Checking the Shape Safety of Pointer Manipulations

Adam Bakewell, Detlef Plump, and Colin Runciman

Department of Computer Science
University of York, UK
{ajb,det,colin}@cs.york.ac.uk

Abstract. We present a new algorithm for checking the shape-safety of pointer manipulation programs. In our model, an abstract, data-less pointer structure is a *graph*. A *shape* is a language of graphs. A pointer manipulation program is modelled abstractly as a set of graph rewrite rules over such graphs where each rule corresponds to a pointer manipulation step. Each rule is annotated with the intended shape of its domain and range and our algorithm checks these annotations.
We formally define the algorithm and apply it to a binary search tree insertion program. Shape-safety is undecidable in general, but our method is more widely applicable than previous checkers, in particular, it can check programs that temporarily violate a shape by the introduction of intermediate shape definitions.

1 Introduction

In imperative programming languages, pointers are key to the efficiency of many algorithms. But pointer programming is an error-prone weak point in software development. The type systems of most current programming languages cannot detect non-trivial pointer errors which violate the intended shapes of pointer data structures. From a programming languages viewpoint, programmers need means by which to specify the shapes of pointer data structures, together with safety checkers to guarantee statically that a pointer program always preserves these shapes.

For example, Figure 1 defines a simple program for insertion in binary search trees, written in a pseudo-C notation. Ideally the type system of this language should allow a definition of `BT` to specify exactly the class of binary trees, and the type checker would verify that whenever the argument `t` is a pointer to a binary tree and `insert` returns, the result is a pointer to a binary tree. Such a system would guarantee that the program does not create any dangling pointers or shape errors such as creating sharing or cycles within the tree and that there are no null pointer dereferences. It would not guarantee the stronger property that `insert` does insert `d` properly at the appropriate place in the tree because that is not a pointer safety issue.

The method developed in our *Safe Pointers by Graph Transformation* project [SPG] is to specify the shape of a pointer data-structure by graph reduction rules, see Section 2 and [BPR03b]. Section 3 models the operations upon the

R. Berghammer et al. (Eds.): RelMiCS/Kleene-Algebra Ws 2003, LNCS 3051, pp. 48–61, 2004.

```
BT *insert(datum d, BT *t) = {
  a~:= t;
  while branch(a) && a->data != d do
    if a->data > d
    then a~:= a->left
    else a~:= a->right;
  if leaf(a)
  then *a := branch{data=d, left=leaf, right=leaf};
  return(t)
}
```

Fig. 1. Binary search tree insertion program

data structure by more general graph transformation rules. Section 5 describes our language-independent algorithm for checking shape preservation, which is based on Fradet and Le Metayer's algorithm [FM97, FM98]. It automatically proves the shape safety of operations such as search, insertion and deletion in cyclic lists, linked lists and binary search trees. It can also handle operations that temporarily violate shapes if the intermediate shapes are specified, see Section 4. Section 6 considers related work and concludes.

This paper formalises the overview we gave in [BPR03a], a much more detailed explanation including the proofs omitted from this paper and a number of alternative checking algorithms is provided by the technical report [Bak03].

2 Specifying Shapes by Graph Reduction

A *shape* is a language of labelled, directed graphs. This sections summarises our method of specifying shapes (see [BPR03b]) and presents an example specification of binary trees.

A *graph* $G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle$ consists of: a finite set of nodes V_G; a finite set of arcs E_G; total functions $s_G, t_G : E_G \to V_G$ assigning a source and target vertex to each arc; a partial node labelling function $l_G : V_G \to \mathcal{C}_V$; and a total arc labelling function $m_G : E_G \to \mathcal{C}_E$. Graph G is an abstract model of a pointer data structure which retains only the pointer fields. Each node models a record of pointers. Nodes are labelled from the *node alphabet* $\mathcal{C}_V$ to indicate their tag. Graph arcs model a pointer field of their source node; their label, drawn from the *arc alphabet* $\mathcal{C}_E$ indicates which pointer field. The *label type* function $type : \mathcal{C}_V \to \wp(\mathcal{C}_E)$ specifies that if node v is labelled l and the source of arc e is v then the label of e must be in $type(l)$ and e must be the only such arc. Together, $\langle \mathcal{C}_V, \mathcal{C}_E, type \rangle$ form a *signature* Σ and G is a *Σ-graph*.

Graphs may occur in rewrite rules or as language (shape) members. Language members are always *Σ-total* meaning that every node v is labelled with some l and the labels of the arcs whose source is v together equal $type(l)$; so they model closed pointer structures with no missing or dangling pointers.

A *graph morphism* $g : G \to H$ consists of a node mapping $g_V : V_G \to V_H$ and an arc mapping $g_E : E_G \to E_H$ that preserve sources, targets and labels: $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $m_H \circ g_E = m_G$ and $l_H(g_V(v)) = l_G(v)$ for all nodes v where $l_G(v) \neq \bot$. An *isomorphism* is a morphism that is injective and surjective in both components and maps unlabelled nodes to unlabelled nodes. If there is an isomorphism from G to H they are *isomorphic*, denoted by $G \cong H$. Applying morphism $g : G \to H$ to graph G yields a graph gG where: $V_{gG} = g_V V_G$ (i.e. apply g_V to each node in V_G); $E_{gG} = g_E E_G$; $s_G(e) = n \Leftrightarrow s_{gG}(g_E(e))$ and similarly for targets; $m_G(e) = m \Leftrightarrow m_{gG}(g_E(e)) = m$; $l_G(n) = l \Leftrightarrow l_{gG}(g_V(n)) = l$.

A *graph inclusion* $H \supseteq G$ is a graph morphism $g : G \to H$ such that $g(x) = x$ for all nodes and arcs x in G.

A *rule* $r = \langle L \supseteq K \subseteq R \rangle$ consists of three graphs: the *interface* graph K and the *left* and *right* graphs L and R which both include K. Intuitively, a rule deletes nodes and arcs in $L - K$, preserves those in K and allocates those in $R - K$. Our pictures of rules show the left and right graphs; the interface is always just their common nodes which are indicated by numbers.

Graph G *directly derives* graph H through rule $r = \langle L \supseteq K \subseteq R \rangle$, injective morphism g and isomorphism i, written $G \Rightarrow H$ or $G \Rightarrow_{r,g,i} H$, if the diagram below consists of pushouts (1) and (2) and an i arrow (see [HMP01] for a full definition of pushouts).

$$\begin{array}{ccccc} L & \supseteq & K & \subseteq & R \\ g \downarrow & (1) & \downarrow & (2) & \downarrow \\ G & \supseteq & D & \subseteq & H' \xrightarrow{i} H \end{array}$$

Injectivity means that distinct nodes in L must be distinct in gL; the pushout construction means that deleted nodes, those in $gL - gK$, cannot be adjacent to any arcs in D if the derivation exists (the *dangling condition*).

If $H \cong G$ or H is derived from G by a sequence of direct derivations using rules in set $\mathcal{R}$ we write $G \Rightarrow^*_{\mathcal{R}} H$ or $G \Rightarrow^* H$. If no graph can be directly derived from G through a rule in $\mathcal{R}$ we say G is $\mathcal{R}$*-irreducible*.

A *GRS* (graph reduction specification) $S = \langle \Sigma, \mathcal{R}, Acc \rangle$ consists of a signature Σ, a set of Σ-total rules $\mathcal{R}$ and a Σ-total $\mathcal{R}$-irreducible *accepting graph* Acc. It defines a language $\mathcal{L}(S) = \{G \mid G \Rightarrow^*_{\mathcal{R}} Acc\}$.

So a GRS is a reversed graph grammar: Acc corresponds to the start graph and $\mathcal{R}$ corresponds to reversed production rules. The rules are Σ*-total* meaning that if $G \Rightarrow_{\mathcal{R}} H$ then G is a Σ-total graph iff H is a Σ-total graph. So GRSs are guaranteed to define languages of pointer structure models.

A GRS S is *polynomially terminating* if there is a polynomial p such that for every derivation $G \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \cdots \Rightarrow_{\mathcal{R}} G_n$, $n \leq p(\#V_G + \#E_G)$. It is *closed* if $G \in \mathcal{L}(S)$ and $G \Rightarrow_{\mathcal{R}} H$ implies $H \in \mathcal{L}(S)$. A *PGRS* is a polynomially terminating and closed GRS. Membership of PGRS languages is decidable in polynomial time; this and sufficient conditions for closedness, polynomial termination and Σ-totality are discussed in [BPR03b].

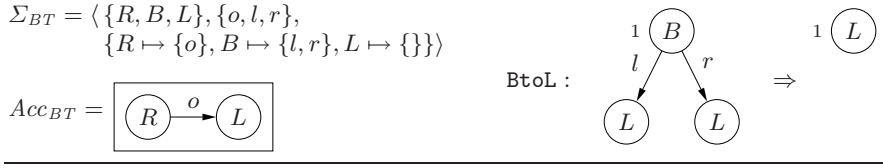

Fig. 2. A PGRS of rooted binary trees, BT.

For example, a *rooted binary tree* is a graph containing one root node labelled R and a number of other nodes labelled B(ranch) or L(eaf). Branch nodes have l(eft) and r(ight) outgoing arcs, the root has one o(rigin) outgoing arc and leaf nodes have no outgoing arcs. Branches and leaves all have one incoming arc, the root has no incoming arcs and every node is reachable from the root. If every branch contains a data item, the leaves contain no data and the data is ordered it is a *binary search tree*.

The data-less shape is specified by the PGRS BT in Fig. 2. The BT signature allows nodes to be labelled R, B or L, where R-nodes have an o-labelled outgoing arc, B-nodes have two outgoing arcs labelled l and r, and L-nodes have no outgoing arcs. The BT accepting graph Acc_{BT} is the smallest possible tree and every other tree reduces to it by repeatedly applying the reduction rule `BtoL` to its branches. No non-tree reduces to Acc_{BT} because `BtoL` is matched injectively and cannot be applied if deleted nodes are adjacent to arcs outside the left-hand side of the rule, see [BPR03b] for an example. BT is polynomially terminating and closed because `BtoL` is size reducing and *non-overlapping*. See [BPR03b] for full details.

3 Graph Transformation Models of Pointer Programs

Textbooks on data structures often present pointer programs pictorially and then formalise them as imperative programs. In our approach a pictorial presentation is a formal graph-transformation model of a program.

A *model pointer program* in the sense of this paper is a set of rules with a strategy for their application (see [HP01] for more on the the syntax and semantics of such programs). Programs may temporarily violate the shape of a graph so the rule construction is not as restricted as the Σ-total reduction rules.

The rules in Fig. 3 model all the pointer manipulation steps in a *binary search tree insertion* program such as that in Figure 1. They manipulate graphs over two signatures, AT is an extension of BT which allows A-labelled nodes with two outgoing arcs labelled a and o. The idea is to model insertion by replacing the R-labelled tree root with an A-labelled *auxiliary root*, moving the a-arc to the insertion position and changing the tree structure at that point appropriately. So the control strategy is to apply the `Begin` rule once, then apply `GoLeft` and `GoRight` any number of times, then apply either `Insert` or `Found`. The *seman-*

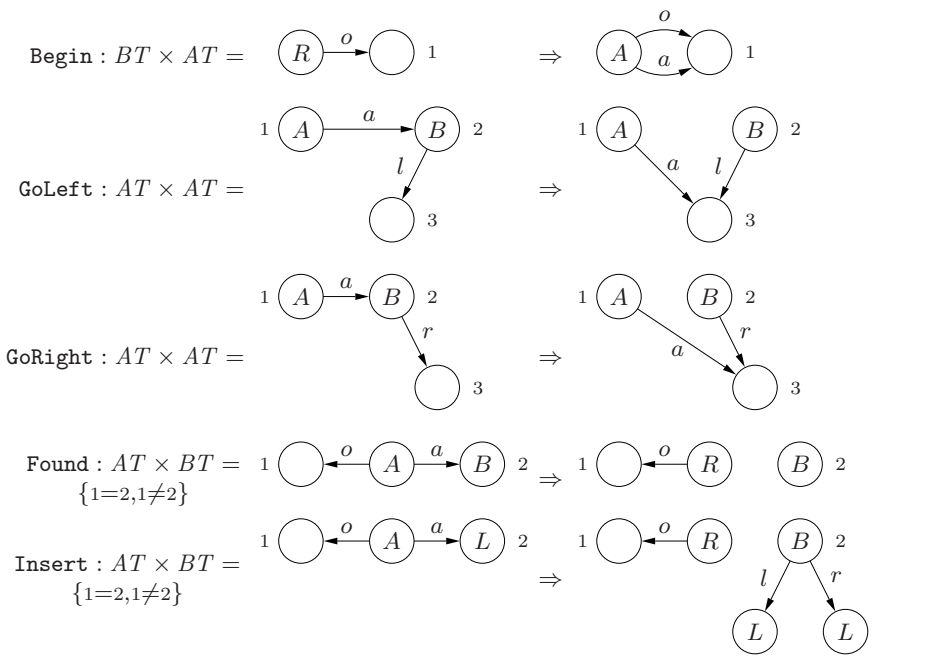

Fig. 3. Transformation rules modelling binary search tree insertion. In `Found` and `Insert`, nodes 1 and 2 may either be distinct or identical

tics of our program is the following binary relation $\rightarrow_{ins}$ on BT graphs, which represents every possible insertion of every possible element in every possible tree.

$$\rightarrow_{ins} = (\Rightarrow_{\{\texttt{Begin}\}}) \circ (\Rightarrow^*_{\{\texttt{GoLeft},\texttt{GoRight}\}}) \circ (\Rightarrow_{\{\texttt{Insert},\texttt{Found}\}})$$

A simple type-checker verifies that the declared range shape of `Begin` matches the domain shape of `GoLeft` and `GoRight` and that the declared range shape of `GoLeft` and `GoRight` matches their domain shape and the domain shape of `Insert` and `Found`. Shape checking aims to prove the individual rule shape annotations.

The `Begin` rule relabels the root A and introduces an auxiliary pointer a to the origin; this is a simple model of procedure call. Then, if the branch pointed to by a contains the datum to insert, the procedure should just return, removing a and relabelling the root back to R, which is done by `Found`. If a points to a leaf the datum is not present in the tree so a new branch should be allocated to hold it and the procedure should return, this is done by `Insert`. If a points to a branch and the datum to insert is less than the branch datum, a should move to insert in its left child, this is done by `GoLeft`; `GoRight` is similar.

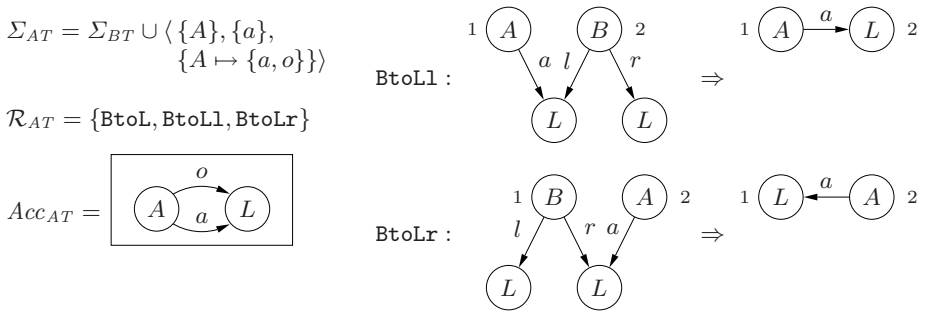

Fig. 4. PGRS of rooted binary trees with an auxiliary pointer, AT

A slightly simpler model of tree insertion is possible by giving the auxiliary pointer a separate parent from the origin (see [BPR03b]) but this version illustrates the ability of our method to check shape-changing rules.

4 Specifying Intermediate Shapes

The insertion rules of Fig. 3 temporarily violate the BT shape by relabelling the root and introducing an auxiliary pointer a. Temporary shape violation is essential in many pointer algorithms such as red-black tree insertion or in-place list reversal. Our approach to such algorithms is to define their intermediate shapes by PGRSs, annotate each of their rules with the intended shapes of their domain and range, and check these shape-changing rules.

Another approach would be to separate the heap-resident branch and leaf nodes from the stack-resident root nodes in our graphs. This way the tree insertion rules can be treated as non heap-shape changing but this approach would not allow the list reversal or red-black insertion examples to be treated as non-shape changing; in general, specifying intermediate shapes seems a better solution.

The rules in Fig. 3 are annotated with the intended shape of their domain and range: during the search phase the shape should be rooted binary trees where the root node is labelled A and has an auxiliary arc a pointing somewhere in the tree, in addition to the origin arc. This shape AT is specified formally by the PGRS in Fig. 4. The new rules `BtoLl` and `BtoLr` reduce branches if one of their children is the target of the auxiliary pointer a, which they move up the tree. Thus the smallest AT-graph, which is the accepting graph, has one leaf pointed to by both arcs of the root A.

5 Checking Shape Safety

An annotated rule $t : S \times T$ is *shape-safe* if for every derivation $G \Rightarrow_t H$, $G \in \mathcal{L}(S)$ implies $H \in \mathcal{L}(T)$.

Our checking algorithm builds an ARG (abstract reduction graph), which is a finite representation of graphs in $\mathcal{L}(S)$ and the domain of $\Rightarrow_t$ and rewrites this ARG to a *normal form* (Section 5.2). Then it builds another ARG to represent graphs in $\mathcal{L}(T)$ and the range of $\Rightarrow_t$ and tests whether this *includes* the left ARG (Section 5.3). To help represent infinite languages finitely, ARGs only include *basic contexts* (Section 5.1); the ARG inclusion test proves shape-safety for basic contexts; the *congeniality* test extends the safety result to all of $\mathcal{L}(S)$ in the domain of $\Rightarrow_t$ (Section 5.4).

The algorithm is necessarily incomplete because shape safety is undecidable in general. This follows from [FM97] which reduces the inclusion problem for context-free graph languages to a variant of shape safety. In practice, ARG construction may not terminate and if it does then some safe rules may fail the tests. The closedness property of PGRSs helps to improve ARG construction and the success rate of the algorithm.

5.1 Graph Contexts

An ARG represents all the contexts for the left (or right) graph of a rule. Intuitively, if $G \in \mathcal{L}(S) \cap dom(\Rightarrow_t)$ then G is the left graph of t glued into some graph context C. We denote this by $C(\!|L|\!)$. The ARG represents the set of all such contexts C as a kind of graph automaton: the derivation $C(\!|L|\!) \Rightarrow^* Acc_S$ can be broken down into a sequence of derivations $C_1(\!|L_1|\!) \Rightarrow L_2, \ldots, C_n(\!|L_n|\!) \Rightarrow Acc_S$ where C_i is the smallest context needed for the ith derivation to take place and C may be obtained by gluing all the C_i together. So the ARG is an automaton whose nodes are the L_i's of all such possible derivation sequences and whose arcs are labelled with the C_i necessary for the derivation from source to target.

There are two issues addressed in this section before the formal definition of ARGs: 1. All the represented contexts must be valid graphs; our definition of graph contexts ensures this. 2. The ARG must be finite; this is not always possible but it is often achievable by restricting the ARG to represent basic contexts only.

A *context* is a graph in which some nodes are internal. *Internal* nodes must be labelled and have a full set of outarcs, their inarcs cannot be extended when the context is extended. This restriction prevents ARGs representing invalid graphs, which otherwise would arise, for example if nodes allocated during a derivation are then glued into a context which must exist before those nodes.

Formally, a Σ-*context* is a pair $C = (G, I)$ where G is a Σ-graph and $I \subseteq V_G$ is a set of *internal* nodes such that $\forall v \in I. l_G(v) \neq \bot \wedge \{m_g(e) \mid s_G(e) = v\} = type\,(l_G(v))$. The *boundary* of C, $boundary(C) = V_G - I$, is all the external nodes of C. Equality, intersection, union and inclusion extend to contexts from graphs in the obvious way. A reduction rule is converted to a pair of contexts by the following function:
$ruletocxts(\langle L \supseteq K \subseteq R \rangle) = ((L, V_L - V_K), (R, V_R - V_K))$.

A *context morphism* $g : (G, I) \to (H, J)$ is a graph morphism $g : G \to H$ which preserves internal nodes and the indegree of internal nodes: $g_V I \subseteq$

J and $v \in I$ implies $indegree_G(v) = indegree_H(g(v))$ where $indegree_A(v) = \#\{e | t_A(e) = v\}$.

A *direct derivation* of context D from C through rule r and morphisms g, h is given by $C \Rightarrow_{r,g,h} D$ where $r = \langle L \supseteq K \subseteq R \rangle$ and the following diagram is two pushouts and h is a context isomorphism.

$$\begin{array}{ccccccc} (L, V_L - V_K) & \supseteq & (K, \emptyset) & \subseteq & (R, V_R - V_K) & & \\ g \downarrow & & \downarrow & & \downarrow & & \\ C & \supseteq & B & \subseteq & D' & \xrightarrow{h} & D \end{array}$$

A direct derivation preserves all the external nodes of C: only internal nodes can be deleted and allocated nodes are internal. $C(\!|D|\!)$ means *glue* D into C. It is defined if the following diagram is a pushout: the arrows are context inclusion morphisms; $C \cap D$ is discrete, unlabelled and all its nodes external; and D and $C(\!|D|\!)$ are contexts.

$$\begin{array}{ccc} C \cap D & \rightarrow & D \\ \downarrow & & \downarrow \\ C & \subseteq & C(\!|D|\!) \end{array}$$

The pushout construction means that $C(\!|D|\!)$ includes everything in C and D and nothing else (and only nodes internal in C or D are internal in $C(\!|D|\!)$); the context inclusions guarantee that every internal node of D has exactly the same inarcs and outarcs in $C(\!|D|\!)$; the restrictions on $C \cap D$ guarantee that C is the smallest context needed to form $C(\!|D|\!)$. Note that internal nodes of D cannot occur in C but external nodes of D can be made internal in $C(\!|D|\!)$ by being internal in C; and C does not have to be a proper context as its internal nodes can lack some of the inarcs or outarcs they have in $C(\!|D|\!)$. So $(\!|\,|\!)$ is associative but not commutative.

A useful property is that $(\!|\,|\!)$ cannot prevent reducibility by breaking the dangling condition, so if $C \Rightarrow D$ then $X(\!|C|\!) \Rightarrow X(\!|D|\!)$.

In a *basic direct derivation* of context C glued in context X, there must be a non-trivial overlap between C and X. Formally, $basic(X(\!|C|\!) \Rightarrow_{r,g,id} D)$ if $r = \langle L \supseteq K \subseteq R \rangle$; context X is minimal, $X(\!|C|\!) = gL \cup C$; the derivation exists, $X(\!|C|\!) \Rightarrow_{r,g,id} D$; and the overlap is non-trivial, $gL \cap C \neq gK \cap C$.

A non-basic derivation leaves C unchanged (but the reduction rule left graph may overlap some of C). Every derivation of the form $X(\!|C|\!) \Rightarrow^* Acc$ can be reordered and split into two consecutive derivations $X(\!|C|\!) \Rightarrow^* Y(\!|C|\!) \Rightarrow^* Acc$ where the direct derivations in the first sequence are all non-basic and those in the second sequence are all basic. The left ARG represents all such Y; the inclusion test checks that the transformation is safe for all graph contexts of the form $Y(\!|C|\!)$ and the shape congeniality test extends the result to all graphs of the form $X(\!|C|\!)$.

$ARG(C,S) = Build_{CXT}(D, \boxed{D}, S)$ *where* $D = reduce(C, \Rightarrow_S)$
$Build_{CXT}(C, A, S = \langle \Sigma, \mathcal{R}, Acc \rangle) =$
for $each(D, X) \in (\{(reduce(D, \Rightarrow_{\mathcal{R}}), X) \mid r \in \mathcal{R}, basic(X(\!|C|\!) \Rightarrow_{r,g,id} D)\}$
$\cup\{(X(\!|C|\!), X) \mid X(\!|C|\!) \cong Acc, C \ncong Acc\})/\cong$
do if $\exists C' \in V_A$, context isomorphism $i.D = iC'$ **then** $A := A \cup \boxed{C \xrightarrow{X} C'}$
else $Build_{CXT}(D, A \cup \boxed{C \xrightarrow{X} D}, S)$
return A
$reduce(C, \Rightarrow) =$ **if** $C \Rightarrow C'$ **then** $reduce(C', \Rightarrow)$ **else** C

Fig. 5. Context-based ARG construction algorithm

5.2 Abstract Reduction Graphs

An *ARG* $A = \langle V, E, m, s, t \rangle$ is a directed graph comprising a set of contexts V (the nodes); a set of arcs E; an arc labelling function $m : E \to Context$ and arc source and target functions $s, t : E \to V$.

For transformation rule $t : S \times T$, where $ruletocxts(t) = (C, D)$, the left ARG, $ARG(C, S)$ is produced by the algorithm in Fig. 5.

Intuitively, an ARG is built by starting with node C. For every context X (up to isomorphism, denoted $/\cong$ in Fig. 5) such that $X(\!|C|\!)$ has a basic derivation to D or $X(\!|C|\!)$ is Acc, we add an arc from C to node D, or node Acc, labelled X (the boxed expressions in Fig. 5 denote graphs pictorially). The process repeats on these new nodes. In general, $Build_{CXT}$ is non-terminating so safety checking often fails with certain GRSs. The report [Bak03] considers conditions for termination — a terminating, or size-reducing, GRS is not sufficient.

The closedness property of PGRSs allows us to *reduce* the contexts D — and the initial context C — before adding it to the ARG (and makes *reduce* deterministic). This reduces ARG size and improves the likelihood of $Build_{CXT}$ terminating.

Raw ARGs can be very large and they can include garbage paths that do not lead to the accepting graph. Therefore we use a system of *ARG normalisation* rules to eliminate excess nodes by merging and deleting arcs where possible. Fig. 6 shows an ARG: the left node is the context C; the right node is the accepting graphs of AT; positively numbered graph nodes are external; Unnumbered or negatively numbered graph nodes in ARG labels are internal. So for example, the top loop arc in this ARG means that gluing the central node into the loop label forms a context from which we can derive the central node (after renaming the boundary node 2 to 3).

Every path from C to Acc_S in $A = ARG(C, S)$ represents a context of C. The *context-paths* of A are $cxtpaths(A) = \{p \in paths(A) | s_p = C \wedge t_p = Acc_S\}$ where the *paths* in graph A are all sequences of arcs such that the target of each arc is the source of its successor in the sequence, $paths(G) = \{\langle e_i \rangle_{i=1}^{n} | e_i \in E_A \wedge 1 \leq$

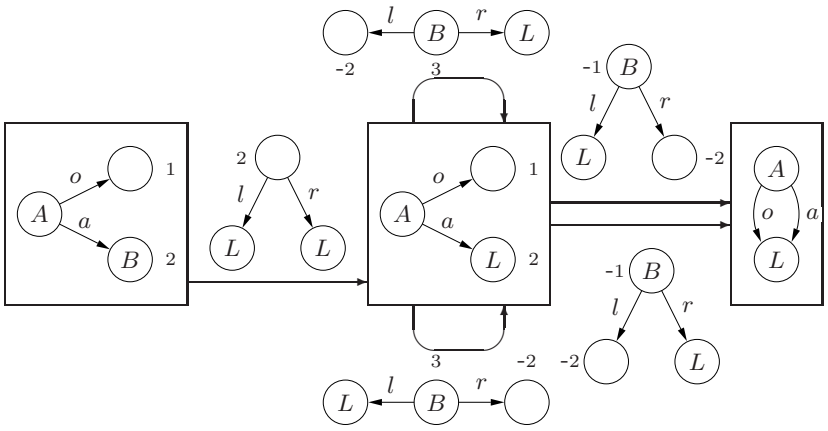

Fig. 6. Normalised ARG for the left hand side of `Found` (nodes 1 and 2 distinct)

$i < n \Rightarrow t_A(e_i) = s_A(e_{i+1})\}$; the *source* of a path $p = \langle e_i \rangle_{i=1}^{n}$ is $s_p = s_A(e_1)$ and the *target* of p is $t_p = t_A(e_n)$.

The *context represented* by a context-path p may be extracted by the following function which glues the arc labels together and uses a renaming morphism α to ensure that the right nodes are identified at each gluing.

$$
\begin{array}{l}
cof\,\langle\rangle = \emptyset \\
cof\,\langle C \xrightarrow{X,g} D\rangle + p = g\alpha_{C,X,g,D,P} P(\!|X|\!) \\
\text{where } D = reduce(X(\!|C|\!), \Rightarrow) \\
\qquad P = cof\,p \\
\qquad \alpha_{C,X,\sigma,D,Y} : (Rng(\sigma) - Dom(\sigma)) \cup (VE_Y - VE_D) \rightarrow \\
\qquad\qquad\qquad (VE - VE_{C \cup X \cup D}) - Rng(\sigma)
\end{array}
$$

The example ARG in Fig. 6 represents all the graphs depicted in Fig. 7: starting from an instance of the `Found` rule left graph we can reduce the branch pointed to by a to a leaf with one `BtoL` derivation then any number of `BtoLl` or `BtoLr` derivations (following the cycle in the centre of the ARG) move the a-arc up until it points to a child of the root branch; finally a `BtoLl` or `BtoLr` derivation at the root results in the accepting graph. So the basic contexts include the path from the a-arc up to the origin; the reduction of the other sub-trees to leaves are always part of the non-basic derivations.

The ARGs generated by our algorithm have the following properties. *ARG context-path completeness* says every basic context is represented by some context-path: if $G \cong X(\!|C|\!)$ and $G \Rightarrow_S^* Acc_S$ then there is a path $p \in cxtpaths(ARG(C,S))$.$G \Rightarrow_S^* cof p(\!|C|\!) \Rightarrow_S^* Acc_S$. *ARG context-path soundness* says every context-path represents some basic context: if $ruletocxts(r) = (C, D)$ and $G \Rightarrow^* cof p(\!|C|\!)$ and $p \in cxtpaths(ARG(C,S))$ then $G \Rightarrow_S^* Acc_S$ and $G \in dom(\Rightarrow_r)$.

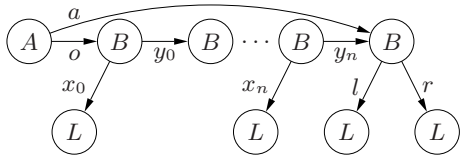

Fig. 7. Graphs represented by Fig. 6

5.3 ARG Inclusion

To check the shape-safety of a transformation for all basic contexts we can construct the ARG of its right graph then check that every context represented by the left ARG is represented by the right ARG. In practice this is undecidable so we ask whether the right ARG *includes* the left ARG. A more powerful algorithm uses the left ARG as a guide for the construction of the right ARG such that the construction succeeds if the inclusion exists.

Let $A = ARG(L, S)$ and $B = ARG(R, T)$. Then B *includes* A, $B \supseteq\!\!\sim A$, if there is:
1. A total injective morphism $m : VE_A \to VE_B$ such that (i). $m(L) = R$ and (ii). $m(Acc_S) = Acc_T$,
2. A total function $\beta : V_A \to (V \to V)$ assigning a node morphism to each node of A such that (i). $\beta(v) : boundary(v) \to boundary(m(v))$ is a total bijection and (ii). $\beta(L)$ is an identity,
3. A total function $\sigma : E_A \to (VE \to VE)$ assigning a graph isomorphism to each arc $e = C \xrightarrow{X} C'$ of A, where $m(e) = D \xrightarrow{Y} D'$, such that (i). $\sigma(e)X = Y$, (ii). $\sigma(e)|_C = \beta(C)$ and (iii). $\sigma(e)|_{C''} = h\alpha_h\beta(C')\alpha_g^{-1}g^{-1}|_{C''}$ where $reduce(X(\!|C|\!), \Rightarrow_S) = gC'$ and $reduce(Y(\!|D|\!), \Rightarrow_S) = hD'$ and $C'' = g\alpha_g C'$, $\alpha_g = \alpha_{C,X,g,C',C'}$ and $\alpha_h = \alpha_{D,Y,h,D',D'}$.

Fig. 8 shows a right ARG of the `Found` rule. This includes the left ARG in Fig. 6: there is an obvious isomorphism between these ARGs; the boundary nodes of corresponding contexts are the same; the context labels of corresponding arcs are the same (though their internal node and arc ids may differ). So all the contexts represented by the left ARG are represented by the right ARG too.

In general, we have the result that if $A = ARG(L, S)$ and $B = ARG(R, T)$ and $B \supseteq\!\!\sim A$ and $p \in cxtpaths(A)$ then there is a $q \in cxtpaths(B)$ such that $cof p(\!|L|\!) \cong cof q(\!|L|\!)$.

5.4 GRS Congeniality

To complete the shape-safety proof we need to extend the result from the basic contexts to all contexts. Formally we say that GRSs S and T are *congenial* for transformation t if whenever $X(\!|C|\!) \Rightarrow_S^* Y(\!|C|\!)$ then $X(\!|D|\!) \Rightarrow_T^* Y(\!|D|\!)$ where $(C, D) = ruletocxts(t)$ and no reduction in the $\Rightarrow_S^*$ sequence is basic.

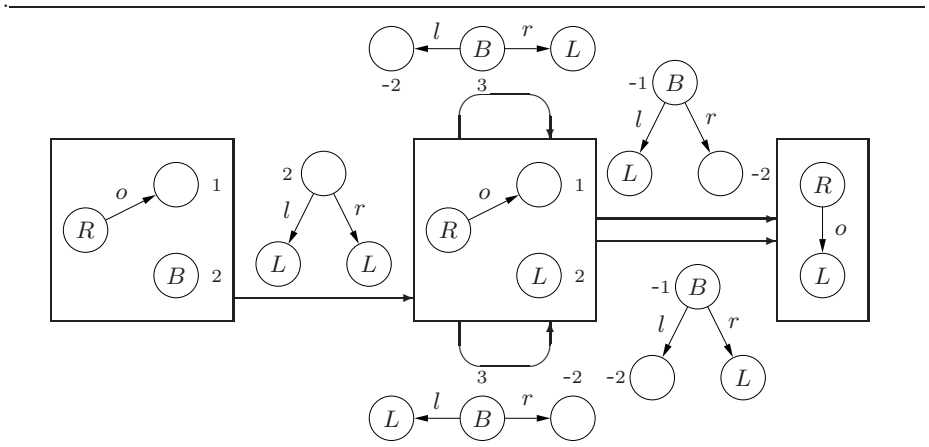

Fig. 8. Normalised ARG for the right hand side of `Found` (nodes 1 and 2 distinct)

The following principles are used to check congeniality. For every rule $r = \langle L \supseteq K \subseteq R \rangle \in \mathcal{R}_S$:
1. $r \in \mathcal{R}_T$ and K is discrete and unlabelled and every node in V_K is a source or target in L.
2. $r \notin \mathcal{R}_T$ and r can only be used in basic reductions of C.

These principles suffice for our example: in `Begin`, `GoLeft` and `GoRight` the rule range shape includes all the reduction rules of the domain shape, and they satisfy principle 1 above. So every reduction of any context to a basic context is still possible after application of these rules. The `Found` and `Insert` rules are more difficult because the `BtoLl` and `BtoLr` reduction rules are not part of BT so we need to use principle 2: as the `Found` and `Insert` left graphs both include the a-arc of the root node, every overlap with these reduction rules must be basic, otherwise the graph would have two roots and could not be in $\mathcal{L}(AT)$.

Thus all the insertion rules are shape safe and so insertion is shape safe.

6 Related Work and Conclusion

Our checking algorithm automatically proves the shape safety of operations such as search, insertion and deletion in cyclic lists, linked lists and binary search trees. It is more widely applicable than Fradet and Le Metayer's original algorithm [FM97, FM98] because our checking method is strictly more powerful and our ARGs are more precise through the use of graph contexts. In Fradet and Le Metayer's framework shape specifications are restricted to context-free graph grammars, this means that ARG construction is guaranteed to terminate and they do not need the congeniality condition (but checking is still undecidable). However their checking method is not applicable to shape changing rules such as the tree insertion algorithm here.

Context-exploiting shapes are generated by hyperedge-replacement rules that are extended with context [DHM03]. This paper shows that there is a reasonable (and decidable) class of *shaped transformation rules* that preserve context-exploiting shapes. Hence there is no need for type checking if shapes and transformations are restricted to these classes. It is unclear at present how severe this restriction is and how it compares to PGRSs with arbitrary transformations.

Other approaches to shape safety are more distant from our work as they are based on logics or types. The *logic of reachability expressions*, or *shape analysis*, [BRS99, SRW02] can be used for deciding properties of linked data structures expressed in a 3-valued logic. The PALE [MS01] tool can check specifications expressed in a *pointer assertion logic*, an extension of the earlier *graph types* specification method [KS93]. The logic of *bunched implications* [IO01] can be used as a language for describing and proving properties of pointer structures. *Alias Types* [WM01] are a pseudo-linear type system which accept shape safe programs on structures such as lists, trees and cycles. *Role analysis* [KLR02] checks programs annotated with role specifications which restrict the incoming and outgoing pointers of each record type and specify properties such as which pointer sequences form identities. Generally speaking, these approaches cannot or do not deal with context-sensitive shapes.

We can specify shapes beyond the reach of context-free graph grammars, such as red-black trees [BPR03b]. However, our current checking method is often non-terminating on such shapes. Essentially the problem occurs when a GRS has rules that can cause context D to be larger than context C in some basic derivation $X(\!|C|\!) \Rightarrow D$, and this growth is repeatable without limit, causing ARG construction to non-terminate. This situation often occurs with non-context-free GRSs such as balanced trees, preventing us from checking operations on such shapes.

The main areas for further work are to develop languages for safe pointer programming based on our existing specification and checking methods, and to overcome some of the limitations of the current approach to enable the automatic checking of operations on non-context-free shapes.

References

[Bak03] A Bakewell. Algorithms for checking the shape safety of graph transformation rules. Technical report, 2003. Available from [SPG]. 49, 56

[BPR03a] A Bakewell, D Plump, and C Runciman. Checking the shape safety of pointer manipulations — extended abstract. In *Participant's Proceedings of the 7th International Seminar on Relational Methods in Computer Science (RelMiCS 7), Malente, Germany*, pages 144–151. University of Kiel, 2003. Available from [SPG]. 49

[BPR03b] A Bakewell, D Plump, and C Runciman. Specifying pointer structures by graph reduction. Technical Report YCS-2003-367, Department of Computer Science, University of York, 2003. Available from [SPG]. 48, 49, 50, 51, 53, 60

[BRS99] M Benedikt, T Reps, and M Sagiv. A decidable logic for describing linked data structures. In *Proc. European Symposium on Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 2–19. Springer-Verlag, 1999. 60

[DHM03] F Drewes, B Hoffmann, and M Minas. Context-exploiting shapes for diagram transformation. *Machine Graphics and Vision*, 12(1):117–132, 2003. 60

[FM97] P Fradet and D Le Métayer. Shape types. In *Proc. Principles of Programming Languages (POPL '97)*, pages 27–39. ACM Press, 1997. 49, 54, 59

[FM98] P Fradet and D Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2–3):263–289, 1998. 49, 59

[HMP01] A Habel, J Müller, and D Plump. Double-pushout graph transformation revisited. *Math. Struct. in Comp. Science*, 11:637–688, 2001. 50

[HP01] A Habel and D Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *LNCS*, pages 230–245. Springer-Verlag, 2001. 51

[IO01] S Ishtiaq and P W O'Hearn. BI as an assertion language for mutable data structures. In *Proc. Principles of Programming Languages (POPL '01)*, pages 14–26. ACM Press, 2001. 60

[KLR02] V Kuncak, P Lam, and M Rinard. Role analysis. In *Proc. Principles of Programming Languages (POPL '02)*, pages 17–32. ACM Press, 2002. 60

[KS93] N Klarlund and M I Schwartzbach. Graph types. In *Proc. Principles of Programming Languages (POPL '93)*, pages 196–205. ACM Press, 1993. 60

[MS01] A Møller and M I Schwartzbach. The pointer assertion logic engine. In *Proc. ACM SIGPLAN '01 Conference on Programming Langauge Design and Implementation (PLDI '01)*. ACM Press, 2001. 60

[SPG] Safe Pointers by Graph Transformation, project webpage. `http://www-users.cs.york.ac.uk/~ajb/spgt/`. 48, 60

[SRW02] M Sagiv, T Reps, and R Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002. 60

[WM01] D Walker and G Morrisett. Alias types for recursive data structures. In *Types in Compilation (TIC '00), Selected Papers*, volume 2071 of *LNCS*, pages 177–206. Springer-Verlag, 2001. 60