ROPAS
Research On Program Analysis System
National Creative Research Initiative Center 1998-2003
Programming Research Laboratory, School of Computer Science & Engineering
Seoul National University

ROPAS MEMO
2005-23
October 21, 2005

Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis

Oukseh Lee, Hongseok Yang, and Kwangkeun Yi
{oukseh; hyang; kwang}@ropas.snu.ac.kr
School of Computer Science & Engineering
Seoul National University, Korea

October 21, 2005

Abstract

We present a program analysis that can automatically discover the shape of complex pointer data structures. The discovered invariants are, then, used to verify the absence of safety errors in the program, to check whether the program preserves the data consistency, or, in some cases, to show the full correctness of the program. Our analysis extends the shape analysis of Sagiv et al. with grammar annotations, which can precisely express the shape of complex data structures. We demonstrate the usefulness of our analysis with three examples; binomial heap construction, the Schorr-Waite tree disposal, and the Schorr-Waite tree traversal. For a binomial heap construction algorithm, our analysis returns a grammar that precisely describes the shape of a binomial heap; for the Schorr-Waite disposal, it proves that the input tree is completely disposed; for the Schorr-Waite tree traversal, our analysis shows that at the end of the execution, the result is a tree and there are no memory leaks. We prove the correctness of our analysis by compiling it into separation logic.

1 Introduction

We show that a static program analysis can automatically verify imperative, pointer programs such as binomial heap construction, the Schorr-Waite tree disposal, and the Schorr-Waite tree traversal. The verified properties are: for a binomial heap construction algorithm, our analysis verifies that the returned heap structure is a binomial heap; for the Schorr-Waite disposal, it verifies that the input tree is completely disposed; for the Schorr-Waite tree traversal, it verifies that the output tree is a binary tree, and there are no memory leaks. In all three cases, the analysis took less than 0.1 second in Intel Pentium 3.0C with 1GB memory and the analysis result is simple and human-readable.

Note that although the three programs handle regular heap structures such as binomial heaps and trees, the web of pointers (e.g., cycles) and their imperative operations (e.g., pointer swapping) are fairly challenging for fully automatic static verification without any annotation from the programmer.

We stand on two shoulders:

- The static analysis is an extension to Sagiv et al.'s shape analysis [SRW02] for separation logic [Rey02, OYR04]. For an improved accuracy, we associate grammars, which finitely summarize run-time heap structures, with the summary nodes of the shape graphs. This enrichment of shape graph by grammars provides the analysis with an ample space for it to precisely capture the imperative effects on dynamic heap structures. The grammar is unfolded, i.e., generates a heap structure on demand to expose an exact heap shape. The grammar is also folded to replace an exact heap structure by a more abstract nonterminal representation. To ensure the termination of the analysis, the grammar merges multiple production rules into a single one, and unify two nonterminals; this simplification of grammar makes the grammar size remain within a practical bound.

- The static analysis' correctness is proven via the separation logic [Rey02]. The static analysis is compositionally defined over the input program structures by using the usual least fixpoint operator for the loops. Involved abstract operations are over the shape graphs with grammars. The semantics (concretization) of the shape graphs are defined as assertions in separation logic. Each abstract operator's correctness is proven by showing that the separation-logic assertion for the input graph to the operator implies the separation-logic assertion for the operator's output graph. The input program C wrapped by the input and output assertions $\{P\}\,C\,\{Q\}$ is always a provable Hoare triple by the separation logic's proof rules.

1.1 Comparison

There have been many works to try to verify pointer programs automatically. However, most of those approaches require the user to provide invariants, as opposed to inferring those invariants automatically [BRS99, JJKS97, MS01], with the notable exception of the shape analysis of Sagiv et al. [SRW98, SRW02]

Our analysis borrows several interesting ideas from Sagiv's shape analysis [SRW02]. Our analysis represents a program invariant using a set of shape graphs, where each shape graph consists of either concrete or abstract nodes. The analysis also uses the idea of refining an abstract node, often called focus or materialization in Sagiv's shape analysis. However, our analysis goes beyond what Sagiv's shape analysis achieves currently, by automatically discovering what Sagiv called instrumentation predicates. We do not ask the user to provide instrumentation predicates, and instead attempts to automatically discover those predicates. Moreover, the discovered instrumentation predicates are high-level in the sense that they match up with the usual intuitions of programmers. The price we pay for this enhanced automation is that theoretically our analysis more often fails to produce meaningful results, simply by returning the top. However, our experimental results show that for programs whose sharing patterns mostly arise from cycles, our analysis infers invariants, which often are precise enough to prove the full correctness.

1.2 Outline

Section 2 briefly describes the programming language that we would analyze, and Section 3 reviews separation logic which we use to give the meaning of abstract values. Then, we explain the key ideas of our analysis, using a simpler version; this simpler version analyzes a program well, only when the recursive data structures in the program are tree-like structures with no shared nodes. Section 4 and 5 explain the abstract domain and operators for this domain, and Section 6 uses these domain and operators to define the analyzer. The simpler version is extended to the full analysis in Section 7; the full version can handle the command that disposes cells, consequently dangling pointers, and it also can discover interesting invariants when programs use data structures with "cyclic" sharing, such as binomial heaps. In Section 8, we demonstrate the accuracy of our analysis using binomial heap construction algorithm, and the Schorr-Waite tree traversing and disposing algorithms. In all three cases, the analysis shows the partial correctness of the algorithms, which includes the simple safety properties, such as the absence of null-pointer or dangling pointer dereferences. Finally, we prove the correctness of our analysis by compiling it into separation logic in Section 9.

2 Programming Language

We use the standard while language with additional constructs for pointer manipulations.

$$\begin{array}{lll} \textit{Vars} \;\; x \qquad \textit{Fields} & f \in \{0,1\} & \\ \textit{Boolean Expressions} & B \;::=\; x \,\texttt{=}\, y \mid \texttt{!}B & \\ \textit{Commands}\; C & ::= & x \,\texttt{:=}\, \texttt{nil} \mid x \,\texttt{:=}\, \texttt{new} \\ & \mid & x \,\texttt{:=}\, x \mid x \,\texttt{:=}\, x\texttt{->}f \mid x\texttt{->}f \,\texttt{:=}\, x \\ & \mid & C\,;C \mid \texttt{if}\; B\; C\; C \mid \texttt{while}\; B\; C \end{array}$$

This language assumes that every heap cell is binary, and has two fields 0 and 1. A heap cell is allocated by `x := new`, and the contents of such an allocated cell is accessed by the field-dereference operation `->`; the f field of cell x is read by $y \,\texttt{:=}\, x\texttt{->}f$, and updated by $x\texttt{->}f \,\texttt{:=}\, y$. Note that the language prevents nil from occurring in the boolean expressions or the field update $x\texttt{->}f \,\texttt{:=}\, E$; we took this constraints in order to simplify the presentation of our algorithm. All the other constructs in the language are standard.

3 Separation Logic with Recursive Predicates

In this section, we review the assertion language in separation logic, and explain our extension of the language with recursive predicates.

Let *Loc* and *Val* be an unspecified infinite set such that nil $\notin$ *Loc* and $\mathit{Loc} \cup \{\text{nil}\} \subseteq \mathit{Val}$. We consider separation logic for the following semantic domains.

$$\begin{array}{rcl} \mathit{Stack} & = & \mathit{Var} \rightharpoonup \mathit{Val} \\ \mathit{Heap} & = & \mathit{Loc} \rightharpoonup \mathit{Val} \times \mathit{Val} \\ \mathit{State} & = & \mathit{Stack} \times \mathit{Heap} \end{array}$$

This domain implies that a state has the stack and heap components, and that the heap component of a state has finitely many binary cells.

The assertion language in separation logic is given by the following grammar:[1]

$$\begin{array}{rcl} P & ::= & E = E \mid \mathsf{emp} \mid (E \mapsto E, E) \mid P * P \\ & & \mid \mathsf{true} \mid P \wedge P \mid P \vee P \mid \neg P \mid \forall x.\, P \end{array}$$

Separating conjunction $P * Q$ is the most important, and it expresses the splitting of heap storage; $P * Q$ means that the heap can be split into two parts, so that P holds for the one and Q holds for the other. The other special constructs are standard, except $(E \mapsto E_1, E_2)$ and `emp`. Points-to predicate $(E \mapsto E_1, E_2)$ means that the heap contains only one cell, the location of the unique cell is E, and the two fields of the cell have E_1 and E_2, respectively; and empty predicate `emp` means that the heap is empty.

We often use precise equality and iterated separating conjunction, both of which we define as syntactic sugars. Let X be a finite set $\{x_1, \ldots, x_n\}$ where all x_i's are different.

$$\begin{array}{rcl} E \doteq E' & \triangleq & E = E' \wedge \mathsf{emp} \\ \bigodot_{x \in X} A_x & \triangleq & \left\{ \begin{array}{ll} A_{x_1} * \ldots * A_{x_n} & \text{if } X \neq \emptyset \\ \mathsf{emp} & \text{otherwise} \end{array} \right. \end{array}$$

In this paper, we use the extension of the basic assertion language with recursive predicates:

$$\begin{array}{rcl} P & ::= & \ldots \alpha(E, \ldots, E) \mid \mathsf{let}\; \Gamma \;\mathsf{in}\; P \\ \Gamma & ::= & \alpha(x_1, \ldots, x_n) = P \mid \Gamma, \Gamma \end{array}$$

[1] The assertion language also has the adjoint $-\!\!*$ of $*$. But this adjoint is not used in this paper, so we omit it here.

The extension allows the definition of new recursive predicates by least-fixed points in "$\mathsf{let}\ \Gamma\ \mathsf{in}\ P$", and the use of such defined recursive predicates in $\alpha(E, \ldots, E)$. To ensure the existence of the least-fixed point in $\mathsf{let}\ \Gamma\ \mathsf{in}\ P$, we will consider only well-formed Γ where all recursively defined predicates appear in positive positions.

A recursive predicate in this extended language means a set of heap objects. A *heap object* is a pair of location (or locations) and heap. Intuitively, the first component denotes the starting address of a data structure, and the second the cells in the data structure. For instance, if a linked list is seen as a heap object, the location of the head of the list becomes the first component, and the cells in the linked list becomes the second component.

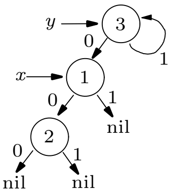
The precise semantics of this assertion language is given by a forcing relation $\models$. For a state (s, h) and an environment η for recursively defined predicates, we define inductively when an assertion P holds for (s, h) and η. We show the sample clauses below; the full definition appear in Appendix A.

$$\begin{array}{ll} (s,h),\eta \models \alpha(E) & \text{iff} \quad ([\![E]\!]s, h) \in \eta(\alpha) \\ (s,h),\eta \models \mathsf{let}\ \alpha(x){=}P\ \mathsf{in}\ Q & \text{iff} \quad (s,h),\eta[\alpha{\to}k] \models P \\ \multicolumn{2}{l}{\quad (\text{where } k = \text{fix}\, \lambda k_0.\{(v,h) \mid (s[x{\to}v],h),\eta[\alpha{\to}k_0] \models P\})} \end{array}$$

4 Abstract Domain

4.1 Shape Graph

Our analysis interprets a program as a (nondeterministic) transformer of *shape graphs*. A shape graph is an abstraction of a concrete state; this abstraction maintains the basic "structure" of the state, but abstracts away all the other details. For instance, consider a state $([x{\to}1, y{\to}3], [1{\to}\langle 2, \text{nil}\rangle, 2{\to}\langle \text{nil}, \text{nil}\rangle, 3{\to}\langle 1, 3\rangle])$:



We obtain a shape graph from this state in two steps. First, we replace the specific addresses, such as 1 and 2, by *symbolic locations*; we introduce three symbol a, b, c, and represent the state by $([x{\to}a, y{\to}c], [a{\to}\langle b, \text{nil}\rangle, b{\to}\langle \text{nil}, \text{nil}\rangle, c{\to}\langle a, c\rangle])$. Note that this process abstracts away the specific addresses from the state, and just keeps the relationship between the addresses in the state. Second, we abstract heap cells a and b by a grammar. Thus, this step transforms the state to $([x{\to}a, y{\to}c], [a{\to}\mathsf{tree}, c{\to}\langle a, c\rangle])$, where $a{\to}\mathsf{tree}$ means that a is the address of the root of a tree, whose structure is summarized by grammar rules fir nonterminal tree. The grammar conversion occurs only when the the tree does not contain other cells in the heap.

The formal definition of a shape graph is given as follows:

$$\begin{aligned} \mathit{SymLoc} &\triangleq \{a, b, c, \ldots\} \\ \mathit{NonTerm} &\triangleq \{\alpha, \beta, \gamma, \ldots\} \\ \mathit{Graph} &\triangleq (\mathit{Vars} \rightharpoonup \mathit{SymLoc}) \times \\ &\quad\ (\mathit{SymLoc} \rightharpoonup \{\text{nil}\} + \mathit{SymLoc}^2 + \mathit{NonTerm}) \end{aligned}$$

Here the set of nonterminals is disjoint from *Vars* and *SymLoc*; these nonterminals represent recursive heap structures such as tree or list. Each shape graph has two components (s, g). The first component s maps describes stack variables to symbolic locations. The other component g describes heap cells reachable from each symbolic location. For each a, either no heap cells can be reached from a ($g(a) = \text{nil}$); or, a is a binary cell with contents $\langle b, c\rangle$ ($g(a) = \langle b, c\rangle$); or, the cells reachable from a form a heap structure specified by a nonterminal α ($g(a) = \alpha$). In the first two cases, the contents of some reachable cells from a are expressed exactly, either none or a binary cell with $\langle b, c\rangle$. The third case summarizes, by the grammar's production rules for α, the shape of reachable cells from a.

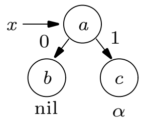
The semantics (or concretization) of a shape graph (s, g) is given by a translation into an assertion in separation logic:

$$\begin{aligned}
\mathsf{means_s}(s, g) &\triangleq \exists\vec{a}.\ \mathtt{true} * (\textstyle\bigodot_{x\in\mathrm{dom}(s)}\, x \doteq s(x))\\
&\qquad * (\textstyle\bigodot_{a\in\mathrm{dom}(g)}\, \mathsf{means_v}(a, g(a)))\\
\mathsf{means_v}(a, \text{nil}) &\triangleq a \doteq \text{nil}\\
\mathsf{means_v}(a, \alpha) &\triangleq \alpha(x)\\
\mathsf{means_v}(a, \langle a_1, a_2\rangle) &\triangleq a \mapsto a_1, a_2
\end{aligned}$$

The translation function $\mathsf{means_s}$ calls a subroutine $\mathsf{means_v}$ to get the translation of the value of $g(a)$, and then, it existentially quantifies all the symbolic locations appearing in the translation. For instance, $\mathsf{means_s}([x\to a, y\to c], [a\to\mathsf{tree}, c\to\langle a, c\rangle])$ is

$$\exists ac.\, (x \doteq a) * (y \doteq c) * \mathsf{tree}(a) * (c \mapsto a, c).$$

When we present a shape graph, we interchangeably use the set notation and a graph picture. Each variable or symbolic location becomes a node in a graph, and s and g are represented by edges or annotations. For instance, we draw a shape graph $(s, g) = ([x\to a], [a\to\langle b, c\rangle, b\to\text{nil}, c\to\alpha])$ as:



Note that $g(a)$ which is a pair is represented as two edges and $g(b)$ and $g(c)$ which are respectively nil and a nonterminal are represented as annotations to the nodes.

4.2 Grammar

A grammar gives the meaning of nonterminals in a shape graph. We define a *grammar* R as a finite partial function from nonterminals (the lhs of production rules) to $\wp_{\mathsf{nf}}(\{\text{nil}\} + (\{\text{nil}\} + \mathit{NonTerm})^2)$ (the rhs of production rules), where $\wp_{\mathsf{nf}}(X)$ is the family of all nonempty finite subsets of X.

$$\begin{aligned}
&\mathit{Grammar} \triangleq\\
&\qquad \mathit{NonTerm} \rightharpoonup \wp_{\mathsf{nf}}(\{\text{nil}\} + (\{\text{nil}\} + \mathit{NonTerm})^2)
\end{aligned}$$

Set $R(\alpha)$ contains all the possible shapes of heap objects for α. If $\text{nil} \in R(\alpha)$, α can be the empty heap object. If $\langle\beta, \gamma\rangle \in R(\alpha)$, then some heap object for α can be split into a root cell, the left heap object β, and the right heap object γ. For instance, if $R(\mathsf{tree}) = \{\text{nil}, \langle\mathsf{tree}, \mathsf{tree}\rangle\}$ (i.e., in the production rule notation, $\mathsf{tree} ::= \text{nil} \mid \langle\mathsf{tree}, \mathsf{tree}\rangle$), then tree represents binary trees.

In our analysis, we use only *well-formed* grammars, where all nonterminals appearing in the range of a grammar are defined in the grammar.

The semantics of a grammar is again defined by translating the grammar to recursive predicate declarations in separation logic. A grammar R is translated into a context Γ for recursive predicates that is defined exactly for $\mathrm{dom}(R)$, and satisfies the following: when $\text{nil} \notin R(\alpha)$, $\Gamma(\alpha)$ is

$$\alpha(a) = \bigvee_{(v,v')\in R(\alpha)} \begin{array}{l}\exists bb'.(a \mapsto b, b') * \mathsf{means_v}(b, v)\\ \quad * \mathsf{means_v}(b', v')\end{array}$$

where neither b nor b' appear in a, v or v'; in the other case, $\Gamma(\alpha)$ is identical as above except that $a \doteq \mathrm{nil}$ is added as an additional disjunct. For instance, $\mathsf{means_g}([\mathsf{tree}{\to}\{\mathrm{nil}, \langle \mathsf{tree}, \mathsf{tree}\rangle\}])$ is a context

$$\{\mathsf{tree}(a) = a \doteq \mathrm{nil} \vee \exists bb'.(a \mapsto b, b') * \mathsf{tree}(b) * \mathsf{tree}(b')\}.$$

4.3 Abstract Domain = Shape Graphs + Grammar

The abstract domain for our analysis consists of pairs of shape graph set and grammar:

$$\widehat{D} \overset{\Delta}{=} \{\top\} + \wp_{\mathsf{nf}}(\mathit{Graph}) \times \mathit{Grammar}$$

The first element indicates that our analysis fails to produce any meaningful results for a given program, because the program has the safety errors, or the program uses complex data structures so that our analysis fails to capture them.

The meaning (or concretization) of each abstract state $(\mathcal{G}, R)$ in $\widehat{D}$ is given by a translation into a separation-logic assertion:

$$\mathsf{means}(\mathcal{G}, R) \;\overset{\Delta}{=}\; \mathsf{let}\ \mathsf{means_g}(R)\ \mathsf{in}\ \bigvee_{(s,g)\in\mathcal{G}} \mathsf{means_s}(s, g)$$

5 Normalized Abstract States and Normalization Function

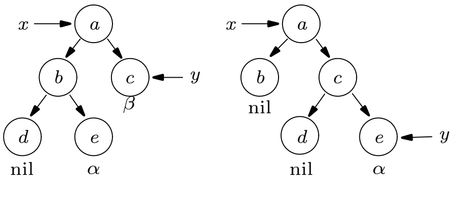
The main strength of our analysis is to automatically discover a grammar which describes, in an "intuitive" level, invariants for heap data structures, and to abstract concrete states according to this discovered grammar. This inference of high-level grammars is mainly done by the normalization function $\mathsf{normalize}$ from $\widehat{D}$ to a subdomain $\widehat{D}^{\nabla}$ of *normalized abstract states*. In this section, we explain these two notions, normalized abstract states and function $\mathsf{normalize}$.

An abstract state $(\mathcal{G}, R)$ is normalized if it satisfies the following two conditions. The first condition is that all the shape graphs (s, g) in $\mathcal{G}$ are abstract enough: all the recognizable heap objects are replaced by nonterminals. Note that this condition on $(\mathcal{G}, R)$ is about individual shape graphs in $\mathcal{G}$. We call a shape graph *normalized* if it satisfies this condition. The second condition is the absence of redundancies: all shape graphs are not *similar*, and all nonterminals have *non-similar* definitions.

5.1 Normalized Shape Graphs

The first requirement of an abstract state $(\mathcal{G}, R)$ being normalized is that all shape graphs in $\mathcal{G}$ are normalized. A shape graph is normalized when it is "maximally" folded. To state the precise definition of a normalized shape graph, we need to classify symbolic locations in a shape graph by whether they are abstractable or not. Let's say that a symbolic location a is *shared* if and only if there are more than one references to a. A symbolic location a is called *abstractable* or *foldable* in (s, g) if $g(a)$ is a pair and there is no path from a to a shared symbolic location.

A shape graph (s, g) is called *normalized* if and only if for all symbolic locations a in $\mathrm{dom}(g)$, a is not foldable.



In the left shape graph, symbolic location b is foldable: b is a pair and does not reach any shared symbolic location. So, the left graph is not normalized. In the right graph, there are no foldable symbolic locations: both a and c are pairs but they reach symbolic location e shared with y. So, the right graph is normalized.

5.2 Similarity

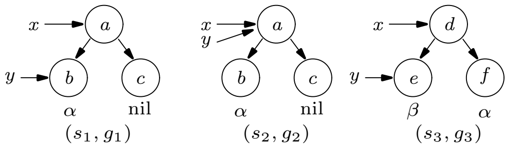
The second requirement for $(\mathcal{G}, R)$ being normalized is that there are no redundancies in shape graphs in $\mathcal{G}$ and grammar R: no two shape graphs in $\mathcal{G}$ have the "similar structures," the production rules of each nonterminal in R do not have two "similar cases," and the definitions of two nonterminals are not "similar." Two shape graphs are similar if they are identical when the symbolic locations, nonterminals, and nil are erased. The two cases in the production rules for one nonterminal are defined similar if they become identical when symbolic locations, nonterminals, and nil are erased from the rules. Two nonterminals are similar if one's production rules have similar cases in the other's production rules.

The precise definition of similar structures is given by a relation $\sim^G$ (*graph similarity*), that of similar cases by $\sim^C$ (*case similarity*), and that of similar grammar definitions by $\sim^D$ (*definition similarity*).

Two shape graphs are similar when they have the similar structures. Let S be a substitution that renames symbolic locations by symbolic locations. We say that two shape graphs (s, g) and (s', g') are *similar upto* S, denoted $(s, g) \sim^G_S (s', g')$, if and only if

1. $\mathrm{dom}(s) = \mathrm{dom}(s')$ and $S(\mathrm{dom}(g)) = \mathrm{dom}(g')$;
2. for all $x \in \mathrm{dom}(s)$, $S(s(x)) = s'(x)$; and
3. for all $a \in \mathrm{dom}(g)$,
   (a) if $g(a)$ is not a pair, the $g'(S(a))$ is not a pair, and
   (b) if $g(a) = \langle b, c\rangle$, then $g'(S(a)) = \langle S(b), S(c)\rangle$.

Intuitively, two shape graphs are S-similar, when equating nil and all nonterminals make the graphs identical upto renaming S. We call (s, g) and (s', g') is similar, denoted $(s, g) \sim (s', g')$, if and only if there is a renaming relation S such that $(s, g) \sim^G_S (s', g')$. For instance, consider the following three shape graphs:



(s_1, g_1) and (s_2, g_2) are not similar because $s_1(x) \neq s_1(y)$ and $s_2(x) = s_2(y)$; that is, we cannot find a renaming substitution S such that $S(s_1(x)) = S(s_1(y))$ (condition 2). (s_1, g_1) and (s_3, g_3) are similar because we can find a renaming substitution $\{d/a, e/b, f/c\}$ that makes (s_1, g_1) identical to (s_3, g_3) when nil and all nonterminals are erased.

Two cases in the grammar definitions are similar when they have similar structures. Cases e_1 and e_2 are similar, denoted $e_1 \sim^C e_2$, if and only if both e_1 and e_2 are pairs or neither e_1 and e_2 are pairs. For instance, all the pair cases are similar, and the nil case is similar to itself. However, the nil case is not similar to the pair case.

The definitions of two nonterminals are similar when their cases are similar. The definitions E_1 and E_2 are similar, denoted $E_1 \sim^D E_2$, if and only if, for all cases e in E_1, E_2 has a similar case e' to e ($e \sim^C e'$), and vice versa. For example, consider the following definitions:

$$\begin{array}{l} \alpha ::= \langle \beta, \mathrm{nil}\rangle \\ \beta ::= \mathrm{nil} \mid \langle \beta, \mathrm{nil}\rangle \\ \gamma ::= \langle \gamma, \gamma\rangle \mid \langle \alpha, \mathrm{nil}\rangle \end{array}$$

The definitions of α and γ are similar because $\langle\gamma,\gamma\rangle$ is similar to $\langle\beta,\mathrm{nil}\rangle$, and $\langle\alpha,\mathrm{nil}\rangle$ is also similar to $\langle\beta,\mathrm{nil}\rangle$. But the definitions of α and β are not similar since α does not have a case similar to nil. By the same reason, the definitions of β and γ are not similar.

5.3 Normalized Abstract States

An abstract state $(\mathcal{G},R)$ is *normalized* if and only if all shape graphs in $\mathcal{G}$ are normalized and $(\mathcal{G},R)$ does not have redundancies.

Definition 1 (Normalized Abstract States) *An abstract state $(\mathcal{G},R)$ is normalized if and only if*

1. *for all shape graphs in $\mathcal{G}$ are normalized; and*
2. *for all shape graphs (s_1,g_1) and (s_2,g_2) in $\mathcal{G}$, $((s_1,g_1)\sim(s_2,g_2))$ implies that $(s_1,g_1)=(s_2,g_2)$.*
3. *for all α in $\mathrm{dom}(R)$ and for all cases e_1 and e_2 in $R(\alpha)$, $e_1\sim^C e_2$ implies that $e_1=e_2$.*
4. *for all α and β in $\mathrm{dom}(R)$, $R(\alpha)\sim^D R(\beta)$ implies that $\alpha=\beta$.*

Normalized abstract states are the analysis results expressed in a human-readable level.

5.4 k-Bounded Normalized States

Another gain from the normalized abstract domain is that it ensures that our analyzer terminates. The normalized abstract domain itself has an infinite length of chain but when we give a restriction on the number of symbolic locations, every chain becomes finite. We say that the abstract domain $\widehat{D}_k$ is k-bounded if and only if every abstract state in $\widehat{D}_k$ has only shape graphs whose number of symbolic locations is less than k. We prove the k-bounded normalized domain has only finite length of (strictly increasing) chains by finding a "quotiented" sub-domain which is finite.

Lemma 2 (Finite Quotiented Domain) *For the k-bounded normalized abstract domain $\widehat{D}^{\nabla}_k$, there exists a finite sub-domain of $\widehat{D}^{\nabla}_{\mathsf{fin}}$ which satisfies that for all $(\mathcal{G},R)\in\widehat{D}^{\nabla}_k$, there exists $(\mathcal{G}',R')\in\widehat{D}^{\nabla}_{\mathsf{fin}}$ such that $\mathsf{means}(\mathcal{G},R)\iff\mathsf{means}(\mathcal{G}',R')$.*

Proof sketch. Consider a normalized abstract domain $\widehat{D}^{\nabla}_{\mathsf{fin}}$ with a set A of k symbolic locations, and a set N of three nonterminals. This domain is finite because it is composed of finite domains. For all $(\mathcal{G},R)$ in the k-bounded normalized abstract domain, we have to find an abstract state whose meaning is the same as $(\mathcal{G},R)$ and which is included in $\widehat{D}^{\nabla}_{\mathsf{fin}}$. It can be obtained by renaming symbolic locations and nonterminals:

- Let's rename symbolic locations of each shape graph in $\mathcal{G}$ so that $\mathcal{G}$ has only symbolic names in A. It is possible because it is k-bounded.
- Let's rename nonterminals in $(\mathcal{G},R)$ so that $(\mathcal{G},R)$ has only nonterminals in N. It is possible because $(\mathcal{G},R)$ has indeed no more than three nonterminals. The number of equivalence classes of $\sim^D$ is 3: $[\{\mathrm{nil}\}]$, $[\{\langle\mathrm{nil},\mathrm{nil}\rangle\}]$ and $[\{\mathrm{nil},\langle\mathrm{nil},\mathrm{nil}\rangle\}]$. By the condition 4 of Definition 1, $(\mathcal{G},R)$ has no more than three nonterminals.

Those renaming does not change the meaning of $(\mathcal{G},R)$ and makes $(\mathcal{G},R)$ be included in $\widehat{D}^{\nabla}_{\mathsf{fin}}$. □

5.5 Normalization Function

The normalization function $\mathsf{normalize}$ transforms $(\mathcal{G}, R)$ to an normalized $(\mathcal{G}', R')$ with a further abstraction. That is, the following holds:

$$\mathsf{means}(\mathcal{G}, R) \implies \mathsf{means}(\mathcal{G}', R')$$

Transformation $\mathsf{normalize}$ is the composition of five subroutines:

$$\mathsf{normalize} = \mathsf{bound}^k \circ \mathsf{simplify} \circ \mathsf{unify} \circ \mathsf{fold} \circ \mathsf{deljunk}.$$

The first subroutine $\mathsf{deljunk}$ removes all the “imaginary” sharing and garbage due to constant symbolic locations, so that it makes the real sharing and garbage easily detectable in syntax. The $\mathsf{deljunk}$ procedure applies the following two rules until an abstract state does not change. In the definition, “$\uplus$” is a disjoint union of sets, and “$\cdot$” is a union of partial maps with disjoint domains.

(alias) $(\mathcal{G} \uplus \{(s \cdot [x \to a], g \cdot [a \to \text{nil}])\}, R)$
$\rightsquigarrow (\mathcal{G} \cup \{(s \cdot [x \to a'], g \cdot [a' \to \text{nil}])\}, R)$
where a should appear in (s, g) and a' is fresh.

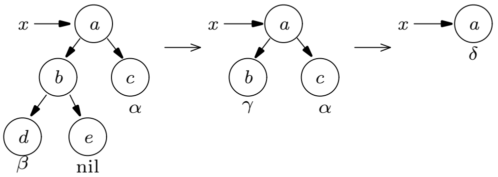
(gc) $(\mathcal{G} \uplus \{(s, g \cdot [a \to \text{nil}])\}, R) \rightsquigarrow (\mathcal{G} \cup \{(s, g)\}, R)$
where a does not appear in (s, g)

For instance, $([x \to a, y \to a], [a \to \text{nil}])$ has the same meaning as $([x \to a, y \to b], [a \to \text{nil}, b \to \text{nil}])$. We rewrite the former by the latter by **(alias)**. $([x \to a], [a \to \text{nil}, b \to \text{nil}])$ has the same meaning as $([x \to a], [a \to \text{nil}])$. We rewrite the former by the latter by **(gc)**.

The second subroutine fold converts shape graphs to normal forms. For each shape graph (s, g) in $\mathcal{G}$, fold replaces all foldable symbolic locations by nonterminals. The fold procedure applies the following rule until an abstract state does not change:

(fold) $(\mathcal{G} \uplus \{(s, g \cdot [a \to \langle a_1, a_2 \rangle, a_1 \to v_1, a_2 \to v_2])\}, R)$
$\rightsquigarrow (\mathcal{G} \cup \{(s, g \cdot [a \to \alpha])\}, R \cdot [\alpha \to \{\langle v_1, v_2 \rangle\}])$
where neither a_1 nor a_2 appears in the range of g and s, and α is fresh.

The rule recognizes that the symbolic locations a_1 and a_2 are accessed only via a. Then, it represents cell a, plus the reachable cells from a_1 and a_2 by a nonterminal α. For example, consider:

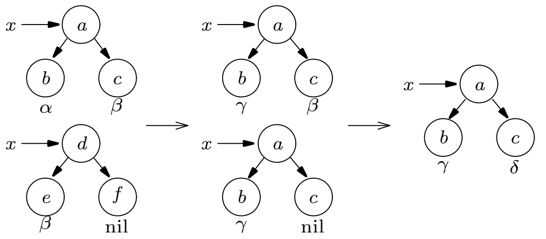


In the first shape graph of the above example, symbolic location b is foldable. By applying rule **(fold)**, we obtain the second one with the production rule $\gamma ::= \langle \beta, \text{nil} \rangle$ for a new nonterminal γ. Again, a is foldable so we achieve the last one by **(fold)** with extra grammar rule $\delta ::= \langle \gamma, \alpha \rangle$.

The third subroutine unify merges two similar shape graphs in $\mathcal{G}$. Let (s, g) and (s', g') be similar shape graphs by the identity renaming Δ (i.e., $(s, g) \sim^G_\Delta (s', g')$.) Then, these two shape graphs are almost identical; the only exception is when $g(a)$ and $g'(a)$ are nonterminals or nil. unify eliminates all such differences in two shape graphs; if $g(a)$ and $g'(a)$ are nonterminals, then unify changes g and g', so that they map a to the same fresh nonterminal γ, and then it defines γ to cover both α and β. The unify procedure applies the following rule to $(\mathcal{G}, R)$ until the abstract state doesn't change:

$$
\begin{array}{ll}
\textbf{(unify)} & (\mathcal{G} \uplus \{(s_1, g_1 \cdot [a_1 \to \alpha_1]), (s_2, g_2 \cdot [a_2 \to \alpha_2])\}) \\
 & \rightsquigarrow \begin{pmatrix} \mathcal{G} \cup \{(Ss_1, Sg_1 \cdot [a_2 \to \beta]), (s_2, g_2 \cdot [a_2 \to \beta])\}, \\ R \cdot [\beta \to R(\alpha_1) \cup R(\alpha_2)] \end{pmatrix} \\
 & \text{where } (s_1, g_1 \cdot [a_1 \to \alpha_1]) \sim^G_S (s_2, g_2 \cdot [a_2 \to \alpha_2]), \\
 & S(a_1) \equiv a_2,\ \alpha_1 \not\equiv \alpha_2, \text{ and } \beta \text{ is fresh.}
\end{array}
$$

The rule recognizes two similar shape graphs which have different nonterminals at the same position, and replaces those nonterminals by fresh nonterminal β that covers the two nonterminals. For example, the left two shape graphs are unified by the following steps:



We first replace the left children α and β by γ which covers both; that is, given grammar R, we add $[\beta \to R(\alpha) \cup R(\beta)]$ to R. Then we replace the right children α and nil by δ which covers both.

The last subroutine simplify reduces the complexity of grammar by combining similar cases or similar definitions. This subroutine applies three rules repeatedly:

- If the definition of a nonterminal has two similar cases $\langle \beta, v \rangle$ and $\langle \beta', v' \rangle$, and β and β' are different nonterminals, unify nonterminals β and β'. Apply the same rule for the second field.
- If the definition of a nonterminal has two similar cases (β, v_2) and (nil, v_2'), add the nil case to $R(\beta)$. Apply the same rule for the second field.
- If the definition of two nonterminals are similar, the nonterminals are unified.

The simplify procedure applies the following rules to $(\mathcal{G}, R)$ until the abstract value doesn't change:

$$
\begin{array}{ll}
\textbf{(case)} & (\mathcal{G}, R) \rightsquigarrow (\mathcal{G}, R)\{\beta/\alpha\} \\
 & \text{where } \{\langle \alpha, v \rangle, \langle \beta, v' \rangle\} \subseteq R(\gamma), \text{ and } \alpha \not\equiv \beta. \\
 & \text{(same for the second field)} \\
\textbf{(nil)} & (\mathcal{G}, R \cdot [\alpha \to E \uplus \{\langle \beta, v \rangle, \langle \text{nil}, v' \rangle\}]) \\
 & \rightsquigarrow (\mathcal{G}, R'[\beta \to R'(\beta) \cup \{\text{nil}\}]) \\
 & \text{where } R' = R \cdot [\alpha \to E \uplus \{\langle \beta, v \rangle, \langle \beta, v' \rangle\}] \\
 & \text{(same for the second field)} \\
\textbf{(def)} & (\mathcal{G}, R) \rightsquigarrow (\mathcal{G}, R)\{\beta/\alpha\} \text{ where } R(\alpha) \sim R(\beta)
\end{array}
$$

Note that we apply a substitution $\{\alpha/\beta\}$ to an abstract state $(\mathcal{G}, R)$ in a usual way except that we remove the definition of β from R and re-define α so that α covers both α and β:

$$
\begin{array}{l}
(\mathcal{G}, R \cdot [\alpha \to E_1, \beta \to E_2])\{\alpha/\beta\} \stackrel{\Delta}{=} \\
\qquad (\mathcal{G}\{\alpha/\beta\}, R\{\alpha/\beta\} \cdot [\alpha \to (E_1 \cup E_2)\{\alpha/\beta\}]).
\end{array}
$$

For example, consider the following transitions. The initial grammar says that α means complete binary trees whose height is 0, 2 or 3:

$$\begin{array}{ll} & \alpha ::= \mathrm{nil} \mid \langle\beta,\beta\rangle \mid \langle\gamma,\gamma\rangle\,,\ \beta ::= \langle\gamma,\gamma\rangle\,,\ \gamma ::= \langle\mathrm{nil},\mathrm{nil}\rangle \\ \overset{\text{(case)}}{\rightsquigarrow} & \alpha ::= \mathrm{nil} \mid \langle\beta,\beta\rangle\,,\ \beta ::= \langle\beta,\beta\rangle \mid \langle\mathrm{nil},\mathrm{nil}\rangle \\ \overset{\text{(nil)}}{\rightsquigarrow} & \alpha ::= \mathrm{nil} \mid \langle\beta,\beta\rangle\,,\ \beta ::= \langle\beta,\beta\rangle \mid \langle\beta,\mathrm{nil}\rangle \mid \mathrm{nil} \\ \overset{\text{(nil)}}{\rightsquigarrow} & \alpha ::= \mathrm{nil} \mid \langle\beta,\beta\rangle\,,\ \beta ::= \langle\beta,\beta\rangle \mid \mathrm{nil} \\ \overset{\text{(def)}}{\rightsquigarrow} & \alpha ::= \mathrm{nil} \mid \langle\alpha,\alpha\rangle \end{array}$$

In the initial grammar, α's definition has the similar cases $\langle\beta,\beta\rangle$ and $\langle\gamma,\gamma\rangle$, so we apply $\{\beta/\gamma\}$ **(case)**. In the second grammar, the β has the similar cases $\langle\beta,\beta\rangle$ and $\langle\mathrm{nil},\mathrm{nil}\rangle$. Thus, we replace nil by β, and add another case nil to β's definition **(nil)**. We apply **(nil)** once more for the second field. In the fourth grammar, since α and β have similar (in fact, the same) definitions, we apply $\{\alpha/\beta\}$. As a result, we obtain the last grammar that says α describes binary trees, which is much weaker than the original meaning.

The last step bound^k checks the number of symbolic locations in each shape graph. It simply gives $\top$ when one of shape graphs has more than k symbolic locations.

$$\mathsf{bound}^k(\mathcal{G},R) = \\ \begin{cases} (\mathcal{G},R), & \text{if } (s,g) \text{ has no more than } k \text{ symbolic} \\ & \text{locations for all } (s,g)\in\mathcal{G} \\ \top, & \text{otherwise} \end{cases}$$

The normalization function $\mathsf{normalize}$ always terminates for any abstract state: $\mathsf{deljunk}$ strictly decreases the number of shared or unreachable constants; fold strictly decreases the number of symbolic locations; unify strictly decreases the number of shape graphs in a few steps; and $\mathsf{simplify}$ strictly decreases the number of nonterminals plus the number of nil which appears in a pair.

The normalization function $\mathsf{normalize}$ always gives a k-bounded normalized abstract state. The termination of the **(fold)** rule ensures that every shape graph is normalized (the first condition of Definition 1). The termination of the **(unify)** rule ensures that the result does not have similar shape graphs (the second condition). The **(unify)** rule preserves that the every shape graph is normalized. The termination of the **(case)** rule ensures that the definition of a nonterminal does not have similar cases (the third condition), and the termination of the **(def)** rule ensures that the grammar does not have similar definitions (the fourth condition). Note that changing nonterminals in shape graphs does not change the properties that they are normalized and they are not similar to each other. Thus the **(case)** and **(def)** rule does not break the previous conditions. Finally, bound^k gives $\top$ if the number of symbolic locations of a shape graph is more than k.

6 Analysis

Our analyzer consists of two parts: the "forward analysis" of commands C, and the "backward analysis" of boolean expressions B. Both of these interpret C and B as functions on abstract states, and they accomplish the usual goals in the abstract interpretation: for an initial abstract state $(\mathcal{G},R)$, $[\![C]\!](\mathcal{G},R)$ approximates the possible output states, and $[\![B]\!](\mathcal{G},R)$ denotes the result of pruning some states in $(\mathcal{G},R)$ that do not satisfy B.

One particular feature of our analysis is that the analysis also checks the absence of memory errors, such as null-pointer dereference errors. Given a command C and an abstraction $(\mathcal{G},R)$ for input states, the result $[\![C]\!](\mathcal{G},R)$ of analyzing a command C can be either some abstract state $(\mathcal{G}',R')$ or $\top$. $(\mathcal{G}',R')$ means that all the results of C from $(\mathcal{G},R)$ are approximated by

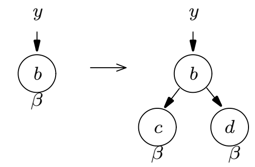
$(\mathcal{G}', R')$, but in addition to this, it also means that no computations of C from $(\mathcal{G}, R)$ can generate memory errors. $\top$, on the other hand, expresses the possibility of memory errors, or indicates that a program uses the data structures whose complexity goes beyond the current capability of the analysis.

The definition of our analyzer is shown in Figure 1.

6.1 Forward Analysis

The forward analysis part of our analyzer abstractly executes each command using the abstract states. The abstract execution of a command follows the real execution of the command. For assignment $x\,\texttt{:=}\,y$ and an input abstract state $(\mathcal{G}, R)$, it "updates" the variable x in each shape graph (s, g) in $\mathcal{G}$ to the abstract value of y. For $x := \texttt{new}$, the analyzer picks three fresh symbols, a for a new location and a_1, a_2 for its contents; and for each shape graph (s, g) in the input $(\mathcal{G}, R)$, the analyzer "allocates" $[a{\to}(a_1, a_2), a_1{\to}\mathrm{nil}, a_2{\to}\mathrm{nil}]$ in g, and updates x to the symbolic location a of the new cell.

The abstract execution of the analyzer often preprocesses "input" shape graphs in the input $(\mathcal{G}, R)$. When the command contains dereferencing operators, the analyzer checks each input shape graph (s, g) to find out whether the dereferenced symbolic location a is mapped to a nonterminal; if so, it looks up the nonterminal $g(a)$ in a grammar, and expands this definition inside the input shape graph. For example, consider the case for computing $[\![x\,\texttt{:=}\,y\,\texttt{->}\,0]\!]\,(\mathcal{G}, R)$ when $\mathcal{G} = \{([x{\to}a, y{\to}b], [a{\to}\mathrm{nil}, b{\to}\beta])\}$ and $R = [\beta ::= \langle\beta, \beta\rangle]$. Since y has a nonterminal β, our analysis unfolds the definition of β once by calling $\mathsf{unfold}(\mathcal{G}, R, y)$. The result of unfolding is:



After this unfolding, it updates the content of x by c. Note that unfold, in fact, gives many (not only one) shape graphs when the grammar definition has many cases. Also, unfold fails when the nonterminal has a nil case. This case means that there may be a null-pointer dereference error during the dereferencing operation. In this case, our analysis gives $\top$ as a result.

For analyzing `if`-statement, prior to analyze each branches, our analysis prunes the input state by using the backward interpretation of our analysis. For "$\texttt{if}\,B\,C_1\,C_2$," it first selects, from the input state, what satisfies B and what satisfies $\texttt{!}B$. Then it analyzes C_1 and C_2 with the pruned results respectively, and joins the two analysis results.

For analyzing a loop, our analysis uses the subdomain $\widehat{D}_k^{\nabla}$ in order to ensure the termination. For "while B C," our analysis does a fixpoint iteration as usual but, additionally, normalizes the abstract state for each fixpoint iteration step. Each fixpoint iteration consists of four procedures: it prunes the previous states with B by the backward analysis; it analyzes C with the pruned result; it joins the analysis result of C, the previous abstract state, and the initial abstract state; and finally, it finds a normal form of the joined result by using $\mathsf{normalize}$. These procedures are repeated until the previous and current states are provably equivalent by our "algorithmic" order $\dot{\sqsubseteq}$ in (in Figure 1). After obtaining a fixpoint, it prunes again by the exit condition $\texttt{!}B$ (not B) of the while loop. This fixpoint iteration always terminates because $\mathsf{normalize}$ is extensive for $\dot{\sqsubseteq}$ and every chain of $\dot{\sqsubseteq}$ is finite in the k-bounded normalized abstract domain by Lemma 2[2].

[2] Lemma 2 works for the approximated order $\dot{\sqsubseteq}$

$$\boxed{\llbracket C \rrbracket : \widehat{D} \to \widehat{D}}$$

$$\begin{array}{rcl}
\llbracket x \texttt{ := new} \rrbracket (\mathcal{G}, R) & = & (\{(s[x{\to}a], g[a{\to} \langle a_1, a_2 \rangle, a_1{\to}\text{nil}, a_2{\to}\text{nil}] \mid (s,g) \in \mathcal{G}\}, R) \\
 & & \text{new } a,\ a_1,\ \text{and } a_2 \\
\llbracket x \texttt{ := nil} \rrbracket (\mathcal{G}, R) & = & (\{(s[x{\to}a], g[a{\to}\text{nil}]) \mid (s,g) \in \mathcal{G}\}, R) \text{ new } a \\
\llbracket x \texttt{ := } y \rrbracket (\mathcal{G}, R) & = & \text{when } y \in \text{dom}(s) \text{ for all } (s,g) \in \mathcal{G}, \\
 & & (\{(s[x{\to}s(y)], g) \mid (s,g) \in \mathcal{G}\}, R) \\
\llbracket x\texttt{->}i \texttt{ := } y \rrbracket (\mathcal{G}, R) & = & \text{when } \mathsf{unfold}(\mathcal{G}, R, x) = \mathcal{G}' \text{ and } y \in \text{dom}(s) \text{ for all } (s,g) \in \mathcal{G}', \\
 & & (\{(s, g[a{\to}(g(a)[i{\to}s(y)])] \mid s(x) = a,\ (s,g) \in \mathcal{G}'\}, R) \\
 & & \text{where } \langle a_1, a_2 \rangle [0{\to}a] = \langle a, a_2 \rangle \text{ and } \langle a_1, a_2 \rangle [1{\to}a] = \langle a_1, a \rangle \\
\llbracket x \texttt{ := } y\texttt{->}i \rrbracket (\mathcal{G}, R) & = & \text{when } \mathsf{unfold}(\mathcal{G}, R, y) = \mathcal{G}', \\
 & & (\{(s[x{\to}a_i], g) \mid g(s(y)) = \langle a_1, a_2 \rangle,\ (s,g) \in \mathcal{G}'\}, R) \\
\llbracket C_1 \texttt{;} C_2 \rrbracket (\mathcal{G}, R) & = & \llbracket C_2 \rrbracket (\llbracket C_1 \rrbracket (\mathcal{G}, R)) \\
\llbracket \texttt{if } B\ C_1\ C_2 \rrbracket (\mathcal{G}, R) & = & \llbracket C_1 \rrbracket (\llbracket B \rrbracket (\mathcal{G}, R)) \mathbin{\dot\sqcup} \llbracket C_2 \rrbracket (\llbracket \texttt{!} B \rrbracket (\mathcal{G}, R)) \\
\llbracket \texttt{while } B\ C \rrbracket (\mathcal{G}, R) & = & \llbracket \texttt{!} B \rrbracket \quad \text{fix}^{\dot\sqsubseteq}\ \lambda A.\mathsf{normalize}(A \mathbin{\dot\sqcup} (\mathcal{G}, R) \mathbin{\dot\sqcup} \llbracket C \rrbracket (\llbracket B \rrbracket A)) \\
\llbracket C \rrbracket A & = & \top \text{ (other cases)}
\end{array}$$

$$\boxed{\llbracket B \rrbracket : \widehat{D} \to \widehat{D}}$$

$$\begin{array}{rcl}
\llbracket x \texttt{=} y \rrbracket (\mathcal{G}, R) & = & \text{when } \mathsf{split}(\mathsf{split}((\mathcal{G}, R), x), y) = (\mathcal{G}', R') \\
 & & (\{(s,g) \mid \mathsf{equal}(g, s(x), s(y)) \text{ hols},\ (s,g) \in \mathcal{G}'\}, R') \\
\llbracket \texttt{!} x \texttt{=} y \rrbracket (\mathcal{G}, R) & = & \text{when } \mathsf{split}(\mathsf{split}((\mathcal{G}, R), x), y) = (\mathcal{G}', R') \\
 & & (\{(s,g) \mid \mathsf{notequal}(g, s(x), s(y)) \text{ hols},\ (s,g) \in \mathcal{G}'\}, R') \\
\llbracket \texttt{!(!} B \texttt{)} \rrbracket (\mathcal{G}, R) & = & \llbracket B \rrbracket (\mathcal{G}, R) \\
\llbracket B \rrbracket A & = & \top \text{ (other cases)}
\end{array}$$

where

$$\mathsf{unfold}((s,g), R, x) = \begin{cases} \{(s,g)\}, & \text{if } g(s(x)) \text{ is a pair} \\ \{(s, g[a{\to} \langle a_1, a_2 \rangle, a_1{\to}v_1, a_2{\to}v_2]) \mid \langle v_1, v_2 \rangle \in R(a)\} & \text{if } s(x) = a,\ g(a) = \alpha \text{ and nil} \notin R(\alpha) \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

$$\begin{array}{rl}
\mathsf{unfold}(\mathcal{G}, R, x) = & \cup_{(s,g) \in \mathcal{G}} \mathsf{unfold}((s,g), R, x) \\
 & \text{if for all } (s,g) \in \mathcal{G},\ \mathsf{unfold}((s,g), R, x) \text{ is defined; otherwise, undefined.}
\end{array}$$

$$\mathsf{split}((s,g), R, x) = \begin{cases} (\{(s, g[a{\to}\text{nil}])\}, R), & \text{if } s(x) = a,\ g(a) = \alpha \text{ and } R(\alpha) = \{\text{nil}\} \\ (\{(s, g[a{\to}\text{nil}]), (s, g[a{\to}\beta])\}, R[\beta{\to}R(\alpha) - \{\text{nil}\}]), & \text{if } s(x) = a,\ g(a) = \alpha, \\ & \quad R(\alpha) \supseteq \{\text{nil}\} \text{ and } R(\alpha) \neq \{\text{nil}\} \\ (\{(s,g)\}, R), & \text{otherwise} \end{cases}$$

$$\mathsf{split}(\mathcal{G}, R, x) = \dot\bigsqcup{}_{(s,g) \in \mathcal{G}} \mathsf{split}((s,g), R, x) \text{ if for all } (s,g) \in \mathcal{G},\ x \in \text{dom}(s);\ \text{otherwise, } \top.$$

$$\begin{array}{rl}
\mathsf{equal}(g, a, b) = & (a = b) \vee (g(a) = \text{nil} \wedge g(b) = \text{nil}) \\
\mathsf{notequal}(g, a, b) = & \neg \mathsf{equal}(g, a, b)
\end{array}$$

$$\begin{array}{rl}
\llbracket \text{nil} \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket \text{nil} \rrbracket_{R_2} & \\
\llbracket \text{nil} \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket \alpha \rrbracket_{R_2} & \text{iff } \{\text{nil}\} \subseteq R_2(\alpha) \\
\llbracket \alpha \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket \text{nil} \rrbracket_{R_2} & \text{iff } R_1(\alpha) \subseteq \{\text{nil}\} \\
\llbracket \alpha \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket \beta \rrbracket_{R_2} & \text{iff nil} \in R_1(\alpha) \Longrightarrow \text{nil} \in R_2(\beta) \\
 & \text{and} \langle v_1, v_2 \rangle \in R_1(\alpha) \Longrightarrow \exists \langle v_1', v_2' \rangle \in R_2(\beta) \text{ s.t. } \llbracket v_1 \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket v_1' \rrbracket_{R_2} \text{ and } \llbracket v_2 \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket v_2' \rrbracket_{R_2} \\
 & \quad \text{co-inductively} \\
\llbracket \langle a_1, a_2 \rangle \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket \langle a_1, a_2 \rangle \rrbracket_{R_2} & \\
\llbracket (s_1, g_1) \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket (s_2, g_2) \rrbracket_{R_2} & \text{iff } (s_1, g_1) \text{ is not normalized} \wedge ((s_1', g_1'), R_1') = \mathsf{fold}(s_1, g_1) \wedge \llbracket (s_1', g_1') \rrbracket_{R_1'} \mathrel{\dot\sqsubseteq} \llbracket (s_2, g_2) \rrbracket_{R_2} \\
 & \text{or } (s_1, g_1) \text{ is normalized} \wedge (s_1, g_1) \sim_S^G (s_2, g_2) \wedge \forall a \in \text{dom}(g_2).\llbracket (S(g_1))(a) \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket g_2(a) \rrbracket_{R_2} \\
A \mathrel{\dot\sqsubseteq} \top & \\
(\mathcal{G}_1, R_1) \mathrel{\dot\sqsubseteq} (\mathcal{G}_2, R_2) & \text{iff } (s_1, g_1) \in \mathcal{G}_1 \Longrightarrow \exists (s_2, g_2) \in \mathcal{G}_2.\llbracket (s_1, g_1) \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket (s_2, g_2) \rrbracket_{R_2}
\end{array}$$

$$\begin{array}{rl}
A \mathbin{\dot\sqcup} \top & \triangleq \top \\
(\mathcal{G}_1, R_1) \mathbin{\dot\sqcup} (\mathcal{G}_2, R_2) & \triangleq (\mathcal{G}_1 \cup \mathcal{G}_2, R_1 \cup R_2),\ \text{no nonterminals in } (\mathcal{G}_1, R_1) \text{ appear in } (\mathcal{G}_2, R_2) \text{ and vice versa}
\end{array}$$

Figure 1: The Analysis.

6.2 Backward Analysis

The main step of the backward analysis is to transform an input abstract state $(\mathcal{G}, R)$ to $(\mathcal{G}', R')$ where it can be syntactically decided whether two symbolic locations denote the same concrete locations or not; that is, for each shape graph (s, g) in $\mathcal{G}'$ and non-nil symbolic locations $a_1, a_2 \in \mathrm{dom}(g)$ (i.e, $g(a_1) \not\equiv \mathrm{nil}$ and $g(a_2) \not\equiv \mathrm{nil}$), a_1 and a_2 mean different concrete values if and only if they are syntactically different. Note that this property does not hold in general, because $g(a_1)$ and $g(a_2)$ are nonterminals, and these nonterminals have the nil case. Then, although a_1 and a_2 are syntactically different, both of them can mean nil. So, our analysis eliminates such a case, prior to comparing two symbolic locations.

We define such an elimination, which we call split, as follows. Given an input abstract state $(\mathcal{G}, R)$ and a variable x, it first collects all shape graphs (s, g) in $\mathcal{G}$ where $g(s(x))$ is a nonterminal. For all such graphs (s, g), split looks at the definition of nonterminal $g(s(x))$, named α, in R, and it modifies (s, g) in the following two cases:

- If $R(\alpha) = \{\mathrm{nil}\}$, replace the content of x by nil.
- If $R(\alpha)$ contains both nil and a pair, extend R with $[\beta \to R(\alpha) - \{\mathrm{nil}\}]$, and split (s, g) into two shape graphs (s, g_1) and (s, g_2) such that g_1 and g_2 are equal to g except for $s(x)$: $g_1(s(x)) \equiv \mathrm{nil}$, and $g_2(s(x)) \equiv \beta$.

Note that after this transformation, if $g(s(x))$ is not nil, it is a pair or a nonterminal whose definition contains only pairs; thus, it denotes the address of an allocated cell.

Using split, we define the backward analysis of equality and inequality by: preprocessing with split followed by filtering with simple syntactic criteria. For $x_1 \mathtt{=} x_2$, the backward analyzer first applies split twice, to get $(\mathcal{G}', R') = \mathsf{split}(\mathsf{split}((\mathcal{G}, R), x_1), x_2)$. Then, it removes shape graphs (s, g) in $\mathcal{G}'$ such that $s(x_1) \not\equiv s(x_2)$ and at least one of $g(s(x_1))$ and $g(s(x_2))$ is not nil. Note that this filtering is correct, because if $g(s(x_i))$ is not nil, then it denotes the address of an allocated cell. Thus, when precisely one of $g(s(x_1))$ and $g(s(x_2))$ is nil, the other denotes the address of an allocated cell, which cannot be nil. When neither $g(s(x_1))$ nor $g(s(x_2))$ is nil, both $g(s(x_1))$ and $g(s(x_2))$ should denote the addresses of allocated cells; now, the meaning of a shape graph ensures that these addresses must be different because $s(x_1) \not\equiv s(x_2)$. The backward analysis of inequality is defined similarly, and its correctness can be checked by a similar argument.

7 Full Analysis

The basic version of our analysis, which we have presented so far, has two immediate shortcomings. First, the basic analysis cannot deal with data structures with sharing, such as doubly linked lists and binomial heaps; as a consequence, if a program uses such data structures, the analysis usually gives up, and returns $\top$. Second, the basic version cannot handle the deallocation command `dispose`, because there are no ways to express dangling pointers.

The full analysis extends the basic version to overcome these shortcomings. First, it uses a more expressive language for a grammar, where a nonterminal is allowed to have parameters. The main feature of this new parameterized grammar is that an invariant for a data structure with sharing is expressible by a grammar, as long as the sharing is "cyclic." A parameter plays a role of "targets" of such cycles. The analyzer is modified to fully exploit this increased expressivity, so that it can handle data structures with cyclic sharing. Second, it includes an approximate constant $-$, which denotes any non-nil addresses that are not necessarily allocated. The analyzer uses this new constant $-$ to represent dangling pointers.

In this section, we explain the full analysis by focusing on these two extensions.

7.1 Abstract Domain

The two extensions in the full analysis appear most explicitly in the new abstract domains. Let self and arg be two different symbolic locations. In the full analysis, the domains for shape graphs and grammars are modified as follows:

$$
\begin{aligned}
&\mathit{Cons} \triangleq \{\mathrm{nil}, c_1, \ldots, c_n, -\}\\
&\mathit{NTermApp} \triangleq \mathit{NonTerm} \times (\mathit{SymLoc} + \bot)\\
&\mathit{NTermAppR} \triangleq \mathit{NonTerm} \times (\{\mathsf{self}, \mathsf{arg}\} + \bot)\\
&\mathit{Graph} \triangleq\\
&\quad (\mathit{Var} \rightharpoonup \mathit{SymLoc}) \times\\
&\quad (\mathit{SymLoc} \rightharpoonup \mathit{Cons} + \mathit{NTermApp} + \mathit{SymLoc}^2)\\
&\mathit{Grammar} \triangleq\\
&\quad \mathit{NonTerm} \rightharpoonup \wp_{\mathsf{nf}} \begin{pmatrix} \mathit{Cons} + \\ (\mathit{Cons}{+}\{\mathsf{self}, \mathsf{arg}\}{+}\mathit{NTermAppR})^2 \end{pmatrix}
\end{aligned}
$$

The first change in the new definitions is that all the nonterminals have parameters. All the uses of nonterminals in the old definitions are replaced by the applications of nonterminals, and the declarations of nonterminals in a grammar can use two symbolic locations self and arg, as opposed to none, which denote the implicit self parameter and the explicit parameter. Note that $\bot$ can be applied to a nonterminal; this means that we consider subcases of the nonterminal where the arg parameter is not used. For instance, if a grammar R maps β to $\{\mathrm{nil}, \langle \mathsf{arg}, \mathsf{arg} \rangle\}$, then $\beta(\bot)$ excludes $\langle \mathsf{arg}, \mathsf{arg} \rangle$, and means the empty heap object.

The second change is that the new definitions allow several constants, instead of just nil; for all that places where a singleton set $\{\mathrm{nil}\}$ appears in the old definitions, set Cons now takes up those positions in the new definitions. The newly added constant $-$ means any non-null address, and it is used to represent a dangling pointer. This constant plays a crucial role for making the analysis of `dispose` possible. The other new constants $c_1, \ldots, c_n$ are non-address concrete values, which are usually used to express the status of a heap cell. For example, in a Schorr-Waite tree marking algorithm, R and L are such constants, and these constants indicate whether the left or right field of a cell is reversed.

The third change is the addition of a boolean component in shape graphs. This additional component indicates whether all heap cells are represented by the second partial function on symbolic locations or not. For example, a shape graph $([x{\to}a], [a{\to}\mathsf{tree}], \mathtt{true})$ means that the heap might contain some cells besides the tree a, while $([x{\to}a], [a{\to}\mathsf{tree}], \mathtt{false})$ means that the heap contain only the tree a.

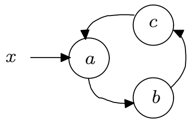
As in the base case, the precise meaning of a shape graph and a grammar is given by a translation into separation-logic assertions. Appendix B shows this translation.

7.2 Normalization Function

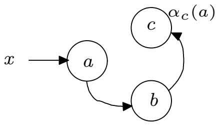
To fully exploit the increased expressivity of the abstract domain, we change the abstract function in the full analysis. This new normalization function is shown in Appendix C.3.

The most important change in the new normalization function is an extension to the **(fold)** rule and the addition of a new rule **(bfold)**. Recall that the goal of **(fold)** is to recognize heap objects in a shape graph, and to replace them by nonterminals. The limitation of the old **(fold)** rule is that the rule can recognize a heap object only when the object does not have shared cells internally. The extension of the **(fold)** rule is for overcoming this limitation. The key idea is to "cut" a "noncritical" link to a shared cell, and represent the removed link by a parameter to a nonterminal. If enough such links are cut from an heap object, the object no longer has (explicitly) shared cells, so that the wrapping step in the old definition of **(fold)** can be applied.

To understand the “cutting” of **(fold)**, let’s consider a shape graph containing a cycle from x:

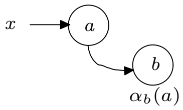


In this shape graph, “cell” a is shared, because variable x points to a and “cell” c points to a. For this shape graph, the extended **(fold)** rule concludes that the link from c to a is not critical, because even without this link, a is still reachable from variables. Thus, the rule cuts the link from c to a, introduces a nonterminal α_c with the definition $\{\langle \mathsf{arg}\rangle\}$, and changes the shape graph as follows:



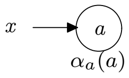
$$\alpha_c ::= \langle \mathsf{arg}\rangle$$

Note that the resulting graph does not have explicit sharing. So, we can apply the usual wrapping in the old definition of **(fold)** to c to get a shape graph and a grammar:



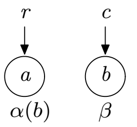
$$\begin{aligned}\alpha_c &::= \langle \mathsf{arg}\rangle\\ \alpha_b &::= \langle \alpha_c(\mathsf{arg})\rangle\end{aligned}$$

By repeating this process, we can obtain a shape graph with the following grammar:



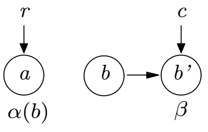
$$\begin{aligned}\alpha_c &::= \langle \mathsf{arg}\rangle\\ \alpha_b &::= \langle \alpha_c(\mathsf{arg})\rangle\\ \alpha_a &::= \langle \alpha_b(\mathsf{arg})\rangle\end{aligned}$$

The other change in the normalization step is the addition of the new rule **(bfold)**. This **(bfold)** rule changes the direction of wrapping a concrete cell in **(fold)**, and wraps a cell “from the back.” Recall that the **(fold)** rule puts a cell at the front of a heap object; it adds the cell as a root of a nonterminal. The **(bfold)** rule, on the other hand, puts a cell a at the exit of a heap object. When a is used as a parameter for a nonterminal α, the rule “combines” a and α. This rule can best be explained using an list-traversing algorithm. Consider a program that traverses a linked list where variable r points to the head cell of the list, and variable c to the current cell of the list. The usual loop invariant of such a program is expressed by the following shape graph and grammar:

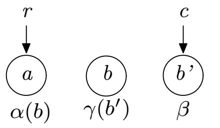


$$\begin{aligned}\alpha &::= \langle \alpha(\mathsf{arg})\rangle \ \mid\ \langle \mathsf{arg}\rangle\\ \beta &::= \mathrm{nil} \mid\ \langle \beta\rangle\end{aligned}$$

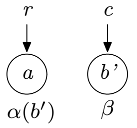
However, only with the **(fold)** rule, which adds a cell to the front, we cannot discover this invariant; one iteration of the program moves c to the next cell, and thus changes the above shape graph into the following unsimilar shape graph:



The **(bfold)** rule changes this shape graph back to the one for the invariant, by merging $\alpha(b)$ with cell b. The rule first extends a grammar with $[\gamma \to \{\langle \mathsf{arg}\rangle\}]$.



Then, it finds all the places where arg is used as itself in the definition of α, and replaces arg there by $\beta(\mathsf{arg})$. Finally, the rule changes the binding for a from $\alpha(b)$ to $\alpha(b')$, and eliminates cell b, thus resulting the following shape graph and a grammar[3]:



The precise definition of **(bfold)** does what we call *linearity* checks, in order to ensure the soundness of replacing arg by nonterminals. For the details, see the definition of **(bfold)** in Appendix C.3.

7.3 Forward and Backward Analysis

The forward analysis part of the full analyser is almost identical to that of the basic one (Figure 1). The only important difference is that the full forward analysis can handle the deallocation command:

$$\begin{aligned}[\![\mathtt{dispose}\,x]\!]\,(\mathcal{G},R) \;=\; & \text{when } \mathsf{unfold}(\mathcal{G},R,x)=\mathcal{G}',\\ & (\{(s,g[s(x){\to}-]) \mid (s,g)\in G'\}\,,R)\end{aligned}$$

The analyzer first checks whether x always points to an allocated cell; for all (s,g) in $\mathcal{G}$, it checks whether $g(s(x))$ is a tuple, which denotes a concrete cell, or a nonterminal whose only cases are tuples. If so, it materializes the cell for x in each shape graph, by unrolling the definition of nonterminals if necessary. Finally, the analyser eliminates this materialized cell, by newly binding $s(x)$ to $-$.

The backward analysis part of the full analyser is also very similar to that of the basic analyser (Figure 1). The difference is that the new backward analysis does more case analysis because of new constants. In particular, the backward analysis uses the fact that $-$ can be any addresses, but never nil, and prunes more states. For instance, when performing the backward analysis of $x = y$, the analyzer eliminates a shape graph (s,g) from $(\mathcal{G},R)$ if $g(s(x)) \equiv -$ and $g(s(y)) \equiv \text{nil}$. However, if $g(s(x)) \equiv -$ and $g(s(y))$ is a tuple, then the analyser includes (s,g) in its output, because $-$ can mean the address of cell $g(s(y))$.

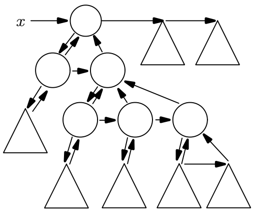
8 Example Verifications

We tested our analysis with three example programs; binomial heap construction, Schorr-Waite tree disposal, and Schorr-Waite tree traversal. For each of these programs, we ran the analyzer, and obtained abstract states for a loop invariant and the result. In all three cases, the analysis took less than 0.1 second in Intel Pentium 3.0C with 1GB memory. The obtained abstract states $(\mathcal{G},R)$ were simple and human-readable, mainly because $\mathcal{G}$ contains only a single shape graph. The obtained abstract states were strong enough to show several interesting properties of programs, including the absence of null-pointer dereference errors or memory leak, the "binomial heapness," and the complete disposal of the input tree. In this section, we will explain these obtained abstract states.

[3] The grammar is slightly different from the one for the invariant. However, if we combine two abstract states and apply unify and simplify, then the grammar for the invariant is recovered.

8.1 Binomial Heap Construction

In this experiment, we took an implementation of binomial heap construction in [CLRS01], where each cell has a pointer to the left-most child, another pointer to the next sibling, and the parent pointer. This construction algorithm produces a forest of the following shape:



In the figure, the horizontal edges are for next siblings, the down edges are for left-most children, and the up edges are for parents.

We ran the analyzer with this binomial heap construction program and the empty abstract state $(\{\}, [])$. Then, the analyzer inferred the following same abstract state $(\mathcal{G}, R)$ for the result of the construction as well as for the loop invariant.

$$\mathcal{G} = \big\{\big([x{\to}a], [a{\to}\mathsf{forest}]\big)\big\}$$
$$R = \left[\begin{array}{l}\mathsf{forest} ::= \mathrm{nil} \mid \langle \mathsf{stree}(\mathsf{self}), \mathsf{forest}, \mathrm{nil}\rangle \\ \mathsf{stree} ::= \mathrm{nil} \mid \langle \mathsf{stree}(\mathsf{self}), \mathsf{stree}(\mathsf{arg}), \mathsf{arg}\rangle\end{array}\right]$$

Here we omit $\perp$ from $\mathsf{forest}(\perp)$.

The unique shape graph in $\mathcal{G}$ means that the heap has only a single heap object whose root is stored in x, and the heap object is an instance of forest. Grammar R defines the structure of this heap object. It says that the heap object is a linked list of instances of stree, and that each instance of stree in the list is given the address of the containing list cell. These instances of stree are, indeed, precisely those trees with pointers to the left-most children and to the next sibling, and the parent pointer. To see how the grammar R implies this fact, let's consider the definition of stree by assuming that arg is a pointer to the parent cell. Then, $\langle \mathsf{stree}(\mathsf{self}), \mathsf{stree}(\mathsf{arg}), \mathsf{arg}\rangle$ means that the third field points to the parent, and that the first two fields point to instances of stree; the first instance, the one from the first field, has the current cell as its parent, and the second instance has the same parent as the current cell. Note that this is precisely the characterization of the trees in a binomial heap.

8.2 Schorr-Waite Tree Disposal

We applied our analysis to the Schorr-Waite tree-disposing algorithm. We took a slightly optimized Schorr-Waite traversing algorithm, where the algorithm uses three fields instead of four, and we inserted a "`dispose`" command appropriately. The resulting algorithm traverses a binary tree in the depth-first manner, using pointer reversals, and disposes a cell when the cell is last visited.

As an initial abstract state, we used the following abstract state $(\mathcal{G}_0, R_0)$:[4]

$$\mathcal{G}_0 = \big\{\big([x{\to}a], [a{\to}\mathsf{tree}]\big)\big\}$$
$$R_0 = [\mathsf{tree} ::= \mathrm{nil} \mid \langle \mathrm{I}, \mathsf{tree}, \mathsf{tree}\rangle]$$

This abstract state means that the initial heap contains a binary tree a whose cells are marked I, and that this tree is the only thing in the heap.

[4] Here again we omitted $\perp$ from $\mathsf{tree}(\perp)$.

Given the program and the initial value $(\mathcal{G}_0, R_0)$, the analyzer returned $(\mathcal{G}_1, R_1)$ for a final result, and $(\mathcal{G}_2, R_2)$ for an loop invariant:

$$\begin{aligned}
\mathcal{G}_1 &= \big\{ \big([x{\to}a], [a{\to}\mathrm{nil}] \big) \big\} \\
R_1 &= []
\end{aligned}$$

$$\begin{aligned}
\mathcal{G}_2 &= \big\{ \big([x{\to}a, y{\to}b], [a{\to}\mathsf{tree}, b{\to}\mathsf{rtree}] \big) \big\} \\
R_2 &= \left[\begin{array}{l} \mathsf{rtree} ::= \mathrm{nil} \mid \langle \mathrm{R}, \mathrm{nil}, \mathsf{rtree} \rangle \mid \langle \mathrm{L}, \mathsf{rtree}, \mathsf{tree} \rangle \\ \mathsf{tree} ::= \mathrm{nil} \mid \langle \mathrm{I}, \mathsf{tree}, \mathsf{tree} \rangle \end{array} \right]
\end{aligned}$$

The abstract state $(\mathcal{G}_1, R_1)$ says that the variable x contains nil, and that the heap is empty. Thus, this abstract state proves that all the heap cells in the input tree are disposed by the algorithm.[5]

The inferred invariant $(\mathcal{G}_2, R_2)$ describes two disjoint heap objects, one reachable from the current cell x and the other reachable from the original parent y of x. The invariant says that the heap object x is either empty, or a binary tree all of whose nodes are marked I. The first empty case means that cell x has already been visited before; thus, all the cells from x in the initial state must have been disposed already. The second case, on the other hand, means that cell x is first visited; so, no cells from the binary tree from x have been modified by the algorithm. For the other heap object y, the invariant implies that (1) a cell is marked R, the left subtree is completely disposed, and the right field is reversed; or (2) a cell is marked L, the left field is reversed, and the right subtree has not been changed.

8.3 Schorr-Waite Tree Traversal

The last example is the Schorr-Waite tree traversing algorithm. Although this traversing algorithm is almost identical to the previous Schorr-Waite tree disposal, we analyzed this algorithm because its verification is more demanding. The full verification of the algorithm needs to show that the final heap contains a binary tree all of whose cells are marked by R, and that this tree is identical to the original tree except for the first marking field. In this experiment, we wanted to see how much of this invariant can be discovered by our analyzer.

Given the traversing algorithm and the input abstract state in the previous section, the analyzer produced $(\mathcal{G}_1, R_1)$ for final states, and $(\mathcal{G}_2, R_2)$ for a loop invariant:

$$\begin{aligned}
\mathcal{G}_1 &= \big\{ \big([x{\to}a], [a{\to}\mathsf{treeR}] \big) \big\} \\
R_1 &= [\mathsf{treeR} ::= \mathrm{nil} \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{treeR} \rangle]
\end{aligned}$$

$$\begin{aligned}
\mathcal{G}_2 &= \big\{ \big([x{\to}a, y{\to}b], [a{\to}\mathsf{treeRI}, b{\to}\mathsf{rtree}] \big) \big\} \\
R_2 &= \left[\begin{array}{l} \mathsf{rtree} ::= \mathrm{nil} \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{rtree} \rangle \mid \langle \mathrm{L}, \mathsf{rtree}, \mathsf{tree} \rangle \\ \mathsf{tree} ::= \mathrm{nil} \mid \langle \mathrm{I}, \mathsf{tree}, \mathsf{tree} \rangle \\ \mathsf{treeR} ::= \mathrm{nil} \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{treeR} \rangle \\ \mathsf{treeRI} ::= \mathrm{nil} \mid \langle \mathrm{I}, \mathsf{tree}, \mathsf{tree} \rangle \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{treeR} \rangle \end{array} \right]
\end{aligned}$$

The abstract state $(\mathcal{G}_1, R_1)$ means that the heap contains only a single heap object x, and that this heap object is a binary tree containing only R-marked cells. Note that this abstract state implies the absence of memory leaks, because the tree x is the only thing in the heap. However, the abstract state is not strong enough to guarantee that for each cell, the original values of the second and third fields are restored. In fact, our analyzer cannot prove such a

[5] Some reader might wonder why the variable x is nil. This is because we obtained the Schorr-Waite disposal from the Schorr-Waite traversal. In the traversing algorithm, the first field of the current cell x is checked if x is not nil. So, after disposing the current cell, we need to set x to nil, in order to avoid dangling-pointer dereference errors.

property that connects the initial and final states. In section 10, we mention some ideas for inferring even such properties.

The loop invariant $(\mathcal{G}_2, R_2)$ means that the heap contains two disjoint heap objects x and y. Since the heap object x is an instance of treeRI, the object x is a I-marked binary tree, or a R-marked binary tree. This first case indicates that x is first visited, and the second case that x has been visited before. The nonterminal rtree for the other heap object y implies that one of left or right field of cell y is reversed. The second case, $\langle \mathrm{R}, \mathsf{treeR}, \mathsf{rtree}\rangle$, in the definition of rtree means that the current cell is marked R, its right field is reversed, and the left subtree is a R-marked binary tree. The third case, $\langle \mathrm{L}, \mathsf{rtree}, \mathsf{tree}\rangle$, in the definition means that the current cell is marked L, the left field is reversed, and the right subtree is a I-marked binary tree. Note that this invariant, indeed, holds because y points to the parent of x, so the left or right field of cell y must be reversed.

9 Correctness

The correctness of our analysis is expressed by the following theorem:

Theorem 3 *For all programs C and abstract states $(\mathcal{G}, R)$, if $[\![C]\!](\mathcal{G}, R)$ is a non-$\top$ abstract state $(\mathcal{G}', R')$, then triple $\{\mathsf{means}(\mathcal{G}, R)\}C\{\mathsf{means}(\mathcal{G}', R')\}$ holds in separation logic.*

Note that the theorem considers only non-$\top$ abstract states. This exclusion of $\top$ is related to the tight interpretation of a Hoare triple in separation logic. In separation logic, $\{P\}C\{Q\}$ implies that C does not dereference nil or dangling pointers if it is started at a state satisfying P. Our analysis can guarantee such memory safety only when the analysis result is not $\top$.

The proof of this theorem uses the correctness of abstract operators and the backward analysis:

Lemma 4 *All of* unfold*,* $\mathsf{normalize}$*,* $\dot\sqcup$ *,* $\dot\sqsubseteq$ *and the backward analysis are correct.*
(1) If $\mathsf{unfold}(\mathcal{G}, R, x) = (\mathcal{G}', R')$*, then*

$$\mathsf{means}(\mathcal{G}, R) \implies \mathsf{means}(\mathcal{G}', R').$$

(2) If $\mathsf{normalize}(\mathcal{G}, R) = (\mathcal{G}', R')$*, then*

$$\mathsf{means}(\mathcal{G}, R) \implies \mathsf{means}(\mathcal{G}', R').$$

(3) If $(\mathcal{G}'', R'') = (\mathcal{G}, R) \,\dot\sqcup\, (\mathcal{G}', R')$, then

$$\mathsf{means}(\mathcal{G}, R) \vee \mathsf{means}(\mathcal{G}', R') \iff \mathsf{means}(\mathcal{G}'', R'').$$

(4) If $(\mathcal{G}, R) \,\dot\sqsubseteq\, (\mathcal{G}', R')$, then

$$\mathsf{means}(\mathcal{G}, R) \implies \mathsf{means}(\mathcal{G}', R').$$

(5) If $[\![B]\!](\mathcal{G}, R) = (\mathcal{G}', R')$, then

$$(B \wedge \mathsf{means}(\mathcal{G}, R)) \implies \mathsf{means}(\mathcal{G}', R').$$

Using Lemma 4, we prove the main theorem. We use the induction on the structure of C to show that $\{\mathsf{means}(\mathcal{G}, R)\}C\{\mathsf{means}(\mathcal{G}', R')\}$ is derivable in separation logic; then, the theorem follows from the soundness of separation-logic proof rules. In the proof, we use the operator $\mathsf{closure}$ that takes an assertion and existentially quantifies all the symbolic locations in the assertions. For instance, $\mathsf{closure}(x \doteq a * y \doteq b)$ is $\exists ab.\, x \doteq a * y \doteq b$. We also use $\mathsf{means}_{\mathrm{sd}}(s, g)$ to express $\mathsf{means}_{\mathrm{s}}(s, g)$ without existential quantifications.

We first consider cases when C is an atomic command A. In this case, the grammar part of the input abstract value doesn't change, so that $R = R'$. Using this fact, we construct a part of the required proof tree:

$$\dfrac{\dfrac{\dfrac{\tau}{\mathsf{means}_{\mathrm{g}}(R)\ \vdash\ \{\bigvee_{(s,g)\in\mathcal{G}}\mathsf{means}_{\mathrm{s}}(s,g)\}A\{\bigvee_{(s',g')\in\mathcal{G}'}\mathsf{means}_{\mathrm{s}}(s',g')\}}}{\{\mathsf{let}\ \mathsf{means}_{\mathrm{g}}(R)\ \mathsf{in}\ \mathsf{means}_{\mathrm{ss}}(\mathcal{G})\}A\{\mathsf{let}\ \mathsf{means}_{\mathrm{g}}(R)\ \mathsf{in}\ \mathsf{means}_{\mathrm{ss}}(\mathcal{G}')\}}}{\{\mathsf{means}(\mathcal{G},R)\}A\{\mathsf{means}(\mathcal{G}',R)\}}$$

In the above derivation, the missing subtree τ varies depending on the specific case of A. For each atomic command A, we will explain, one-by-one, how to to construct such a subtree τ.

- $A \equiv x := \mathtt{new}$: In this case, there are auxiliary variables a, b_1, b_2 such that

  1. none of a, b_1 and b_2 occur in $\mathcal{G}$ or R; and
  2. $\mathcal{G}' = \{(s[x{\to}a], g[a{\to}\langle b_1, b_2\rangle, b_1{\to}\mathrm{nil}, b_2{\to}\mathrm{nil}] \mid (s,g) \in \mathcal{G}\}$.

  Thus, it suffices to show that for each (s,g) in $\mathcal{G}$,

  $$\mathsf{means}_{\mathrm{g}}(R)\ \vdash\ \{\mathsf{means}_{\mathrm{s}}(s,g)\}x := \mathtt{new}\{\mathsf{means}_{\mathrm{s}}(s[x{\to}a], g[g'])\}$$

  where $g' = [a{\to}\langle b_1, b_2\rangle, b_1{\to}\mathrm{nil}, b_2{\to}\mathrm{nil}]$. Let s_0 the restriction of s to $\mathrm{dom}(s) - \{x\}$. We derive the triple as follows:

  $$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{means}_{\mathrm{g}}(R)\ \vdash\ \{\mathtt{emp}\}x := \mathtt{new}\{x \mapsto \langle\mathrm{nil},\mathrm{nil}\rangle\}}\ \text{New}}{\mathsf{means}_{\mathrm{g}}(R)\ \vdash\ \{\mathtt{emp}\}x := \mathtt{new}\{\exists a\vec{b}.\, x \doteq a * a \mapsto \langle b_1, b_2\rangle * \bigodot_i b_i \doteq \mathrm{nil}\}}\ \text{Conseq}}{\mathsf{means}_{\mathrm{g}}(R)\ \vdash\ \{\mathsf{means}_{\mathrm{sd}}(s_0,g)\}x := \mathtt{new}\{\mathsf{means}_{\mathrm{sd}}(s_0,g) * \exists a\vec{b}.\, x \doteq a * (a \mapsto b_1, b_2) * \bigodot_i b_i \doteq \mathrm{nil}\}}\ \text{FR}}{\mathsf{means}_{\mathrm{g}}(R)\ \vdash\ \{\mathsf{means}_{\mathrm{sd}}(s,g)\}x := \mathtt{new}\{\exists a\vec{b}.\,\mathsf{means}_{\mathrm{sd}}(s[x{\to}a], g[g'])\}}\ \text{Conseq}}{\mathsf{means}_{\mathrm{g}}(R)\ \vdash\ \{\mathsf{closure}(\mathsf{means}_{\mathrm{sd}}(s,g))\}x := \mathtt{new}\{\mathsf{closure}(\mathsf{means}_{\mathrm{sd}}(s[x{\to}a], g[g']))\}}\ \text{AuxElim}$$

  In the derivation, we used the "freshness" of a and $\vec{b}$ to move the existential quantification outside of $*$ (in the below application of Consequence). We also used the assumption about a program variable in the shape graph; each program variable y appears at most once in $\mathsf{means}_{\mathrm{sd}}(s,g)$ for its own "definition" $y \doteq s(y)$. Thus, $\mathsf{means}_{\mathrm{sd}}(s_0,g)$ does not contain x, because $x \notin \mathrm{dom}(s_0)$, and so, $\mathsf{means}_{\mathrm{sd}}(s_0,g)$ can be added by the frame rule. Also, $\mathsf{means}_{\mathrm{sd}}(s,g)$ has to be $\mathsf{means}(s_0,g) * (x \doteq s(x))$, so it implies $\mathsf{means}(s_0,g)$. This implication is used in the below application of Consequence.

- $A \equiv x := \mathrm{nil}$: Then, there is a variable a such that

  1. a does not occur in $\mathcal{G}$ and R; and
  2. $\mathcal{G}' = \{(s[x{\to}a], g[a{\to}\mathrm{nil}]) \mid (s,g) \in \mathcal{G}\}$.

  For each $(s,h,t) \in \mathcal{G}$, we will construct a proof tree for the triple

  $$\mathsf{means}_{\mathrm{g}}(R)\ \vdash\ \{\mathsf{means}_{\mathrm{sd}}(s,g)\}x := E\{\exists a.\,\mathsf{means}_{\mathrm{sd}}(s[x{\to}a], g[a{\to}\mathrm{nil}])\}.$$

  Once having such a tree, we can build the required τ easily:

  $$\dfrac{\dfrac{\dfrac{\vdots}{\mathsf{means}_{\mathrm{g}}(R)\ \vdash\ \{\mathsf{means}_{\mathrm{sd}}(s,g)\}x := \mathrm{nil}\{\exists a.\,\mathsf{means}_{\mathrm{sd}}(s[x{\to}a], g[a{\to}\mathrm{nil}])\}}}{\mathsf{means}_{\mathrm{g}}(R)\ \vdash\ \{\mathsf{closure}(\mathsf{means}_{\mathrm{sd}}(s,g))\}x := \mathrm{nil}\{\mathsf{closure}(\mathsf{means}_{\mathrm{sd}}(s[x{\to}a], g[a{\to}\mathrm{nil}]))\}}\ \text{AuxElim}}{\mathsf{means}_{\mathrm{g}}(R)\ \vdash\ \{\bigvee_{(s,g)\in\mathcal{G}}\mathsf{means}_{\mathrm{s}}(s,g)\}x := \mathrm{nil}\{\bigvee_{(s',g')\in\mathcal{G}'}\mathsf{means}_{\mathrm{s}}(s',g')\}}\ \text{Disj}$$

Consider a shape graph (s, g) in $\mathcal{G}$. Let s_0 be the restriction of s to $\mathrm{dom}(s) - \{x\}$. We construct the required tree as follows:

$$\begin{array}{cl}
\hline
\mathsf{means}_{\mathrm{g}}(R) \ \vdash \ \{(\exists a.\, \mathsf{means}_{\mathrm{sd}}(s[x{\to}a], g[a{\to}\mathrm{nil}]))[\mathrm{nil}/x]\}x := \mathrm{nil}\{\exists a.\, \mathsf{means}_{\mathrm{sd}}(s[x{\to}a], g[a{\to}\mathrm{nil}])\} & \text{Assign} \\
\hline
\mathsf{means}_{\mathrm{g}}(R) \ \vdash \ \{\exists a.\, \mathsf{means}_{\mathrm{sd}}(s_0, g) * \mathrm{nil} \doteq a * a \doteq \mathrm{nil}\}x := \mathrm{nil}\{\exists a.\, \mathsf{means}_{\mathrm{sd}}(s[x{\to}a], g[a{\to}\mathrm{nil}])\} & \text{Conseq} \\
\hline
\mathsf{means}_{\mathrm{g}}(R) \ \vdash \ \{\mathsf{means}_{\mathrm{sd}}(s, g)\}x := \mathrm{nil}\{\exists a.\, \mathsf{means}_{\mathrm{sd}}(s[x{\to}a], g[a{\to}\mathrm{nil}])\} & \text{Conseq}
\end{array}$$

In the last two steps of the derivation, we used the following implications, which follow from the definition of $\mathsf{means}_{\mathrm{sd}}$ and the "freshness" of a:

$$\begin{array}{rcl}
\exists a.\, \mathsf{means}_{\mathrm{sd}}(s[x{\to}a], g[a{\to}\mathrm{nil}]) \ \ [\mathrm{nil}/x] & \equiv & \exists a. \ \ \mathsf{means}_{\mathrm{sd}}(s[x{\to}a], g[a{\to}\mathrm{nil}])[\mathrm{nil}/x] \\
& \equiv & \exists a. \ \ (\mathsf{means}_{\mathrm{sd}}(s_0, g) * x \doteq a * a \doteq \mathrm{nil})[\mathrm{nil}/x] \\
& \equiv & \exists a.\, \mathsf{means}_{\mathrm{sd}}(s_0, g) * \mathrm{nil} \doteq a * a \doteq \mathrm{nil} \\
\\
\mathsf{means}_{\mathrm{sd}}(s, g) & \Longrightarrow & \exists a.\, \mathsf{means}_{\mathrm{sd}}(s, g) * \mathrm{nil} \doteq a * a \doteq \mathrm{nil} \\
& \Longrightarrow & \exists a.\, \mathsf{means}_{\mathrm{sd}}(s_0, g) * \mathrm{nil} \doteq a * a \doteq \mathrm{nil}
\end{array}$$

- $A \equiv x := y$: In this case, $y \in \mathrm{dom}(s)$ for all $(s, g) \in \mathcal{G}$, and

$$\mathcal{G}' = \{(s[x{\to}s(y)], g) \mid (s, g) \in \mathcal{G}\}.$$

Using these facts, we construct the require derivation:

$$\begin{array}{cl}
\hline
\mathsf{means}_{\mathrm{g}}(R) \ \vdash \ \{\mathsf{means}_{\mathrm{sd}}(s[x{\to}s(y)], g)[y/x]\}x := y\{\mathsf{means}_{\mathrm{sd}}(s[x{\to}s(y)], g)\} & \text{Assign} \\
\hline
\mathsf{means}_{\mathrm{g}}(R) \ \vdash \ \{\mathsf{means}_{\mathrm{sd}}(s, g)\}x := y\{\mathsf{means}_{\mathrm{sd}}(s[x{\to}s(y)], g)\} & \text{Conseq} \\
\hline
\mathsf{means}_{\mathrm{g}}(R) \ \vdash \ \{\mathsf{closure}(\mathsf{means}_{\mathrm{s}}(s, g))\}x := y\{\mathsf{closure}(\mathsf{means}_{\mathrm{sd}}(s[x{\to}s(y)], g))\} & \text{AuxElim} \\
\hline
\mathsf{means}_{\mathrm{g}}(R) \ \vdash \ \{\bigvee_{(s,h,t)\in\mathcal{G}} \mathsf{means}_{\mathrm{s}}(s, h, t)\}x := y\{\bigvee_{(s',g')\in\mathcal{G}'} \mathsf{means}_{\mathrm{s}}(s', g')\} & \text{Disj}
\end{array}$$

In the application of Consequence, we used the following implication, which follows from the definition of $\mathsf{means}_{\mathrm{sd}}$:

$$\begin{array}{rcl}
\mathsf{means}_{\mathrm{sd}}(s, g) & \Longrightarrow & \mathsf{means}_{\mathrm{sd}}(s, g) * y \doteq s(y) \\
& \Longrightarrow & \mathsf{means}_{\mathrm{sd}}(s|_{\mathrm{dom}(s)-\{x\}}, g) * y \doteq s(y) \\
& \equiv & (\mathsf{means}_{\mathrm{sd}}(s|_{\mathrm{dom}(s)-\{x\}}, g) * x \doteq s(y))[y/x] \\
& \equiv & \mathsf{means}_{\mathrm{sd}}(s[x{\to}s(y)], g)[y/x].
\end{array}$$

- $A \equiv x\text{->}i := y$: WLOG, we assume that $i = 0$. In this case, there exists a graph set $\mathcal{G}_0$ such that

  1. $\mathcal{G}_0 = \mathsf{unfold}(\mathcal{G}, R, x)$;
  2. for all $(s, g) \in \mathcal{G}_0$, we have that $(x, y \in \mathrm{dom}(s)) \ \wedge \ (s(x) \in \mathrm{dom}(g))$;
  3. $\mathcal{G}' = \{(s, g[a{\to}\langle s(y), a_1\rangle] \mid \exists a_0.\, s(x) = a \wedge g(a) = \langle a_0, a_1\rangle \wedge (s, g) \in \mathcal{G}_0\}$.

Since $\mathcal{G}_0 = \mathsf{unfold}(\mathcal{G}, R, x)$, by Lemma 4, the following implication holds:

$$\mathsf{means}_{\mathrm{g}}(R) \ \vdash \ \Big(\bigvee_{(s,g)\in\mathcal{G}} \mathsf{means}_{\mathrm{s}}(s, g)\Big) \Longrightarrow \Big(\bigvee_{(s,g)\in\mathcal{G}_0} \mathsf{means}_{\mathrm{s}}(s, g)\Big).$$

Thus, it suffices to derive the following triple for every (s, g) in $\mathcal{G}_0$:

$$\mathsf{means}_{\mathrm{g}}(R) \ \vdash \ \{\mathsf{means}_{\mathrm{s}}(s, g)\}x\text{->}i := E\{\mathsf{means}_{\mathrm{s}}(s, g[a{\to}\langle s(y), a_1\rangle])\}$$

where $a = s(x)$ and $\langle a_0, a_1\rangle = g(a)$. Let $b = s(y)$. Consider a restriction (s_0, g_0) of (s, g):

$$s_0(z) = \begin{cases} \text{undefined} & \text{if } z \equiv x \text{ or } z \equiv y \\ s(z) & \text{otherwise} \end{cases} \qquad g_0(c) = \begin{cases} \text{undefined} & \text{if } c \equiv a \\ g(c) & \text{otherwise} \end{cases}$$

The required triple is proved as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \{x \mapsto -, a_1\}x\texttt{->}0 \mathrel{:=} y\{x \mapsto y, a_1\}}}{\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \{\mathsf{means}_{\mathrm{sd}}(s_0, g_0) * x\doteq a * y\doteq b * x\mapsto -, a_1\}x\texttt{->}0 \mathrel{:=} y\{\mathsf{means}_{\mathrm{sd}}(s_0, g_0) * x\doteq a * y\doteq b * x\mapsto y, a_1\}}\;\text{FR}}{\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \{\mathsf{means}_{\mathrm{sd}}(s_0, g_0) * x\doteq a * y\doteq b * a\mapsto -, a_1\}x\texttt{->}0 \mathrel{:=} y\{\mathsf{means}_{\mathrm{sd}}(s_0, g_0) * x\doteq a * y\doteq b * a\mapsto b, a_1\}}\;\text{Conseq}}{\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \{\mathsf{means}_{\mathrm{sd}}(s, g)\}x\texttt{->}0 \mathrel{:=} y\{\mathsf{means}_{\mathrm{sd}}(s, g[a\to\langle b, a_1\rangle])\}}\;\text{Conseq}}{\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \{\mathsf{closure}(\mathsf{means}_{\mathrm{sd}}(s, g))\}x\texttt{->}0 \mathrel{:=} y\{\mathsf{closure}(\mathsf{means}_{\mathrm{sd}}(s, g[a\to\langle b, a_1\rangle]))\}}\;\text{AuxElim}$$

In the derivation, we used the following facts:

$$\mathsf{means}_{\mathrm{sd}}(s, g[a\to\langle b, a_1\rangle]) \;\equiv\; \mathsf{means}_{\mathrm{sd}}(s_0, g_0) * x \doteq a * y \doteq b * a \mapsto b, a_1$$

$$\begin{aligned} \mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \mathsf{means}_{\mathrm{sd}}(s, g) \quad &\Longrightarrow \quad \mathsf{means}_{\mathrm{sd}}(s_0, g_0) * x \doteq a * y \doteq b * a \mapsto a_0, a_1 \\ &\Longrightarrow \quad \mathsf{means}_{\mathrm{sd}}(s_0, g_0) * x \doteq a * y \doteq b * a \mapsto -, a_1 \end{aligned}$$

- $A \equiv x \mathrel{:=} y\texttt{->}i$: WLOG, we assume that $i = 0$. In this case, there is a graph set $\mathcal{G}_0$ such that

  1. $\mathcal{G}_0 = \mathsf{unfold}(\mathcal{G}, R, y)$;
  2. for all $(s, g) \in \mathcal{G}_0$, we have that $y \in \mathrm{dom}(s)$ and $s(y) \in \mathrm{dom}(g)$; and
  3. $\mathcal{G}' = \{(s[x\to a_0], g) \mid \exists a_1.\, (s, g) \in \mathcal{G}_0.\ g(s(y)) = \langle a_0, a_1\rangle\}$.

  Since $\mathcal{G}_0 = \mathsf{unfold}(\mathcal{G}, R, y)$, by Lemma 4, the following implication holds:

  $$\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \Big(\bigvee_{(s,g)\in\mathcal{G}} \mathsf{means}_{\mathrm{s}}(s, g)\Big) \Longrightarrow \Big(\bigvee_{(s,g)\in\mathcal{G}_0} \mathsf{means}_{\mathrm{s}}(s, g)\Big).$$

  Thus, it suffices to derive the following triple for every (s, g) in $\mathcal{G}_0$:

  $$\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \{\mathsf{means}_{\mathrm{s}}(s, g)\}x \mathrel{:=} y\texttt{->}0\{\mathsf{means}_{\mathrm{s}}(s[x\to a_0], g)\}$$

  where $\langle a_0, a_1\rangle = g(s(y))$. Let $b = s(y)$. Consider a restriction (s_0, g_0) of (s, g):

  $$s_0(z) = \begin{cases} \text{undefined} & \text{if } z \equiv x \text{ or } z \equiv y \\ s(z) & \text{otherwise} \end{cases} \qquad g_0(c) = \begin{cases} \text{undefined} & \text{if } c \equiv b \\ g(c) & \text{otherwise} \end{cases}$$

  The required proof tree is given below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \{y\mapsto a_0, a_1\}x \mathrel{:=} y\texttt{->}0\{x\doteq a_0 * y\mapsto a_0, a_1\}}}{\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \{\mathsf{means}_{\mathrm{sd}}(s_0, g_0) * y\doteq b * y\mapsto a_0, a_1\}x \mathrel{:=} y\texttt{->}0\{\mathsf{means}_{\mathrm{sd}}(s_0, g_0) * y\doteq b * x\doteq a_0 * y\mapsto a_0, a_1\}}\;\text{FR}}{\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \{\mathsf{means}_{\mathrm{sd}}(s_0, g_0) * y\doteq b * b\mapsto a_0, a_1\}x \mathrel{:=} y\texttt{->}0\{\mathsf{means}_{\mathrm{sd}}(s_0, g_0) * y\doteq b * x\doteq a_0 * b\mapsto a_0, a_1\}}\;\text{Conseq}}{\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \{\mathsf{means}_{\mathrm{sd}}(s, g)\}x \mathrel{:=} y\texttt{->}0\{\mathsf{means}_{\mathrm{sd}}(s[x\to a_0], g)\}}\;\text{Conseq}}{\mathsf{means}_{\mathrm{g}}(R) \;\vdash\; \{\mathsf{closure}(\mathsf{means}_{\mathrm{sd}}(s, g))\}x \mathrel{:=} y\texttt{->}0\{\mathsf{closure}(\mathsf{means}_{\mathrm{sd}}(s[x\to a], g))\}}\;\text{AuxElim}$$

  In the tree, we use the following facts:

  $$\mathsf{means}_{\mathrm{sd}}(s[x\to a_0], g) \equiv \mathsf{means}_{\mathrm{sd}}(s_0, g_0, t) * y \doteq b * x \doteq a_0 * b \mapsto a_0, a_1.$$

  $$\begin{aligned} \mathsf{means}_{\mathrm{g}}(R) \vdash \mathsf{means}_{\mathrm{sd}}(s, g) \;&\Longrightarrow\; \mathsf{means}_{\mathrm{sd}}(s_0, g_0) * x\doteq s(x) * y\doteq b * b\mapsto a_0, a_1 \quad (\because \text{Def. of } \mathsf{means}_{\mathrm{sd}}) \\ &\Longrightarrow\; \mathsf{means}_{\mathrm{sd}}(s_0, g_0) * y\doteq b * b\mapsto a_0, a_1. \end{aligned}$$

Next, we prove the cases for the composite commands. Here we focus on the case for the loop. The other cases are straightforward. Suppose that $[\![\texttt{while}\,B\,C]\!](\mathcal{G}, R) = (\mathcal{G}', R')$. Then, by the definition of the forward analysis, there exist abstract states $(\mathcal{G}_0, R_0)$ and $(\mathcal{G}_1, R_1)$ such that

1. $(\mathcal{G}_1, R_1) = [\![C]\!]([\![B]\!](\mathcal{G}_0, R_0))$;
2. $[\![!B]\!](\mathcal{G}_0, R_0) = (\mathcal{G}', R')$; and
3. $\mathsf{normalize}\big((\mathcal{G}_0, R_0) \mathrel{\dot\sqcup} (\mathcal{G}, R) \mathrel{\dot\sqcup} (\mathcal{G}_1, R_1)\big) \mathrel{\dot\sqsubseteq} (\mathcal{G}_0, R_0)$.

Then, $\{\mathsf{means}[\![B]\!](\mathcal{G}_0, R_0)\}C\{\mathsf{means}(\mathcal{G}_1, R_1)\}$ is derivable by induction hypothesis, and the following implications hold by Lemma 4:

$$\begin{array}{ll} \mathsf{means}(\mathcal{G}_0, R_0) \wedge B \implies \mathsf{means}[\![B]\!](\mathcal{G}_0, R_0) & \mathsf{means}(\mathcal{G}_0, R_0) \wedge \neg B \implies \mathsf{means}(\mathcal{G}', R') \\ \mathsf{means}(\mathcal{G}, R) \implies \mathsf{means}(\mathcal{G}_0, R_0) & \mathsf{means}(\mathcal{G}_1, R_1) \implies \mathsf{means}(\mathcal{G}_0, R_0) \end{array}$$

Using two such facts, we derive the required triple:

$$\dfrac{\dfrac{\dfrac{\{\mathsf{means}([\![B]\!](\mathcal{G}_0, R_0))\}C\{\mathsf{means}(\mathcal{G}_1, R_1)\}}{\{\mathsf{means}(\mathcal{G}_0, R_0) \wedge B\}C\{\mathsf{means}(\mathcal{G}_0, R_0)\}}\,\text{Conseq}}{\{\mathsf{means}(\mathcal{G}_0, R_0)\}\texttt{while}\,B\,C\{\mathsf{means}(\mathcal{G}_0, R_0) \wedge \neg B\}}\,\text{Loop}}{\{\mathsf{means}(\mathcal{G}, R)\}\texttt{while}\,B\,C\{\mathsf{means}(\mathcal{G}', R')\}}\,\text{Conseq}$$

10 Conclusion

We have designed a fully automatic analysis that infers heap invariants of pointer programs, and applied the resulting analysis to three nontrivial pointer programs. In all three cases, the analysis run very fast, taking at most 0.1 second, and produced invariants strong enough to show interesting properties of the program. The obtained invariant for the binomial heap construction algorithm proves that the construction algorithm indeed builds a binomial heap; the invariant for the Schorr-Waite tree disposing algorithm verifies that the algorithm removes all the cells in the tree; the invariant for the Schorr-Waite tree traversing algorithm shows that the result of the algorithm is a marked-binary tree and that there are no memory leaks. The current implementation and the three examples are available on the following URL:

`http://ropas.snu.ac.kr/grammar`

One limitation of our analysis is that the analysis cannot handle dags and general graphs. Frankly, we are in a very early stage for attacking this problem. We are currently considering to use a more general language for a grammar, where the definition of a nonterminal can talk not just about the root cell, but also about shared cells.

Another limitation is that the analysis cannot prove a relational property regarding both the initial and final states. For instance, the full specification for the Schorr-Waite tree traversing algorithm includes a requirement that the initial and final trees are identical except for the marking field. Our analysis cannot prove such a property. The current idea is to add "auxiliary" fields to every cell, and to cache the initial field values of each cell to these auxiliary fields. Then, the analyser can show that the heap is restored to the initial heap by inferring that at the end of the execution, the actual and auxiliary fields of each cell have the same contents.

References

[BRS99] M. Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for describing linked data structures. In *ESOP '99: European Symposium on Programming*, pages 2–19. Lecture Notes in Computer Science, Vol. 1576, S.D. Swierstra (ed.), Springer-Verlag, New York, NY, 1999.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill Book Company, 2001.

[JJKS97] J. Jenson, M. Jorgensen, N. Klarkund, and M. Schwartzback. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 225–236, 1997. SIGPLAN Notices 32(5).

[MS01] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *the Proceedings of Conference on Programming Language Design and Implementation.* ACM, June 2001.

[OYR04] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *ACM Symp. on Princ. of Program. Lang.*, pages 268–280. ACM Press, January 2004.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, July 2002.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, January 1998.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

A The Semantics Of Assertions In Separation Logic

For heaps h_1, h_2 such that $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$, we write $h_1 \cdot h_2$ for heap $h_1 \cup h_2$. The formal definition of $\models$ is given as follows:

$$
\begin{array}{ll}
(s,h),\eta \models (E \mapsto E_1, E_2) & \text{iff} \\
\quad \mathrm{dom}(h) = \{[\![E]\!]s\} \text{ and } h([\![E]\!]s) = ([\![E_1]\!]s, [\![E_2]\!]) & \\
(s,h),\eta \models E = E' & \text{iff } [\![E]\!]s = [\![E']\!]s \\
(s,h),\eta \models P * Q & \text{iff} \\
\quad \exists h_0, h_1 \text{ s.t. } h_0 \cdot h_1 = h & \\
\qquad\qquad \text{and } (s,h_0),\eta \models P \text{ and } (s,h_1),\eta \models Q & \\
(s,h),\eta \models \mathtt{true} & \text{always} \\
(s,h),\eta \models P \wedge Q & \text{iff } (s,h),\eta \models P \text{ and } (s,h),\eta \models Q \\
(s,h),\eta \models P \vee Q & \text{iff } (s,h),\eta \models P \text{ or } (s,h),\eta \models Q \\
(s,h),\eta \models \neg P & \text{iff } (s,h),\eta \not\models P \\
(s,h),\eta \models \forall x.P & \text{iff } \forall v \in \mathit{Loc}.(s[x{\to}v],h),\eta \not\models P \\
(s,h),\eta \models \alpha(E_1,\ldots,E_n) & \text{iff } (([\![E_1]\!]s,\ldots,[\![E_n]\!]s),h) \in \eta(\alpha) \\
(s,h),\eta \models \mathsf{let}\ \Gamma\ \mathsf{in}\ Q & \text{iff } (s,h),\eta\Big[\beta{\to}k(\beta)\Big]_{\beta\in\mathrm{dom}(\Gamma)} \models P \\
\quad \text{where } k = \text{fix}\ \ \lambda k_0.\lambda\alpha \in \mathrm{dom}(\Gamma). & \\
\qquad\qquad\qquad \text{let } (\alpha(\vec{x}) = Q) \equiv \Gamma(\alpha) & \\
\qquad\qquad\qquad \text{in } \{(\vec{v},h) \mid (s[x{\to}\vec{v}],h),\eta\Big[\beta{\to}k_0(\beta)\Big] \models P\} &
\end{array}
$$

B The Meaning Of An Abstract State Of The Full Analysis

The meaning of an abstract state in the full analysis is given by compiling it into a separation-logic assertion. To simplify the presentation, let's assume that *Cons* contains only a single non-address constant c, and that the abstract domain is for binary cells. We can define a translation means for such a setting, by modifying only $\mathsf{means}_{\mathrm{v}}$ and $\mathsf{means}_{\mathrm{g}}$.

$$\begin{array}{rcl}
\mathsf{means}_{\mathrm{v}}(a, \mathrm{nil}) & \stackrel{\Delta}{=} & a \doteq \mathrm{nil} \\
\mathsf{means}_{\mathrm{v}}(a, -) & \stackrel{\Delta}{=} & \mathtt{emp} \land a \neq \mathrm{nil} \land a \neq c \\
\mathsf{means}_{\mathrm{v}}(a, c) & \stackrel{\Delta}{=} & x \doteq c \\
\mathsf{means}_{\mathrm{v}}(a, b) & \stackrel{\Delta}{=} & a \doteq b \\
\mathsf{means}_{\mathrm{v}}(a, \alpha(b)) & \stackrel{\Delta}{=} & \alpha(a, b) \\
\mathsf{means}_{\mathrm{v}}(a, \alpha(\bot)) & \stackrel{\Delta}{=} & \forall b. \alpha(a, b)
\end{array}$$

In the last clause, b is a different variable from a. The meaning of a grammar is a context defining a set of recursive predicates.

$$\mathsf{means}_{\mathrm{g}}(R) \stackrel{\Delta}{=} \{\alpha(a,b) {=} \bigvee_{e \in R(\alpha)} \mathsf{means}_{\mathrm{gc}}(a, b, e)\}_{\alpha \in \mathrm{dom}(R)}$$

where $\mathsf{means}_{\mathrm{gc}}$ is defined as follows:

$$\begin{array}{rcl}
\mathsf{means}_{\mathrm{gc}}(a, b, c) & \stackrel{\Delta}{=} & \mathsf{means}_{\mathrm{v}}(a, c) \\
\mathsf{means}_{\mathrm{gc}}(a, b, \langle v_1, v_2 \rangle) & \stackrel{\Delta}{=} & \exists a_1 a_2. a \mapsto a_1, a_2 \\
 & * & \mathsf{means}_{\mathrm{v}}(a_1, v_1 \, \{a/\mathsf{self}, b/\mathsf{arg}\}) \\
 & * & \mathsf{means}_{\mathrm{v}}(a_2, v_2 \, \{a/\mathsf{self}, b/\mathsf{arg}\})
\end{array}$$

In the second clause, a_1 and a_2 are variables that do not appear in v_1, v_2, a, and b.

C Full Analysis

The main definition of our full analysis is the same as Figure 1 but we use different subroutines.

C.1 An Algorithm To Compute The Order

We extend the algorithm to compute the order for parameterized nonterminals and constants:

$$\begin{array}{rll}
\llbracket c \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket c \rrbracket_{R_2} & & \\
\llbracket c \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket \alpha(w) \rrbracket_{R_2} & \text{iff} & \{c\} \subseteq R_2(\alpha) \\
\llbracket \alpha(w) \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket c \rrbracket_{R_2} & \text{iff} & R_1(\alpha) \subseteq \{c\} \\
\llbracket \mathsf{arg} \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket \mathsf{arg} \rrbracket_{R_1} & & \\
\llbracket \mathsf{self} \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket \mathsf{self} \rrbracket_{R_1} & &
\end{array}$$

$$\begin{array}{l}
\llbracket \alpha(w_1) \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket \beta(w_2) \rrbracket_{R_2} \quad \text{iff} \\
\quad w_1 \equiv \bot \text{ or } w_1 \equiv w_2, \text{ and } c \in R_1(\alpha) \Longrightarrow c \in R_2(\beta), \\
\quad \text{and } \langle v_1, v_2 \rangle \in R_1(\alpha) \Longrightarrow \exists \, \langle v_1', v_2' \rangle \in R_2(\beta) \text{ s.t.} \\
\qquad\qquad \llbracket v_1 \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket v_1' \rrbracket_{R_2} \text{ and } \llbracket v_2 \rrbracket_{R_1} \mathrel{\dot\sqsubseteq} \llbracket v_2' \rrbracket_{R_2} \\
\quad \text{co-inductively}
\end{array}$$

C.2 Similarity

We first define the value similarity, and then define the case similarity, the definition similarity, and the graph similarity:

$$
\begin{array}{lll}
\multicolumn{3}{l}{c \sim^V c \quad c \sim^V \alpha(w) \quad \alpha(w) \sim^V c} \\
\multicolumn{3}{l}{\mathsf{self} \sim^V \mathsf{self} \quad \mathsf{arg} \sim^V \mathsf{arg}} \\
\alpha(w_1) \sim^V \beta(w_2) & \text{iff} & w_1 \equiv w_2,\ w_1 \equiv \bot,\ \text{or } w_2 \equiv \bot
\end{array}
$$

$$
\begin{array}{lll}
c \sim^C c & & \\
\langle v_1, v_2 \rangle \sim^C \langle v_1', v_2' \rangle & \text{iff} & v_1 \sim^V v_1' \text{ and } v_2 \sim^V v_2' \\
E_1 \sim^D E_2 & \text{iff} & e \in E_1 \Longrightarrow \exists e' \in E_2 \text{ s.t. } e \sim^C e' \\
 & \text{and} & e \in E_2 \Longrightarrow \exists e' \in E_1 \text{ s.t. } e \sim^C e'
\end{array}
$$

$(s_1, g_1) \sim^G_S (s_2, g_2)$ iff
 $\mathrm{dom}(s_1) = \mathrm{dom}(s_2)$ and $S(\mathrm{dom}(g_1)) = \mathrm{dom}(g_2)$
 and for all $x \in \mathrm{dom}(s_1)$, $S(s_1(x)) \equiv s_2(x)$
 and for all $a \in \mathrm{dom}(g_1)$,

- if $g_1(a)$ is not a pair, $g_2(S(a))$ is not a pair and $S(g_1(a)) \sim^V g_2(S(a))$, and
- if $g_1(a) = \langle b, c \rangle$, $g_2(S(a)) = \langle S(b), S(c) \rangle$.

C.3 Normalization Function

The $\mathsf{deljunk}$ procedure has one more rule to remove "imaginary" cycles:

$$
\begin{array}{ll}
\textbf{(cycle)} & (\mathcal{G} \uplus \{(s, g \cdot [a \to \alpha(a)])\}, R) \\
 & \rightsquigarrow \left(\begin{array}{l} \mathcal{G} \cup \{(s, g \cdot [a \to \beta(\bot)])\}, \\ R \cdot [\beta \to R(\alpha)\,\{\mathsf{self}/\mathsf{arg}\}] \end{array} \right)
\end{array}
$$

The fold procedure has the following rules:

$$
\begin{array}{ll}
\textbf{(fold)} & (\mathcal{G} \uplus \{(s, g \cdot [a \to \langle a_1, a_2 \rangle, a_1 \to v_1, a_2 \to v_2])\}, R) \\
 & \rightsquigarrow \left(\begin{array}{l} \mathcal{G} \cup \{(s, g \cdot [a \to \alpha(w)])\}, \\ R \cdot [\alpha \to \{\langle v_1, v_2 \rangle \{\mathsf{self}/a, \mathsf{arg}/w\}\}] \end{array} \right) \\
 & \text{where neither } a_1 \text{ nor } a_2 \text{ appears in the range} \\
 & \text{of } g \text{ and } s,\ \mathsf{free}(\langle v_1, v_2 \rangle \subseteq \{a, b\}, \text{ and } \alpha \text{ is fresh.}
\end{array}
$$

$$
\begin{array}{ll}
\textbf{(cut)} & (\mathcal{G} \uplus \{(s, g \cdot [a \to \langle a_1, a_2 \rangle])\}, R) \\
 & \rightsquigarrow \left(\begin{array}{l} \mathcal{G} \cup \{(s, g \cdot [a \to \alpha(w)])\}, \\ R \cdot [\alpha \to \{\langle a_1, a_2 \rangle \{\mathsf{self}/a, \mathsf{arg}/w\}\}] \end{array} \right) \\
 & \text{where there are paths from variables to } a_1 \text{ and} \\
 & a_2 \text{ in } g,\ \mathsf{free}(\langle v_1, v_2 \rangle) \subseteq \{a, b\}, \text{ and } \alpha \text{ is fresh.}
\end{array}
$$

We have two more rules for the mixed cases that left (or right) field is foldable and right (or left) field can be cut. The mixed rules can be defined straightforwardly by mixing the (fold) and (cut) rules. Additional rule is for the backward folding:

$$
\begin{array}{ll}
\textbf{(bfold)} & (\mathcal{G} \cup \{(s, g \cdot [a \to \alpha(b), b \to \beta(w)])\}, R) \\
 & \rightsquigarrow (\mathcal{G} \cup \{(s, g \cdot [a \to \alpha'(w)])\}, R \cdot [\alpha' \to E])
\end{array}
$$

where b does not appear in g, α is linear (that is, arg appears exactly once in each case of $R(\alpha)$), and $E = \{c \in R(\alpha)\} \cup \{\langle f(v_1), f(v_2) \rangle \mid \langle v_1, v_2 \rangle \in R(\alpha)\}$ where

$$
f(v) = \begin{cases} \beta(\mathsf{arg}), & \text{if } v \equiv \mathsf{arg} \\ \alpha'(\mathsf{arg}), & \text{if } v \equiv \alpha(\mathsf{arg}) \\ v & \text{otherwise} \end{cases}
$$

Here we present only for the case that the parameter of α is not passed to another different nonterminals. In that case, we have to check those nonterminals are linear, introduce new nonterminals for those, and similarly define their production rules to the case of α.

The unify procedure is straightforwardly extended for handling parameterized nonterminals and additional constants as follows:

(unify)

$$\begin{array}{l}
(\mathcal{G} \uplus \{(s_1, g_1 \cdot [a_1{\to}\alpha_1(w_1)]), (s_2, g_2 \cdot [a_2{\to}\alpha_2(w_2)])\}) \\
\leadsto \begin{pmatrix} \mathcal{G} \cup \{(Ss_1, Sg_1 \cdot [a_2{\to}\beta(w_1)]), (s_2, g_2 \cdot [a_2{\to}\beta(w_2)])\}, \\ R \cdot [\beta{\to}R(\alpha_1) \cup R(\alpha_2)] \end{pmatrix} \\
\text{where } (s_1, g_1 \cdot [a_1{\to}\alpha_1]) \sim^G_S (s_2, g_2 \cdot [a_2{\to}\alpha_2]), \\
S(a_1) \equiv a_2,\ \alpha_1 \not\equiv \alpha_2, \text{ and } \beta \text{ is fresh.}
\end{array}$$

(unil)

$$\begin{array}{l}
(\mathcal{G} \uplus \{(s_1, g_1 \cdot [a_1{\to}\alpha(w)]), (s_2, g_2 \cdot [a_2{\to}\text{nil}])\}, R) \\
\leadsto \begin{pmatrix} \mathcal{G} \cup \left\{ \begin{array}{l} (S(s_1), S(g_1) \cdot [a_2{\to}\beta(w)]), \\ (s_2, g_2 \cdot [a_2{\to}\beta(w)]) \end{array} \right\}, \\ R \cdot [\beta{\to}R(\alpha) \cup \{\text{nil}\}] \end{pmatrix} \\
\text{where } (s_1, g_1 \cdot [a_1{\to}\alpha(w)]) \sim^G_S (s_2, g_2 \cdot [a_2{\to}\text{nil}]), \\
\text{and } \beta \text{ is fresh.}
\end{array}$$

(uarg)

$$\begin{array}{l}
(\mathcal{G} \uplus \{(s_1, g_1 \cdot [a_1{\to}\alpha(w)]), (s_2, g_2 \cdot [a_2{\to}\alpha(\bot)])\}, R) \\
\leadsto \left(\mathcal{G} \cup \left\{ \begin{array}{l} (S(s_1), S(g_1) \cdot [a_2{\to}\alpha(w)]), \\ (s_2, g_2 \cdot [a_2{\to}\alpha(w)]) \end{array} \right\}, R \right) \\
\text{where } (s_1, g_1 \cdot [a_1{\to}\alpha(w)]) \sim^G_S (s_2, g_2 \cdot [a_2{\to}\alpha(\bot)]), \\
\text{and } w \not\equiv \bot.
\end{array}$$

The $\mathsf{simplify}$ procedure is also straightforwardly extended as:

$$\begin{array}{ll}
\textbf{(case)} & (\mathcal{G}, R) \leadsto (\mathcal{G}, R)\,\{\beta/\alpha\} \\
& \text{where } \{\langle \alpha(w_1), v \rangle, \langle \beta(w_2), v' \rangle\} \subseteq R(\gamma), \\
& \alpha(w_1) \sim^V \alpha(w_2), \text{ and } \alpha \neq \beta, \\
& \text{(the same rule for the second field)} \\
\textbf{(nil)} & (\mathcal{G}, R \cdot [\alpha{\to}E \cdot \{\langle \beta(w), v \rangle, \langle c, v' \rangle\}]) \\
& \leadsto (\mathcal{G}, R'[\beta{\to}R'(\beta) \cup \{c\}]) \\
& \text{where } R' = R \cdot [\alpha{\to}E \cup \{\langle \beta(w), v \rangle, \langle \beta(\bot), v' \rangle\}] \\
& \text{(the same rule for the second field)} \\
\textbf{(arg)} & (\mathcal{G}, R \cdot [\alpha{\to}E \cdot \{\langle \beta(w), v \rangle, \langle \beta(\bot), v' \rangle\}]) \\
& \leadsto (\mathcal{G}, R \cdot [\alpha{\to}E \cup \{\langle \beta(w), v \rangle, \langle \beta(w), v' \rangle\}]) \\
& \text{where } w \not\equiv \bot \\
& \text{(the same rule for the second field)} \\
\textbf{(def)} & (\mathcal{G}, R) \leadsto (\mathcal{G}, R)\,\{\beta/\alpha\} \text{ where } R(\alpha) \sim^D R(\beta)
\end{array}$$

C.4 Unfolding

When we unfold the definition of a parameterized nonterminal, we replace arg by the parameter and self by the current symbolic location:

$\mathsf{unfold}((s, g), R, x) =$

- if $g(s(x))$ is a pair: $\{(s, g)\}$
- if $s(x) = a$, $g(a) = \alpha(w)$, and $R(\alpha)$ has only pairs: $\{(s, g[a{\to}\langle a_1, a_2 \rangle, a_1{\to}v_1, a_2{\to}v_2]) \mid \langle v'_1, v'_2 \rangle \in R(\alpha)\,\{a/\mathsf{self}, w/\mathsf{arg}\},\ \bot \text{ doesn't appear in } v'_1 \text{ or } v'_2\}$

- otherwise, undefined.

C.5 Case Splitting and Equality

Case splitting is extended to handle multiple constants:

$\mathsf{split}((s, g), R, x) =$

- if $s(x) = a$, $g(a) = \alpha$, and $R(\alpha) \subseteq \mathit{Const}$: $(\{(s, g[a {\to} c]) \mid c \in R(\alpha)\}, R)$.
- if $s(x) = a$, $g(a) = \alpha$, and $R(\alpha)$ contains both pairs and constants: $(\{(s, g[a {\to} c]) \mid c \in R(\alpha)\} \cup \{(s, g[a {\to} \beta])\}, R \cdot [\beta {\to} R(\alpha) - \mathit{Const}])$.
- otherwise: $(\{(s, g)\}, R)$.

The equality and non-equality checks are:

$$\begin{array}{l}
\mathsf{equal}(g, a, b) = \\
\quad (g(a) \in \mathit{Cons} \wedge g(b) \in \mathit{Cons} \Longrightarrow g(a) \equiv g(b)) \wedge \\
\quad (g(a) \notin \mathit{Cons} \vee g(b) \notin \mathit{Cons} \Longrightarrow \\
\qquad\qquad g(a) \equiv - \vee g(b) \equiv - \vee a \equiv b) \\
\mathsf{notequal}(g, a, b) = \\
\quad (g(a) \in \mathit{Cons} \wedge g(b) \in \mathit{Cons} \Longrightarrow g(a) \not\equiv g(b)) \wedge \\
\quad (g(a) \notin \mathit{Cons} \vee g(b) \notin \mathit{Cons} \Longrightarrow \\
\qquad\qquad g(a) \equiv - \vee g(b) \equiv - \vee a \not\equiv b)
\end{array}$$

Note that $\mathsf{notequal}(g, a, b) \neq \neg\mathsf{equal}(g, a, b)$ because we have an approximated constant $-$.