Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis[⋆]

Oukseh Lee[1], Hongseok Yang[2], and Kwangkeun Yi[3]

[1] Dept. of Computer Science & Engineering, Hanyang University, Korea
[2] ERC-ACI, Seoul National University, Korea
[3] School of Computer Science & Engineering, Seoul National University, Korea

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, or to check whether the program preserves the data consistency. 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 binomial heap construction 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 tree traversal, our analysis shows that at the end of the execution, the result is a tree and there are no memory leaks.

1 Introduction

We show that a static program analysis can automatically verify pointer programs, such as binomial heap construction 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 tree traversal, it verifies that the output tree is a binary tree, and there are no memory leaks. In both cases, the analysis took less than 0.2 second in Intel Pentium 3.0C with 1GB memory, and its result is simple and human-readable.

Note that although these programs handle regular heap structures such as binomial heaps and trees, the topology 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.

The static analysis is an extension to Sagiv et al.'s shape analysis [13] by grammars. To improve 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 an ample space for precisely capturing the imperative effects on heap structures. The grammar is unfolded to expose an exact heap structure on demand. The grammar is also folded to replace an exact heap structure by an abstract nonterminal. To ensure the termination of the analysis, the grammar merges multiple

⋆ Lee and Yi were supported by the Brain Korea 21 project in 2004, and Yang was supported by R08-2003-000-10370-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

production rules into a single one, and unify multiple nonterminals; this simplification makes the grammar size remain within a practical bound.

The analysis's correctness is proved via separation logic [12, 11]. The analysis is a composition of abstract operations over the grammar-based shape graphs. The semantics (concretization) of the shape graphs is defined as assertions in separation logic. Each abstract operator is proved safe by showing that the separation-logic assertion for the input graph implies that for the 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 proof rules.

The main limitation of our analysis is that the analysis cannot handle DAGs and general graphs. To overcome this limitation, we need to use a more general grammar, where the nonterminals can talk about shared cells.

Related Work We borrowed several interesting ideas from the shape analysis [14]. Our analysis represents a program invariant using a set of shape graphs where each shape graph consists of either concrete or abstract nodes. It uses the idea of refining an abstract node, often called focus or materialization, and also the idea of merging the shape graphs which have a similar structure [9, 5].

The difference is the use of grammar; it is the main reason for the improved precision of our analysis. Another difference is that our analysis separates node-summarizing criteria from the properties of the summary nodes. Normally, the shape analysis of Sagiv et al. partitions all the concrete nodes according to the instrumentation predicates that they satisfy, and groups each partition into a single summary node. Thus, two different summary nodes must satisfy different sets of instrumentation predicates. Our analysis, on the other hand, groups the concrete nodes using the most approximate grammar: each group is a maximal set of concrete nodes that can be expressed by the most approximate grammar. Then, the analysis summarizes each group by a single summary node, and annotates the summary node with a new grammar that "best" describes the pointer structure of the summarized concrete nodes. As a consequence, two different summary nodes in our analysis can have the identical grammar annotations.

Graph type [7, 10] and shape type [6] are also closely related to our work. Both of them express invariants of heap objects (or data structures) using grammar-based languages, which are more expressive than the grammars we used. However, they assume that all the loop invariants of a program are provided, while our work infers such invariants.

Outline Section 2 describes the source programming language. Section 3 overviews separation logic that we use to give the meaning of abstract values. Then, we explain the key ideas of our analysis, using a simpler version that can handle tree-like structures with no shared nodes. Section 4 and 5 explain the abstract domain and abstract operators, and Section 6 defines the analyzer. The simpler version is extended to the full analysis in Section 7. Section 8 demonstrates the accuracy of our analysis using binomial heap construction algorithm and the Schorr-Waite tree traversing algorithm.

2 Programming Language

We use the standard while language with additional pointer operations.

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

This language assumes that every heap cell is binary, having 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 `->`. All the other constructs in the language are standard.

3 Separation Logic with Recursive Predicates

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

$$\mathit{Stack} \triangleq \mathit{Vars} \rightharpoonup_{\text{fin}} \mathit{Val} \qquad \mathit{Heap} \triangleq \mathit{Loc} \rightharpoonup_{\text{fin}} \mathit{Val} \times \mathit{Val} \qquad \mathit{State} \triangleq \mathit{Stack} \times \mathit{Heap}$$

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:[4]

$$P ::= E = E \mid \texttt{emp} \mid (E \mapsto E, E) \mid P * P \mid \texttt{true} \mid P \wedge P \mid P \vee P \mid \neg P \mid \forall x.\, P$$

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 for the other. 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.

$$E \doteq E' \ \triangleq\ E{=}E' \wedge \texttt{emp} \qquad \textstyle\bigodot_{x \in X} A_x \ \triangleq\ \text{if } (X = \emptyset) \text{ then } \texttt{emp} \text{ else } (A_{x_1} * \ldots * A_{x_n})$$

In this paper, we use the extension of the basic assertion language with recursive predicates [15]:

$$P ::= \ldots \mid \alpha(E, \ldots, E) \mid \mathsf{rec}\ \Gamma\ \mathsf{in}\ P \qquad \Gamma ::= \alpha(x_1, \ldots, x_n) = P \mid \Gamma, \Gamma$$

The extension allows the definition of new recursive predicates by least-fixed points in "$\mathsf{rec}\ \Gamma\ \mathsf{in}\ P$", and the use of such defined recursive predicates in $\alpha(E, \ldots, E)$. To ensure the existence of the least-fixed point in $\mathsf{rec}\ \Gamma\ \mathsf{in}\ P$, 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, when 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 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 appears in [8].

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

$$\begin{array}{lll}(s,h),\eta \models \alpha(E) & \text{iff} & ([\![E]\!]s,h) \in \eta(\alpha) \\ (s,h),\eta \models \mathsf{rec}\ \alpha(x){=}P\ \mathsf{in}\ Q & \text{iff} & (s,h),\eta[\alpha{\to}k] \models P \\ \multicolumn{3}{l}{(\text{where } k = \mathrm{lfix}\,\lambda k_0.\{(v',h') \mid (s[x{\to}v'],h'),\eta[\alpha{\to}k_0] \models P\})}\end{array}$$

4 Abstract Domain

Shape Graph Our analysis interprets a program as a (nondeterministic) transformer of *shape graphs*. A shape graph is an abstraction of concrete states; this abstraction maintains the basic "structure" of the state, but abstracts away all the other details. For instance, consider a state $([x{\to}1, y{\to}3], [1{\to}\langle 2,\mathrm{nil}\rangle, 2{\to}\langle \mathrm{nil},\mathrm{nil}\rangle, 3{\to}\langle 1,3\rangle])$. 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 symbols a, b, c, and represent the state by $([x{\to}a, y{\to}c], [a{\to}\langle b,\mathrm{nil}\rangle, b{\to}\langle \mathrm{nil},\mathrm{nil}\rangle, c{\to}\langle a,c\rangle])$. Note that this process abstracts away the specific addresses and just keeps the relationship between the addresses. 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 for nonterminal tree.

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

$$\begin{array}{rl} \mathit{SymLoc} \stackrel{\Delta}{=} \{a,b,c,\ldots\} & \qquad \mathit{NonTerm} \stackrel{\Delta}{=} \{\alpha,\beta,\gamma,\ldots\} \\ \mathit{Graph} \stackrel{\Delta}{=} & (\mathit{Vars} \rightharpoonup_{\mathrm{fin}} \mathit{SymLoc}) \times (\mathit{SymLoc} \rightharpoonup_{\mathrm{fin}} \{\mathrm{nil}\} + \mathit{SymLoc}^2 + \mathit{NonTerm}) \end{array}$$

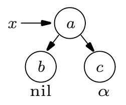
Here the set of nonterminals is disjoint from *Vars* and *SymLoc*; these nonterminals represent recursive heap structures such as tree or list. Each shape graph has two components (s,g). The first component s maps stack variables to symbolic locations. The other component g describes heap cells reachable from each symbolic location. For each a, either no heap cells can be reached from a, i.e, $g(a) = \mathrm{nil}$; or, a is a binary cell with contents $\langle b,c\rangle$; or, the cells reachable from a form a heap object specified by a nonterminal α. We also require that g describes all the cells in the heap; for instance, if g is the empty partial function, it means the empty heap.

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

$$\begin{array}{c} \mathsf{means}_{\mathrm{v}}(a,\mathrm{nil}) \stackrel{\Delta}{=} a \doteq \mathrm{nil} \quad \mathsf{means}_{\mathrm{v}}(a,\alpha) \stackrel{\Delta}{=} \alpha(a) \quad \mathsf{means}_{\mathrm{v}}(a,\langle b,c\rangle) \stackrel{\Delta}{=} (a \mapsto b,c) \\ \mathsf{means}_{\mathrm{s}}(s,g) \stackrel{\Delta}{=} \exists \boldsymbol{a}.\ (\bigodot_{x\in\mathrm{dom}(s)} x \doteq s(x)) * (\bigodot_{a\in\mathrm{dom}(g)} \mathsf{means}_{\mathrm{v}}(a,g(a))) \end{array}$$

The translation function $\mathsf{means}_{\mathrm{s}}$ calls a subroutine $\mathsf{means}_{\mathrm{v}}$ 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, $\mathsf{means}_{\mathrm{s}}([x{\to}a, y{\to}c], [a{\to}\mathsf{tree}, c{\to}\langle a,c\rangle])$ 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 $(s,g) = ([x{\to}a], [a{\to}\langle b,c\rangle, b{\to}\mathrm{nil}, c{\to}\alpha])$ as:



Note that pair $g(a)$ is represented by two edges (the left one is for field 0 and the right one for field 1), and non-pair values $g(b)$ and $g(c)$ by annotations to the nodes.

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.

$$\mathit{Grammar} \overset{\Delta}{=} \mathit{NonTerm} \rightharpoonup_{\text{fin}} \wp_{\mathsf{nf}}(\{\text{nil}\} + (\{\text{nil}\} + \mathit{NonTerm})^2)$$

Set $R(\alpha)$ contains all the possible shapes of heap objects for α. If 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 meaning $\mathsf{means}_{\mathrm{g}}(R)$ of a grammar R is given by a recursive predicate declaration $\mathit{\Gamma}$ in separation logic. $\mathit{\Gamma}$ is defined exactly for $\text{dom}(R)$, and satisfies the following: when nil $\notin R(\alpha)$, $\mathit{\Gamma}(\alpha)$ is

$$\alpha(a) = \bigvee_{\langle v,w\rangle \in R(\alpha)} \exists bc.\, (a {\mapsto} b, c) * \mathsf{means}_{\mathrm{v}}(b,v) * \mathsf{means}_{\mathrm{v}}(c,w),$$

where neither b nor c appears in a, v or w; otherwise, $\mathit{\Gamma}(\alpha)$ is identical as above except that $a \doteq \text{nil}$ is added as a disjunct. For instance, $\mathsf{means}_{\mathrm{g}}([\mathsf{tree} {\to} \{\text{nil}, \langle\mathsf{tree},\mathsf{tree}\rangle\}])$ is $\{\mathsf{tree}(a) = a {\doteq} \text{nil} \vee \exists bc.(a \mapsto b,c) {*} \mathsf{tree}(b) {*} \mathsf{tree}(c)\}$.

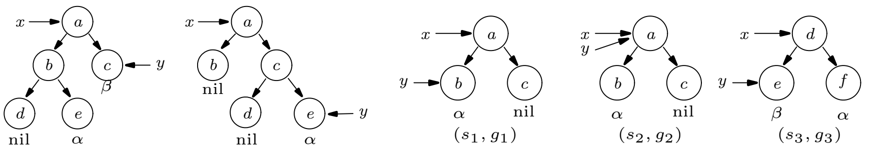
Abstract Domain The abstract domain $\widehat{D}$ for our analysis consists of pairs of a shape graph set and a grammar: $\widehat{D} \overset{\Delta}{=} \{\top\} + \wp_{\mathsf{nf}}(\mathit{Graph}) \times \mathit{Grammar}$. The element $\top$ indicates that our analysis fails to produce any meaningful results for a given program because the program has safety errors, or the program uses data structures too complex for our analysis to capture. The meaning of each abstract state $(\mathcal{G}, R)$ in $\widehat{D}$ is $\mathsf{means}(\mathcal{G},R) \overset{\Delta}{=} \mathsf{rec}\ \mathsf{means}_{\mathrm{g}}(R)\ \mathsf{in} \bigvee_{(s,g)\in\mathcal{G}} \mathsf{means}_{\mathrm{s}}(s,g)$.

5 Normalized Abstract States and Normalization

The main strength of our analysis is to automatically discover a grammar that 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 from $\widehat{D}$ to a subdomain $\widehat{D}^{\triangledown}$ of *normalized abstract states*. In this section, we explain these two notions, normalized abstract states and normalization function.

5.1 Normalized Abstract States

An abstract state $(\mathcal{G}, R)$ is normalized if it satisfies the following two conditions. First, 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. Second, an abstract state does not have redundancies: all shape graphs are not *similar*, and all nonterminals have *non-similar* definitions.



(a) not normalized (b) normalized (c) $(s_1, g_1) \not\sim (s_2, g_2)$ but $(s_1, g_1) \sim (s_3, g_3)$

Fig. 1. Examples of the Normalized Shape Graphs and Similar Shape Graphs.

Normalized Shape Graphs A shape graph is normalized when it is "maximally" folded. A symbolic location a is *foldable* in (s, g) if $g(a)$ is a pair and there is no path from a to a *shared* symbolic location that is referred more than once. When $\text{dom}(g)$ of a shape graph (s, g) does not have any foldable locations, we say that (s, g) is *normalized.* For instance, Figure 1.(a) is not normalized, because b is foldable: b is a pair and does not reach any shared symbolic locations. On the other hand, Figure 1.(b) is normalized, because all the pairs in the graph (i.e., a and c) can reach shared symbolic location e.

Similarity We define three notions of similarity: one for shape graphs, another for two cases of the grammar definitions, and the third for the grammar definitions of two nonterminals.

Two shape graphs are similar when they have the similar structures. Let S be a substitution that renames symbolic locations. Two shape graphs (s, g) and (s', g') are *similar up to* S, denoted $(s, g) \sim_S^G (s', g')$, if and only if

1. $\text{dom}(s) = \text{dom}(s')$ and $S(\text{dom}(g)) = \text{dom}(g')$;
2. for all $x \in \text{dom}(s)$, $S(s(x)) = s'(x)$; and
3. for all $a \in \text{dom}(g)$, if $g(a)$ is a pair $\langle b, c\rangle$ for some b and c, then $g'(S(a)) = \langle S(b), S(c)\rangle$; if $g(a)$ is not a pair, neither is $g'(S(a))$.

Intuitively, two shape graphs are S-similar, when equating nil and all nonterminals makes the graphs identical up to renaming S. We say that (s, g) and (s', g') are similar, denoted $(s, g) \sim (s', g')$, if and only if there is a renaming relation S such that $(s, g) \sim_S^G (s', g')$. For instance, in Figure 1.(c), (s_1, g_1) and (s_2, g_2) are not similar because we cannot find a renaming substitution S such that $S(s_1(x)) = S(s_1(y))$ (condition 2). However, (s_1, g_1) and (s_3, g_3) are similar because a renaming substitution $\{d/a, e/b, f/c\}$ makes (s_1, g_1) identical to (s_3, g_3) when nil and all nonterminals are erased from the graphs.

Cases e_1 and e_2 in the grammar definitions are similar, denoted $e_1 \sim^C e_2$, if and only if either both e_1 and e_2 are pairs, or they are both non-pair values. The similarity $E_1 \sim^D E_2$ between grammar definitions E_1 and E_2 uses this case similarity: $E_1 \sim^D E_2$ 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, in the grammar

$$\alpha ::= \langle \beta, \text{nil}\rangle\,, \quad \beta ::= \text{nil} \mid \langle \beta, \text{nil}\rangle\,, \quad \gamma ::= \langle \gamma, \gamma\rangle \mid \langle \alpha, \text{nil}\rangle$$

the definitions of α and γ are similar because both $\langle \gamma, \gamma\rangle$ and $\langle \alpha, \text{nil}\rangle$ are similar to $\langle \beta, \text{nil}\rangle$. But the definitions of α and β are not similar since α does not have a case similar to nil.

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

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

We write $\widehat{D}^{\nabla}$ for the set of normalized abstract states.

***k*-Bounded Normalized States** Unfortunately, the normalized abstract domain $\widehat{D}^{\nabla}$ does not ensure the termination of the analysis, because it has infinite chains. For each number k, we say that an abstract state $(\mathcal{G}, R)$ is k-bounded iff all the shape graphs in $\mathcal{G}$ use at most k symbolic locations, and we define $\widehat{D}^{\nabla}_k$ to be the set of k-bounded normalized abstract states. This finite domain $\widehat{D}^{\nabla}_k$ is used in our analysis.

5.2 Normalization Function

The normalize function transforms $(\mathcal{G}, R)$ to a normalized $(\mathcal{G}', R')$ with a further abstraction (i.e., $\mathsf{means}(\mathcal{G}, R) \Rightarrow \mathsf{means}(\mathcal{G}', R')$).[5] It is defined by the composition of five subroutines: $\mathsf{normalize} = \mathsf{bound}^k \circ \mathsf{simplify} \circ \mathsf{unify} \circ \mathsf{fold} \circ \mathsf{rmjunk}$.

The first subroutine rmjunk 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 subroutine rmjunk 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**) $(\mathcal{G} \uplus \{(s \cdot [x{\to}a], g \cdot [a{\to}\mathrm{nil}])\}, R) \rightsquigarrow (\mathcal{G} \cup \{(s \cdot [x{\to}a'], g \cdot [a{\to}\mathrm{nil}, a'{\to}\mathrm{nil}])\}, R)$
where a should appear in (s, g) and a' is fresh.

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

For instance, given a shape graph $([x{\to}a, y{\to}a], [a{\to}\mathrm{nil}, c{\to}\mathrm{nil}])$, (**gc**) collects the "garbage" c, and (**alias**) eliminates the "imaginary sharing" between x and y by renaming a in $y{\to}a$. So, the shape graph becomes $([x{\to}a, y{\to}b], [a{\to}\mathrm{nil}, b{\to}\mathrm{nil}])$.

The second subroutine fold converts a shape graph to a normal form, by replacing all foldable symbolic locations by nonterminals. The subroutine fold repeatedly applies the following rule until the abstract state does not change:

(**fold**) $(\mathcal{G} \uplus \{(s, g{\cdot}[a{\to}\langle b, c\rangle, b{\to}v, c{\to}v'])\}, R) \rightsquigarrow (\mathcal{G} \cup \{(s, g{\cdot}[a{\to}\alpha])\}, R{\cdot}[\alpha{\to}\{\langle v, v'\rangle\}])$
where neither b nor c appears in (s, g), α is fresh, and v and v' are not pairs.

The rule recognizes that the symbolic locations b and c are accessed only via a. Then, it represents cell a, plus the reachable cells from b and c by a nonterminal α. Figure 2 shows how the (**fold**) folds a tree.

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 rules to an abstract state $(\mathcal{G}, R)$ until the abstract state does not change:

[5] The normalize function is a reminiscent of the widening in [2, 3].

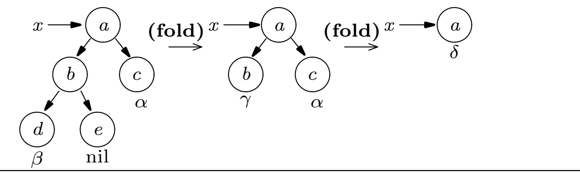
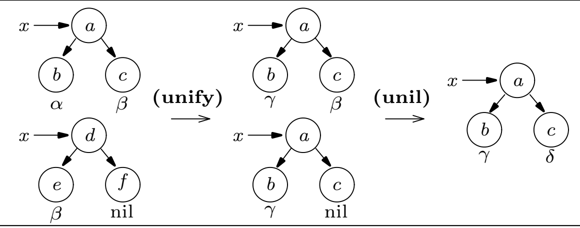
(a) fold		New grammar rules $\gamma ::= \langle \beta, \text{nil} \rangle$ $\delta ::= \langle \gamma, \alpha \rangle$
(b) unify and unil		New grammar rules $\gamma ::= R(\alpha) \cup R(\beta)$ $\delta ::= R(\beta) \cup \{\text{nil}\}$

Fig. 2. Examples of (**fold**), (**unify**), and (**unil**).

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

The (**unify**) rule recognizes two similar shape graphs that have different nonterminals at the same position, and replaces those nonterminals by fresh nonterminal β that covers the two nonterminals. The (**unil**) rule deals with the two similar graphs that have, respectively, nonterminal and nil at the same position. For instance, in Figure 2.(b), the left two shape graphs are unified by (**unify**) and (**unil**). We first replace the left children α and β by γ that covers both; that is, to a given grammar R, we add $[\gamma \to R(\alpha) \cup R(\beta)]$. Then we replace the right children β and nil by δ that covers both.

The fourth subroutine simplify reduces the complexity of grammar by combining similar cases or similar definitions.[6] It 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 for the second field.
- If the definition of a nonterminal has two similar cases $\langle \beta, v \rangle$ and $\langle \text{nil}, v' \rangle$, add the nil case to $R(\beta)$ and replace $\langle \text{nil}, v' \rangle$ by $\langle \beta, v' \rangle$. Apply the same for the second field.
- If the definitions of two nonterminals are similar, unify the nonterminals.

Formally, the three rules are:

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

Here, $(\mathcal{G}, R)\{\alpha/\beta\}$ substitutes α for β, and in addition, it removes the definition of β from R and re-defines α such that α covers both α and β:

[6] The simplify subroutine is similar to the widening operator in [4].

$$(\mathcal{G}, R \cdot [\alpha{\to}E_1, \beta{\to}E_2])\,\{\alpha/\beta\} \triangleq (\mathcal{G}\,\{\alpha/\beta\}, R\,\{\alpha/\beta\} \cdot [\alpha{\to}(E_1 \cup E_2)\,\{\alpha/\beta\}]).$$

For example, consider the following transitions:

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

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

The last subroutine bound^k checks the number of symbolic locations in each shape graph. The subroutine bound^k simply gives $\top$ when one of shape graphs has more than k symbolic locations, thereby ensuring the termination of the analysis.[7]

$$\mathsf{bound}^k(\mathcal{G}, R) = \text{if (no } (s, g) \text{ in } \mathcal{G} \text{ has more than } k \text{ symbolic locations) then } (\mathcal{G}, R) \text{ else } \top$$

Lemma 1. *Given every abstract state $(\mathcal{G}, R)$, $\mathsf{normalize}(\mathcal{G}, R)$ always terminates, and its result is a k-bounded normalized abstract state.*

6 Analysis

Our analyzer (defined in Figure 3) 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 static analysis: 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 the 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 analyzer unfolds the grammar definition by calling the subroutine unfold. Given a shape graph (s, g), a variable x and a grammar R, the subroutine unfold first checks whether $g(s(x))$ is a nonterminal or not. If $g(s(x))$ is a nonterminal α, unfold looks up the definition of α in R and unrolls this definition in the shape graph (s, g): for each case e in $R(\alpha)$, it updates g by $[s(x){\to}e]$. For instance, when $R(\beta) = \{\langle\beta,\gamma\rangle, \langle\delta,\delta\rangle\}$, $\mathsf{unfold}(([x{\to}a], [a{\to}\beta]), R, x)$ is shape-graph set $\{([x{\to}a], [a{\to}\langle\beta,\gamma\rangle]),\ ([x{\to}a], [a{\to}\langle\delta,\delta\rangle])\}$.

[7] Limiting the number of symbolic locations to be at most k ensures the termination of the analyzer in the *worst case*. When programs use data structures that our grammar captures well, the analysis usually terminates without using this k limitation, and yields meaningful results.

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

$$\begin{aligned}
\llbracket x \texttt{ := new} \rrbracket\,(\mathcal{G},R) &= (\{(s[x{\to}a],\ g[a{\to}\langle b,c\rangle, b{\to}\mathrm{nil}, c{\to}\mathrm{nil}]) \mid (s,g)\in\mathcal{G}\}, R)\ \text{new } a,b,c\\
\llbracket x \texttt{ := nil} \rrbracket\,(\mathcal{G},R) &= (\{(s[x{\to}a], g[a{\to}\mathrm{nil}]) \mid (s,g)\in\mathcal{G}\}, R)\ \text{new } a\\
\llbracket x \texttt{ := } y \rrbracket\,(\mathcal{G},R) &= \text{when } y\in\mathrm{dom}(s) \text{ for all } (s,g)\in\mathcal{G},\\
&\qquad (\{(s[x{\to}s(y)], g) \mid (s,g)\in\mathcal{G}\}, R)\\
\llbracket x\texttt{->}0 \texttt{ := } y \rrbracket\,(\mathcal{G},R) &= \text{when } \mathsf{unfold}(\mathcal{G},R,x)=\mathcal{G}' \text{ and } \forall (s,g)\in\mathcal{G}'.\, y\in\mathrm{dom}(s),\\
&\qquad (\{(s, g[s(x){\to}\langle s(y),c\rangle] \mid (s,g)\in\mathcal{G}',\ g(s(x))=\langle b,c\rangle\}, R)\\
\llbracket x \texttt{ := } y\texttt{->}0 \rrbracket\,(\mathcal{G},R) &= \text{when } \mathsf{unfold}(\mathcal{G},R,y)=\mathcal{G}',\\
&\qquad (\{(s[x{\to}b], g) \mid (s,g)\in\mathcal{G}',\ g(s(y))=\langle b,c\rangle\}, R)\\
\llbracket C_1 ; C_2 \rrbracket\,(\mathcal{G},R) &= \llbracket C_2\rrbracket\,(\llbracket C_1\rrbracket\,(\mathcal{G},R))\\
\llbracket \texttt{if } B\ C_1\ C_2 \rrbracket\,(\mathcal{G},R) &= \llbracket C_1\rrbracket\,(\llbracket B\rrbracket\,(\mathcal{G},R))\ \dot{\sqcup}\ \llbracket C_2\rrbracket\,(\llbracket \texttt{!}B\rrbracket\,(\mathcal{G},R))\\
\llbracket \texttt{while } B\ C \rrbracket\,(\mathcal{G},R) &= \llbracket \texttt{!}B\rrbracket \left(\mathrm{lfix}^{\dot{\sqsubseteq}}\ \lambda A\colon \widehat{D}^{\nabla}_{k}.\ \mathsf{normalize}(A\ \dot{\sqcup}\ (\mathcal{G},R)\ \dot{\sqcup}\ \llbracket C\rrbracket\,(\llbracket B\rrbracket A))\right)\\
\llbracket C \rrbracket A &= \top \text{ (other cases)}
\end{aligned}$$

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

$$\begin{aligned}
\llbracket x \texttt{ = } y \rrbracket\,(\mathcal{G},R) &= \text{when } \mathsf{split}(\mathsf{split}((\mathcal{G},R),x),y) = (\mathcal{G}',R')\\
&\qquad (\{(s,g)\in\mathcal{G}' \mid s(x){\equiv}s(y) \lor g(s(x)){=}g(s(y)){=}\mathrm{nil}\},\ R')\\
\llbracket \texttt{!}x \texttt{ = } y \rrbracket\,(\mathcal{G},R) &= \text{when } \mathsf{split}(\mathsf{split}((\mathcal{G},R),x),y) = (\mathcal{G}',R')\\
&\qquad (\{(s,g)\in\mathcal{G}' \mid s(x){\not\equiv}s(y) \land (g(s(x)){\neq}\mathrm{nil} \lor g(s(y)){\neq}\mathrm{nil})\},\ R')\\
\llbracket \texttt{!(!}B\texttt{)} \rrbracket\,(\mathcal{G},R) &= \llbracket B\rrbracket\,(\mathcal{G},R)\\
\llbracket B \rrbracket\ A &= \top \text{ (other cases)}
\end{aligned}$$

Subroutine unfold unrolls the definition of a grammar:

$$\mathsf{unfold}((s,g),R,x) = \begin{cases} \{(s, g[a{\to}\langle b,c\rangle, b{\to}v, c{\to}u]) \mid \langle v,u\rangle \in R(a)\} & \text{if } g(s(x)){=}\alpha \land \mathrm{nil}{\notin}R(\alpha)\\ \{(s,g)\} & \text{if } g(s(x)) \text{ is a pair}\\ \top & \text{otherwise}\end{cases}$$

$$\mathsf{unfold}(\mathcal{G},R,x) = \begin{cases} \bigcup_{(s,g)\in\mathcal{G}} \mathsf{unfold}((s,g),R,x) & \text{if } \forall (s,g)\in\mathcal{G}.\ \mathsf{unfold}((s,g),R,x)\neq\top\\ \top & \text{otherwise}\end{cases}$$

Subroutine $\mathsf{split}((s,g),R,x)$ changes (s,g) to (s',g') s.t. $s'(x)$ means nil iff $g'(s'(x))$=nil.

$$\begin{aligned}
\mathsf{split}((s,g),R,x) = {}&\text{if } (\exists\alpha.\, g(s(x)){=}\alpha \land R(\alpha){\supseteq}\{\mathrm{nil}\} \land R(\alpha)\neq\{\mathrm{nil}\})\\
&\quad \text{then } (\{(s, g[s(x){\to}\mathrm{nil}]), (s, g[s(x){\to}\beta])\}, R[\beta{\to}R(\alpha){-}\{\mathrm{nil}\}]) \text{ for fresh } \beta\\
&\quad \text{else if } (\exists\alpha.\, g(s(x)){=}\alpha \land R(\alpha){=}\{\mathrm{nil}\}) \text{ then } (\{(s, g[s(x){\to}\mathrm{nil}])\}, R)\\
&\qquad\qquad \text{else } (\{(s,g)\}, R)
\end{aligned}$$

$$\mathsf{split}(\mathcal{G},R,x) = \begin{cases} \dot{\bigsqcup}_{(s,g)\in\mathcal{G}} \mathsf{split}((s,g),R,x) & \text{if } \forall (s,g){\in}\mathcal{G}.\, x\in\mathrm{dom}(s)\\ \top & \text{otherwise}\end{cases}$$

The algorithmic order $\dot{\sqsubseteq}$ defined in [8] satisfies that if $A \mathrel{\dot{\sqsubseteq}} B$, $\mathsf{means}(A) \Rightarrow \mathsf{means}(B)$

Fig. 3. Analysis.

7 Full Analysis

The basic version of our analysis, which we have presented so far, cannot deal with data structures with sharing, such as doubly linked lists and binomial heaps. If a program uses such data structures, the analysis gives up and returns $\top$.

The full analysis overcomes this shortcoming by using 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 overall structure of the full analysis is almost identical to the basic version in Figure 3. Only the subroutines, such as normalize, are modified. In this section, we will explain the full analysis by focusing on the new parameterized grammar, and the modified normalization function for this grammar. The full definition is in [8].

7.1 Abstract Domain

Let self and arg be two different symbolic locations. In the full analysis, the domains for shape graphs and grammars are modified as follows:

$$\begin{array}{l} \mathit{NTermApp} \triangleq \mathit{NonTerm}\times(\mathit{SymLoc}+\bot) \quad \mathit{NTermAppR} \triangleq \mathit{NonTerm}\times(\{\mathsf{self},\mathsf{arg}\}+\bot) \\ \mathit{Graph} \triangleq (\mathit{Vars} \rightharpoonup_{\text{fin}} \mathit{SymLoc}) \times (\mathit{SymLoc} \rightharpoonup_{\text{fin}} \{\text{nil}\} + \mathit{SymLoc}^2 + \mathit{NTermApp}) \\ \mathit{Grammar} \triangleq \mathit{NonTerm} \rightharpoonup_{\text{fin}} \wp_{\mathsf{nf}}(\{\text{nil}\} + (\{\text{nil}\}+\{\mathsf{self},\mathsf{arg}\}+\mathit{NTermAppR})^2) \end{array}$$

The main 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.[8] For instance, a doubly-linked list is defined by $\mathsf{dll} ::= \text{nil} \mid \langle\mathsf{arg},\mathsf{dll}(\mathsf{self})\rangle$. This grammar maintains the invariant that arg points to the previous cell. So, the first field of a node always points to the previous cell, and the second field the the next cell. 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 $\{\text{nil},\langle\mathsf{arg},\mathsf{arg}\rangle\}$, then $\beta(\bot)$ excludes $\langle\mathsf{arg},\mathsf{arg}\rangle$, and means the empty heap object.

As in the basic case, the precise meaning of a shape graph and a grammar is given by a translation into separation-logic assertions. We can define a translation means by modifying only $\mathsf{means}_{\mathrm{v}}$ and $\mathsf{means}_{\mathrm{g}}$.

$$\begin{array}{rl} \mathsf{means}_{\mathrm{v}}(a,\text{nil}) \triangleq a \doteq \text{nil} & \mathsf{means}_{\mathrm{v}}(a,\alpha(b)) \triangleq \alpha(a,b) \\ \mathsf{means}_{\mathrm{v}}(a,b) \triangleq a \doteq b & \mathsf{means}_{\mathrm{v}}(a,\alpha(\bot)) \triangleq \forall b.\alpha(a,b) \end{array}$$

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.

$$\begin{array}{rcl} \mathsf{means}_{\mathrm{g}}(R) & \triangleq & \{\alpha(a,b)=\bigvee_{e\in R(\alpha)} \mathsf{means}_{\mathrm{gc}}(a,b,e)\}_{\alpha\in\text{dom}(R)} \\ \mathsf{means}_{\mathrm{gc}}(a,b,\text{nil}) & \triangleq & \mathsf{means}_{\mathrm{v}}(a,\text{nil}) \\ \mathsf{means}_{\mathrm{gc}}(a,b,\langle v_1,v_2\rangle) & \triangleq & \exists a_1a_2.\,(a\mapsto a_1,a_2)*\mathsf{means}_{\mathrm{v}}(a_1,v_1\,\{a/\mathsf{self},b/\mathsf{arg}\}) \\ & & \qquad *\mathsf{means}_{\mathrm{v}}(a_2,v_2\,\{a/\mathsf{self},b/\mathsf{arg}\}) \end{array}$$

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

7.2 Normalization Function

To fully exploit the increased expressivity of the abstract domain, we change the normalization function in the full analysis. The most important change in the new normalization function is the addition of new rules (**cut**) and (**bfold**) into the fold procedure.

[8] We allow only "one" explicit parameter. So, we can use pre-defined name arg.

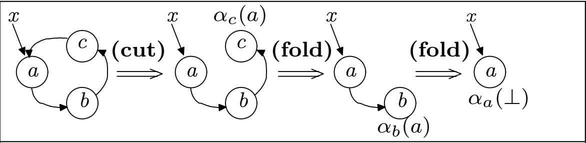
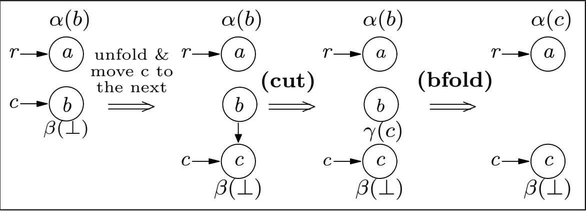
(a) cut	 x $\alpha_c(a)$ (cut) (fold) (fold) $\alpha_b(a)$ $\alpha_a(\bot)$	$\alpha_c ::= \langle \mathsf{arg} \rangle$ $\alpha_b ::= \langle \alpha_c(\mathsf{arg}) \rangle$ $\alpha_a ::= \langle \alpha_b(\mathsf{arg}) \rangle$
(b) bfold	 $\alpha(b)$ r c $\beta(\bot)$ unfold & move c to the next (cut) $\gamma(c)$ (bfold) $\alpha(c)$	Initial grammar $\alpha ::= \langle \alpha(\mathsf{arg}) \rangle \mid \langle \mathsf{arg} \rangle$ $\beta ::= \langle \beta(\bot) \rangle \mid \mathrm{nil}$ Final grammar $\alpha ::= \langle \alpha(\mathsf{arg}) \rangle \mid \langle \gamma(\mathsf{arg}) \rangle$ $\beta ::= \langle \beta(\bot) \rangle \mid \mathrm{nil}$ $\gamma ::= \langle \mathsf{arg} \rangle$

Fig. 4. Examples of (**cut**) and (**bfold**).

The (**cut**) rule enables the conversion of a cyclic structure to grammar definitions. Recall that the (**fold**) rule can recognize a heap object only when the object does not have shared cells internally. The key idea is to "cut" a "non-critical" 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 of (**fold**) can be applied. The formal definition of the (**cut**) rule is:

$$(\textbf{cut})\ (\mathcal{G} \uplus \{(s, g \cdot [a \rightarrow \langle a_1, a_2 \rangle])\}, R) \rightsquigarrow \begin{pmatrix} \mathcal{G} \cup \{(s, g \cdot [a \rightarrow \alpha(b)])\}, \\ R \cdot [\alpha \rightarrow \{\langle a_1, a_2 \rangle \{\mathsf{self}/a, \mathsf{arg}/b\}\}] \end{pmatrix}$$

where there are paths from variables to a_1 and a_2 in g, $\mathsf{free}(\langle v_1, v_2 \rangle) \subseteq \{a, b\}$, and α is fresh. (If $\mathsf{free}(\langle v_1, v_2 \rangle) \subseteq \{a\}$, we use $\alpha(\bot)$ instead of $\alpha(b)$.)

Figure 4.(a) shows how a cyclic structure is converted to grammar definitions.[9] In the first shape graph, "cell" a is shared because variable x points to a and "cell" c points to a, but the link from c to a is not critical because even without this link, a is still reachable from x. Thus, the (**cut**) rule cuts the link from c to a, introduces a nonterminal α_c with the definition $\{\langle \mathsf{arg} \rangle\}$, and annotates node c with $\alpha_c(a)$. Note that the resulting graph (the second shape graph in Figure 4.(a)) does not have explicit sharing. So, we can apply the (**fold**) rule to c, and then to b as shown in the last two shape graphs in Figure 4.(a).

The (**bfold**) rule 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 for a nonterminal. The (**bfold**) rule, on the other hand, puts a cell a at the exit of a heap object. When b is used as a parameter for a nonterminal α, the rule "combines" b and α. This rule can best be explained using a 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 first shape graph in Figure 4.(b). 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 shape graph into the second shape graph in Figure 4.(b), but this new graph is not similar to the initial one. The (**bfold**) rule changes the new shape graph back to the one for the invariant, by merging $\alpha(b)$ with cell b. The (**cut**) rule

[9] To simplify the presentation, we assume that each cell in the figure has only a single field.

first cuts the link from b to c, extends a grammar with $[\gamma{\to}\{\langle\mathsf{arg}\rangle\}]$, and annotates the node b with $\gamma(c)$. Then, the (**bfold**) rule finds all the places where arg is used as itself in the definition of α, and replaces arg there by $\gamma(\mathsf{arg})$. Finally, the rule changes the binding for a from $\alpha(b)$ to $\alpha(c)$, and eliminates cell b, thus resulting the last shape graph in Figure 4(b).[10] The precise definition of (**bfold**) does what we call *linearity* check, in order to ensure the soundness of replacing arg by nonterminals:[11]

(**bfold**) $(\mathcal{G} \cup \{(s, g \cdot [a{\to}\alpha(b), b{\to}\beta(w)])\}, R) \rightsquigarrow (\mathcal{G} \cup \{(s, g \cdot [a{\to}\alpha'(w)])\}, R \cdot [\alpha'{\to}E])$
where b does not appear in g, α is linear (that is, arg appears exactly once in each case of $R(\alpha)$), and $E = \{\mathrm{nil} \in R(\alpha)\} \cup \{\langle f(v_1), f(v_2)\rangle \mid \langle v_1, v_2\rangle \in R(\alpha)\}$
where $f(v) = \mathsf{if}\ (v \equiv \mathsf{arg})\ \mathsf{then}\ \beta(\mathsf{arg})\ \mathsf{else}\ \Big(\mathsf{if}\ (v \equiv \alpha(\mathsf{arg}))\ \mathsf{then}\ \alpha'(\mathsf{arg})\mathsf{else}\ v\Big)$

7.3 Correctness

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

Theorem 1. *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.*

We proved this theorem in two steps. First, we showed a lemma that all subroutines, such as normalize and unfold, and the backward analysis are correct. Then, with this lemma, we applied the induction on the structure of C, and showed that $\{\mathsf{means}(\mathcal{G}, R)\}C\{\mathsf{means}(\mathcal{G}', R')\}$ is derivable in separation logic. The validity of the triple now follows, because separation-logic proof rules are sound. The details are in [8].

8 Experiments

We have tested our analysis with the six programs in Table 1. For each of the programs, we ran the analyzer, and obtained abstract states for a loop invariant and the result. In this section, we will explain the cases of binomial heap construction and the Schorr-Waite tree traversal. The others are explained at `http://ropas.snu.ac.kr/grammar`.

Binomial Heap Construction In this experiment, we took an implementation of binomial heap construction in [1], where each cell has three pointers: one to the left-most child, another to the next sibling, and the third to the parent. 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. Here we omit $\perp$ from $\mathsf{forest}(\perp)$.

[10] 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.

[11] Here we present only for the case that the parameter of α is not passed to another different nonterminals. With such nonterminals, we need to do a more serious linearity check on those nonterminals, before modifying the grammar.

program	description	cost(sec)	analysis result
`listrev.c`	list construction followed by list reversal	0.01	the result is a list
`dbinary.c`	construction of a tree with parent pointers	0.01	the result is a tree with parent pointers
`dll.c`	doubly-linked list construction	0.01	the result is a doubly-linked list
`bh.c`	binomial heap construction	0.14	the result is a binomial heap
`sw.c`	Schorr-Waite tree traversal	0.05	the result is a tree
`swfree.c`	Schorr-Waite tree disposal	0.02	the tree is completely disposed

For all the examples, our analyzer proves the absence of null pointer dereference errors and memory leaks.

Table 1. Experimental Results

$$\mathcal{G} = \big\{ \big([x{\rightarrow}a], [a{\rightarrow}\mathsf{forest}] \big) \big\} \quad R = \begin{bmatrix} \mathsf{forest} ::= \mathrm{nil} \mid \langle \mathsf{stree}(\mathsf{self}), \mathsf{forest}, \mathrm{nil} \rangle \,, \\ \mathsf{stree} ::= \mathrm{nil} \mid \langle \mathsf{stree}(\mathsf{self}), \mathsf{stree}(\mathsf{arg}), \mathsf{arg} \rangle \end{bmatrix}$$

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.

Schorr-Waite Tree Traversal We used the following $(\mathcal{G}_0, R_0)$ as an initial abstract state:

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

Here we omit $\perp$ from $\mathsf{tree}(\perp)$. This abstract state means that the initial heap contains only a binary tree a whose cells are marked I.

Given the traversing algorithm and the abstract state $(\mathcal{G}_0, R_0)$, the analyzer produced $(\mathcal{G}_1, R_1)$ for final states, and $(\mathcal{G}_2, R_2)$ for a loop invariant:

$$\mathcal{G}_1 = \big\{ \big([x{\rightarrow}a], [a{\rightarrow}\mathsf{treeR}] \big) \big\} \qquad R_1 = [\mathsf{treeR} ::= \mathrm{nil} \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{treeR} \rangle]$$
$$\mathcal{G}_2 = \big\{ \big([x{\rightarrow}a, y{\rightarrow}b], [a{\rightarrow}\mathsf{treeRI}, b{\rightarrow}\mathsf{rtree}] \big) \big\}$$
$$R_2 = \begin{bmatrix} \mathsf{rtree} ::= \mathrm{nil} \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{rtree} \rangle \mid \langle \mathrm{L}, \mathsf{rtree}, \mathsf{tree} \rangle \,, \quad \mathsf{tree} ::= \mathrm{nil} \mid \langle \mathrm{I}, \mathsf{tree}, \mathsf{tree} \rangle \,, \\ \mathsf{treeR} ::= \mathrm{nil} \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{treeR} \rangle \,, \quad \mathsf{treeRI} ::= \mathrm{nil} \mid \langle \mathrm{I}, \mathsf{tree}, \mathsf{tree} \rangle \mid \langle \mathrm{R}, \mathsf{treeR}, \mathsf{treeR} \rangle \end{bmatrix}$$

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.

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 an I-marked binary tree, or an 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 an R-marked binary tree. The third case, $\langle \mathrm{L}, \mathsf{rtree}, \mathsf{tree} \rangle$, means that the current cell is marked L, the left field is reversed, and the right subtree is an 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.

References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill Book Company, 2001.
2. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
3. Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Logic and Comput.*, 2(4):511–547, 1992.
4. Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, June 1995. ACM Press, New York, NY.
5. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 230–241. ACM Press, 1994.
6. Pascal Fradet and Daniel Le Métayer. Shape types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 27–39. ACM Press, January 1997.
7. Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1993.
8. Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. Tech. Memo. ROPAS-2005-23, Programming Research Laboratory, School of Computer Science & Engineering, Seoul National University, March 2005.
9. R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Proceedings of the International Symposium on Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 265–279, August 2004.
10. A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* ACM, June 2001.
11. Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 268–280. ACM Press, January 2004.
12. 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.
13. 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.
14. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
15. Élodie-Jane Sims. Extending separation logic with fixpoints and postponed substitution. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology*, volume 3116 of *Lecture Notes in Computer Science*, pages 475–490, 2004.