Specifying Pointer Structures by Graph Reduction

ADAM BAKEWELL\textsuperscript{1} and DETLEF PLUMP\textsuperscript{1} and COLIN RUNCIMAN\textsuperscript{1}
\textsuperscript{1} Department of Computer Science, University of York, York YO10 5DD, UK.
\{ajb, det, colin\}@cs.york.ac.uk

Received 11 May 2004

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 article formally defines GRSs as graph reduction systems with a signature restriction and an accepting graph. We are mainly interested in PGRSs, which are polynomially-terminating GRSs whose graph languages are closed under reduction. PGRS languages have a polynomial membership test, making them a computationally well-behaved formalism for specifying graph languages.

We investigate the power of the PGRS framework by presenting example shapes within and beyond its scope and by considering its language closure properties under intersection, union and complement: 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.

We show how nil pointers can be modelled and present a wide variety of example PGRSs including lists, cyclic lists, trees, threaded trees, various balanced trees and grids. In each case we try to provide a simplest possible PGRS — ideally one with the fewest rules, the simplest possible termination and confluence proofs and the fewest 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);

\textsuperscript{1} Partly funded by the EPSRC project Safe Pointers by Graph Transformation, grant GR/R72440/01.
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 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. Section 7 includes an outline of our approach to this problem.

Example 1.1 (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 "i" 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 more difficult 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 by taking 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. In term rewriting it is not usual to specify whether multiple occurrences of the same sub-term are stored separately or shared, so it is not strictly possible to specify the tree shape where all subtrees of every branch must be disjoint. This remark also applies to algebraic type specifications. Previous work on shape specifications uses variants of context-free graph grammars, or certain logics, which are unable to express properties like balance (Hendren, Hummel & Nicolau 1992, Kuncak, Lam & Rinard 2002, Benedikt, Reps & Sagiv 1999, Klarlund & Schwartzbach 1993, Fradet & Métayer 1997). GRSs are far more powerful than the syntactic type restrictions expressible in languages like AGG, PROGRES and
Fujaba\textsuperscript{1}. The suitability of general-purpose specification languages like OCL (OCL 1997) for specifying and checking shapes is unclear.

We present GRSs as reduction systems with accepting graphs rather than production rule systems with a start graph. This emphasises the importance of efficient membership checking (graph parsing) for checking pointer structures and pointer operations. Of course there is no fundamental difference between production and reduction, but we wish to stress that restrictions common in many formulations of grammars do not apply: our accepting graph does not have to be a single non-terminal symbol and the right-hand sides of reduction rules do not have to be single non-terminals. So GRSs are like very flexible context-sensitive graph grammars but they can have slightly different properties. For example, we show later that languages including the empty graph can be specified by a terminating GRS with the empty graph as the accepting graph; if the accepting graph had to be some symbol such a language cannot have a terminating specification. PGRSs can also define shapes with sharing and cycles. The second introductory example presents cyclic lists.

Example 1.2 (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. \hfill \qed

The rest of the article 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 considers how nil pointer should be modelled with GRSs. Section 6 applies the theory to specify many example shapes. Section 7 discusses related work including verifying operations and other specification methods. Section 8 summarises the properties of the example GRSs and indicates how we intend to develop this approach.

2. Graph Reduction Specifications

This section describes our framework for specifying graph languages by reduction systems. Section 2.1 introduces the signature restriction we use to ensure that graphs are

\footnote{\url{http://www.gratra.org/}.}
models of data structures. Section 2.2 defines graphs, rules and derivations as in the double-pushout approach (Habel, Müller & Plump 2001). Section 2.3 presents restrictions used to guarantee that rules preserve the signature restriction. Section 2.4 presents the (P)GRS shape specification method. 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.

2.1. Signatures

All graphs are defined over some signature.

**Definition 2.1 (Signature).** A signature \( \Sigma = \langle C_V, C_N, C_E, \text{type} : C_V \rightarrow \varphi(C_E) \rangle \) consists of a finite set of node labels \( C_V \), a set of non-terminal node labels \( C_N \) such that \( C_N \subseteq C_V \), a finite set of arc labels \( C_E \) and a total function \( \text{type} \) assigning a set of arc labels to each node label.

Intuitively, graph nodes represent tagged records. Node labels are the tags. Outgoing arcs represent the record pointer fields of which each tag has a fixed selection defined by \( \text{type} \). The edge label alphabet \( C_E \) correspond to the names of pointer fields. Non-terminal node labels may occur in intermediate graphs during reduction but not in any graph representing a pointer structure. In the following, \( \Sigma \) always denotes an arbitrary but fixed signature \( \langle C_V, C_N, C_E, \text{type} \rangle \).

**Example 2.1 (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 \( B(\text{binary branch}), U(\text{unary branch}) \) or \( L(\text{leaf}). \) There are no non-terminals. Arcs are labelled \( l(\text{left}), r(\text{right}) \) or \( c(\text{child}). \) Binary branches have left and right outgoing arcs, unary branches have a child and leaves have no arcs.

2.2. Graph reduction

The following definitions are consistent with the double-pushout approach to defining labelled graphs, morphisms, rules and derivations (see (Habel et al. 2001); (Habel & Plump 2002) considers graph relabelling). Fig. 3 shows two example graphs over \( \Sigma_{BT} \).

**Definition 2.2 (Graph).** A graph over \( \Sigma, G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle \) consists of: a finite set of nodes \( V_G \); a finite set of arcs \( E_G \); total functions \( s_G, t_G : E_G \rightarrow V_G \) assigning a source and target node to each arc; a partial node labelling function \( l_G : V_G \rightarrow C_V \); and a total arc labelling function \( m_G : E_G \rightarrow C_E \).
Definition 2.3 (Morphism, application, inclusion and rule). A graph morphism $g : G \rightarrow H$ consists of a node mapping $g_V : V_G \rightarrow V_H$ and an arc mapping $g_E : E_G \rightarrow E_H$ 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 $l_G(x) \neq \bot^1$. 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 $g : G \rightarrow H$ to graph $G$ yields a graph $gG$ where: $V_{gG} = gV V_G$ (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 $g : G \rightarrow H$ such that $g(x) = x$ for all nodes and arcs $x$ in $G$. Note that inclusions may map unlabelled nodes to labelled nodes.

A rule $r = (L \supseteq K \subseteq R)$ consists of three graphs: the interface graph $K$ and the left and right graphs $L$ and $R$ which both include $K$. \hfill $\square$

Intuitively, a rule deletes nodes in $L - K$, preserves nodes in $K$ and allocates nodes in $R - K$. In (Habel et al. 2001) rules may merge nodes but we have no need for that more general formulation here. Our pictures of rules show the left and right graphs; the interface graph is always the set of numbered nodes 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 2.4 (Graph union and intersection). $G \cup H = C$ where $C \supseteq G$, $C \supseteq H$, and for every graph $D$ such that $D \supseteq G$ and $D \supseteq H$ we have $D \supseteq C$. $G \cap H = C$ where $C \subseteq G$, $C \subseteq H$, and for every graph $D$ such that $D \subseteq G$ and $D \subseteq H$ we have $D \subseteq C$. \hfill $\square$

The union of two graphs is the least-defined graph which includes both and the intersection of two graphs is the most-defined graph included by both.

Definition 2.5 (Direct derivation). Graph $G$ directly derives graph $H$ through rule $r = (L \supseteq K \subseteq R)$ and morphism $g$, written $G \Rightarrow H$, $G \Rightarrow_r H$ or $G \Rightarrow_{r,g} H$, if there is an injective graph morphism $g : L \rightarrow G$ such that: 1. no arc in $G - gL$ is incident to

\[ f(x) = \bot \] means $f$ is undefined for $x$. 

---

**Fig. 3.** Two $\Sigma_{BT}$-total graphs. The right one is a BBT, the left one is not.
a node in $gL - gK$ (the dangling condition); 2. $H \cong H'$ where $H'$ is constructed from $G$ as follows: (i) remove all nodes and arcs 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 nodes and arcs (and their labels) in $R - K$ to $D$ to form $H'$: so there is another injective morphism $h : R \rightarrow H'$ with $h(R - K) \cap D = \emptyset$; if the source of an arc $e \in R - K$ is $x \in V_K$ then $s_{H'}(h(e))$ is $g(x)$ otherwise it is $h(x)$; similarly for targets; for every node $x \in V_K$ where $l_L(x) \neq l_R(x)$, the label of $g(x)$ in $H'$ becomes $l_R(x)$.\hfill \Box

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 $\mathcal{R}$ 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.

2.3. Signature preservation

The graph and rule definitions 2.2 and 2.3 are too general for modelling data structures because the outdegree of nodes is unlimited, and the graphs and rules need not respect the intentions of our signatures.

Example 2.2 (Unrestricted graph reduction is too general). Fig. 4 shows a simple rule B1 which relabels a node, and an example derivation in which the relabelling results in a graph containing a leaf with a child. Such unrestricted rules could reduce cyclic “trees” or branches with multiple left-children.\hfill \Box

This motivates the following $\Sigma$-graph restriction.

Definition 2.6 (Outlabels and $\Sigma$-graph). The outlabels of node $v$ in graph $G$ are the set of labels of arcs whose source is $v$: outlabels$_G(v) = \{m_G(e) \mid s_G(e) = v\}$. A graph $G$ respects $\Sigma$, or $G$ is a $\Sigma$-graph for short, if: (1) $\forall e, e' \in E_G : s_G(e) = s_G(e') \Rightarrow m_G(e) \neq m_G(e') \vee e = e'$ and (2) $\forall v \in V_G : l_G(v) \neq \bot \Rightarrow$ outlabels$_G(v) \subseteq$ type ($l_G(v)$).\hfill \Box

Every node has at most one outgoing arc with any given label, and the outlabels of a node labelled $l$ form a subset of the type of $l$. Note the set of $\Sigma$-graphs is closed under subgraph selection.

Definition 2.7 ($\Sigma$-total graphs). A $\Sigma$-graph $G$ is $\Sigma$-total if $l_G$ is total and for every node $v \in V_G$, outlabels$_G(v) =$ type ($l_G(v)$).\hfill \Box

A $\Sigma$-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. Note that nil pointers must be modelled as pointers to nil node(s); Section 5 considers a variant which allows a more faithful model of nil pointers. Non-total $\Sigma$-graphs are used in rules where it is essential, or convenient, to have unlabelled nodes and missing outlabels. Languages where nodes need to have a variable
Specifying Pointer Structures by Graph Reduction

Fig. 4. A rule $\text{Rel}_1$, which does not respect the BT signature, and the effect of applying it to a graph which does respect the BT signature.

outdegree — series-parallel graphs, for example — can be modelled either by providing a series of node labels (for each desired outarc combination) or by using the correspondence between hypergraphs and bipartite graphs (where arcs are represented by nodes).

**Example 2.3 ($\Sigma_{BT}$ and $\Sigma_{BT}$-total graphs).** In the right half of Fig. 4, the left graph respects $\Sigma_{BT}$ and the right graph does not. In Fig. 3 both graphs are $\Sigma_{BT}$-total.

To prevent reduction rules breaking either the signature or the totality of graphs, we define $\Sigma$-total rules. This is proved in Theorem 2.1; all five conditions are necessary for this theorem, but conditions 4 and 5 are only needed for the preservation of $\Sigma$-totality.

**Definition 2.8 ($\Sigma$-total rule).** A rule $\langle L \supseteq K \subseteq R \rangle$ is a $\Sigma$-total rule if $L, R$ are $\Sigma$-graphs and for every node $x$:

1. $l_L(x) = \bot \Rightarrow x \in V_K \land l_R(x) = \bot \land outlabels_L(x) = outlabels_R(x)$. That is, unlabelled nodes in $L$ are preserved and remain unlabelled with the same outlabels.

2. $x \in V_K \land l_L(x) \neq \bot \land l_L(x) = l_R(x) \Rightarrow outlabels_L(x) = outlabels_R(x)$. That is, labelled nodes in $L$ which are preserved with the same label have the same outlabels in $L$ and $R$.

3. $x \in V_K \land l_R(x) \neq \bot \land l_L(x) \neq l_R(x) \Rightarrow$

   $l_R(x) \neq \bot \land outlabels_L(x) \supseteq type (l_L(x)) - type (l_R(x))$

   $\land outlabels_R(x) = (type (l_R(x)) - type (l_L(x))) \cup (outlabels_L(x) \cap type (l_R(x)))$. That is, for relabelled nodes $x$: all outlabels belonging only to $type (l_L(x))$ are removed; all outlabels belonging only to $type (l_R(x))$ are added; the outlabels common to both are preserved. Nodes may not be labelled in $L$ and unlabelled in $R$, or vice versa.

4. $x \in V_L - V_K \Rightarrow outlabels_L(x) = type (l_R(x))$. That is, deleted nodes have a complete set of outlabels.

5. $x \in V_R - V_K \Rightarrow l_R(x) \neq \bot \land outlabels_R(x) = type (l_R(x))$. That is, allocated nodes are labelled and have a complete set of outlabels. 

Note that to add or remove some outarc of a node, that node must be relabelled.

**Example 2.4 (Rules specifying balanced binary trees).** Example 2.5 specifies BBTs with the $\Sigma_{BT}$-total rules $\mathcal{R}_{BBT} = \{\text{PickLeaf}, \text{PushBranch}, \text{FellTrunk}\}$, 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 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 2.1 (\(\Sigma\)-total rules preserve \(\Sigma\) and \(\Sigma\)-totality). Let \(r\) be a \(\Sigma\)-total rule and \(G \Rightarrow_r H\) a direct derivation on graphs over \(\Sigma\). Then \(G\) is a \(\Sigma\)-graph iff \(H\) is a \(\Sigma\)-graph. Moreover, \(G\) is \(\Sigma\)-total iff \(H\) is \(\Sigma\)-total.

Proof. \(H\) is constructed as in Definition 2.5; so \(r = \langle L \supseteq K \subseteq R \rangle\), there is a morphism 
\[g : L \rightarrow G,\]
and there is an intermediate graph
\[D = \langle V_G - g(V_L - V_K), E_G - g(E_L - E_K), s_G, t_G, l_G, m_G \rangle.\]
Let \(R' = (g \cup h)R\) where morphism \(h\) comprises the injective mappings 
\[h_V : V_R \rightarrow V - V_D, \quad h_E : E_R \rightarrow E - E_D.\]
Then \(H \cong H'\) where \(H' = \langle V_D \cup V_R, E_D \cup E_R, s_D, t_D, l_D, m_D \rangle\).

Preservation of graph properties: (i) New nodes and arcs in \(H\) do not clash with those in \(D\) as they are added disjointly; (ii) \(s_H\) and \(t_H\) are total functions from \(E_H\) to \(V_H\) as \(s_D\) is total on \(E_D\) and new arcs are all assigned a source and target either in \(V_D\) or the new nodes; (iii) \(l_H\) is a partial function from \(E_H\) to \(C_H\) as \(l_D\) is partial, its restriction in \(H'\) is partial with a disjoint domain to \(V_R\) and \(V_R \subseteq V_H\) and \(R'\) is a \(\Sigma\)-graph; (iv) \(m_H\) is a total function from \(E_H\) to \(C_H\) as \(m_D\) and \(m_R\) are total.

Preservation of \(\Sigma\): (i) \(D\) is a \(\Sigma\)-graph because it is a subgraph of \(G\), if a node is removed then so are all its outarcs and inarcs, arc removal from preserved nodes preserves \(\Sigma\); (ii) Outlabels of allocated nodes respect \(\Sigma\) because \(R\) respects \(\Sigma\) and new nodes and arcs are added disjointly; (iii) For a preserved node \(v\), if it retains its label then we have \(\text{outlabels}_G(v) = \text{outlabels}_H(v)\), if it is relabelled then all of \(\text{outlabels}_G(v)\) are removed in \(D\) so \(\text{outlabels}_H(v) = \text{type}\ l_H(v)\).

Preservation of \(\Sigma\)-totality: (i) \(l_H\) is total as \(l_D\) is total, every preserved node labelled in \(L\) is labelled in \(R\) and nodes in \(R - K\) are labelled; (ii) \(\forall v \in V_H - \text{outlabels}_H(v) = \text{type}\ l_H(v)\) by proof of preservation of \(\Sigma\).

Reverse direction: if \(r\) is a \(\Sigma\)-total rule then so is \(\langle R \supseteq K \subseteq L \rangle\).
2.4. Shape specifications

GRSs are formally defined as reduction systems with a signature restriction and an accepting graph — essentially graph grammars. Nonterminal-free GRSs are formally simpler; their language members reduce in one step to another language member whereas with nonterminals much subtler language definition schemes are possible.

**Definition 2.9 (GRS, nonterminal-free GRS).** A graph reduction specification (GRS) $S = (\Sigma, R, Acc)$ consists of signature $\Sigma$, finite set of $\Sigma$-total rules $R$ and $R$-irreducible $\Sigma$-total graph $Acc$, the accepting graph. The graph language of $S$ is $L(S) = \{ G \mid G \Rightarrow^{*} R Acc \land l_{G}(V_{G}) \cap \mathcal{C}_{N} = \emptyset \}$. If $\mathcal{C}_{N} = \emptyset$ we say that $S$ is nonterminal-free. $\square$

Termination and closedness are discussed in Section 3. Note that $Acc$ is $\Sigma$-total, so every graph in $L(S)$ is $\Sigma$-total by Theorem 2.1.

**Example 2.5 (Specification of balanced binary trees).** We define BBTs by the nonterminal-free GRS $BBT = (\Sigma_{BBT}, R_{BBT}, Acc_{L})$, where $R_{BBT}$ is defined in Example 2.4. That is, $R_{BBT}$ reduces BBTs, and nothing else, to $Acc_{L}$. Fig. 6 shows an example reduction. The left graph in Fig. 3 is irreducible under $R_{BBT}$, 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. $\square$

**Theorem 2.2 (BBT specifies balanced binary trees).** For every $\Sigma_{BT}$-graph $G$, $G \in L(BBT)$ iff $G$ is a balanced binary tree.

**Proof.**

1. If $G$ reduces it is a BBT: $Acc_{B}$ is a BBT; Applying the inverse of a rule to any BBT results in a larger BBT.
2. If $G$ is a BBT it reduces: In outline, every non-$Acc_{B}$ BBT reduces to a smaller BBT; we show this in detail by induction, any BBT of height $n$ reduces to a chain of $n - 1$ $U$-nodes terminated by an $L$ using PickLeaf and PushBranch. Then $n - 1$ FellTrunk derivations reduce this chain to $Acc_{B}$.

The inductive proof: (i) BBT of height 1 is already a leaf; (ii) at height $n$ the subtree(s) reduce to chains by induction, then a $U$ branch is a chain or a $B$ branch of two chains of length $n - 1$ becomes a single chain of length $n$ by $n - 1$ PushBranch derivations and then one PickLeaf. $\square$

**Example 2.6 (Binary tree and cyclic list specifications).** We give formal specifications of the opening examples (Example 1.1 and Example 1.2): binary trees ($BT$), full binary trees ($FBT$) and cyclic lists ($CLIST$) as follows. Their formal proofs are simple (see Theorem 6.1).

$BT = (\Sigma_{BT}, \{ \text{UtoL}, \text{BtoL} \}, \text{Acc}_{L})$

$FBT = (\Sigma_{BBT}, \{ \text{BtoL} \}, \text{Acc}_{L})$

$CLIST = \langle \langle \{C\}, \{n\}, \{C \rightarrow \{n\}\} \rangle, \{ \text{TwoLoop}, \text{Unlink} \}, \text{Acc}_{C} \rangle$ $\square$

To understand the power of GRSs, first consider Chomsky grammars. A grammar $C = (V, N, P, S)$ has a finite set $V$ of symbols, a set $N \subseteq V$ of non-terminals, a finite set $P \subseteq V^{+} \times V^{*}$ of productions and a start symbol $S \in N$. It specifies the string language $L(C) = \{ w \in (V - N)^{*} \mid S \rightarrow^{*} P w \}$ where $\rightarrow P$ denotes the replacement of the left side of
a production by its right side in a string. By Lemma 2.1 it follows that GRSs can specify every recursively enumerable set of strings and so GRSs can specify graph languages with undecidable membership problems.

**Lemma 2.1 (Simulation of Chomsky grammars).** For every Chomsky grammar $C$ there is a GRS $S$ s.t. $L(C) = L(S)$.

**Proof.** Let $\Sigma = \langle V \cup \{\bullet\}, N, \{n\}, \bullet \mapsto 0 \cup \{v \mapsto \{n\} | v \in V \rangle$. A string graph is a $\Sigma$-total graph of the form:

$$
\begin{array}{ccc}
  x_1 & \overset{n}{\longrightarrow} & x_2 \\
  \hspace{1cm} & \cdots \hspace{1cm} & \cdots \hspace{1cm} \\
  \vdots & \ddots & \ddots \\
  x_i & \overset{n}{\longrightarrow} & \bullet
\end{array}
$$

where $i \geq 0$ and $\{x_1, \ldots, x_i\} \in V$. For every $w \in V^*$, $w^*$ denotes the string graph of $w$. For each production $p = y_1 \ldots y_j \Rightarrow x_1 \ldots x_i$ in $P$, let $p^*$ be the following rule.

$${\begin{array}{ccc}
  1 & \overset{x_1}{\longrightarrow} & 2 \\
  \vdots & \ddots & \ddots \\
  1 & \overset{x_i}{\longrightarrow} & 2
\end{array}} \Rightarrow
{\begin{array}{ccc}
  1 & \overset{y_1}{\longrightarrow} & 2 \\
  \vdots & \ddots & \ddots \\
  1 & \overset{y_j}{\longrightarrow} & 2
\end{array}}$$

Consider the reduction specification $C^* = \langle \Sigma, \{p^* | p \in P\}, S^* \rangle$. By construction, for all $u, v \in V^*$, $u \Rightarrow_p v$ iff $v^* \Rightarrow_p^* u^*$. Hence $L(C^*) = \{w^* | w \in L(C)\}$. If $S^*$ is reducible we can add a new non-terminal $N$ as the accepting graph and a reduction rule to rewrite $S^*$ to the accepting graph. So $C^*$ is a GRS.

More generally, graph reduction rules are just reversed graph-grammar production rules. So from Uesu’s result that double-pushout graph grammars can generate every recursively enumerable set of graphs (Uesu 1978), it follows that GRSs can define every recursively enumerable set of $\Sigma$-total graphs (that exclude the empty graph — see Lemma 4.2 later).

3. Membership Checking

For testing example structures we need specifications for which language membership can be checked — preferably in polynomial time. Therefore we wish to disallow the GRSs with
undecidable membership problems. For practical specifications we require polynomially terminating GRSs whose languages are closed under reduction. Then testing membership is simple: given a graph $G$, check that $G$ only has terminal labels and apply the rules in $R$ (nondeterministically) as long as possible; $G$ belongs to $\mathcal{L}(S)$ iff the resulting graph is isomorphic to $\text{Acc}$. Section 3.1 considers termination in more detail and Section 3.2 considers confluence and the complexity of testing membership in more detail.

### 3.1. Termination

**Definition 3.1 (Graph size, polynomially terminating, size-reducing).** Graph size is defined by $\text{size}(G) = \#V_G + \#E_G$ where $\#$ denotes set cardinality. A GRS $S = \langle \Sigma, R, \text{Acc} \rangle$ is terminating if there is no infinite derivation $G_0 \Rightarrow_R G_1 \Rightarrow_R \cdots$. It is polynomially terminating if there is a polynomial $p$ such that for every derivation $G \Rightarrow_R G_1 \Rightarrow_R \cdots \Rightarrow_R G_n$, $n \leq p(\text{size}(G))$. It is size-reducing if $\text{size}(L) > \text{size}(R)$ for every rule $(L \geq K \subseteq R)$ in $R$.

Our example specifications mostly have linear reduction lengths. For example, $BBT$ is size-reducing and $RBT$ (Section 6.4.1) reduces the natural number $\text{size}(G) + \#\{v \mid l_G(v) = B\}$ at each step. The following example presents a GRS with slightly less obvious linear termination.

**Example 3.1 (Binary DAGs).** Binary DAGs can be specified by giving reduction rules to convert them to trees (see Fig. 7) in combination with the normal full binary tree reduction rule: $BDAG = \langle \Sigma_{BT}, \{\text{BtoL}, \text{DagL1}, \text{DagL2}\}, \text{Acc}_L \rangle$.

**Theorem 3.1 (Linear termination of BDAG).** $\mathcal{R}_{BDAG}$ is linearly terminating.

**Proof.** Each $\text{DagL}$ rule in $\mathcal{R}_{BDAG}$ reduces the amount of sharing in a graph $G$, that is, $\Sigma\{\text{indegree } v \mid v \in V_G\}$ where $\text{indegree } v = \#\{e \mid l_G(e) = v\}$. $\text{BtoL}$ preserves the amount of sharing and reduces the number of nodes. The amount of sharing is bounded by $\#E_G$, therefore $\mathcal{R}_{BDAG}$ is linearly terminating.

If we allowed unsharing of branches as well as leaves by including $\text{DagB1}$ and $\text{DagB2}$ of Fig. 7 in $\mathcal{R}_{BDAG}$ then the reduction of a DAG graph could become exponential. For example, a linear chain of $n$ branch nodes whose left and right children are the next node in the chain could be expanded into a full binary tree of depth $n$ before being reduced as a tree. Worse, a cyclic graph could expand without limit using the $\text{DagB2}$ rule.

To summarise, linear termination may easily be demonstrated by size reduction, reduction in an ordering on node labels or reduction of node indegree. But no general decision method exists so new GRSs may need a new termination analysis.

### 3.2. Closedness, confluence and complexity

A closed GRS cannot reduce a language member to a non-member; this is a special case of confluence. The class of GRSs we propose for practical shape specifications are PGRSs, which guarantee a polynomial membership test by Theorem 3.2.
\textbf{Definition 3.2 (Closedness, Confluence, PGRS).} A GRS $S = (\Sigma, R, \text{Acc})$ is closed if for every direct derivation $G \Rightarrow_R H$, $G \Rightarrow_R^* \text{Acc}$ implies $H \Rightarrow_R^* \text{Acc}$. $S$ is confluent if for every pair of derivations $H_1 \Rightarrow_R^* H_2$, $H_2 \Rightarrow_R^* H_3$ over $\Sigma$, there is a graph $H$ such that $H_1 \Rightarrow_R^* H \Rightarrow_R^* H_2$ over $\Sigma$, there is a graph $H$ such that $H_1 \Rightarrow_R^* H \Rightarrow_R^* H_2$. A polynomially terminating and closed GRS is a polynomial GRS, PGRS for short.

\textbf{Theorem 3.2 (Complexity of testing membership).} If $S$ is a PGRS then membership of $L(S)$ is decidable in polynomial time.

\textit{Proof.} We assume $S$ is fixed, so the number of rules is fixed and the size of the largest left graph in $R$ is a constant $c$. Checking whether any rule in $R$ matches a graph $G$ requires $O(size(G)^c)$ time. This is because there are at most $size(G)^c$ injective mappings $V_L \rightarrow V_G$ for any left graph $L$, and checking whether a mapping induces a graph morphism $L \rightarrow G$ 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 backtracking is unnecessary for closed $S$.

To show confluence we extend the critical pair method of (Plump 1993) to labelled systems. Two reduction rules form a $\Sigma$-critical pair if they can be applied to the same $\Sigma$-graph in such a way that one rule removes part of the graph required to apply the other rule (we only need to consider $\Sigma$-graphs, unlike the definition in (Plump 1993)).

\textbf{Definition 3.3 (\Sigma-critical pair).} Let $r_i = (L_i \supseteq K_i \subseteq R_i)$ be $\Sigma$-rules for $i = 1, 2$. A pair of direct derivations $T_{r_1, r_2} S \Rightarrow u \Rightarrow_{r_2, r_2} U$ is a $\Sigma$-critical pair if $S = g_1L_1 \cup g_2L_2$ is a $\Sigma$-graph and $g_1L_1 \cap g_2L_2 \neq g_1K_1 \cap g_2K_2$. Furthermore, if $r_1 = r_2$ then $g_1 \neq g_2$.

Critical pairs are not distinguished if the only difference is in the naming of nodes or arcs. If a GRS has no critical pairs it is strongly confluent. The following example illustrates how critical pairs may arise.
Example 3.2 (Critical pair). Fig. 8 shows a harmless additional reduction rule \texttt{Unlink2} that could be used for the reduction of cyclic lists. But its addition to \( R_{\text{CLIST}} \) gives rise to the critical pair shown: a chain of four nodes can now be reduced directly to two nodes or just to three nodes by \texttt{Unlink}.

If there are critical pairs the following lemma may be used to show confluence of a terminating reduction system. This is only a sufficient test as confluence is undecidable in general (Plump 1993).

**Definition 3.4 (Strongly joinable critical pair).** A \( \Sigma \)-critical pair \( T \triangleleft S \Rightarrow U \) (where \( S, T, U \) are \( \Sigma \)-graphs but may be not total) is strongly joinable if there are derivations \( T \Rightarrow \ast X \ast \Rightarrow U \) such that for all nodes \( v \) in \( \text{Protect}(S) \), \( \text{track}_{S \Rightarrow T}(v) \) and \( \text{track}_{S \Rightarrow U}(v) \) are defined and equal.

\( \text{Protect}(S) \) is the set of all nodes \( v \in V_S \) for which \( \text{track}_{S \Rightarrow T}(v) \) and \( \text{track}_{S \Rightarrow U}(v) \) are defined.

The \( \text{track} \) function maps a node to \( \perp \) if it is deleted during a derivation, otherwise it follows a node through a derivation. Let the derivation \( G \Rightarrow H \) be constructed as in Definition 2.5. Then \( \text{track}_{G \Rightarrow H}(v) = c(v) \), where \( c : H' \Rightarrow H \) if \( v \in V_D; \perp, \) otherwise. This extends to derivation sequences as follows: \( \text{track}_{G \Rightarrow \ast} = i \) if \( G \Rightarrow \ast H \) by an isomorphism \( i \); \( \text{track}_{G \Rightarrow \ast}(v) = \text{track}_{H' \Rightarrow \ast} \circ \text{track}_{G \Rightarrow H'} \) if \( G \Rightarrow \ast H' \Rightarrow H \) by a sequence \( G \Rightarrow \ast H' \Rightarrow H \).

**Lemma 3.1 (Critical pair lemma).** A reduction system is locally confluent if all its \( \Sigma \)-critical pairs are strongly joinable.

**Proof.** As in (Plump 1993) but with reduction of labelled graphs.

Example 3.3 (Showing confluence). The \( \Sigma \)-critical pairs of \texttt{Unlink} and \texttt{Unlink2} are all strongly joinable. For the example in Fig. 8, the right graph is derived from one application of \texttt{Unlink} to the lower graph. The \texttt{Protect} nodes are 1 and 2 which are preserved by both derivation sequences.

Example 3.4 (Non-strongly-joinable critical pair). Although intuitively, the \texttt{BDAG} specification in Fig. 7 is confluent, it is beyond the scope of the critical pair lemma to show this. Fig. 9 shows a \( \Sigma_{BT} \)-critical pair of \texttt{DagL2} and \texttt{DagL2}: the left and right derived graphs are both irreducible, they are isomorphic but the \( \text{track} \) function maps node 3 in the left derivation to the unnumbered leaf in the right derivation. Therefore the critical pair is not strongly joinable.
Several of our examples have Σ-critical pairs, and some of these are not strongly joinable. But we conjecture that all our examples are closed. Closedness can be tested by disregarding any Σ-critical pair which only occurs as part of a non-language member graph; no formal method is given in this paper.

4. Closure properties of GRSs

Nonterminal-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 can be specified as nonterminal-free (P)GRSs or not. From this we develop results about how allowing intersection, union or complement of specifications affects their power.

Section 4.1 shows that intersection extends the range of shapes definable by nonterminal-free (P)GRSs to all the (P)GRS-definable shapes, and that GRSs are closed under intersection. Section 4.2 shows that union extends the range of shapes definable by nonterminal-free (P)GRSs and (P)GRSs, but terminating and possibly non-confluent GRSs are closed under union (provided Acc ≠ ∅). Section 4.3 shows that complement extends the range of shapes definable by nonterminal-free (P)GRSs and (P)GRSs.

4.1. Intersection

To show that intersection of nonterminal-free GRSs define more languages, consider complete binary trees (CBTs) — these are BBTs where every branch is binary. Theorem 4.1 says they cannot be defined by a nonterminal-free GRS. It uses Lemma 4.1 which presents a general method for showing that an nonterminal-free GRS cannot define a given shape.

Lemma 4.1 (Proving graph languages are undefinable). Graph language L cannot be defined by a nonterminal-free GRS if:

∀k ∈ N, R ⊆ L × L · max{δ(G, H) | (G, H) ∈ R} ≥ k ∨ ∀G ∈ L · L × {G} ⊃ R*, that is, for any relation R (that we might try to use as a reduction relation defining L), either:

(1) to define R needs a rule bigger than any k, so it cannot be defined by a GRS (where the graph rewrite rule size function rsize(G, H) = min{max{size(L), size(R)} | r = (L ⊃ K ⊆ R), G ⇒ r H} is the size of the smallest rule that can rewrite G to H); Or (2) the transitive closure of R cannot be a reduction relation defining all of L.
Specifying Pointer Structures by Graph Reduction

Proof. To be definable by a nonterminal-free GRS, there must be a finite set of rewrite rules defining a relation \( \mathcal{R} \) such that every graph \( G' \in \mathcal{L} \) is related to some graph \( G \in \mathcal{L} \) by the reflexive-transitive closure of \( \mathcal{R} \) (condition 2). That GRSs are defined by a finite number of rules means there is some bound \( k \) on the size of rules defining \( \mathcal{R} \) (condition 1).

Conversely, if there is a finite set of rules covering \( \mathcal{L} \), there may not be a nonterminal-free GRS defining \( \mathcal{L} \). For example, some context-sensitive properties cannot be expressed without the use of intermediate states which are not in \( \mathcal{L} \) as in the strings defined by \( B^*(AB^n)^* \).

To use Lemma 4.1 we show that for every \( k \) there is a graph \( G \in \mathcal{L} \) which cannot be rewritten to some smaller or larger graph \( H \in \mathcal{L} \) without using a rule of size at least \( k \).

Theorem 4.1 (CBTs cannot be defined by a nonterminal-free GRS). No nonterminal-free GRS can specify complete binary trees.

Proof. By Lemma 4.1: Let \( G \) be a CBT of depth \( k \). Every smaller CBT is at least \( 2^{(k-1)} \) nodes smaller; every larger CBT is at least \( 2^k \) nodes larger.

We can often make a language specifiable by using non-terminals. Alternatively, we can take the intersection of two nonterminal-free GRS languages. This section shows that using non-terminals is equivalent to using intersection and hence (P)GRSs are closed under intersection. First, a non-terminal PGRS of CBTs.

Example 4.1 (Specification of complete binary trees). Let \( CBT = \langle \Sigma_{BT} + \{\}, \{\}, \mathcal{L}_{BT}, \mathcal{L}_{CBT} \rangle \) where \( \mathcal{L}_{BT} \) is given in Fig. 5. Hence CBTs are BBTs which do not contain any unary branches.

The graph language of the intersection of two GRSs is simply the intersection of their languages, so we can define CBTs in terms of previously defined GRSs.

Example 4.2 (CBTs by intersection). Let \( \mathcal{L}(CBT) = \mathcal{L}(FBT) \cap \mathcal{L}(BBT) \). CBTs are full binary trees (left conjunct) and they are balanced (right conjunct). Note that both GRSs are nonterminal-free.

By Theorem 4.1 and Example 4.2, the languages of nonterminal-free (P)GRSs are not closed under intersection: FBTs and BBTs have NT-free GRSs but their intersection CBTs does not. Theorem 4.2 shows that (P)GRSs and intersections of nonterminal-free (P)GRSs have equivalent power and (P)GRSs are closed under intersection.

Theorem 4.2 (Graph reduction languages closed under intersection). 1. If \( N \) is a (P)GRS there are nonterminal-free (P)GRSs \( S \) and \( T \) s.t. \( \mathcal{L}(N) = \mathcal{L}(S) \cap \mathcal{L}(T) \). Further, if \( N \) is closed or confluent, so is \( S \); the termination complexity of \( N \) is the termination complexity of \( S \). The GRS \( T \) is confluent and linearly terminating.

2. If \( S \) and \( T \) are (P)GRSs there is a (P)GRS \( N \) s.t. \( \mathcal{L}(N) = \mathcal{L}(S) \cap \mathcal{L}(T) \). Further, if \( S \) and \( T \) are closed or confluent so is \( N \); the termination complexity of \( N \) is the greatest of linear, the termination complexity of \( S \) and the termination complexity of \( T \).
Proof. 1. Any (P)GRS $N$ is equivalent to the intersection of a nonterminal-free (P)GRS $S$ which does the same as $N$ but treats all labels as terminals and another nonterminal-free PGRS $T$ which accepts exactly all nonterminal-free graphs. The details work out as follows.

Let $N = (\langle C_V, C_N, C_E, \text{type} \rangle, \mathcal{R}, \text{Acc})$. Let $C_T = C_V - C_N$ be the set of terminal node labels. Let $S$ be $\langle C_V, \{\}, C_E, \text{type}, \mathcal{R}, \text{Acc} \rangle$, the same specification where no labels are non-terminals. Let $T$ be $\langle C_V, \{\}, C_E, \text{type}, \mathcal{R}', \emptyset \rangle$, where $\emptyset$ denotes the empty graph and

$$\mathcal{R}' = \{ \text{DeArc}(x) \mid x \in C_T \} \cup \{ \text{DeNode}(x) \mid x \in C_T \} \text{ where:}$$

$$\text{DeArc}(x):\quad y \in \text{type x}$$

$$\xymatrix{1 & x & y & 2 \Rightarrow 1 & x & y & 2}$$

$$\text{DeNode}(x):\quad \{ y_1, \ldots, y_n \} = \text{type x}$$

$$\xymatrix{y_1 \ar@{-}[r]^x & \cdots & y_n} \Rightarrow \emptyset$$

So $\# \mathcal{R}' = \# C_T + \sum_{x \in C_T} \# \text{type} x$. The instances of DeArc replace arcs from terminal nodes with loops. The instances of DeNode remove any terminal node with indegree 0 whose arcs are all loops. Any graph containing no non-terminals reduces to the empty graph under $\mathcal{R}'$. Therefore $N = S \cap T$. Termination and confluence of $S$ follows from termination of $N$. $T$ is strongly confluent and linearly terminating because its rules decrease either the number of non-loop arcs or graph size.

2. An intersection can be re-expressed as a single system which makes two copies of a graph then reduces the first copy with $\mathcal{R}_S$ and the second copy with $\mathcal{R}_T$. The accepting graph is the union of the original accepting graphs. The labels of the copies need to be new non-terminals to ensure that this scheme does not extend the original specification. The details work out as follows.

Let $S$ and $T$ be non-terminal-free GRSs where $\Sigma_S = \Sigma_T = (\langle C_V, \{\}, C_E, \text{type} \rangle, \text{type})$. Let $C'_V$, $C''_V$ and $C'_N$, $C''_N$ be distinct renamings of $C_V$ and $C_N$ and $L$ and $N$ be new labels not occurring in any of these sets; $p$ and $q$ are new arc labels. Our signature is:

$$\Sigma_N = \langle C_V \cup C'_N, C'_E, C''_E \cup \{p, q\}, \{L \mapsto \{p, q\}, N \mapsto \{\}\} \cup$$

$$\bigcup\{ l \mapsto \{p, q\} \cup t, l \mapsto t, l'' \mapsto t, l''' \mapsto t \mid l \in C_V, t = \text{type } l \} \rangle$$

where $C_N = C'_N \cup C''_N \cup \{L, N\}$

The rules in Fig. 10 create two distinctly-named copies of any $\Sigma_S$-total graph. DupNode replaces each $C_V$-node by three nodes: two nodes to hold the copies, labelled $N$ meanwhile, and a renaming of the original node which also has $p$ and $q$ arcs to the copies.

When a node and all its successors have been copied its arcs can be copied. DupArcs relabels and adds arcs to such nodes. Some arcs $a_i$ could be loops or shared and the copying must preserve this, so we actually require all the quotient rules $\text{DupA} = \{ g(\text{DupArcs}(l)) \mid l \in C_V, g \text{ is a surjective graph morphism s.t.} \forall i,j \cdot g(i1) \neq g(j2) \land g(i2) \neq g(j3) \land g(i1) \neq g(j3) \land \forall i, j : g(2i1) = g(2j2) \leftrightarrow g(1i1) = g(1j1) \land g(3i1) = g(3j1) \}$ for arc copying.

After arc copying original nodes are labelled $L$. They can be removed by DupDel when they are no longer the target of any arcs.

Let $\mathcal{R}'_S$ and $\mathcal{R}'_T$ be renamings of the two original sets of reduction rules. These will
reduce the copies. The new accepting graph is just the union of renamings of the original accepting graphs. The GRS is:

\[ N = (\Sigma_N, \{\text{DupNode}(l) \mid l \in C_V \setminus C_N\} \cup \text{DupA} \cup \{\text{DupDel}\} \cup R''_S \cup R''_T, \text{Acc}''_S \cup \text{Acc}''_T) \]

\( N \) has \( 2 + 3 \times \#C_V \) new non-terminal labels and \( 1 + \#C_V + \#\text{DupA} \) new rules. This huge increase is caused mainly by our insistence that nodes have a full set of outlabels, so arcs cannot be copied one by one. Copying is linearly terminating as \( \text{DupNode} \) and \( \text{DupArcs} \) can be applied once per node, owing to the relabelling. Therefore the termination complexity of \( N \) is the greater of the termination complexities of \( S \) and \( T \). The copying rules are confluent and there are no critical pairs involving any two of: the copying rules, \( R''_S \) or \( R''_T \); so \( N \) is confluent if \( S \) and \( T \) are confluent.

4.2. Union

Language union offers another way to compose specifications. It is easy to see that union extends the range of languages specifiable by PGRSs and nonterminal-free PGRSs. For instance, a PGRS cannot define a finite language that includes the empty graph.

**Lemma 4.2 (Properties of GRS languages including \( \emptyset \)).** Let \( S \) be a GRS with \( \emptyset \in L(S) \).

1. \( \text{Acc}''_S = \emptyset \).
2. \( L(S) \) is closed under disjoint union of graphs.
3. \( L(S) \) is infinite if it is not \{\emptyset\}.

**Proof.** 1. If \( \text{Acc}''_S \neq \emptyset \) then \( \emptyset \Rightarrow^* \text{Acc}''_S \) and therefore \( \text{Acc}''_S \) is reducible. Therefore if \( \text{Acc}''_S \) is irreducible it must be \( \emptyset \).
2. For any $G, H \in \mathcal{L}(S)$ we have $G \Rightarrow^* \emptyset$ and $H \Rightarrow^* \emptyset$, hence for their disjoint union $G + H \Rightarrow \emptyset$.

3. If $G \neq \emptyset$ and $G \in \mathcal{L}(S)$ then the graph containing $n$ disjoint copies of $G$ is in $\mathcal{L}(S)$ for every $n \in \mathbb{N}$.

Any finite language can be specified as a union of PGRSs with no reduction rules whose accepting graphs are the language members. This non-closure property is not restricted to finite languages.

**Theorem 4.3 (Non-closure under union).** Infinite (nonterminal-free) PGRS languages are not closed under union.

*Proof.* Let $D_i$ denote a discrete graph of $i$ $L$-labelled nodes (a graph with no arcs).

The language of all $D_i$ where $i$ is a multiple of 2 or 3 is easily defined by a union of two nonterminal-free PGRSs: $\mathcal{L}(D_{23}) = \mathcal{L}(\Sigma_B, \{D_2 \Rightarrow \emptyset, \emptyset\} \cup \mathcal{L}(\Sigma_B, \{D_3 \Rightarrow \emptyset, \emptyset\})$. To construct a single nonterminal-free PGRS to accept the same language we must define $\text{Acc} = D_{6n}$ for some $n$. As $D_{6n+2}$ and $D_{6n+3}$ must both reduce to $D_{6n}$ it follows that $D_{6n+1}$, which is not in the language, will also reduce to $D_{6n}$ as it reduces to $D_{6n+2}$.

The same language cannot be defined by any single PGRS $S$ because if $D_2$ reduces to $\text{Acc}_S$ then $D_3$ reduces to $D_1 \cup \text{Acc}_S$; as $S$ is closed $D_1 \cup \text{Acc}_S$ must reduce to $\text{Acc}_S$ and therefore $D_i$ reduces to $\text{Acc}_S$ for every $i \geq 2$.

A single specification can replace a union if it can add some information to the graph to say which of the original reduction systems a rule belongs to, and use this information to prevent graphs that reduce under some combination of both systems from being accepted.

If we allow terminating but non-confluent reduction specifications, we can show that they are closed under union by Theorem 4.4. A technicality (the restriction that accepting graphs are irreducible) prevents this theorem applying to languages that include the empty graph. Note that excluding the empty graph does not affect the result of Theorem 4.3.

**Theorem 4.4 (Closure under union of non-$\emptyset$ GRSSs).** If $S$ and $T$ are (perhaps non-confluent) GRSSs and $\emptyset \notin \mathcal{L}(S) \cup \mathcal{L}(T)$ then there is a GRSS $U$ such that $\mathcal{L}(U) = \mathcal{L}(S) \cup \mathcal{L}(T)$. Moreover, if $S$ and $T$ are terminating then so is $U$.

*Proof.* Let $S = (\Sigma, \mathcal{R}_1, \text{Acc}_i)$ and $T = (\Sigma, \mathcal{R}_2, \text{Acc}_2)$. The new signature has three new non-terminals: $\Sigma_U = \Sigma + \{\{1, 2, A\}, \{1, 2, A\}, \emptyset\}, \{1 \rightarrow \emptyset, 2 \rightarrow \emptyset, A \rightarrow \emptyset\}$.

The reduction rules are modified as shown in Fig. 11. When a rule from $\mathcal{R}_i$ is used an $i$-labelled node is added to the graph. The new $\text{Del}$ rules replace two $i$ nodes by a single $i$ node; the $\text{Accept}$ rules rewrite an original accepting graph $\text{Acc}_i$ or an original accepting graph with one $i$ node to the new accepting graph. $\mathcal{R}_U = \{\text{Rule}(r, i) \mid i \in \{1, 2\}, r \in \mathcal{R}_i\} \cup \{\text{Del}(i), \text{Accept}(\text{Acc}_1), \text{Accept}(\text{Acc}_2), i \rightarrow \{1, 2\}\}$

The new accepting graph just contains the new non-terminal $A$. Clearly every $\mathcal{R}_i$ derivation maps to an $\mathcal{R}_U$ derivation and no derivation involving rules from both $\mathcal{R}_i$ and $\mathcal{R}_2$ can lead to $\text{Acc}_U$. This scheme preserves termination: we just need a $\text{Del}$ step after all but the first reduction and an $\text{Accept}$ step at the end. It does not preserve size reduction but if the original systems were size reducing then $\mathcal{R}_U$ guarantees termination in twice
the original number of steps. It does not preserve confluence because, for example, the \texttt{Del} steps must occur before the \texttt{Accept} step.

4.3. Complement

A further consequence of Lemma 2.1 (simulation of Chomsky grammars) is that (P)GRSs are not closed under complement. Even nonterminal-free size-reducing PGRSs are not closed under complement. As with union, languages including \( \emptyset \) are problematic and we can use this, as in the following example, to show that a (P)GRS language has an undefinable complement. A consequence of Lemma 4.2 is that languages including \( \emptyset \) are undefinable if they are not closed under disjoint union.

\textbf{Example 4.3 (Language with undefinable complement).} Consider the nonterminal-free PGRS \( AB = (\Sigma, \{AABB\}, Acc_{AB}) \) where \( \Sigma \) contains the nullary terminals \( A \) and \( B \) only.

\[
Acc_{AB} = \begin{array}{c}
A \\
B
\end{array}
\]

\[
AABB : \begin{array}{cccc}
A & A & B & B \\
\Rightarrow & A & B
\end{array}
\]

\( L(AB) \) contains all non-empty \( \Sigma \)-graphs which have the same number of \( A \) and \( B \) nodes. Therefore \( L(AB) \) includes \( \emptyset \) and the graph containing one \( A \) only and the graph containing one \( B \) only. As \( L(AB) \) does not include the graph containing one \( A \) and one \( B \), it is not closed under disjoint union so it cannot be defined by a GRS by Lemma 4.2.

5. Modelling Nil Pointers

Our example GRSs are simple abstract models of shapes but they differ from standard practice for pointer data structures in that nil is not a single shared object. This means that to be faithful to our specifications an implementation must incur a constant-factor inefficiency overhead by storing each tree leaf at a separate address: the graph models quite clearly say that leaves are all distinct and to share them in the implementation without careful analysis could easily lead to pointer errors. For example, if an operation deletes a leaf and the implementation shares all leaves, then the implemented operation will create dangling pointers. To avoid this effect a GRS-based implementation would need a special analysis to enable nil sharing.

Alternatively, we can give specifications which lead to implementations with the conventional representation of nil. Section 5.1 demonstrates the specification of a shared nil
and Section 5.2 shows how we could use partial graphs to represent nil and why we prefer not to do so.

5.1. Shared nil specifications

Example 5.1 (Full binary trees with a shared leaf). Full binary trees with a single shared leaf are defined by this PGK: \( SFBT = (\Sigma_{BT}, \{\text{BL, oL, OneBL}, \text{AccL}\}) \). The reduction rules are given in Fig. 12. For tree-like structures, sharing leaves inevitably needs a larger specification: branches must not be shared so special rules are needed for the leaves.

All the other tree specifications in this paper could be rewritten with a shared leaf, but we do not do so, because they would require more rules. On the other hand, branching structures that have shared sub-trees are likely to have simpler specifications if we follow the shared leaf convention.

5.2. \( \Sigma \)-partial specifications

Graphs offer another obvious model for the nil pointer: let non-total \( \Sigma \)-graphs model structures and any missing arc is a nil pointer. We briefly investigate the possibilities of this approach here. Firstly we adapt Definition 2.8 so partial graphs can be reduced.

Definition 5.1 (\( \Sigma \)-partial rule). A rule \( \langle L \supseteq K \subseteq R \rangle \) is \( \Sigma \)-partial if \( L, R \) are \( \Sigma \)-graphs and:

1. \( l_L(x) = \bot \Rightarrow x \in V_K \wedge l_R(x) = \bot \wedge \text{outlabels}_L(x) \supseteq \text{outlabels}_R(x) \)

Unlabelled nodes in \( L \) are preserved, remain unlabelled, some arcs may be removed.

2. \( x \in V_K \wedge l_L(x) \neq \bot \wedge l_L(x) = l_R(x) \Rightarrow \text{outlabels}_L(x) \supseteq \text{outlabels}_R(x) \)

A labelled node in \( L \) which is preserved with the same label may have some arcs removed in \( R \).

3. \( x \in V_K \wedge l_L(x) \neq \bot \wedge l_L(x) \neq l_R(x) \Rightarrow l_R(x) \neq \bot \wedge \text{outlabels}_R(x) \supseteq (\text{type } (l_L(x)) - \text{ type } (l_R(x))) \wedge \text{outlabels}_R(x) \subseteq \text{outlabels}_L(x) \cup (\text{type } (l_R(x)) - \text{type } (l_L(x))) \)

A relabelled node: in \( L \) it has at least all the outlabels of its \( L \)-label which are not outlabels of its \( R \)-label to ensure the outlabels in \( R \) are all in the appropriate type; in \( R \) it can only have outlabels which are present in \( L \) or which are not outlabels of its \( L \)-label to prevent the introduction of arcs already present in instances of \( L \).

Nodes may not be labelled in \( L \) (or \( R \)) and unlabelled in \( R \) (or \( L \)).

4. Deleted nodes are labelled and have a subset of the outlabels for that label as \( L \) is a \( \Sigma \)-graph (there is nothing to check).

5. \( x \in V_R - V_K \Rightarrow l_R(x) \neq \bot \). Allocated nodes are labelled and have a subset of the outlabels for that label as \( R \) is a \( \Sigma \)-graph.

Theorem 5.1 (\( \Sigma \)-partial rules preserve \( \Sigma \)). If \( G \) is a \( \Sigma \)-graph and \( r \) is a \( \Sigma \)-partial rule and \( G \Rightarrow_{r,g} H \) then \( H \) is a \( \Sigma \)-graph.
Specifying Pointer Structures by Graph Reduction

![Diagram of reduction rules for full binary trees with a shared leaf](image)

**Fig. 12.** Reduction rules for full binary trees with a shared leaf.

![Diagram of a Σ-partial specification of full binary trees](image)

**Fig. 13.** A Σ-partial specification of full binary trees.

**Proof.** Follows from Definition 5.1.

Now we might expect a simple specification of full binary trees which encodes the empty tree as the empty graph, a singleton as a single arc-less branch node and so on. Alas, this specification is impossible: if Acc = ∅ it will define forests (see Lemma 4.2). Instead we make Acc the singleton.

**Example 5.2 (Partial full binary trees).** Full binary trees excluding ∅ are defined by PFBT = <ΣBT, {Pick}, AccF> (see Fig. 13). This recognises any tree whose nodes are labelled B and which may have a left arc, a right arc, neither or both. Using Pick repeatedly we can remove nodes from the tree bottom-up until we reach the singleton which is accepted.

Note that Pick can also be used to reduce a branch with many left-labelled outgoing arcs to the accepting graph. Therefore with a partial specification we must check that graphs are Σ-graphs before checking their membership by reduction.

By adding the reduction rule AccLeaf : ∅ ⇒ AccF we could add the empty tree to the language. But this would make the accepting graph reducible, so PFBT would be non-terminating and not a partial GRS — even allowing partial rules and graphs. A terminating specification of this language including ∅ is impossible by Lemma 4.2.

Example 5.2 illustrates why we prefer the total graph model for data structures; including ∅ in a language can make its specification impossible or non-terminating, therefore the empty structure should be represented by a leaf node. This problem is alleviated if we assume that trees always have a root node (representing the location of the root pointer), but we still need more rules and we still have the problem that reduction does not preserve the signature in both directions. So we prefer to model nil pointers by nullary graph nodes.
6. Specifications of Popular Shapes

This section applies the GRS theory to specify a number of popular pointer data structure shapes. In each case we give a specification by properties (mostly taken from (Cormen, Leiserson & Rivest 1990)) and a proof that the PGRS is correct (sound and complete relative to the property specification). The aim is always to present a simplest possible PGRS, that is, as few rules as possible, as few non-terminals as possible and, ideally, confluent and terminating, preferably strongly confluent and size-reducing. The examples show that these simplicity criteria raise some interesting conflicts. The examples in sections 6.1 to 6.5 present list variants, threaded tree variants, balanced trees, red-black trees, AVL trees and grids respectively. Section 8 provides a summary of all the specifications. The correctness proof is usually an instance of the following.

**Theorem 6.1 (GRS correctness argument).** Nonterminal-free GRS $S$ is a sound and complete specification of language $L$ whose members satisfy $p$ if (1) $\text{Acc}_S$ is the single smallest member of $L$, (2) every rule in $R_S$ preserves $p$ in both directions and (3) every non-$\text{Acc}_S$ member of $L$ is reducible. If GRS $S$ uses non-terminals we require in addition that every nonterminal-free graph satisfying $p$ is in $L$.

**Proof.** Soundness: every graph obtained by inverse derivation from $\text{Acc}_S$ satisfies $p$. Therefore every such graph without non-terminals is in $L$. Completeness: from every graph satisfying $p$ another graph satisfying $p$ can be derived; eventually $\text{Acc}_S$ is reached as $S$ is terminating and closed.

6.1. Lists

The basic singly linked list is just a tree whose branches are all unary. So we can specify it by the following PGRS.

**Example 6.1 (PGRS of linked lists).** $\text{LIST} = (\Sigma_{\text{BT}}, \{\text{UtoL}\}, \text{Acc}_L)$ where $\text{UtoL}$ and $\text{Acc}_L$ are in Fig. 1.

The following definitions provide PGRSs of some simple list variants taken from (Cormen et al. 1990, Klarlund & Schwartzbach 1993, Fradet & Mêtayer 1998). The reduction rules are shown in Fig. 14. They are all correct (as are $\text{LIST}$ and $\text{CLIST}$) by Theorem 6.1. They are all strongly confluent, size reducing and nonterminal-free.

**Example 6.2 (PGRS of last-element lists).** Every $C$-labelled cons node has an $n$-arc to the next list element and an $l$-arc to the last element. The PGRS is $\text{LAST} = (\Sigma_{\text{Last}}, \{\text{ConsOne}, \text{ConsSs}\}, \text{Acc}_L)$ where $\Sigma_{\text{Last}} = \langle\{C, L\}, \{n, l\}, \{C \mapsto \{n, l\}, L \mapsto \{\}\}\rangle$.

**Example 6.3 (PGRS of doubly-linked lists).** Every $D$-labelled double-cons node has an $n$-arc to the next list element and a $p$-arc to the previous list element. The previous node of the first cons and the next node of the last cons are distinct leaves but the empty list is just a single leaf as usual. The PGRS is $\text{DLIST} = (\Sigma_{\text{DLat}}, \{\text{DconsOne}, \text{DconsSs}\}, \text{Acc}_L)$ where $\Sigma_{\text{DLat}} = \langle\{D, L\}, \{n, p\}, \{D \mapsto \{n, p\}, L \mapsto \{\}\}\rangle$. 

Example 6.4 (PGRS of skip lists). These lists are chains of \( C \)-labelled cons nodes (with an \( n \)-arc to the next list element) and \( S \)-labelled skip-cons nodes (which also have an \( s \)-arc to the next skip-cons or the list end leaf in the case of the last skip-cons). The PGRS is \( \text{SKIP} = \langle \text{SKIP}, \{\text{Cons}, \text{Skip}\}, \text{Acc}_L \rangle \) where \( \Sigma_{\text{SKIP}} = \langle \{S, C, L\}, \{\}, \{n, l\}, \{C \mapsto \{n\}, S \mapsto \{n, l\}, L \mapsto \{\}\} \rangle \).

6.2. Threaded trees

Adding extra pointer to tree nodes is a simple way to improve the speed of operations. This section presents a selection of trees with additional pointers.

6.2.1. Singly threaded trees are binary search trees whose branches all hold an item of data and whose leaves do not. All data in the left subtree of a branch are less than the datum in that branch; all data in the right subtree are greater. In addition to the tree structure each branch node has a next pointer to the branch containing the smallest datum in the tree which is greater than its own datum. The next pointer of the greatest datum branch points to a nil node. The tree has a root node with pointers to the top of the tree and the least datum branch.

Example 6.5 (Nonterminal-free size-reducing PGRS). The PGRS is \( \text{TT} = \langle \Sigma_{\text{TT}}, \mathcal{R}_{\text{TT}}, \text{Acc}_{\text{TT}} \rangle \) where \( \Sigma_{\text{TT}} = \langle \{R, T, L, N\}, \{\}, \{t, l, r, n\}, \{R \mapsto \{n, t\}, T \mapsto \{l, r, n\}, L \mapsto \{\}, N \mapsto \{\}\} \rangle \) and \( \text{Acc}_{\text{TT}} \) and \( \mathcal{R}_{\text{TT}} \) are in Fig. 15. \( \text{Cast} \) reduces the singleton \( \text{TT} \) to the empty \( \text{TT} \). \( \text{LeftStitch} \) replaces a branch of leaves which is a left child by a single leaf. It preserves the threaded tree properties by moving the next pointer of its predecessor to its successor. \( \text{RightStitch} \) works analogously.

---

Specifying Pointer Structures by Graph Reduction
The GRS $TT$ is correct by Theorem 6.1; it is not confluent as LeftStitch and RightStitch can reduce graphs which are not in $L(TT)$ to distinct and irreducible graphs. However, we conjecture that it is closed.

6.2.2. **Linked-leaf trees** are full binary trees with a root node and a nil node. The root node points to the top and to the left-most leaf of the tree. Each leaf points to the next leaf encountered in an in-order traversal. The right-most leaf points to the nil node. A PGRS $TLEAF$ is given as follows. It is correct by Theorem 6.1; like $TT$ it is not confluent but we conjecture it is closed.

**Example 6.6 (Size-reducing PGRS).** $TLEAF = \langle \Sigma_{TLEAF}, \{\text{Branch}\}, \text{Acc}_{TLEAF} \rangle$ is the PGRS, where Fig. 16 gives the reduction rule and accepting graph and $\Sigma_{TLEAF} = \langle \{R, B, L, N\}, \{l, r, n, o\}, \{R \mapsto \{n, o\}, B \mapsto \{l, r\}, L \mapsto \{n\}, N \mapsto \{\}\rangle$ ⊞

6.2.3. **Root-connected trees** are full binary trees in which every leaf points to the root. A PGRS $TROOT$ is given as follows. It is size-reducing and strongly confluent; it is correct by Theorem 6.1.

**Example 6.7 (Size-reducing PGRS).** $TROOT = \langle \Sigma_{TROOT}, \{\text{Branch}\}, \text{Acc}_{TROOT} \rangle$ is the PGRS where the reduction rule and accepting graph are in Fig. 17 and $\Sigma_{TROOT} = \langle \{B, L\}, \{l, r, o\}, \{B \mapsto \{l, r\}, L \mapsto \{o\}\rangle$ ⊞
6.3. Balanced n-ary trees

Balanced binary trees have a simple graph reduction specification but they are very difficult to use (as search trees) because complex re-arrangements are needed after insertion or deletion. Allowing higher degrees of branching is one solution. Here we generalise the \( \text{BBT} \) specification to such trees.

6.3.1. 2-3 trees are perhaps the simplest kind of balanced tree with practicable insertion and deletion algorithms (see (Reade 1992) for example). 2-3 tree nodes can be 2 or 3-way branches, or leaves. All leaves have the same depth.

**Example 6.8 (2-3 tree PGRS).** Signature \( \Sigma_{23} \) extends the binary tree signature \( \Sigma_{\text{BBT}} \) (see Example 2.1) with a ternary branch: \( \Sigma_{23} = \Sigma_{\text{BBT}} + \{\{T\}, \{\}, \{T \mapsto \{l, c, r\}\}\} \). The PGRS is \( 23 = \langle \Sigma_{23}, R_{\text{BBT}} \cup \{\text{PickLeaf}', \text{PushBranch}'\}, \text{Acc\_L} \rangle \) where \( R_{23} \) uses the BBT reduction rules (see Fig. 5) and the two new rules in figures 18. PickLeaf and PickLeaf' reduce branches which are leaf-parents to a U-branch leaf-parent. PushBranch and PushBranch' move branches down towards the leaves. FellTrunk reduces the height of a tree whose root is a U-branch. The accepting graph a single leaf as usual (see Fig. 1).

PGRS 23 is strongly confluent and size reducing; it is correct by Theorem 6.1 where property \( p \) defines all 1-2-3 trees (trees with unary, binary and ternary branches). This specification easily extends to any kind of balanced tree with a fixed selection of branching arities. All such specifications are strongly confluent and size reducing. Balanced trees with variable branching arities (and therefore B-trees) cannot be directly specified as GRSs, but a specification based on sibling trees — where each node is represented as a
list of branches — would be possible. We do not know if nonterminal-free specifications of 2-3 trees exist.

6.3.2. 2-3-4 trees can be specified by generalising the 2-3 specification. But here we give a slightly shorter and nonterminal-free PGRS. 2-3-4 tree nodes can be 2, 3 or 4-way branches, or leaves where all leaves have the same depth. The PGRS 234 is size reducing and strongly confluent; it is correct by Theorem 6.1.

**Example 6.9 (Nonterminal-free PGRS of 2-3-4 trees).** 234 = (Σ<sub>234</sub>, R<sub>234</sub>, Acc<sub>L</sub>) is the PGRS where Σ<sub>234</sub> = {1, 2, 3, 4, L}, {a, b, c, d}, 2 \(\rightarrow\) {a, b}, 3 \(\rightarrow\) {a, b, c}, 4 \(\rightarrow\) {a, b, c, d}, L \(\rightarrow\) {L}) and R<sub>234</sub> comprises the six rules in figures 19. FellStump2 reduces a 2-branch 'stump' to Acc<sub>L</sub>. PickLeaf3 and PickLeaves4 replace bunches of leaves with two leaves. PushBranch3 and PushBranch4 force heavier branches to the leaves where they can be picked. PushBranch2 reduces tree depth by replacing three 2-branches with a 4-branch. Like the BBT specification the only depth-reducing rules apply at the root, this guarantees the balancing property. Unlike Complete Binary Trees no non-terminals are needed owing to the way PushBranch2 works.
Fig. 19 continued. 2-3-4 tree reduction rules (part 2).
6.4. Red-black trees

Red-black trees are trees of binary-branches and leaves. Branches are labelled red or black. Children of red branches are black or leaves. All paths from the root to a leaf have the same number of black nodes.

6.4.1. A nonterminal-free PGRS Our simplest specification is interesting because it is nonterminal-free and terminating but it is not size-reducing. Using a simplification of Lemma 4.1 we show in Theorem 6.2 that a size-reducing specification needs non-terminals. Such a specification is given in Section 6.4.2; compared to the nonterminal-free PGRS it has more rules but terminates in about half as many steps.

**Theorem 6.2 (A size-reducing GRS of RBTs needs non-terminals).** Red-black trees cannot be specified by a size-reducing nonterminal-free GRS.

**Proof.** Using the following simplification of Lemma 4.1 (which defines rsize, see Page 14). If \( \forall k \in \mathbb{N} \exists G \in \mathcal{L} \cdot \forall G' \in \mathcal{L} \cdot \text{size}(G') < \text{size}(G) \Rightarrow rsize(G, G') \geq k \) then language \( \mathcal{L} \) cannot be defined by a size-reducing nonterminal-free GRS. Because to be defined by a size-reducing nonterminal-free GRS there must be a finite rule which derives some smaller graph from every non-Acc graph in \( \mathcal{L} \).

For RBTs, consider an arbitrary black-only RBT of height \( n \) (so it is a CBT). To remove one black leaf-parent and re-colour the tree such that it is black-balanced we must re-colour nodes in both sub-trees of the root. Therefore the left graph of a rule which causes this change has size greater than \( n \). Similarly, to remove up to \( k \) nodes and re-colour requires a rule which changes both sub-trees of the root and some leaf-parent and whose left graph has size greater than \( n \).

**Example 6.10 (Nonterminal-free red-black tree PGRS).**

\( RBT = (\Sigma_{RBT}, \mathcal{R}_{RBT}, \text{Acc}_L) \), where Fig. 21 shows the reduction rules and Fig. 1 shows \( \text{Acc}_L \) and \( \Sigma_{RBT} = \langle \{R, B, L\}, \{\}, \{l, r\}, \{R \mapsto \{l, r\}, B \mapsto \{l, r\}, L \mapsto \{\} \} \rangle \)

Each rule preserves the red-black properties and produces either a smaller or a redder tree (therefore \( \mathcal{R}_{RBT} \) terminates). \( RBT \) is correct by Theorem 6.1. 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 RedenRoot, safely reducing the black height, and then pushed down and picked by the other rules. Eventually we reach a singleton which is rewritten to \( \text{Acc}_L \) by FellStump.

6.4.2. A size-reducing PGRS A size-reducing specification of red-black trees is possible if we use a non-terminal node label \( G \) — a green node — which plays a similar role to the unary branch in CBTs.
Example 6.11 (Size-reducing red-black tree PGRS).

SRBT = (ΣSRBT, RSRBT, Accl) where the reduction rules are in Fig. 21 and ΣSRBT = \{\{R, B, G, L\}, \{G\}, \{l, r, c\}, \{R \mapsto \{l, r\}, B \mapsto \{l, r\}, G \mapsto \{c\}, L \mapsto \{\}\}\}

This specification can be thought of as removing all red nodes that occur between black nodes and checking that the remaining black structure is a complete balanced tree. To preserve the signature the red removal and black checking steps need to be
Fig. 21. Size-reducing red-black tree reduction rules (part 1).
intermingled. So we can explain the reduction process bottom-up. It is safe to remove any red leaf-parent whose parent is black (PickRedLeaf). Red grandparents occurring between black nodes can be removed by PickRedFork, or PickRedRoot if the tree is of depth 3. Black grandparents are replaced by unary green branches by PickBlackFork. Then higher up in the tree, black branches can be pushed down through green nodes by the height-preserving PushBlackFork, similarly by PushRedFork where there is a red node with green children and a black parent, or by PushRedRoot where the root is red. A red-black tree will reduce to a trunk of green nodes leading to a single fork, these are reduced by FallRedRoot and FallGreenRoot. SRBT is correct by Theorem 6.1 where is all red-black-green trees where green is a unary black branch.

6.5. AVL trees

AVL tree nodes are binary branches or leaves (labelled N). Branches are balanced (labelled B), left-leaning (labelled L) or right-leaning (labelled R). The subtree depths of
balanced branches are equal. Left subtree depth of a left-leaning node is one plus right subtree depth. Right subtree depth of a right-leaning node is one plus left subtree depth.

**Example 6.12 (AVL tree PGRS).** \(\text{AVL} = (\Sigma_{AVL}, R_{AVL}, Acc_N)\) where \(Acc_N\) and the rules in \(R_{AVL}\) are shown in Fig. 22 and \(\Sigma_{AVL} = \{\{B, L, R, N, S, B?, L?, R?\}, \{S, B?, L?, R?\}, \{l, r, s\}, \{B, L, R \mapsto \{l, r\}, N \mapsto \{\}, S \mapsto \{s\}, B?, L?, R? \mapsto \{l, s, r\}\}\) \(\square\)

The reduction rules replace an AVL tree of depth \(n\) with an \(S\)-chain of length \(n\). This chain is reduced to \(Acc_N\) by \(\text{Fells}\). A tree is checked bottom-up. A branch labelled \(B\), claiming to be balanced, is first converted to a \(B?\) node with arcs to its left and right subtree \(S\)-chains and an arc to its own \(S\)-chain. \(\text{CheckBLR}\) descends the sub-tree \(S\)-chains simultaneously, extending the new \(S\)-chain at each step. If both sub-tree chains have the same length then \(\text{Balanced}\) rewrites the \(B?\) node to an \(S\) node. Checking branches claiming to be left or right leaning follows the same patterns, using \(\text{LeanL}\) or \(\text{LeanR}\) as appropriate.
Specifying Pointer Structures by Graph Reduction

PGRS AVL is strongly confluent; it is linearly terminating as each derivation from \(G\) reduces \(\#V_G + 2 \times \#\{v \in V_G \mid \text{type}(v) \in \{B,L,R\}\}\). It is correct by Theorem 6.1 where \(p\) defines all reducing AVL trees. These are trees which are \(\Sigma_{AVL}\)-total graphs and whose nodes have the following height properties. The height of an B,L,R node is one plus the maximum height of its children; the height of an S,B?,L?,R? node is one plus the height of its s-child plus the maximum height of its other children; the height of an N node is 0. The height of the left and right children of B and B? nodes are equal; the height of the left (right) child of an L or L? (R and R?) node is one plus the height of its right (left) child.

AVL trees can also be specified by a size-reducing PGRS: change CheckBLR so that the chain lengths are at least 3 instead of 1 to make it size-reducing and introduce six new rules to directly reduce branches of shorter chains. The other rules are unchanged. So this size-reducing PGRS needs 16 rules compared to the 10 in AVL. Any specification will need non-terminals.

**Theorem 6.3 (An AVL specification needs non-terminals).** AVL trees cannot be specified by a nonterminal-free GRS.

**Proof.** Using Lemma 4.1. Consider an arbitrary AVL tree of depth \(n\). To rewrite it to any smaller AVL tree requires a rule whose left graph includes at least the root and a leaf (and so has size at least \(n\)); If the root is balanced the smallest reduction unbalances it and removes leaves from one sub-tree; if the root leans left the simplest reduction relabels it as balanced and removes a leaf from the left sub-tree; similarly for a right-leaning root.

\[\square\]

6.6. Rectangular grids

A graph is a grid if it has \(n \times m\) nodes labelled \(B\) and \(n + m\) nil nodes labelled \(N\) where \(n, m > 0\) and: Each \(B\) node has a down and a right pointer; Each \(B\) node is assigned a unique coordinate \((i,j) \in \{1, \ldots , n\} \times \{1, \ldots , m\}\); Each \(N\) node is assigned a coordinate \((n + 1, j)\) or \((i, m + 1)\); The down arc of node \((i,j)\) points to node \((i,j+1)\); The right arc of node \((i,j)\) points to node \((i+1,j)\).

**Example 6.13 (Grid PGRS).** \(GRID = (\Sigma_{GRID}, \mathcal{R}_{GRID}, \text{Acc}_{GRID})\) where \(\text{Acc}_{GRID}\), the \(\mathcal{R}_{GRID}\) rules are in Fig. 23 and \(\Sigma_{GRID} = \{\{B,C,N\}, \{C\}, \{d,r\}, \{B,C \rightarrow \{d,r\}, N \rightarrow \{\}\}\}\). Grids are reduced by relabelling their top-left node \(C\) with a ColourTL rule; then the grid is dismantled one row at a time, checking that nodes in the top row are aligned with the nodes in the row below. PickTop removes \(B\) nodes from the top row; when the top row contains at most one \(B\) it is removed and the C label moves down to what should be the top-left node of the new grid by a Skim rule. When the grid becomes \(n \times 1\) it is reduced like a list by SkimLeft. If the grid is \(1 \times n\) it is reduced by SkimTop’. \(\square\)

**Theorem 6.4 (GRID specifies rectangular grids).** \(G \in \mathcal{L}(GRID)\) iff \(G\) is a grid.
Fig. 23. Grid accepting graph and reduction rules.
Proof. Let a reducing grid be a grid except its top-left node (1, 1) is labelled C. But the d-arc of this node points to (1, 2) as usual. The top row may be incomplete; so the r-arc of the C may point to node (i, 1) for any 1 \( i \leq n + 1 \). The nodes (2, 1) to \((i - 1, 1)\) do not exist in a reducing grid. Nonterminal-free reducing grids are grids so the result follows by Theorem 6.1.

The GRID rules are not size-reducing but they are linearly terminating as they reduce graph size plus the number of B-labelled nodes. They have non-strongly-joinable critical pairs but we conjecture that they are closed. They need non-terminals.

**Theorem 6.5 (Grids need non-terminals).** A grid GRS needs non-terminals.

Proof. Using Lemma 4.1. Consider an arbitrary \( n \times m \) grid. To rewrite it to any other grid requires a rule whose left graph includes at least \( \min\{n, m\} \) nodes.

7. Related Work

There are other approaches to shape specification. Most closely related to GRSs are the Shape Types of Fradet and Le Métayer discussed in Section 7.1. Some other type or logic-based approaches are mentioned in Section 7.2. In Section 7.3 we briefly review our work on the second goal mentioned in the introduction: pointer algorithm shape-safety checking.

7.1. Shape types

Shape Types (Fradet & Métayer 1996, Fradet & Métayer 1997, Fradet & Métayer 1998) are specified by context-free hypergraph grammars. All Shape Types can be converted to reduction specifications, as we show below, but the classes of context-free graph languages and PGRS languages are incomparable. The literature shows that there are graph languages satisfiable by context-free graph grammars whose membership problem is NP-complete (Drewes 1993, Drewes, Habel & Krewski 1997), these cannot be specified by PGRSs. However, we are not aware of any common data structure with a context-free specification and no PGRS. Further, PGRSs can specify shapes beyond the scope of shape types, like balanced trees and grids (Fradet & Métayer 1997).

In (Bakewell, Plump & Runciman 2003) we give a rule for converting hypergraphs and hypograph grammars into \( \Sigma \)-graphs and GRSs. Here we just consider an example. The conversion relies on the well-known correspondence between hypergraphs and bipartite graphs.

**Example 7.1 (Converting the Shape Types list specification).** In (Fradet & Métayer 1997) lists are specified by the hypergraph grammar \( \{L, List\}, \{next\}, \{L \rightarrow 1, List \rightarrow 0, next \rightarrow 2\}, P, List \). These lists are chains of nodes joined by binary next edges where the next edge of the end node goes from the end node to the end node. We can encode such lists as \( \Sigma_{STLIST} \) graphs which are chains of O-labelled nodes joined by n-labelled nodes (corresponding to the next edges).
$\Sigma_{STLIST} = \langle \{L,S,n,O\}, \{L,S\}, \{1,2\}, \{L \mapsto \{1\}, S \mapsto \emptyset, n \mapsto \{1,2\}, O \mapsto \emptyset \rangle$

The production rules in $P$ and their GRS reduction rule conversions are shown with the accepting graph in Fig. 24. A list is reduced by replacing the $n$ pointer at the end by a unary non-terminal $L$ pointer which can be interpreted as meaning 'a list began here'; then the $L$ is moved up to the front of the list and finally rewritten to the start symbol $S$ when there are no nodes left.

Context-exploiting shapes (Drewes, Hoffmann & Minas 2003), like PGRSs, are another interesting compromise between context-sensitive and context-free graph grammars, aimed at shape specification. This extends the work on hypergraph shapes specified by context-free rules in (Hoffmann 2001), the precise relation to PGRSs is unclear: the context-exploiting rule formulation appears more restrictive than $\Sigma$-rules which are context-sensitive but the membership checking is exponential. Most significantly, there is a restricted, but decidable, class of shaped transformation rules that preserve context-exploiting shapes.

7.2. Other shape specification methods

There are a number of other approaches to shape specification in the literature.

In functional programming, Nested types can be used to specify perfect binary trees (Hinze 2000); however, these are only complete balanced binary DAGs as they do not preclude sub-tree sharing.

The following approaches 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 (Hendren et al. 1992) 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 (Benedikt et al. 1999) allows the reachability, cyclicity 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 (Kuncak et al. 2002) 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 (Klarlund & Schwartzbach 1993) are recursive data types extended with routing expressions which allow the target of a pointer to be specified relative to its source. In (Mäler & Schwartzbach 2001), 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.

The problem of translating monadic 2nd-order logical shape specifications, of bounded treewidth, into terminating reduction specifications is considered in (Arnborg, Courcelle, Proskurowski & Seese 1993). The specifications produced are similar in spirit to GRSs although the graphs are undirected and unlabelled and there may be several accepting graphs.

Alias types are an advanced pseudo-linear type system for specifying store shapes with strictly controlled sharing (Walker & Morrisett 2001).

7.3. Checking pointer manipulations

This section briefly demonstrates our method for verifying the shape safety of an algorithm, detailed in (Bakewell, Plump & Runciman 2004). To check a pointer algorithm we first derive (or specify) an abstraction of the algorithm in the form of a set of shape-annotated graph transformation rules (Σ-total rules) with a control strategy. Running the algorithm on a data structure is modelled as applying these rules to the graph representing the data structure shape.

Example 7.2 (Insertion in binary search trees). Fig. 25 gives five rules which model insertion into binary search trees (BSTs) as a transformation on their shape. BSTs (see Definition 7.1) have a single root node labelled $R$ whose $a$-arc points to the root of the tree. We assume that each branch holds a data item in the concrete BST, and leaves do not.

Insertion first applies the Begin rule once. This takes a BST and adds a new auxiliary node labelled $I$ whose $a$-arc points to the root, $I$ indicates the current position of the insertion algorithm in the tree. Begin breaks the BST shape, so during insertion we expect the graph to have the shape BST with an auxiliary, defined by the PGRS $\text{BST}(I)$ in Definition 7.1.

Insertion then applies the other rules in Fig. 25 non-deterministically until termination. So every possible insertion into every possible tree is represented by some rule sequence. GoLeft and GoRight move the $a$-arc down the tree, to model searching the tree for the
insertion position. **DontInsert** removes the $I$ and $a$-arc when they point to a branch, restoring the original tree; this models the insertion of an item already in the tree at the current node. **DoInsert** replaces the leaf pointed to by $I$ with a branch of leaves and removes $I$; this models the insertion of an item.

**Definition 7.1 (Binary search trees and insertion states).** Binary search trees (BSTs) are rooted full binary trees defined by $BST = \langle \Sigma_{BST}, \{BtoL\}, Acc_{BST} \rangle$ (see figures 1 and 26) where $\Sigma_{BST} = \Sigma_{BT} + \{ (R, l, j), \{0, a\}, \{R \mapsto \{0\}, l \mapsto \{a\}, J \mapsto \{\} \} \}$. BSTs with an auxiliary are rooted full binary trees with an $l$-node whose arc a may point anywhere in the tree: $BST(l) = \langle \Sigma_{BST}, \{BtoL, BtoLl, BtoLr\}, Acc_{BST(l)} \rangle$ (see figures 1 and 26).

The shape-safety checker attempts to verify the shape annotation of each transformation. So the abstract algorithm is safe if the following property can be proved for each of its transformations.

**Definition 7.2 (Shape-safe rule).** Rule $t : S \times T$ is shape safe if $G \in L(S) \land G \Rightarrow_1 H \Rightarrow H \in L(T)$.

The safety checker available from (Safe Pointers by Graph Transformation, project webpage) can check the BST insertion algorithm. Shape safety is an undecidable problem, as it amounts to a graph language inclusion problem, so not every safe transformation can be checked.
**Specifying Pointer Structures by Graph Reduction**

\[ A \overset{\omega_{\text{BST}}}{\longrightarrow} L \overset{\circ}{\longrightarrow} R \]

\[ A \overset{\omega_{\text{BST}}(l)}{\longrightarrow} J \]

\[ x \in \{L, B\} \]

\[ 1 \overset{a}{\longrightarrow} x \]

\[ [1] \quad x \quad 1 \Rightarrow [J] \quad x \quad 1 \]

**Fig. 26. BST and BST(l) accepting graphs and reduction rules.**

8. **Conclusion**

Graph reduction specifications are a powerful formal framework, capable of defining data structures with non-context-free properties. Although PGRSs are much more restricted than general graph grammars, the examples presented here show how they can specify a wide variety of practically useful shapes — indeed they seem not to preclude any commonly used shapes — so the PGRS framework usefully tames the universal power of non-context-free graph grammars. The GRS tool available from (Safe Pointers by Graph Transformation, project webpage) implements GRS checking including confluence, membership and operation checks.

Table 1 summarises the example shapes considered in this paper and classifies their specifications according to their termination, confluence and use of non-terminals, intersection or union. Broadly, the simplest shapes to work with have a specification with the classification \( P - f - S - s \). For others it took more work to give a specification and prove its properties. For a few specifications the \( m \) classification indicates that we still lack a general method for demonstrating closedness.

The classification \( U \) indicates GRS-undefinable languages. In general there are three causes of unspecifiability. (1) The logical specification of some languages demands graphs that do not fit our signature restrictions; examples include B-trees and series-parallel graphs, which have unlimited outdegree. Such languages can typically be specified by use of some encoding transformation, just as implementors use sibling trees to encode B-trees for example. (2) The fundamental result that some languages are unspecifiable is not a problem as programmers cannot demand such shapes! (3) The PGRS restrictions of irreducible accepting graph, polynomial termination and closedness exclude otherwise specifiable languages. This includes languages that require NP-complete specifications (Drewes 1993) and the problems we encountered with languages including \( \emptyset \) such as \( AB \).

The evidence of Section 6 is that a tolerably natural PGRS exists for most practical pointer structure shapes.

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 Section 7 and (Bakewell et al. 2004) is a starting point. For operations like insertion into red-black trees (Cormen et al. 1990) 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


A. Bakewell, D. Plump, and C. Runciman (2004), Checking the shape safety of pointer manip-
Specifying Pointer Structures by Graph Reduction

ulations, in Proc. 7th International Seminar on Relational Methods in Computer Science (RelMICS '7), LNCS, Springer-Verlag. Extended abstract available from (Safe Pointers by Graph Transformation, project webpage ).


Safe Pointers by Graph Transformation, project webpage.
http://www-users.cs.york.ac.uk/~ajb/spgt/.