Specifying Pointer Structures by Graph Reduction*

Adam Bakewell, Detlef Plump, and Colin Runciman

Department of Computer Science, University of York, York YO10 5DD, UK
{ajb,det,colin}@cs.york.ac.uk

Abstract. Graph reduction specifications (GRSs) are a powerful new method for specifying classes of pointer data structures (shapes). They cover important shapes, like various forms of balanced trees, that cannot be handled by existing methods.
This paper formally defines GRSs as graph reduction systems with a signature restriction and an accepting graph. We are mainly interested in PGRSs — polynomially-terminating GRSs whose graph languages are closed under reduction and have a polynomial membership test.
We investigate the power of the PGRS framework by presenting example specifications and by considering its language closure properties: PGRS languages are closed under intersection; not closed under union (unless we drop the closedness restriction and exclude languages with the empty graph); and not closed under complement.
Our practical investigation presents example PGRSs including cyclic lists, trees, balanced trees and red-black trees. In each case we try to make the PGRS as simple as possible where simpler means fewer rules, simpler termination and closure proofs and fewer non-terminals. We show how to prove the correctness of a PGRS and give methods for demonstrating that a given shape cannot be specified by a PGRS with certain simplicity properties.

1 Introduction

Pointer manipulation is notoriously dangerous in languages like C where there is nothing to prevent: the creation and dereferencing of dangling pointers; the dereferencing of nil pointers or structural changes that break the assumptions of a program, such as turning a list into a cycle.

Our goal is to improve the safety of pointer programs by providing (1) means for programmers to specify pointer data structure shapes, and (2) algorithms to check statically whether programs preserve the specified shapes. We approach these aims as follows.
1. Develop a formal notation for specifying shapes (languages of pointer data structures); that is the main concern of this paper. We show how shapes can be defined by graph reduction specifications (GRSs), which are the dual of graph grammars in that graphs in a language are reduced to an accepting graph rather

* Work partly funded by EPSRC project *Safe Pointers by Graph Transformation*[1].

J.L. Pfaltz, M. Nagl, and B. Böhlen (Eds.): AGTIVE 2003, LNCS 3062, pp. 30–44, 2004.
© Springer-Verlag Berlin Heidelberg 2004

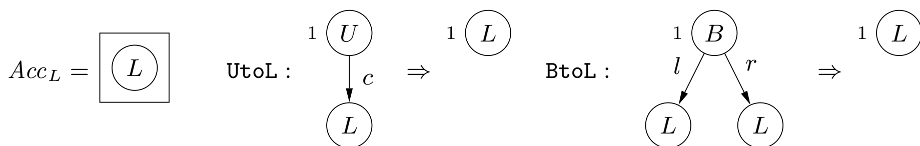

Fig. 1. A graph reduction specification of binary trees

than generated from a start graph. Polynomially terminating GRSs whose languages are closed under reduction (PGRSs) allow a simple and efficient membership test for individual structures, yet seem powerful enough to specify all common data structures.
2. The effect of a pointer algorithm on the shape of a data structure is captured by abstracting the algorithm to a graph rewrite system annotated with the intended structure shape at the start, end and intermediate points if needed. A static verifier then checks the shape annotations (see [3]).

Example 1 (Specifications of binary trees and full binary trees)
Fig. 1 gives a graph reduction specification of binary trees. The smallest binary tree is a leaf. We can draw it as Acc_L, the *accepting graph*, a single node labelled L. Trees may contain unary or binary branches. Therefore any other binary tree can be reduced to Acc_L by repeatedly applying the reduction rules `UtoL` and `BtoL`. These replace bottom-most branches, whose arcs point to leaves, by a leaf. The "1" indicates that any arcs pointing to the branch are left in place by the reduction rule. Full binary trees are specified by omitting the rule `UtoL` so that each node is either a leaf or a binary branch.

This reduction system only recognises trees because applying the inverse of its rules to any tree always produces a tree. Intuitively, forests cannot reduce to a single leaf as the rules do not break up graphs or connect broken graphs; no rule reduces a cycle; rules are matched *injectively* so `BtoL` cannot reduce a DAG with shared sub-trees; our signatures, introduced later, limit node outdegree so branches must be unary or binary. □

Graph reduction is a very powerful specification mechanism, we show how it can be used to define various kinds of *balanced* binary trees. Some shapes are harder to specify than others; we categorise shapes according to whether their PGRS needs non-terminal node labels; the difficulty of proving termination and closedness under reduction are also indicative of shape complexity. Some difficult languages can be specified as the union or intersection of simpler languages; we consider how the power of single PGRSs compares with such combinations.

Although many of our examples are trees, a *graph*-based specification framework is essential because we need precise control over the degree of sharing. Term rewriting ignores this issue and algebraic type specifications are unable to guarantee that members of tree data types are trees. Previous work on shape specifications uses variants of context-free graph grammars, or certain logics,

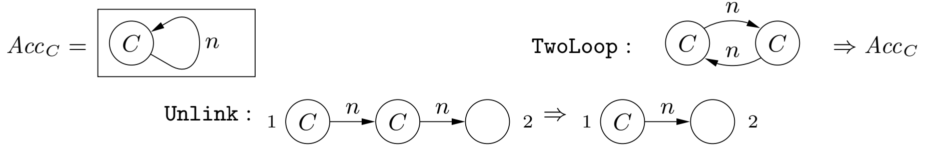

Fig. 2. A graph reduction specification of cyclic lists

which are unable to express properties like balance [12, 15, 5, 14, 8]. GRSs are far more powerful than the syntactic type restrictions expressible in languages like AGG, PROGRES and Fujaba. The suitability of general-purpose specification languages like OCL for specifying and checking shapes is unclear. PGRSs can define shapes with sharing and cycles. Our second example presents cyclic lists.

Example 2 (Specification of cyclic lists)
Fig. 2 gives rules defining cyclic lists. A single loop, Acc_C, is a cyclic list and all other cyclic lists reduce to Acc_C. Two-link cycles are reduced by `TwoLoop`. Longer cycles are reduced a link at a time by `Unlink`.

Clearly a graph of several disjoint cycles will not reduce to a single loop; no rules reduce branching or merging structures, and acyclic chains cannot become loops. □

The rest of this paper is organised as follows. Section 2 defines GRSs. Section 3 discusses polynomial GRSs (PGRSs) and their complexity for shape checking. Section 4 discusses power, showing when shapes are undefinable without non-terminals and demonstrating the closure properties of PGRS languages. Section 5 applies our theory to specify red-black trees. Section 6 discusses related work. Section 7 concludes. Proofs are omitted from this paper, they are given in the full technical report [2].

2 Graph Reduction Specifications

This section describes our framework for specifying graph languages by reduction systems. We define graphs, rules and derivations as in the double-pushout approach [10], and add a signature restriction to ensure that graphs are models of data structures and that rules preserve the restriction. The running example builds a specification of balanced binary trees (BBTs) — binary trees in which all paths from the root to a leaf have the same length.

Definition 1 (Signature)
A *signature* $\Sigma = \langle \mathscr{C}_V, \mathscr{C}_N, \mathscr{C}_E, type : \mathscr{C}_V \rightarrow \wp(\mathscr{C}_E) \rangle$ consists of a finite set of *vertex labels* $\mathscr{C}_V$, a set of *non-terminal* vertex labels $\mathscr{C}_N$ such that $\mathscr{C}_N \subseteq \mathscr{C}_V$, a finite set of *edge labels* $\mathscr{C}_E$ and a total function *type* assigning a set of edge labels to each vertex label. □

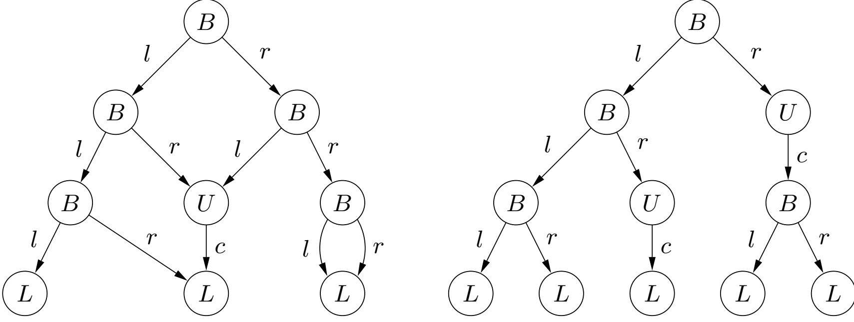

Fig. 3. Two Σ_{BT}-total graphs. The right one is a BBT, the left one is not

Intuitively, graph vertices represent tagged records. Their labels are the tags. Outgoing edges represent the record pointer fields of which each tag has a fixed selection defined by *type*. Edge labels in $\mathscr{C}_E$ correspond to the names of pointer fields. Non-terminal labels may occur in intermediate graphs during reduction but not in any graph representing a pointer structure. There is no need to restrict the permissible target node labels for edges in the signature because the reduction rules introduced below can encode any such restrictions. In the following, Σ always denotes an arbitrary but fixed signature $\langle \mathscr{C}_V, \mathscr{C}_N, \mathscr{C}_E, type \rangle$.

Example 3 (Binary tree signature)
Let $\Sigma_{BT} = \langle \{B, U, L\}, \{\}, \{l, r, c\}, \{B \mapsto \{l, r\}, U \mapsto \{c\}, L \mapsto \{\}\} \rangle$. Tree nodes are labelled B(inary branch), U(nary branch) or L(eaf). There are no non-terminals. Arcs are labelled l(eft), r(ight) or c(hild). Binary branches have left and right outgoing arcs, unary branches have a child and leaves have no arcs. □

Definitions 2, 3 and 4 below are consistent with the double-pushout approach to defining labelled graphs, morphisms, rules and derivations (see [10]; [11] considers graph relabelling). Fig. 3 shows two example graphs over Σ_{BT}.

Definition 2 (Graph)
A *graph* over Σ, $G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle$ consists of: a finite set of vertices V_G; a finite set of edges E_G; total functions $s_G, t_G : E_G \to V_G$ assigning a source and target vertex to each edge; a partial node labelling function $l_G : V_G \to \mathscr{C}_V$ (a partial function $f : A \to B$ maps $dom f$, a subset of A, to B. We write $f(x) = \perp$ when $x \notin dom f$); and a total edge labelling function $m_G : E_G \to \mathscr{C}_E$. □

Definition 3 (Morphism, inclusion and rule)
A *graph morphism* $g : G \to H$ consists of a node mapping $g_V : V_G \to V_H$ and an edge 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 $l_H(g_V(x)) = l_G(x)$

for all nodes x where $l_G(x) \neq \bot$ ($f(x) = \bot$ means f is undefined for x). 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 vertices and edges x in G. Note that inclusions may map unlabelled nodes to labelled nodes.

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 in $L - K$, preserves nodes in K and allocates nodes in $R - K$. In [10] rules may merge nodes but we have no need for this more general formulation here. Our pictures of rules show the left and right graphs; the interface graph is always just the set of numbered vertices common to left and right. For example, the interface of BtoL in Fig. 1 consists of the unlabelled node 1. So BtoL deletes two leaf nodes and two arcs, and preserves node 1 which is relabelled as a leaf.

Definition 4 (Direct derivation)
Graph G *directly derives* graph H through rule $r = \langle L \supseteq K \subseteq R \rangle$ and morphism g, written $G \Rightarrow H$, $G \Rightarrow_r H$ or $G \Rightarrow_{r,g} H$, if there is an injective graph morphism $g : L \to G$ such that: 1. no edge in $G - gL$ is incident to a node in $gL - gK$ (the *dangling condition*); 2. $H \cong H'$ where H' is constructed from G as follows: (i) remove all vertices and edges in $gL - gK$ (and restrict s_G, t_G, l_G and m_G accordingly) to obtain a subgraph D of G, (ii) add disjointly all vertices and edges (and their labels) in $R - K$ to D to form H': so there is another injective morphism $h : R \to H'$ with $h(R - K) \cap D = \emptyset$; if the source of an edge $e \in R - K$ is $x \in V_K$ then $s_{H'}(h(e))$ is $g(x)$ otherwise it is $h(x)$; similarly for targets; for every vertex $x \in V_K$ if $l_L(x) \neq l_R(x)$, the label of $g(x)$ in H' becomes $l_R(x)$. □

Injectivity of the matching morphism g means that `BtoL` in Fig. 1 is only applicable to a graph in which some B-labelled node has left and right arcs to distinct L-labelled nodes; the dangling condition means the L-labelled nodes must have no other in-arcs and the B-labelled node may have in-arcs.

If $H \cong G$ or H is derived from G by a sequence of direct derivations through rules in set $\mathscr{R}$ we write $G \Rightarrow^*_{\mathscr{R}} H$ or $G \Rightarrow^* H$. If no graph can be directly derived from G through a rule in $\mathscr{R}$ we say G is *$\mathscr{R}$-irreducible*. Definitions 2 and 3 are too general for modelling data structures because the outdegree of nodes is unlimited, and graphs and rules can disrespect the intentions of our signatures.

Example 4 (Unrestricted graph reduction is too general)
Fig. 4 shows a simple rule `Rel` which relabels a node, and an example derivation in which the relabelling results in a graph containing a leaf with a child. Un-

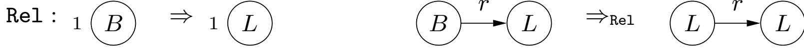

Fig. 4. A rule Rel, which does not respect the BT signature, and the effect of applying it to a graph which does respect the BT signature

restricted rules could make trees cyclic or give branches multiple left-children. This motivates the following restrictions. □

Definition 5 (Outlabels and Σ-graph)
The *outlabels* of node v in graph G are the set of labels of edges whose source is v: $outlabels_G(v) = \{m_G(e) \mid s_G(e) = v\}$.
A graph G *respects* Σ, or G is a Σ-graph for short, if: (1) $\forall e, e' \in E_G \cdot s_G(e) = s_G(e') \Rightarrow m_G(e) \neq m_G(e') \vee e = e'$ and (2) $\forall v \in V_G \cdot l_G(v) \neq \bot \Rightarrow outlabels_G(v) \subseteq type\,(l_G(v))$. Note the set of Σ-graphs is closed under subgraph selection. □

Every node has at most one outgoing edge with any given label, and the outlabels of a node labelled l form a subset of the type of l.

Definition 6 (Σ-total graphs)
A Σ-graph G is Σ-*total* if l_G is total and for every node $v \in V_G$, $outlabels_G(v) = type\,(l_G(v))$. □

A Σ-total graph models a data structure: all its nodes are labelled and each node has a full set of outlabels. Apart from these restrictions nodes may be connected to others in the same graph arbitrarily. In this paper we do not model nil pointers. Alternatives are considered in [2]. Non-total Σ-graphs are used in rules where it is essential, or convenient, to have unlabelled nodes and missing outlabels.

Example 5 (Σ_{BT} and Σ_{BT}-total graphs)
In the right half of Fig. 4, the left graph respects Σ_{BT} and the right graph does not. In Fig. 3 both graphs are Σ_{BT}-total. □

To prevent reduction rules breaking either the signature or the totality of graphs we define a simple restricted rule form: Σ-total rules.

Definition 7 (Σ-total rule)
A rule $\langle L \supseteq K \subseteq R\rangle$ is a Σ-*total rule* if L, R are Σ-graphs and for every node x:
1. $l_L(x) = \bot \Rightarrow x \in V_K \wedge l_R(x) = \bot \wedge outlabels_L(x) = outlabels_R(x)$.
That is, unlabelled nodes in L are preserved and remain unlabelled with the same outlabels.
2. $x \in V_K \wedge l_L(x) \neq \bot \wedge l_L(x) = l_R(x) \Rightarrow outlabels_L(x) = outlabels_R(x)$.
That is, labelled nodes in L which are preserved with the same label have the same outlabels in L and R.

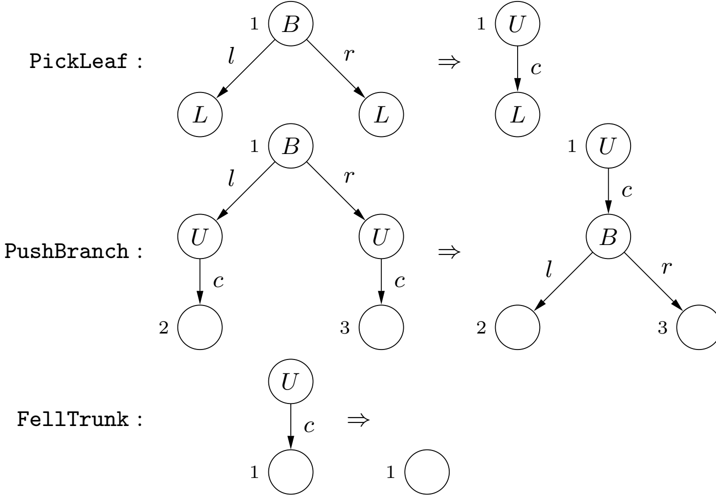

Fig. 5. BBT shape specification rules

3. $x \in V_K \wedge l_L(x) \neq \bot \wedge l_L(x) \neq l_R(x) \Rightarrow$
$l_R(x) \neq \bot \wedge \mathit{outlabels}_L(x) = \mathit{type}\,(l_L(x)) \wedge \mathit{outlabels}_R(x) = \mathit{type}\,(l_R(x))$.
That is, relabelled nodes have a complete set of outlabels in L and R. Nodes may not be labelled in L and unlabelled in R, or vice versa.
4. $x \in V_L - V_K \Rightarrow \mathit{outlabels}_L(x) = \mathit{type}\,(l_L(x))$.
That is, deleted nodes have a complete set of outlabels.
5. $x \in V_R - V_K \Rightarrow l_R(x) \neq \bot \wedge \mathit{outlabels}_R(x) = \mathit{type}\,(l_R(x))$.
That is, allocated nodes are labelled and have a complete set of outlabels. □

Example 6 (Rules specifying balanced binary trees)
Example 7 specifies BBTs with the Σ_{BT}-total rules $\mathscr{R}_{BBT} = \{$`PickLeaf`, `PushBranch`, `FellTrunk`$\}$, given in Fig. 5. `PickLeaf` replaces a binary branch of leaves by a unary branch of a leaf; `PushBranch` forces a binary branch of unary branches one level down, it applies anywhere in a tree. Note that both rules preserve height and balance. `FellTrunk` removes unary branches which are not the target of any arcs, it preserves balance but decreases height. □

Theorem 1 (Σ-total rules preserve Σ and Σ-totality)
Let r be a Σ-total rule and $G \Rightarrow_r H$ a direct derivation on graphs over Σ. Then G is a Σ-graph iff H is a Σ-graph. Moreover, G is Σ-total iff H is Σ-total. □

Definition 8 (GRS, NT-free GRS)
A *graph reduction specification* (GRS) $S = \langle \Sigma, \mathscr{R}, \mathit{Acc} \rangle$ consists of a signature Σ, a finite set of Σ-total rules $\mathscr{R}$ and an $\mathscr{R}$-irreducible Σ-total graph Acc, the

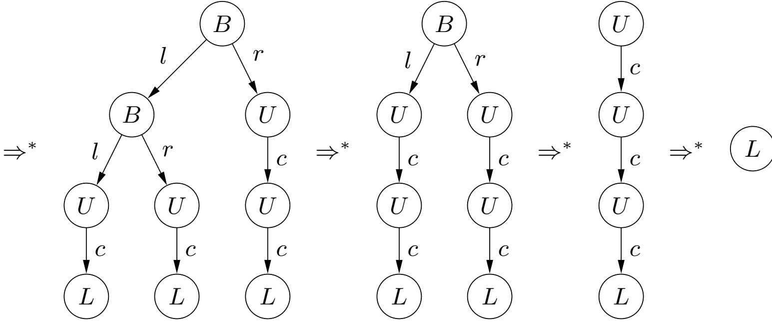

Fig. 6. A reduction of the right graph in Fig. 3. The steps in the four reduction sequences are: `PickLeaf`, `PickLeaf`; `PushBranch`, `PickLeaf`; `PushBranch`, `PushBranch`, `PickLeaf`; `FellTrunk`, `FellTrunk`, `FellTrunk`

---

accepting graph. The *graph language* of S is $\mathscr{L}(S) = \{G \mid G \Rightarrow^*_{\mathscr{R}} Acc \wedge l_G(V_G) \cap \mathscr{C}_N = \emptyset\}$. If $\mathscr{C}_N = \emptyset$ we say that S is *NT-free*. □

Termination and closedness are discussed in Section 3. Note that Acc is Σ-total, so every graph in $\mathscr{L}(S)$ is Σ-total by Theorem 1.

Example 7 (Specification of balanced binary trees)
We define BBTs by the NT-free GRS $BBT = \langle \Sigma_{BT}, \mathscr{R}_{BBT}, Acc_L \rangle$, where $\mathscr{R}_{BBT}$ is defined in Example 6. That is, $\mathscr{R}_{BBT}$ reduces BBTs, and nothing else, to Acc_L. Fig. 6 shows an example reduction. The left graph in Fig. 3 is irreducible under $\mathscr{R}_{BBT}$, owing to the various forms of sharing it contains, and therefore is not a BBT (it is a balanced binary DAG); the right graph is a BBT. □

Theorem 2 (*BBT* specifies balanced binary trees)
For every Σ_{BT}-graph G, $G \in \mathscr{L}(BBT)$ iff G is a balanced binary tree. □

3 Membership Checking

Graph reduction rules are just reversed graph-grammar production rules so reduction specifications can define every recursively enumerable set of Σ-total graphs (that exclude the empty graph, see [2]). This follows from Uesu's result that double-pushout graph grammars can generate every recursively enumerable set of graphs [18]. Consequentially, arbitrary reduction rules can specify languages with an undecidable membership problem.

For testing example structures we need specifications for which language membership can be checked — preferably in polynomial time. Therefore we will require that GRSs are polynomially terminating and their languages closed under reduction. Testing membership of such languages is simple: given a graph G,

check that G only has terminal labels and apply the rules in $\mathscr{R}$ (nondeterministically) as long as possible; G belongs to $\mathscr{L}(S)$ iff the resulting graph is isomorphic to Acc. First we consider termination.

Definition 9 (Graph size, polynomially terminating, size-reducing) Graph size is defined by $size(G) = \#V_G + \#E_G$ where $\#$ denotes set cardinality. A GRS $S = \langle \Sigma, \mathscr{R}, Acc \rangle$ is *terminating* if there is no infinite derivation $G_0 \Rightarrow_{\mathscr{R}} G_1 \Rightarrow_{\mathscr{R}} \cdots$. It is *polynomially terminating* if there is a polynomial p such that for every derivation $G \Rightarrow_{\mathscr{R}} G_1 \Rightarrow_{\mathscr{R}} \cdots \Rightarrow_{\mathscr{R}} G_n$, $n \leq p(size(G))$. It is *size-reducing* if $size(L) > size(R)$ for every rule $\langle L \supseteq K \subseteq R \rangle$ in $\mathscr{R}$. □

The example specifications in this paper have linear reduction lengths; this is usually easily shown, but there is no general decision method, so new GRSs may require individual termination analysis. For example, BBT is size-reducing, while RBT (Section 5) reduces the natural number $size(G) + \#\{v \mid l_G(v) = B\}$ at each step. Now we consider closedness and complexity.

Definition 10 (Closedness, Confluence, PGRS) A GRS $S = \langle \Sigma, \mathscr{R}, Acc \rangle$ is *closed* if for every direct derivation $G \Rightarrow_{\mathscr{R}} H$, $G \Rightarrow^*_{\mathscr{R}} Acc$ implies $H \Rightarrow^*_{\mathscr{R}} Acc$. S is *confluent* if for every pair of derivations $H_1 \,{}_{\mathscr{R}}^{*}\!\!\Leftarrow G \Rightarrow^*_{\mathscr{R}} H_2$ over Σ, there is a graph H such that $H_1 \Rightarrow^*_{\mathscr{R}} H \,{}_{\mathscr{R}}^{*}\!\!\Leftarrow H_2$. A polynomially terminating and closed GRS is a *polynomial* GRS (PGRS). □

Confluence implies closedness (the converse does not hold). Confluence of a terminating specification can be shown by adapting the *critical pair method* of [17] to GRSs (see [2]). All examples in this paper are confluent by this method as all their critical pairs are *strongly joinable*. Two reduction rules form a critical pair if they can be applied to the same graph such that one rule removes part of the graph required to apply the other rule. Closedness can be tested by disregarding any critical pair which only occurs as part of non-language member graphs.

Theorem 3 (Complexity of testing membership) If S is a PGRS then membership of $\mathscr{L}(S)$ is decidable in polynomial time. □

We assume S is fixed, so the number of rules is fixed and the size of the largest left graph in $\mathscr{R}$ is a constant c. Checking whether any rule in $\mathscr{R}$ matches a graph G requires $O(size(G)^c)$ time. This is because there are at most $size(G)^c$ injective mappings $V_L \to V_G$ for any left graph L, and checking whether a mapping induces a graph morphism $L \to G$ and the dangling condition can be done in constant time if graphs are suitably represented. Given a match, rule application is constant time. Hence the procedure sketched in the introduction to this section runs in polynomial time. The procedure is correct as the closedness of S makes backtracking unnecessary.

4 Extensions and Closure Properties

NT-free PGRSs are powerful but there are still lots of shapes they cannot describe; PGRSs are more powerful and GRSs have the universal specification power of graph grammars. This section develops the idea of classifying the simplicity of shapes by showing whether they have an NT-free specification or not. We show that: intersection extends the range of shapes definable by NT-free (P)GRSs to all the (P)GRS-definable shapes, and that (P)GRSs are closed under intersection; union extends the range of shapes definable by NT-free PGRSs and PGRSs, but terminating and possibly non-confluent GRSs are closed under union (provided $Acc \neq \emptyset$); complement extends the range of shapes definable by NT-free (P)GRSs and (P)GRSs.

Complete binary trees (CBTs) are BBTs where every branch is binary. Theorem 4 says they cannot be defined by an NT-free GRS. Lemma 1 presents a general method for showing that an NT-free GRS cannot define a given shape.

Lemma 1 (Proving graph languages are undefinable)
Graph language $\mathscr{L}$ cannot be defined by an NT-free GRS if:
$\forall k \in \mathbb{N}, \mathscr{R} \subseteq \mathscr{L} \times \mathscr{L} \cdot \max\{\delta(G,H) \mid (G,H) \in \mathscr{R}\} \geq k \vee \mathscr{R}^* \neq \mathscr{L} \times \mathscr{L}$
where $\delta(G,H) = \min\{\max\{\mathit{size}(L), \mathit{size}(R)\} \mid r = \langle L \supseteq K \subseteq R\rangle \wedge G \Rightarrow_r H\}$. □

To use Lemma 1 we show that for every k there is a graph $G \in \mathscr{L}$ which cannot be rewritten to some other graph $H \in \mathscr{L}$ without a rule of size at least k.

Theorem 4 (CBTs cannot be defined by an NT-free GRS)
No NT-free GRS can specify complete binary trees. □

We can often make a language specifiable by using non-terminals. Alternatively, we can take the intersection or union of two NT-free GRS languages. We show that using non-terminals is equivalent to using intersection and hence GRSs are closed under intersection. The following examples give non-terminal and intersection specifications of CBTs.

Example 8 (Specification of complete binary trees)
Let $CBT = \langle \Sigma_{BT} + \langle\{\},\{U\},\{\},\{\}\rangle, \mathscr{R}_{BBT}, Acc_B\rangle$. Hence CBTs are BBTs which do not contain any unary branches. □

Example 9 (CBTs by intersection)
Let $\mathscr{L}(CBT) = \mathscr{L}(FBT) \cap \mathscr{L}(BBT)$. CBTs are full binary trees (left conjunct, Example 1). CBTs are balanced (right conjunct). Both GRSs are NT-free. □

By Theorem 4 and Example 9, the languages of NT-free (P)GRSs are not closed under intersection. Theorem 5 shows that (P)GRSs and intersections of NT-free (P)GRSs have equivalent power. Theorem 6 shows that (P)GRSs are closed under intersection.

Theorem 5 (GRSs equivalent to intersections of NT-free GRSs)
1. If N is a GRS there are NT-free GRSs S and T s.t. $\mathscr{L}(N) = \mathscr{L}(S) \cap \mathscr{L}(T)$. Further, if N is a PGRS then so are S and T.
2. If S and T are NT-free GRSs there is a GRS N s.t. $\mathscr{L}(N) = \mathscr{L}(S) \cap \mathscr{L}(T)$. Further, if S and T are PGRSs then so is N. □

Theorem 6 (Graph reduction languages closed under intersection)
If S and T are (P)GRSs, then $\mathscr{L}(S) \cap \mathscr{L}(T)$ can be defined by a (P)GRS N. □

Language union offers another way to compose specifications. It is easy to see that union extends the range of languages specifiable by PGRSs and NT-free PGRSs. For example, a GRS cannot define a finite language that includes the empty graph and some other graph, but such a language is easily specified as a union of PGRSs with no reduction rules whose accepting graphs are the language elements. Similarly, PGRSs are not closed under union for infinite languages with or without the empty graph. If we allow terminating but possibly non-confluent GRSs, we can show that they are closed under union, provided their languages exclude the empty graph (see [2]).

GRS languages are not closed under complement. This follows from the ability of reduction specifications to simulate Chomsky grammars.

5 Red-Black Trees

This section applies the theory to specify red-black trees (RBTs). Our specification in Definition 12 is interesting because it is an NT-free PGRS but is not size-reducing. Theorem 7 says that a size-reducing RBT specification needs non-terminals (using a simplification of Lemma 1; see [2] for the proof and a size-reducing specification with non-terminals).

Definition 11 (Textbook red-black tree definition [6])
Red-black trees are trees of binary-branches and leaves where branches are labelled red or black, children of red branches are black or leaves and all paths from root to leaf have the same number of black nodes. □

Theorem 7 (A size-reducing GRS of RBTs needs non-terminals)
Red-black trees cannot be specified by a size-reducing NT-free GRS. □

Definition 12 (Specification of red-black trees)
Let $\Sigma_{RBT} = \langle \{R, B, L\}, \{\}, \{l, r\}, \{R \mapsto \{l, r\}, B \mapsto \{l, r\}, L \mapsto \{\}\} \rangle$ and $RBT = \langle \Sigma_{RBT}, \mathscr{R}_{RBT}, Acc_L \rangle$, where Fig. 7 shows the reduction rules in $\mathscr{R}_{RBT}$ and Fig. 1 shows Acc_L. □

Note that RBT is not size reducing but it linearly terminates as every reduction step reduces $size(G) + \#\{v \in V_G \mid l_G(v) = B\}$.

Theorem 8 (Correctness of RBT)
For every Σ_{RBT}-graph G, $G \in \mathscr{L}(RBT)$ if and only if G is a red-black tree. □

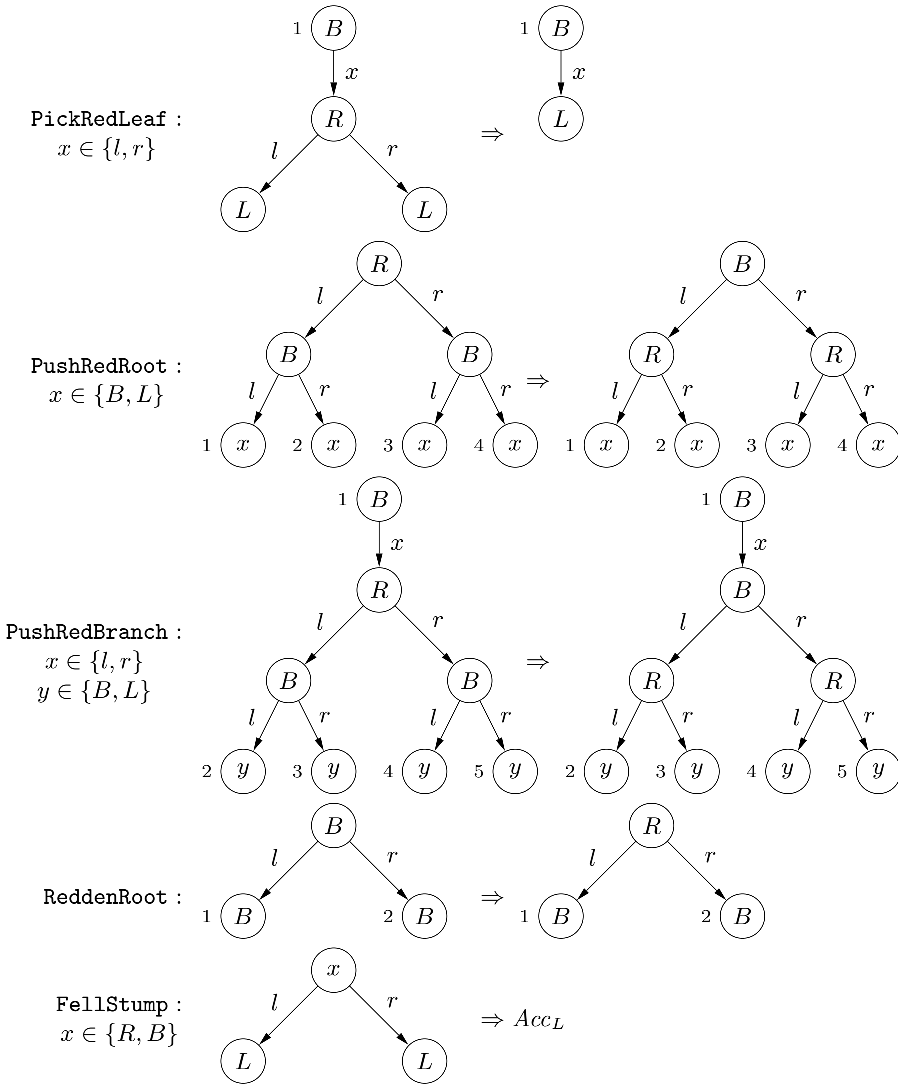

Fig. 7. Red-black tree reduction rules

Each rule preserves the red-black properties and produces either a smaller or a redder tree (therefore $\mathscr{R}_{RBT}$ terminates). The smallest RBT is a leaf. We can think of the tree reduction process as follows. `PickRedLeaf` can remove any red leaf-parent with a black parent. Any red node higher up the tree can be pushed by the tree by recolouring it and its children as in `PushRedRoot` or `PushRedBranch`, provided that its grandchildren are black or leaves. These rules

alone produce a complete black tree. The root can be coloured by `ReddenRoot`, safely reducing the black height, and then pushed down and picked by the other rules. Eventually we reach a singleton which is rewritten to Acc_L by `FellStump`.

6 Related Work

In functional programming, *Nested types* can be used to specify *perfect* binary trees [13]; however, these are only complete balanced binary DAGs as they do not preclude sub-tree sharing.

The following papers specify shapes using variants of context-free graph grammars, or certain logics. They can all specify trees, but none tackle the problem of specifying non-context-free properties like balance. ADDS [12] specifies structures by a number of dimensions where arcs are restricted to point away from, or towards, the root in a specified dimension. It can also limit node indegree. The *logic of reachability expressions* [5] allows the reachability, cyclicality and sharing properties of pointer variables to be specified as logical formulae. It is decidable whether a structure satisfies such a specification (but the complexity is unclear) and the logic is closed under intersection, union and complement. In *role analysis* [15] the shapes of pointer data structures are restricted by specifying whether pointers are on cyclic paths and by stating which pointer sequences form identities. The number and kind of incoming pointers are also specified. An algorithm verifies programs annotated with role specifications. *Graph types* [14] are recursive data types extended with routing expressions which allow the target of a pointer to be specified relative to its source. In [16], graph types are defined by monadic 2nd-order logic formulae and a *pointer assertion logic* is used to annotate C-like programs with partial correctness specifications; a tool checks that programs preserve their graph type invariants.

Shape types [8, 9] are specified by context-free graph grammars. They can always be converted to equivalent GRSs [2], but the classes of context-free graph languages and PGRS languages are incomparable. However, PGRSs can specify context-sensitive shapes and we are not aware of any common data structure with a context-free specification and no PGRS. Shape types have a method for checking the shape-invariance of atomic transformations (individual pointer manipulations). GRSs have a similar method which can verify the shape safety of an algorithm; more detailed explanations are available in [3, 1]. *Context-exploiting shapes* [7] are generated by hyperedge-replacement rules extended with context; the precise relation to PGRSs is unclear. Membership checking is exponential but there is a restricted, decidable class of *shaped transformation rules* that preserve context-exploiting shapes.

For related work on graph parsing see [4] which discusses context-free graph grammars and *layered graph grammars* which are powerful but have an exponential membership algorithm

7 Conclusion and Future Work

Graph reduction specifications are a powerful formal framework, capable of defining data structures with non-context-free properties. The examples presented here show how they can specify complete binary trees (Section 4) and red-black trees (Section 5). Many other examples, including AVL trees and grids, are given in [2]. The GRS tool available from [1] implements GRS checking including confluence, membership and operation checks.

We intend to develop programming languages which offer safe pointer manipulation based on GRSs. We are investigating two approaches.
1. *A new pointer programming paradigm.* Algorithms will be described as operations on graphs with data fields; the shapes of intermediate structures will be specified or inferred and checked. Checking is undecidable in general; we plan to investigate its feasibility on practical examples, the method described in [3] is a starting point. For operations like insertion into red-black trees [6] a better checker will be required, and possibly more informative specifications, because the current checker is often non-terminating on non-context-free shapes.
2. *An imperative programming language.* Combining conventional pointer manipulation with types specified by GRSs: pointer algorithms will be abstracted and then checked as in the first approach. Here the main challenge is to fit the operational semantics of a garbage-collected imperative language to the semantics of double-pushout graph rewriting.

References

[1] Safe Pointers by Graph Transformation, project webpage. `http://www-users.cs.york.ac.uk/~ajb/spgt/`. 30, 42, 43

[2] 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 [1]. 32, 35, 37, 38, 40, 42, 43

[3] A Bakewell, D Plump, and C Runciman. Checking the shape safety of pointer manipulations. In *Proc. 7th International Seminar on Relational Methods in Computer Science (RelMiCS 7)*, LNCS. Springer-Verlag, 2004. 31, 42, 43

[4] R Bardohl, M Minas, A Schürr, and G Taentzer. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter Application of Graph Transformation to Visual Languages, pages 105–180. World Scientific, 1999. 42

[5] 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. 32, 42

[6] T H Cormen, C E Leiserson, and R L Rivest. *Introduction to algorithms*. MIT Press, 1990. 40, 43

[7] F Drewes, B Hoffmann, and M Minas. Context-exploiting shapes for diagram transformation. *Machine Graphics and Vision*, 12(1):117–132, 2003. 42

[8] P Fradet and D Le Métayer. Shape types. In *Proc. Principles of Programming Languages (POPL '97)*, pages 27–39. ACM Press, 1997. 32, 42

[9] P Fradet and D Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2–3):263–289, 1998. 42

[10] A Habel, J Müller, and D Plump. Double-pushout graph transformation revisited. *Math. Struct. in Comp. Science*, 11:637–688, 2001. 32, 33, 34

[11] A Habel and D Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 135–147. Springer-Verlag, 2002. 33

[12] L J Hendren, J Hummel, and A Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proc. ACM SIGPLAN '92 Conference on Programming Langauge Design and Implementation (PLDI '92)*, pages 249–260. ACM Press, 1992. 32, 42

[13] R Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 10(3):305–317, May 2000. 42

[14] N Klarlund and M I Schwartzbach. Graph types. In *Proc. Principles of Programming Languages (POPL '93)*, pages 196–205. ACM Press, 1993. 32, 42

[15] V Kuncak, P Lam, and M Rinard. Role analysis. In *Proc. Principles of Programming Languages (POPL '02)*, pages 17–32. ACM Press, 2002. 32, 42

[16] 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. 42

[17] D Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In M R Sleep, M J Plasmeijer, and M C van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 15, pages 201–213. John Wiley & Sons Ltd, 1993. 38

[18] T Uesu. A system of graph grammars which generates all recursively enumerable sets of labelled graphs. *Tsukuba J. Math.*, 2:11–26, 1978. 37