

Specifying Pointer Structures by Graph Reduction*

Technical Report YCS-2003-367

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

August 29, 2003

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 report 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.

Our practical investigation shows how nil pointers can be modelled and presents 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 the simplest possible PGRS where simplicity means 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.

The theory is implemented in a checking tool available from the project webpage [SPG].

*Work partly funded by the EPSRC project *Safe Pointers by Graph Transformation*[SPG], grant GR/R72440/01. A short version of this report will appear in [BPR03b].

Contents

1	Introduction	3
2	Graph-Reduction Specifications	5
2.1	Signatures	5
2.2	Graph reduction	5
2.3	Signature preservation	7
2.4	Shape specifications	10
3	Membership Checking	11
3.1	Termination	11
3.2	Closedness, confluence and complexity	13
4	Closure properties of GRSs	15
4.1	Intersection	15
4.2	Union	19
4.3	Complement, \emptyset -languages and Chomsky grammars	20
5	Modelling Nil Pointers	22
5.1	Shared nil specifications	22
5.2	Σ -partial specifications	23
6	Example Shapes	24
6.1	Lists	25
6.2	Threaded trees	26
6.3	Balanced n -ary trees	28
6.4	Red-black trees	31
6.5	AVL trees	36
6.6	Rectangular grids	39
7	Related Work	41
7.1	Shape types	41
7.2	Other shape specification methods	42
7.3	Checking pointer manipulations	43
8	Conclusion	45
8.1	Summary of GRSs	46
8.2	Future work	46
	References	47

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 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 (Specifications of binary trees and full binary trees)

Figure 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. \square

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

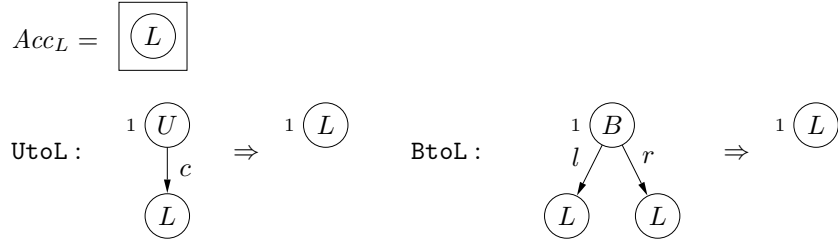


Figure 1: A graph-reduction specification of binary trees.

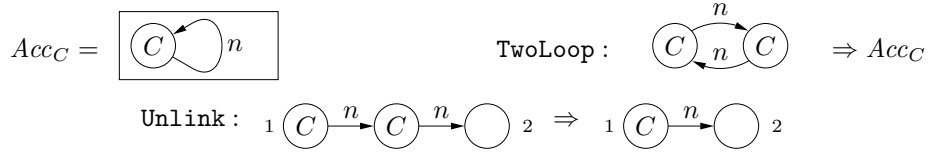


Figure 2: A graph-reduction specification of cyclic lists.

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, which are unable to express properties like balance [HHN92, KLR02, BRS99, KS93, FM97]. PGRSs are also capable of defining shapes with sharing and cycles. Our second introductory example presents cyclic lists.

Example 2 (Specification of cyclic lists)

Figure 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. \square

The rest of this report 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 GRSs can model nil pointers. Section 6 applies our theory to specify many example shapes. Section 7 discusses related work including verifying operations and other specification methods. Section 8 concludes.

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 models of data structures. Section 2.2 defines graphs, rules and derivations as in the double-pushout approach [HMP01]. 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

Definition 1 (Signature)

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

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 \mathcal{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. In the following, Σ always denotes an arbitrary but fixed signature $\langle \mathcal{C}_V, \mathcal{C}_N, \mathcal{C}_E, type \rangle$.

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. \square

2.2 Graph reduction

Definitions 2, 3 and 4 below are consistent with the double-pushout approach to defining labelled graphs, morphisms, rules and derivations (see [HMP01]; [HP02] considers graph relabelling).

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 $s_G, t_G : E_G \rightarrow V_G$ assigning a source and target vertex to each edge; a partial node labelling function $l_G : V_G \rightarrow \mathcal{C}_V$; and a total edge labelling function $m_G : E_G \rightarrow \mathcal{C}_E$. \square

Figure 3 shows two example graphs over Σ_{BT} .

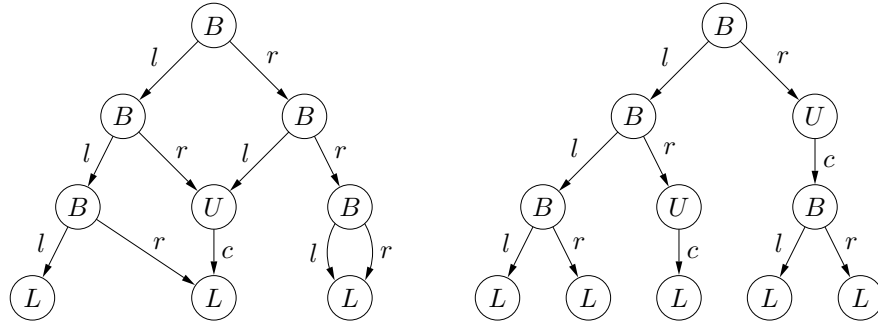


Figure 3: Two Σ_{BT} -total graphs. The right one is a BBT, the left one is not.

Definition 3 (Morphism, inclusion and rule)

A *graph morphism* $g : G \rightarrow H$ consists of a node mapping $g_V : V_G \rightarrow V_H$ and an edge 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 $l_G(x) \neq \perp^1$. 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 $G \cong H$.

A *graph inclusion* $H \supseteq G$ is a graph morphism $g : G \rightarrow H$ 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 . \square

Intuitively, a rule deletes nodes in $L - K$, preserves nodes in K and allocates nodes in $R - K$. In [HMP01] 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 just the set of numbered vertices common to left and right. For example, the interface of $BtoL$ in Figure 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 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 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

¹ $f(x) = \perp$ means f is undefined for x .

$$\text{Re1} : 1 \textcircled{B} \Rightarrow 1 \textcircled{L} \quad \textcircled{B} \xrightarrow{r} \textcircled{L} \Rightarrow_{\text{Re1}} \textcircled{L} \xrightarrow{r} \textcircled{L}$$

Figure 4: A rule **Re1**, which does not respect the BT signature, and the effect of applying it to a graph which does respect the BT signature.

$h : R \rightarrow H'$ 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 $x \in V_K$, the label of $g(x)$ in H' becomes $l_R(x)$. \square

Injectivity of the matching morphism g means that **BtoL** in Figure 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

Definitions 2 and 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 4 (Unrestricted graph reduction is too general)

Figure 4 shows a simple rule **Re1** which relabels a node, and an example derivation in which the relabelling results in a graph containing a leaf with a child. Unrestricted rules could make trees cyclic or give branches multiple left-children. This motivates the following restrictions. \square

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 : $\text{outlabels}_G(v) = \{m_G(e) \mid s_G(e) = v\}$.

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) $\forall v \in V_G \cdot l_G(v) \neq \perp \Rightarrow \text{outlabels}_G(v) \subseteq \text{type}(l_G(v))$. \square

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 $v \in V_G$, $\text{outlabels}_G(v) = \text{type}(l_G(v))$. \square

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 nil pointers are modelled as nullary nodes — the leaves in trees. Alternatives are considered in Section 5. 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 Figure 4, the left graph respects Σ_{BT} and the right graph does not. In Figure 3 both graphs are Σ_{BT} -total. \square

To prevent reduction rules breaking either the signature or the totality of graphs, we define 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. $l_L(x) = \perp \Rightarrow x \in V_K \wedge l_R(x) = \perp \wedge \text{outlabels}_L(x) = \text{outlabels}_R(x)$.

That is, unlabelled nodes in L are preserved and remain unlabelled with the same outlabels.

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

That is, labelled nodes in L which are preserved with the same label have the same outlabels in L and R .

3. $x \in V_K \wedge l_L(x) \neq \perp \wedge l_L(x) \neq l_R(x) \Rightarrow$

$l_R(x) \neq \perp \wedge \text{outlabels}_L(x) = \text{type}(l_L(x)) \wedge \text{outlabels}_R(x) = \text{type}(l_R(x))$.

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. $x \in V_L - V_K \Rightarrow \text{outlabels}_L(x) = \text{type}(l_L(x))$.

That is, deleted nodes have a complete set of outlabels.

5. $x \in V_R - V_K \Rightarrow l_R(x) \neq \perp \wedge \text{outlabels}_R(x) = \text{type}(l_R(x))$.

That is, allocated nodes are labelled and have a complete set of outlabels. \square

Example 6 (Rules specifying balanced binary trees)

Example 7 specifies BBTs with the Σ_{BT} -total rules $\mathcal{R}_{BBT} = \{\text{PickLeaf}, \text{PushBranch}, \text{FellTrunk}\}$, given in Figure 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. \square

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.

Proof

H is constructed as in Definition 4: so $r = \langle L \supseteq K \subseteq R \rangle$, there is a morphism $g : L \rightarrow G$, and there is an intermediate graph

$D = \langle V_G - g(V_L - V_K), E_G - g(E_L - E_K), s_G|_{E_D}, t_G|_{E_D}, l_G|_{V_D}, m_G|_{E_D} \rangle$.

Let $R' = (g \cup h)R$ where morphism h comprises the injective mappings $h_V :$

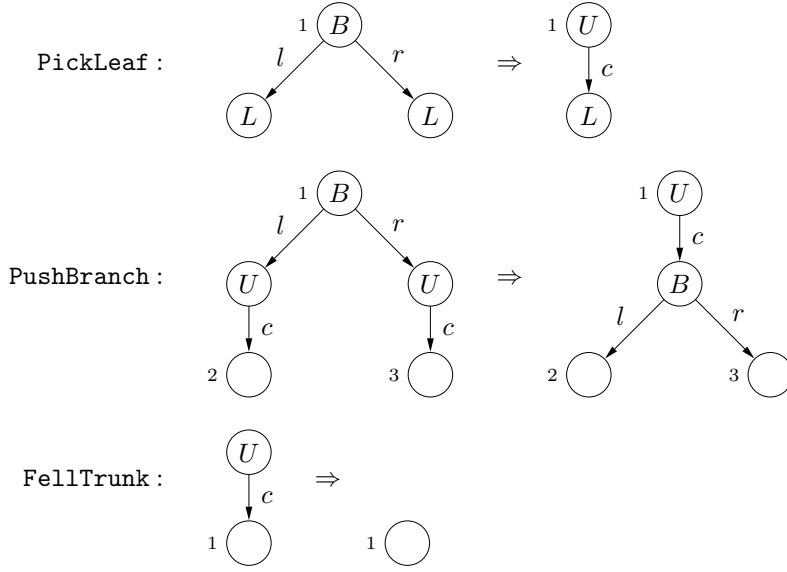


Figure 5: BBT shape specification rules.

$V_R - V_K \rightarrow V - V_D$ and $h_E : E_R - E_K \rightarrow E - E_D$. Then $H \cong H'$ where $H' = \langle V_D \cup V_{R'}, E_D \cup E_{R'}, s_D \cup s_{R'}, t_D \cup t_{R'}, (l_D - l_{gL}) \cup l_{R'}, m_D \cup m_{R'} \rangle$.

Preservation of graph properties: (i) New nodes and edges 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 edges are all assigned a source and target either in V_D or the new vertices; (iii) l_H is a partial function from V_H to \mathcal{C}_V 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 Σ -graph; (iv) m_H is a total function from E_H to \mathcal{C}_E as m_D and $m_{R'}$ are total.

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

Preservation of Σ -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) $\forall v \in V_H \cdot outlabels_H(v) = type\ l_H(v)$ by proof of preservation of Σ .

Reverse direction: if r is a Σ -total rule then $\langle R \supseteq K \subseteq L \rangle$ is also a Σ -total rule. \square

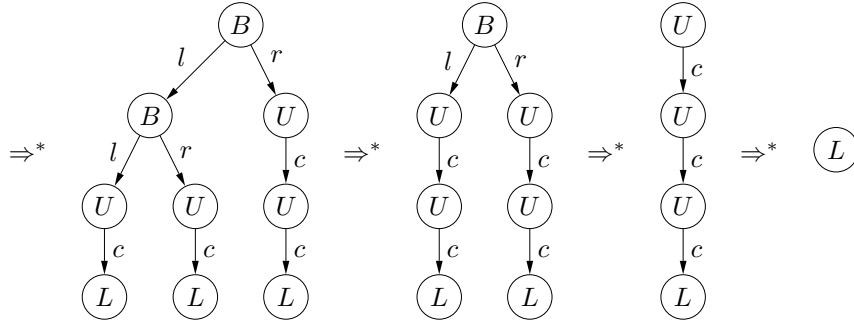


Figure 6: A reduction of the right graph in Figure 3. The steps in the four reduction sequences are: `PickLeaf`, `PickLeaf`; `PushBranch`, `PickLeaf`; `PushBranch`, `PushBranch`, `PickLeaf`; `FellTrunk`, `FellTrunk`, `FellTrunk`.

2.4 Shape specifications

Definition 8 (GRS, NT-free GRS)

A *graph reduction specification* (GRS) $S = \langle \Sigma, \mathcal{R}, Acc \rangle$ consists of a signature Σ , a finite set of Σ -total rules \mathcal{R} and an \mathcal{R} -irreducible Σ -total graph Acc , the *accepting graph*. The *graph language* of S is $\mathcal{L}(S) = \{G \mid G \Rightarrow_{\mathcal{R}}^* Acc \wedge l_G(V_G) \cap \mathcal{C}_N = \emptyset\}$. If $\mathcal{C}_N = \emptyset$ we say that S is *NT-free*. \square

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

Example 7 (Specification of balanced binary trees)

We define BBTs by the NT-free GRS $BBT = \langle \Sigma_{BBT}, \mathcal{R}_{BBT}, Acc_L \rangle$, where \mathcal{R}_{BBT} is defined in Example 6. That is, \mathcal{R}_{BBT} reduces BBTs, and nothing else, to Acc_L . Figure 6 shows an example reduction. The left graph in Figure 3 is irreducible under \mathcal{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 (BBT specifies balanced binary trees)

For every Σ_{BBT} -graph G , $G \in \mathcal{L}(BBT)$ 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 sub-tree(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 8 (Binary tree and cyclic list PGRSs)

We can also give formal PGRSs of the opening examples (Example 1 and Example 2): binary trees, full binary trees and cyclic lists as follows. Their formal proofs are simple (see Theorem 11).

$$\begin{aligned} BT &= \langle \Sigma_{BT}, \{\text{UtoL}, \text{BtoL}\}, Acc_L \rangle \\ FBT &= \langle \Sigma_{BT}, \{\text{BtoL}\}, Acc_L \rangle \\ CLIST &= \langle \langle \{C\}, \{\}, \{n\}, \{C \mapsto \{n\}\} \rangle, \{\text{TwoLoop}, \text{Unlink}\}, Acc_C \rangle \end{aligned} \quad \square$$

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 Lemma 3 later). This follows from Uesu’s result that double-pushout graph grammars can generate every recursively enumerable set of graphs [Ues78]. 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 \mathcal{R} (nondeterministically) as long as possible; G belongs to $\mathcal{L}(S)$ iff the resulting graph is isomorphic to 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 9 (Graph size, polynomially terminating, size-reducing)

Graph size is defined by $size(G) = \#V_G + \#E_G$ where $\#$ denotes set cardinality. A GRS $S = \langle \Sigma, \mathcal{R}, Acc \rangle$ is *terminating* if there is no infinite derivation $G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \dots$. It is *polynomially terminating* if there is a polynomial p such that for every derivation $G \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_n$, $n \leq p(size(G))$. It is *size-reducing* if $G \Rightarrow_{\mathcal{R}} H$ implies that $size(G) > size(H)$. \square

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

Example 9 (Binary DAGs)

Binary DAGs can be specified by giving reduction rules to convert them to trees (see Figure 7) in combination with the normal full binary tree reduction rule:

$$BDAG = \langle \Sigma_{BT}, \{\text{BtoL}, \text{UnsLeaf1}, \text{UnsLeaf2}\}, Acc_L \rangle. \quad \square$$

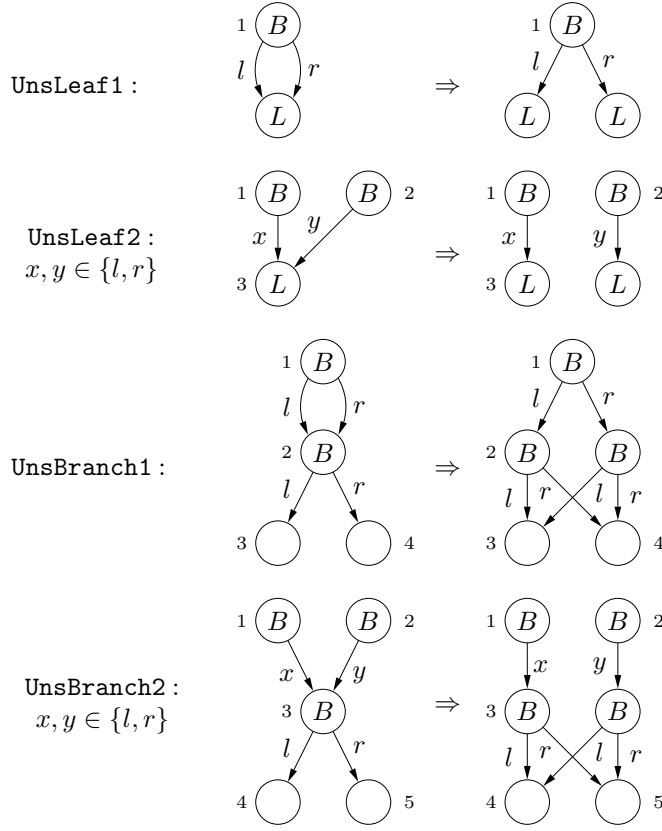


Figure 7: Rules for unsharing the leaves and branches of a binary DAG.

Theorem 3 (Linear termination of $BDAG$)

\mathcal{R}_{BDAG} is linearly terminating.

Proof

Each **UnLeaf** rule in \mathcal{R}_{BDAG} reduces the amount of sharing in a graph G , that is, $\Sigma\{indegree\ v \mid v \in V_G\}$ where $indegree\ v = \#\{e \mid t_G(e) = v\}$. **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. \square

If we allowed unsharing of branches as well as leaves by including **UnBranch1** and **UnBranch2** of Figure 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 reduces as a tree. Worse, a cyclic graph could expand without limit using the **UnBranch2** 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 require individual termination analysis.

3.2 Closedness, confluence and complexity

Definition 10 (Closedness, Confluence, PGRS)

A GRS $S = \langle \Sigma, \mathcal{R}, Acc \rangle$ is *closed* if for every $G \in \mathcal{L}(S)$, $G \Rightarrow_{\mathcal{R}} H$ implies $H \in \mathcal{L}(S)$. S is *confluent* if for every pair of derivations $H_1 \xrightarrow{*}_{\mathcal{R}} G \xrightarrow{*}_{\mathcal{R}} H_2$ over Σ , there is a graph H such that $H_1 \xrightarrow{*}_{\mathcal{R}} H \xrightarrow{*}_{\mathcal{R}} H_2$. A polynomially terminating and closed GRS is a *polynomial GRS*, PGRS for short. \square

Clearly confluence implies closedness (the converse does not hold).

Theorem 4 (Complexity of testing membership)

If S is a PGRS then membership of $\mathcal{L}(S)$ is decidable in polynomial time.

Proof

We assume S is fixed, so the number of rules is fixed and the size of the largest left graph in \mathcal{R} is a constant c . Checking whether any rule in \mathcal{R} matches a graph G requires $O(\text{size}(G)^c)$ time. This is because there are at most $\text{size}(G)^c$ 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 the closedness of S makes backtracking unnecessary. \square

Confluence of a terminating specification can be shown by adapting the *critical pair method* of [Plu93] to GRSs. Two reduction rules form a critical pair if they can be applied to the same graph in such a way that one rule removes part of the graph required to apply the other rule.

Definition 11 (Critical pair)

Let $r_i = (L_i \supseteq K_i \subseteq R_i)$ be rules for $i = 1, 2$. A pair of direct derivations $T \xrightarrow{r_1, g_1} S \xrightarrow{r_2, g_2} U$ is a critical pair if $S = g_1 L_1 \cup g_2 L_2$ and $g_1 L_1 \cap g_2 L_2 \neq g_1 K_1 \cup g_2 K_2$. Furthermore, if $r_1 = r_2$ then $g_1 \neq g_2$. \square

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 10 (Critical pair)

Figure 8 shows a harmless additional reduction rule `Unlink2` that could be used for the reduction of cyclic lists. But its addition to \mathcal{R}_{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 `Unlink`. \square

If there are critical pairs the following lemma may be used to show confluence of a terminating reduction system. This is a sufficient test as confluence is undecidable in general [Plu93].

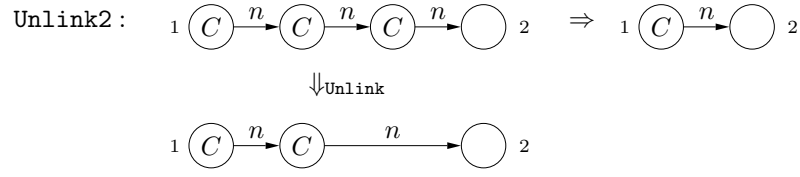


Figure 8: A cyclic list reduction rule `Unlink2` and one of the critical pairs it forms with `Unlink`.

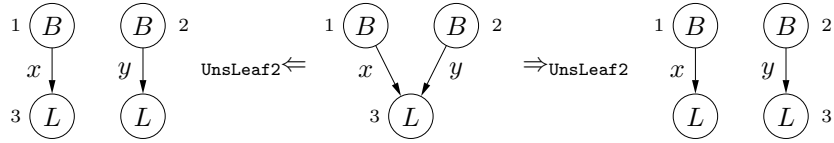


Figure 9: A non-strongly-joinable critical pair of `UnsLeaf2`.

Lemma 1 (Critical pair lemma)

A reduction system is locally confluent if all its critical pairs are *strongly joinable*.

A critical pair $T \leftarrow S \Rightarrow U$ is strongly joinable if there are derivations $T \Rightarrow^* X \leftarrow^* U$ such that for all nodes v in $\text{Protect}(S)$, $\text{track}_{S \Rightarrow T \Rightarrow^* X}(v)$ and $\text{track}_{S \Rightarrow U \Rightarrow^* X}(v)$ 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 *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 4. 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: $\text{track}_{G \Rightarrow^* H} = i$ if $G \Rightarrow^* H$ by an isomorphism i ; $\text{track}_{G \Rightarrow^* H} = \text{track}_{H' \Rightarrow^* H} \circ \text{track}_{G \Rightarrow H'}$ if $G \Rightarrow^* H$ by a sequence $G \Rightarrow^* H' \Rightarrow H$.

Proof

See [Plu93]. □

Example 11 (Showing confluence)

The critical pairs of `Unlink` and `Unlink2` are all strongly joinable. For the example in Figure 8, the right graph is derived from one application of `Unlink` to the lower graph. The *Protect* nodes are 1 and 2 which are preserved by both derivation sequences. □

Example 12 (Non-strongly-joinable critical pair)

Although intuitively, the *BDAG* specification in Figure 7 is confluent, it is beyond the scope of the critical pair lemma to show this. Figure 9 shows a critical pair of `UnsLeaf2` and `UnsLeaf2`: the left and right derived graphs are both irreducible, they are isomorphic but the *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. \square

Several of our examples have critical pairs, and some have non-strongly-joinable critical pairs. But we conjecture that all our examples are closed. Closedness can be tested by disregarding any 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

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 can be specified as NT-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 NT-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 NT-free PGRSs and PGRSs, but terminating and possibly non-confluent GRSs are closed under union (provided $Acc \neq \emptyset$). Section 4.3 shows that complement extends the range of shapes definable by NT-free (P)GRSs and (P)GRSs.

4.1 Intersection

Complete binary trees (CBTs) are BBTs where every branch is binary. Theorem 5 says they cannot be defined by an NT-free GRS. Lemma 2 presents a general method for showing that an NT-free GRS cannot define a given shape.

Lemma 2 (Proving graph languages are undefinable)

Graph language \mathcal{L} cannot be defined by an NT-free GRS if:

$$\forall k \in \mathbb{N}, \mathcal{R} \subseteq \mathcal{L} \times \mathcal{L} \cdot \max\{\delta(G, H) \mid (G, H) \in \mathcal{R}\} \geq k \vee \mathcal{R}^* \neq \mathcal{L} \times \mathcal{L}$$

where $\delta(G, H) = \min\{\max\{size(L), size(R)\} \mid r = \langle L \supseteq K \subseteq R \rangle, G \Rightarrow_r H\}$.

Proof

To be definable by an NT-free GRS, there must be a finite set of rewrite rules defining a relation \mathcal{R} such that every graph $G \in \mathcal{L}$ relates to some other graph $H \in \mathcal{L}$. Finiteness means there is some bound k on the size of rules defining \mathcal{R} . Further, the transitive closure of \mathcal{R} must include every pair of graphs in \mathcal{L} if a finite set of rules can rewrite every \mathcal{L} -graph to a single accepting graph. We can write this: $\exists k \in \mathbb{N}, \mathcal{R} \subseteq \mathcal{L} \times \mathcal{L} \cdot \max\{\delta(G, H) \mid (G, H) \in \mathcal{R}\} < k \wedge \mathcal{R}^* = \mathcal{L} \times \mathcal{L}$, the negation gives this lemma. \square

Conversely, if there is a finite set of rules covering \mathcal{L} , there is not necessarily an NT-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 2 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 5 (CBTs cannot be defined by an NT-free GRS)

No NT-free GRS can specify complete binary trees.

Proof

By Lemma 2: 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. \square

We can often make a language specifiable by using non-terminals. Alternatively, we can take the intersection of two NT-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 13 (Specification of complete binary trees)

Let $CBT = \langle \Sigma_{BT} + \{\}, \{U\}, \{\}, \{\}, \mathcal{R}_{BBT}, Acc_L \rangle$. Hence CBTs are BBTs which do not contain any unary branches. \square

We define GRS language intersection in the obvious way.

Definition 12 (Intersection of GRS languages)

If S and T are GRSs then $\mathcal{L}(S \cap T) = \mathcal{L}(S) \cap \mathcal{L}(T)$. \square

Example 14 (CBTs by intersection)

Let $CBT = FBT \cap BBT$. CBTs are full binary trees (left conjunct) and they are balanced (right conjunct). Note that both GRSs are NT-free. \square

By Theorem 5 and Example 14, the languages of NT-free (P)GRSs are not closed under intersection. Theorem 6 shows that (P)GRSs and intersections of NT-free (P)GRSs have equivalent power. Theorem 7 shows that (P)GRSs are closed under intersection.

Theorem 6 (GRSs equivalent to intersections of NT-free GRSs)

1. If N is a GRS there are NT-free 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 NT-free GRSs there is a 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 an NT-free (P)GRS S

which does the same as N but treats all labels as terminals and another NT-free PGRS T which accepts exactly all NT-free graphs. The details work out as follows.

Let $N = \langle \langle \mathcal{C}_V, \mathcal{C}_N, \mathcal{C}_E, type \rangle, \mathcal{R}, Acc \rangle$. Let $\mathcal{C}_T = \mathcal{C}_V - \mathcal{C}_N$ be the set of terminal node labels. Let S be $\langle \langle \mathcal{C}_V, \{\}, \mathcal{C}_E, type \rangle, \mathcal{R}, Acc \rangle$, the same specification where no labels are non-terminals. Let T be $\langle \langle \mathcal{C}_V, \{\}, \mathcal{C}_E, type \rangle, \mathcal{R}', \emptyset \rangle$, where \emptyset denotes the empty graph and $\mathcal{R}' = \{\text{DeArc}(x) \mid x \in \mathcal{C}_T\} \cup \{\text{DeNode}(x) \mid x \in \mathcal{C}_T\}$ where:

$$\begin{aligned} \text{DeArc}(x) : \quad & \begin{array}{c} y \\ \text{y} \in \text{type } x \end{array} \quad \begin{array}{c} 1 \quad x \quad y \quad \text{---} \quad \text{---} \quad 2 \end{array} \Rightarrow \begin{array}{c} 1 \quad x \quad \text{---} \quad y \quad \text{---} \quad 2 \end{array} \\ \text{DeNode}(x) : \quad & \begin{array}{c} \text{y}_1, \dots, \text{y}_n \\ \text{y}_1, \dots, \text{y}_n \end{array} \quad \begin{array}{c} x \\ \text{---} \quad \text{---} \quad \text{---} \end{array} \Rightarrow \emptyset \end{aligned}$$

So $\#\mathcal{R}' = \#\mathcal{C}_T + \sum_{x \in \mathcal{C}_T} \#\text{type } x$. 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 NT-free GRSs where $\Sigma_S = \Sigma_T = \langle \mathcal{C}_V, \{\}, \mathcal{C}_E, type \rangle$. Let $\mathcal{C}'_V, \mathcal{C}''_V$ and \mathcal{C}'''_V be distinct renamings of \mathcal{C}_V and L and N be new labels not occurring in any of these sets; p and q are new edge labels. Our signature is:

$$\begin{aligned} \Sigma_N = \langle & \mathcal{C}_V \cup \mathcal{C}_N, \mathcal{C}_N, \mathcal{C}_E \cup \{p, q\}, \{L \mapsto \{p, q\}, N \mapsto \{\}\} \cup \\ & \cup \{l' \mapsto \{p, q\} \cup t, l \mapsto t, l'' \mapsto t, l''' \mapsto t \mid l \in \mathcal{C}_V, t = \text{type } l\} \\ & \text{where } \mathcal{C}_N = \mathcal{C}'_V \cup \mathcal{C}''_V \cup \mathcal{C}'''_V \cup \{L, N\} \end{aligned}$$

The rules in Figure 10 create two distinctly-named copies of any Σ_S -total graph. **DupNode** replaces each \mathcal{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. It is possible that the arcs a_i could be loops, or several of them could point to the same node, so for node copying we actually require all the *quotients* [HMP01] of each **DupNode** rule, these are given by $\text{DupN} = \bigcup \{Q(\text{DupNode}(l)) \mid l \in \mathcal{C}_V\}$.

When a node and all its successors have been copied its arcs can be copied. **DupArcs** relabels and adds arcs to such nodes. Again, arcs could be loops or shared and the copying must preserve this so we require all the rules $\text{DupA} = \{g(\text{DupArcs}(l)) \mid l \in \mathcal{C}_V, g \text{ is a surjective graph morphism s.t. } \forall i, j \cdot g(i1) \neq g(j1)\}$

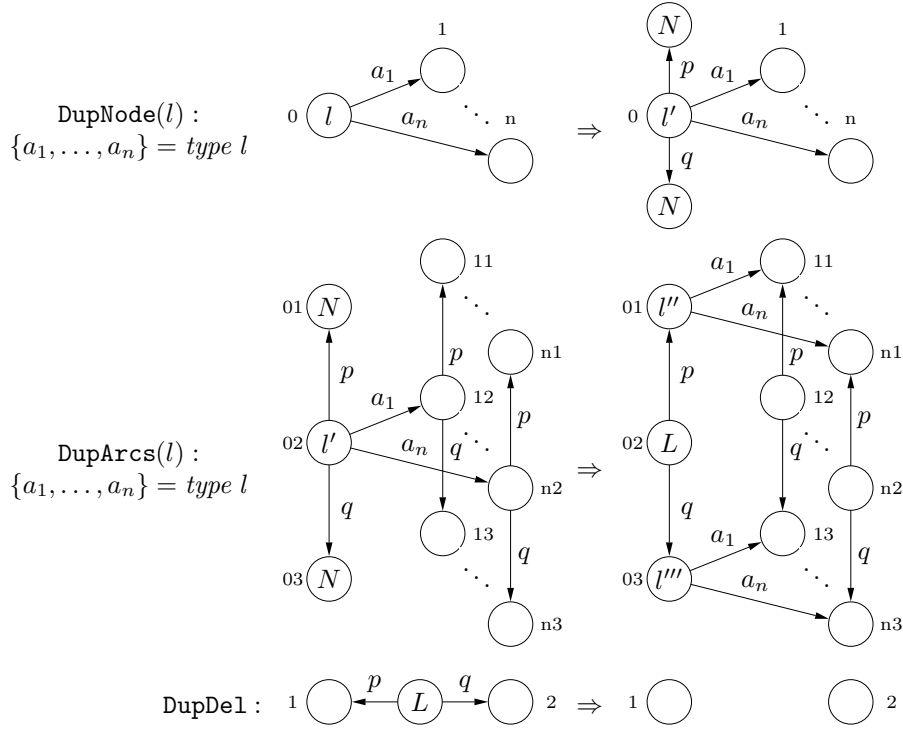


Figure 10: Rules to make two copies of any Σ_S -total graph.

$g(j2) \wedge g(i2) \neq g(j3) \wedge g(i1) \neq g(j3)$ and $\forall i, j \cdot g(2i) = g(2j) \Leftrightarrow g(1i) = g(1j) \wedge g(3i) = g(3j)$ 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:

$$N = \langle \Sigma_N, \text{DupN} \cup \text{DupA} \cup \{\text{DupDel}\} \cup \mathcal{R}_S'' \cup \mathcal{R}_T''', \text{Acc}_S'' \cup \text{Acc}_T''' \rangle$$

N has $2 + 3 \times \#\mathcal{C}_V$ new non-terminal labels and $1 + 2 \times \#\text{DupN}$ 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; and our insistence that relabelled nodes must have a full set of outlabels, so the node copying rule has to mention arcs. Copying is linearly terminating as DupNode and 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 so N is confluent if S and T are confluent. \square

Theorem 7 (Graph-reduction languages closed under intersection)

If S and T are (P)GRSs, then $\mathcal{L}(S \cap T)$ can be defined as a single (P)GRS N . Moreover, if S and T are confluent (or closed) 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

A trivial extension of the argument in Theorem 6 part 2: only nodes labelled with terminals are copied. \square

4.2 Union

Definition 13 (Union of graph-reduction languages)

If S and T are GRSs then $\mathcal{L}(S \cup T) = \mathcal{L}(S) \cup \mathcal{L}(T)$. \square

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 PGRS cannot define a finite language that includes the empty graph (see Lemma 3 later). Union offers a simple way to specify such a language as a union of PGRSs with no reduction rules whose accepting graphs are the elements of the language. This non-closure property is not restricted to finite languages.

Theorem 8 (Non-closure under union)

PGRSs and NT-free PGRSs defining infinite 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 NT-free PGRSs: $D23 = \langle \Sigma_{BT}, \{D_2 \Rightarrow \emptyset\}, \emptyset \rangle \cup \langle \Sigma_{BT}, \{D_3 \Rightarrow \emptyset\}, \emptyset \rangle$. To construct a single NT-free PGRS to accept the same language we must define $Acc = D_{6n}$ for some n . As D_{6n+2} and D_{6n+3} must both reduce to D_{6n} it follows that D_{6n+5} , 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 Acc_S then D_3 reduces to $D_1 \cup Acc_S$; as S is closed $D_1 \cup Acc_S$ must reduce to Acc_S and therefore D_i reduces to Acc_S for every $i \geq 2$. \square

We can replace a union by a single specification if we 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 9. A technicality (the restriction that accepting graphs should be 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 8.

$$\begin{array}{l}
\text{URule}(L \Rightarrow R, i) : \quad L \Rightarrow R \quad \textcircled{i} \\
\text{Del}(i) : \quad \textcircled{i} \quad \textcircled{i} \Rightarrow \textcircled{i} \\
\text{Accept}(Acc) : \quad Acc \Rightarrow Acc_U \\
\text{Accept}(Acc, i) : \quad Acc \quad \textcircled{i} \Rightarrow Acc_U \\
Acc_U = \quad \boxed{\textcircled{A}}
\end{array}$$

Figure 11: Rules and accepting graph specifying a union of GRSs.

Theorem 9 (Closure under union)

If S and T are (perhaps non-confluent) reduction specifications and $\emptyset \notin \mathcal{L}(S) \cup \mathcal{L}(T)$ then there is a reduction specification 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 $S = \langle \Sigma, \mathcal{R}_1, Acc_1 \rangle$ and $T = \langle \Sigma, \mathcal{R}_2, Acc_2 \rangle$. The new signature has three new non-terminals: $\Sigma_U = \Sigma + \langle \{1, 2, A\}, \{1, 2, A\}, \{\}, \{1 \mapsto \emptyset, 2 \mapsto \emptyset, A \mapsto \emptyset\} \rangle$.

The reduction rules are modified as shown in Figure 11. When a rule from \mathcal{R}_i is used an i -labelled node is added to the graph. The new **Del** rules replace two i nodes by a single i node; The **Accept** rules rewrite an original accepting graph Acc_i or an original accepting graph with one i node to the new accepting graph.

$$\mathcal{R}_U = \{ \text{URule}(r, i) \mid i \in \{1, 2\}, r \in \mathcal{R}_i \} \cup \bigcup_{i=1}^2 \{ \text{Del}(i), \text{Accept}(Acc_i), \text{Accept}(Acc_i, i) \}$$

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}_1 and \mathcal{R}_2 can lead to Acc_U . This scheme preserves termination: we just need a **Del** step after all but the first reduction and an **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 **Del** steps must occur before the **Accept** step. \square

4.3 Complement, \emptyset -languages and Chomsky grammars

Definition 14 (Complement of graph-reduction languages)

If S is a GRSs then $\mathcal{L}(\overline{S}) = \{G \mid G \text{ is } \Sigma_S\text{-total}\} - \mathcal{L}(S)$. \square

(P)GRSs and NT-free (P)GRS are not closed under complement. Here we show why this is the case and present some simple principles for showing that a language has an undefinable complement. First we prove some useful properties about languages including \emptyset mentioned earlier.

Lemma 3 (Properties of GRS languages including \emptyset)

Let S be a GRS with $\emptyset \in \mathcal{L}(S)$.

1. $Acc_S = \emptyset$.

2. $\mathcal{L}(S)$ is closed under disjoint union of graphs.
3. $\mathcal{L}(S)$ is infinite if it is not $\{\emptyset\}$.

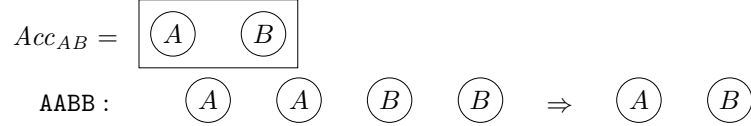
Proof

1. If $Acc_S \neq \emptyset$ then $\emptyset \Rightarrow^* Acc_S$ and therefore Acc_S is reducible. Therefore if 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 $n \in \mathbb{N}$. \square

It follows that a language cannot be defined if it includes \emptyset and it is not closed under disjoint union.

Example 15 (Language with undefinable complement)

Consider the NT-free PGRS $AB = \langle \Sigma, \{\mathbf{AABB}\}, Acc_{AB} \rangle$ where Σ contains the nullary terminals A and B only.



$\mathcal{L}(AB)$ contains all non-empty Σ -graphs which have the same number of A and B nodes. Therefore $\overline{\mathcal{L}(AB)}$ includes \emptyset and the graph containing one A only and the graph containing one B only. As $\overline{\mathcal{L}(AB)}$ 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 3. \square

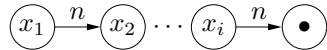
So even the complement of an NT-free and size-reducing PGRS cannot always be defined. If we allow non-terminating GRSs to have reducible accepting graphs then the language $\overline{\mathcal{L}(AB)}$ in Example 15 can be defined. But in general, GRSs are not closed under complement. This can be seen from their ability to simulate Chomsky grammars.

Lemma 4 (Simulation of Chomsky grammars)

A Chomsky grammar $C = \langle V, N, P, S \rangle$ consists of 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 $\mathcal{L} = \{w \in (V - N)^* \mid S \xrightarrow{P} w\}$ where \xrightarrow{P} denotes the replacement of the left side of a production by its right side in a string. There is a GRS whose language is \mathcal{L} .

Proof

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



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

$$\mathbf{p}^\bullet : 1 \text{ (} x_1 \text{)} \xrightarrow{n} \text{ (} x_2 \text{)} \cdots \text{ (} x_i \text{)} \xrightarrow{n} \text{ (} \text{)} 2 \Rightarrow 1 \text{ (} y_1 \text{)} \xrightarrow{n} \text{ (} y_2 \text{)} \cdots \text{ (} y_j \text{)} \xrightarrow{n} \text{ (} \text{)} 2$$

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

Corollary 1 (Consequences of Lemma 4)

1. GRSs can specify every recursively enumerable set of strings.
2. GRSs can specify graph languages with undecidable membership problems.
3. GRSs are not closed under complement. \square

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 16 (Full binary trees with a shared leaf)

Full binary trees with a single shared leaf are defined by the following PGRS.

$$SFBT = \langle \Sigma_{BT}, \{\text{BLtoL}, \text{OneBL}\}, Acc_L \rangle$$

The reduction rules are given in Figure 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. \square

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

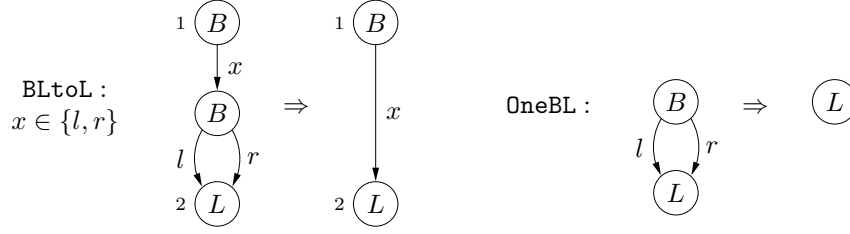


Figure 12: Reduction rules for full binary trees with a shared leaf.

hand, branching structures that have shared sub-trees are likely to have simpler specifications if we follow the shared leaf convention.

5.2 Σ -partial specifications

Graphs offer another obvious model for the nil pointer: let non-total Σ -graphs model structures and any missing arc is a nil pointer. We briefly investigate the possibilities of this approach here. First we need to change Definition 7 so languages can include such partial graphs.

Definition 15 (Σ -partial rule)

A rule $\langle L \supseteq K \subseteq R \rangle$ is Σ -partial if L, R are Σ -graphs and:

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

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

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

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

3. $x \in V_K \wedge l_L(x) \neq \perp \wedge l_L(x) \neq l_R(x) \Rightarrow$

$l_R(x) \neq \perp \wedge \text{outlabels}_L(x) \supseteq (\text{type}(l_L(x)) - \text{type}(l_R(x))) \wedge$
 $\text{outlabels}_R(x) \subseteq \text{outlabels}_L(x) \cup (\text{type}(l_R(x)) - \text{type}(l_L(x)))$

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 Σ -graph.

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

Theorem 10 (Σ -partial rules preserve Σ)

If G is a Σ -graph and r is a Σ -partial rule and $G \Rightarrow_{r,g} H$ then H is a Σ -graph.

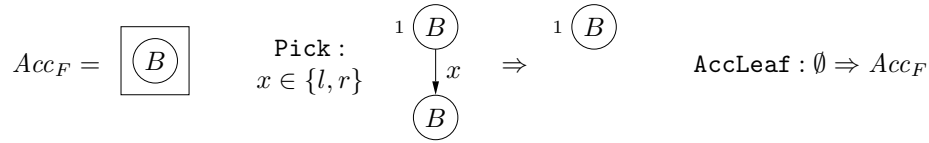


Figure 13: A Σ -partial specification of full binary trees.

Proof

Follows from Definition 15. □

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, such a GRS is not possible: if $Acc = \emptyset$ the the specification will define forests (see Lemma 3). Alternatively, if we make Acc the singleton then we can have a reduction rule to rewrite \emptyset to the singleton.

Example 17 (Partial full binary trees)

Full binary trees are defined by $PFBT = \langle \Sigma_{BT}, \{\text{Pick}, \text{AccLeaf}\}, \text{Acc}_F \rangle$ (see Figure 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. The empty tree is recognised by applying **AccLeaf**. The accepting graph is reducible, so this specification is not a GRS. This specification is non-terminating because **AccLeaf** can be applied to any graph any number of times. A terminating specification of this language is not possible by Lemma 3.

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. □

Example 5 illustrates why we prefer to model data structures as total rather than partial graphs: including \emptyset 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 Example 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 [CLR90]) and a proof that the PGRS is sound and complete relative to the

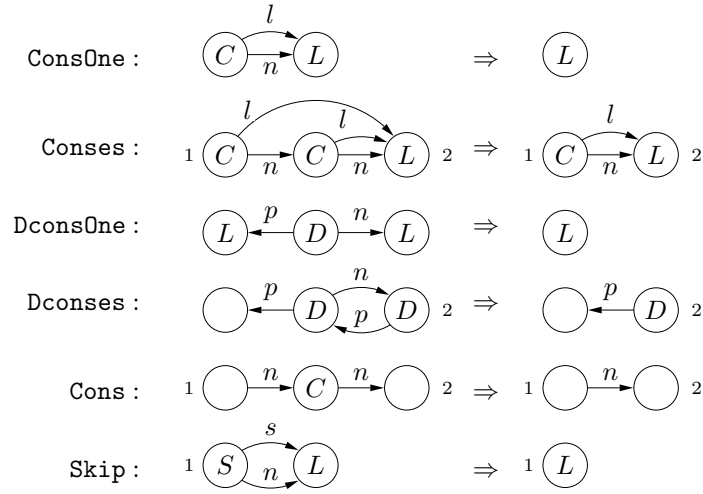


Figure 14: Reduction rules for list variant shapes.

property specification. The aim is always to present the 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 8.1 provides a summary of all the specifications.

The proof of correctness is usually an instance of the following.

Theorem 11 (GRS correctness argument)

NT-free GRS S is a sound and complete specification of language \mathcal{L} whose members satisfy p if (1) Acc_S is the single smallest member of \mathcal{L} , (2) every rule in \mathcal{R}_S preserves p in both directions and (3) every non- Acc_S member of \mathcal{L} is reducible. If GRS S uses non-terminals we require in addition that every NT-free graph satisfying p is in \mathcal{L} .

Proof

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

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.

Definition 16 (PGRS of linked lists)

$$LIST = \langle \Sigma_{BT}, \{\text{UtoL}\}, Acc_L \rangle \quad \square$$

The following definitions provide PGRSs of some simple list variants taken from [CLR90, KS93, FM98]. The reduction rules are shown in Figure 14. They are all correct (as are *LIST* and *CLIST*) by Theorem 11. They are all strongly confluent, size reducing and NT-free.

Definition 17 (Last-element lists)

Every cons node has an n -arc to the next list element and an l -arc to the last element. $\Sigma_{LAST} = \langle \{C, L\}, \{\}, \{n, l\}, \{C \mapsto \{n, l\}, L \mapsto \{\}\} \rangle$
 $LAST = \langle \Sigma_{Last}, \{\text{ConsOne}, \text{Conses}\}, Acc_L \rangle \quad \square$

Definition 18 (Doubly-linked lists)

Every double-cons node has an n -arc to the next list element and a p -arc to the previous list element. $\Sigma_{DLIST} = \langle \{D, L\}, \{\}, \{n, p\}, \{D \mapsto \{n, p\}, L \mapsto \{\}\} \rangle$
 $DLIST = \langle \Sigma_{DList}, \{\text{DconsOne}, \text{Dconses}\}, Acc_L \rangle \quad \square$

Definition 19 (Skip lists)

Every cons node has an n -arc to the next list element. Every skip-cons node also has an s -arc to the next skip-cons (or the end of the list in the case of the last skip-cons). The first node is a skip-cons or a leaf.
 $\Sigma_{SKIP} = \langle \{S, C, L\}, \{\}, \{n, l\}, \{C \mapsto \{n\}, S \mapsto \{n, l\}, L \mapsto \{\}\} \rangle$
 $SKIP = \langle \Sigma_{Skip}, \{\text{Cons}, \text{Skip}\}, Acc_L \rangle \quad \square$

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

Definition 20 (Threaded trees)

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. \square

Definition 21 (NT-free size-reducing PGRS of threaded trees)

$$\Sigma_{TT} = \langle \{R, T, L, N\}, \{\}, \{t, l, r, n\}, \{R \mapsto \{n, t\}, T \mapsto \{l, r, n\}, L \mapsto \{\}, N \mapsto \{\}\} \rangle$$

$$TT = \langle \Sigma_{TT}, \mathcal{R}_{TT}, Acc_{TT} \rangle$$

Acc_{TT} and \mathcal{R}_{TT} are shown in Figure 15. **Cast** reduces the singleton TT to the empty TT . **LeftStitch** replaces a branch of leaves which is a left child by a

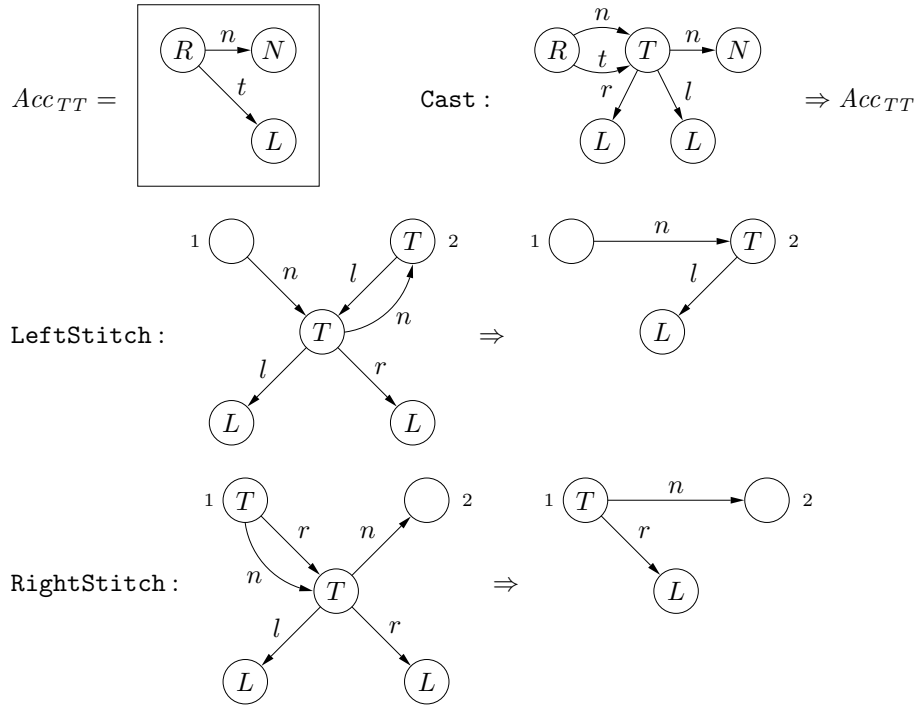


Figure 15: Threaded tree accepting graph and reduction rules.

single leaf. It preserves the threaded tree properties by moving the next pointer of its predecessor to its successor. **RightStitch** works analogously. \square

TT is size-reducing; it is correct by Theorem 11; it is not confluent as **LeftStitch** and **RightStitch** can reduce graphs which are not in $\mathcal{L}(TT)$ to distinct and irreducible graphs. However, we conjecture that it is closed.

6.2.2 Linked-leaf trees

Definition 22 (Linked-leaf trees)

1. Linked-leaf trees are full binary trees with a root node and a nil node.
2. The root node points to the top of the tree and its left-most leaf.
3. Each leaf points to the next leaf encountered in an in-order traversal.
4. The right-most leaf points to the nil node. \square

Definition 23 (A PGRS of linked-leaf trees)

$$\Sigma_{TLEAF} = \langle \{R, B, L, N\}, \{\}, \{l, r, n, o\}, \\ \{R \mapsto \{n, o\}, B \mapsto \{l, r\}, L \mapsto \{n\}, N \mapsto \{\}\} \rangle$$

$$TLEAF = \langle \Sigma_{TLEAF}, \{TBranch\}, Acc_{TLEAF} \rangle$$

The reduction rule and accepting graph are shown in Figure 16. \square

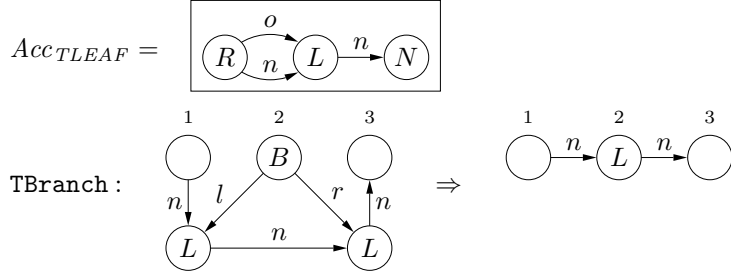


Figure 16: Linked-leaf tree accepting graph and reduction rule.

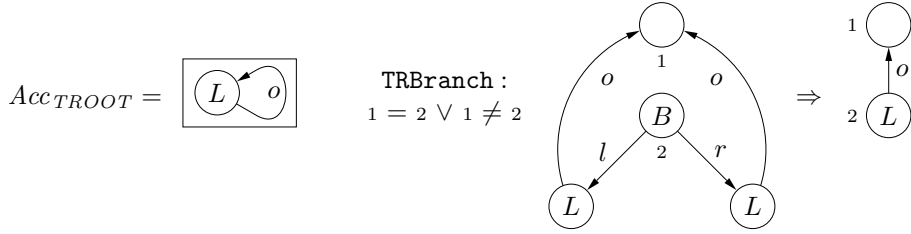


Figure 17: Root-connected tree accepting graph and reduction rule.

$TLEAF$ is size reducing; it is correct by Theorem 11; it is not confluent in general but we conjecture it is closed.

6.2.3 Root-connected trees

Definition 24 (Root-connected trees)

1. Root-connected trees are full binary trees.
2. Every leaf points to the root. □

Definition 25 (A PGRS of root-connected trees)

$\Sigma_{TROOT} = \langle \{B, L\}, \{\}, \{l, r, o\}, \{B \mapsto \{l, r\}, L \mapsto \{o\}\} \rangle$

$TROOT = \langle \Sigma_{TROOT}, \{TRBranch\}, Acc_{TROOT} \rangle$

The reduction rule and accepting graph are shown in Figure 17. □

$TROOT$ is size-reducing and strongly confluent; it is correct by Theorem 11.

6.3 Balanced n -ary trees

Balanced binary trees have a simple graph-reduction specification but they are very difficult to program with because so much re-arrangement is needed after an insertion or deletion. Allowing higher degrees of branching solves this problem. Here we show how to generalise the BBT specification to such trees.

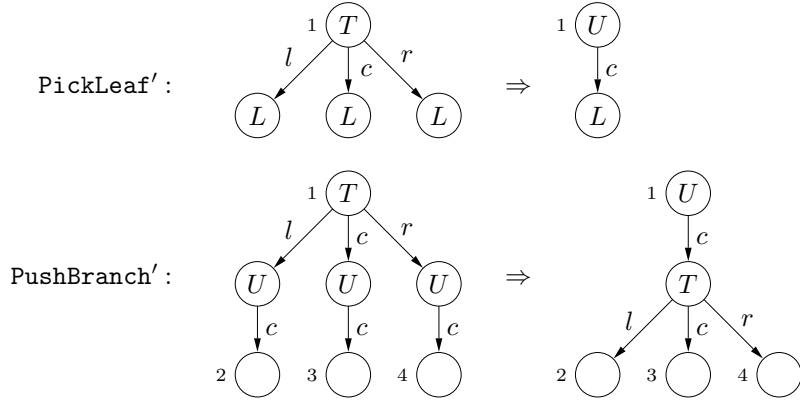


Figure 18: 2-3 tree reduction rules.

6.3.1 2-3 trees

2-3 trees are perhaps the simplest kind of balanced tree with practicable insertion and deletion algorithms (see [Rea92] for example).

Definition 26 (2-3 tree properties)

1. 2-3 tree nodes can be 2 or 3-way branches, or leaves.
2. All leaves have the same depth. □

Definition 27 (A PGRS of 2-3 trees)

$$\Sigma_{23} = \Sigma_{BBT} + \langle \{T\}, \{\}, \{\}, \{T \mapsto \{l, c, r\}\} \rangle$$

$$23 = \langle \Sigma_{23}, \mathcal{R}_{BBT} \cup \{\text{PickLeaf}', \text{PushBranch}'\}, \text{Acc}_L \rangle$$

Σ_{23} extends Σ_{BBT} with a ternary branch. \mathcal{R}_{23} uses the BBT reduction rules and the two new rules in figures 18. **PickLeaf** and **PickLeaf'** reduce branches which are leaf-parents to a 1-branch leaf-parent. **PushBranch** and **PushBranch'** move branches down towards the leaves. **FellTrunk** reduces the height of a tree whose root is a 1-branch. □

23 is strongly confluent and size reducing; it is correct by Theorem 11 where property p defines all 1-2-3 trees. This specification can easily be extended 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 (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 whether an NT-free specification of 2-3 trees is possible.

6.3.2 2-3-4 trees

2-3-4 trees can be specified by generalising the 2-3 specification. But here we show how a slightly shorter, and NT-free, specification is possible.

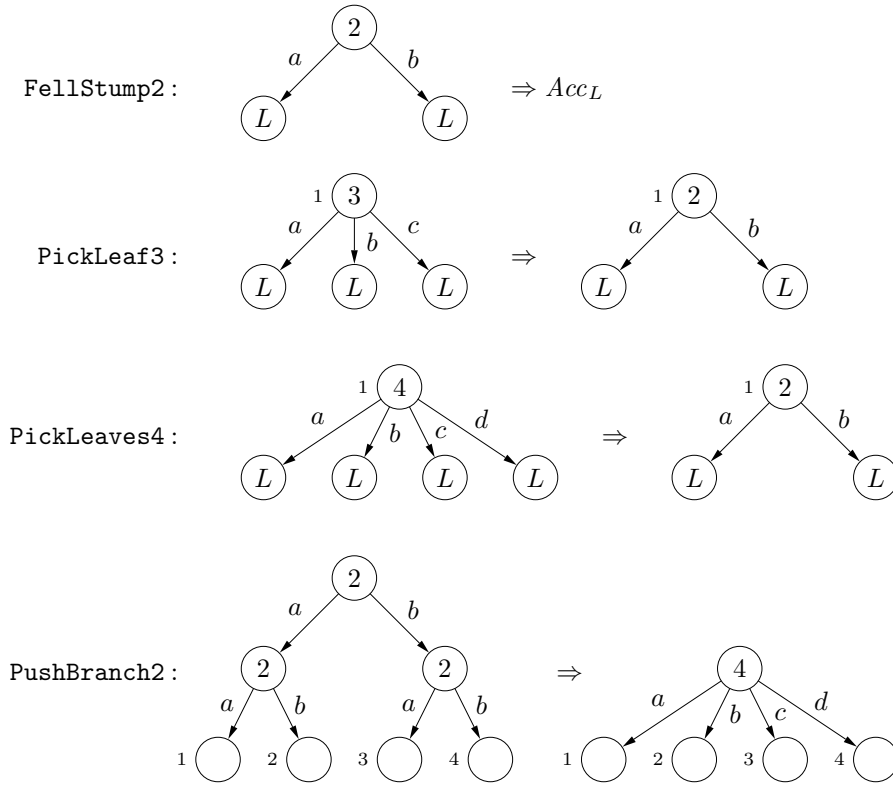


Figure 19: 2-3-4 tree reduction rules 1.

Definition 28 (2-3-4 tree properties)

1. 2-3-4 tree nodes can be 2, 3 or 4-way branches, or leaves.
2. All leaves have the same depth. □

Definition 29 (NT-free PGRS of 2-3-4 trees)

$$\Sigma_{234} = \langle \{2, 3, 4, L\}, \{\}, \{a, b, c, d\}, \{2 \mapsto \{a, b\}, 3 \mapsto \{a, b, c\}, 4 \mapsto \{a, b, c, d\}, L \mapsto \{\}\} \rangle$$

$$234 = \langle \Sigma_{234}, \mathcal{R}_{234}, Acc_L \rangle$$

\mathcal{R}_{234} comprises the six rules in figures 19 and 20. **FellStump2** reduces a 2-branch 'stump' to Acc_L . **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

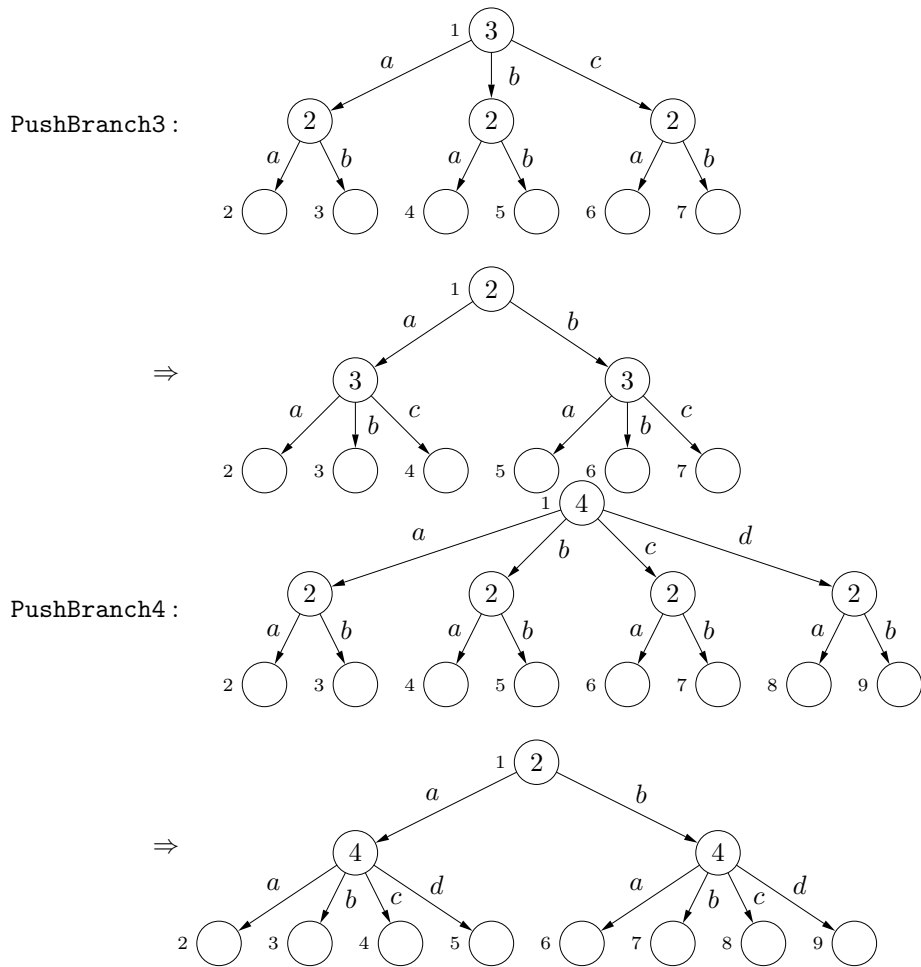


Figure 20: 2-3-4 tree reduction rules 2.

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 CBTs no non-terminals are needed owing to the way `PushBranch2` works. \square

234 is size reducing and strongly confluent; it is correct by Theorem 11.

6.4 Red-black trees

Definition 30 (Red-black tree specification)

1. Red-black trees are trees of binary-branches and leaves.
2. Branches are labelled red or black.

3. Children of red branches are black or leaves.
4. All paths from the root to a leaf have the same number of black nodes. \square

6.4.1 An NT-free PGRS

Our simplest specification is interesting because it is NT-free and terminating but it is not size-reducing. By Lemma 5 (a simplification of Lemma 2) we show in Theorem 12 that a size-reducing specification needs non-terminals. Such a specification is given in Section 6.4.2; compared to the NT-free PGRS it has more rules but terminates in about half as many steps.

Lemma 5 (Languages undefinable as size-reducing NT-free GRSs)

If $\forall k \in \mathbb{N} \cdot \exists G \in \mathcal{L} \cdot \forall G' \in \mathcal{L} \cdot \text{size}(G') < \text{size}(G) \Rightarrow \delta(G, G') \geq k$
then language \mathcal{L} cannot be defined by a size-reducing NT-free GRS.

Proof

To be definable by a size-reducing NT-free GRS there must be a finite rule which derives some smaller graph from every non-*Acc* graph in \mathcal{L} . \square

Theorem 12 (A size-reducing GRS of RBTs needs non-terminals)

Red-black trees cannot be specified by a size-reducing NT-free GRS.

Proof

Using Lemma 5. 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 . \square

Definition 31 (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 $RBT = \langle \Sigma_{RBT}, \mathcal{R}_{RBT}, Acc_L \rangle$, where Figure 21 shows the reduction rules in \mathcal{R}_{RBT} and Figure 1 shows Acc_L . \square

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 11. 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`.

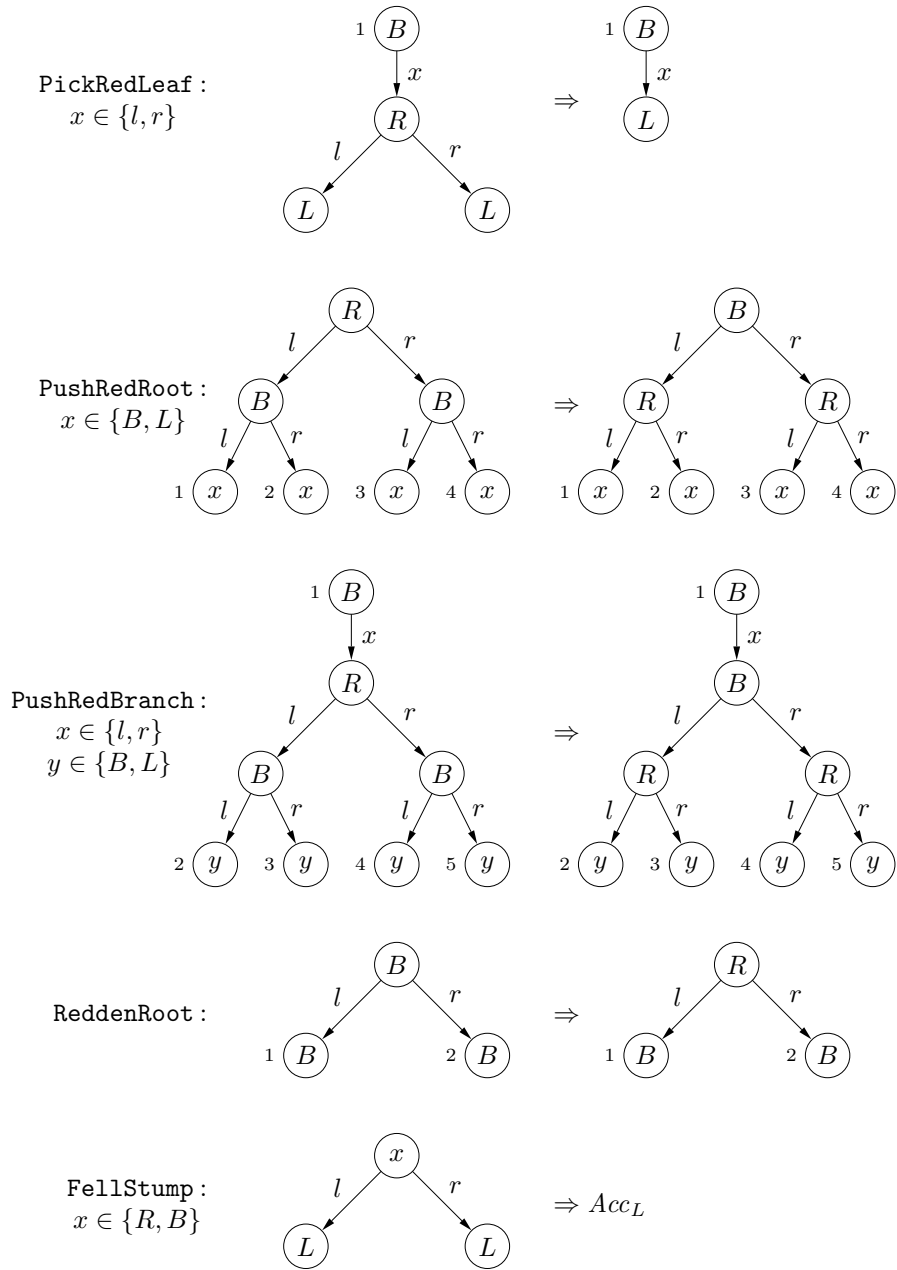


Figure 21: NT-free Red-black tree reduction rules

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.

Definition 32 (Specification of red-black trees)

$$\begin{aligned} \Sigma_{SRBT} &= \langle \{R, B, G, L\}, \{G\}, \{l, r, c\}, \\ &\quad \{R \mapsto \{l, r\}, B \mapsto \{l, r\}, G \mapsto \{c\}, L \mapsto \{\}\} \rangle \\ SRBT &= \langle \Sigma_{SRBT}, \mathcal{R}_{SRBT}, AccL \rangle \end{aligned}$$

The reduction rules in \mathcal{R}_{SRBT} are shown in figures 22 and 23. □

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 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 **FellRBRoot** and **FellGreenRoot**. $SRBT$ is correct by Theorem 11 where \mathcal{L} is all red-black-green trees where green is a unary black branch.

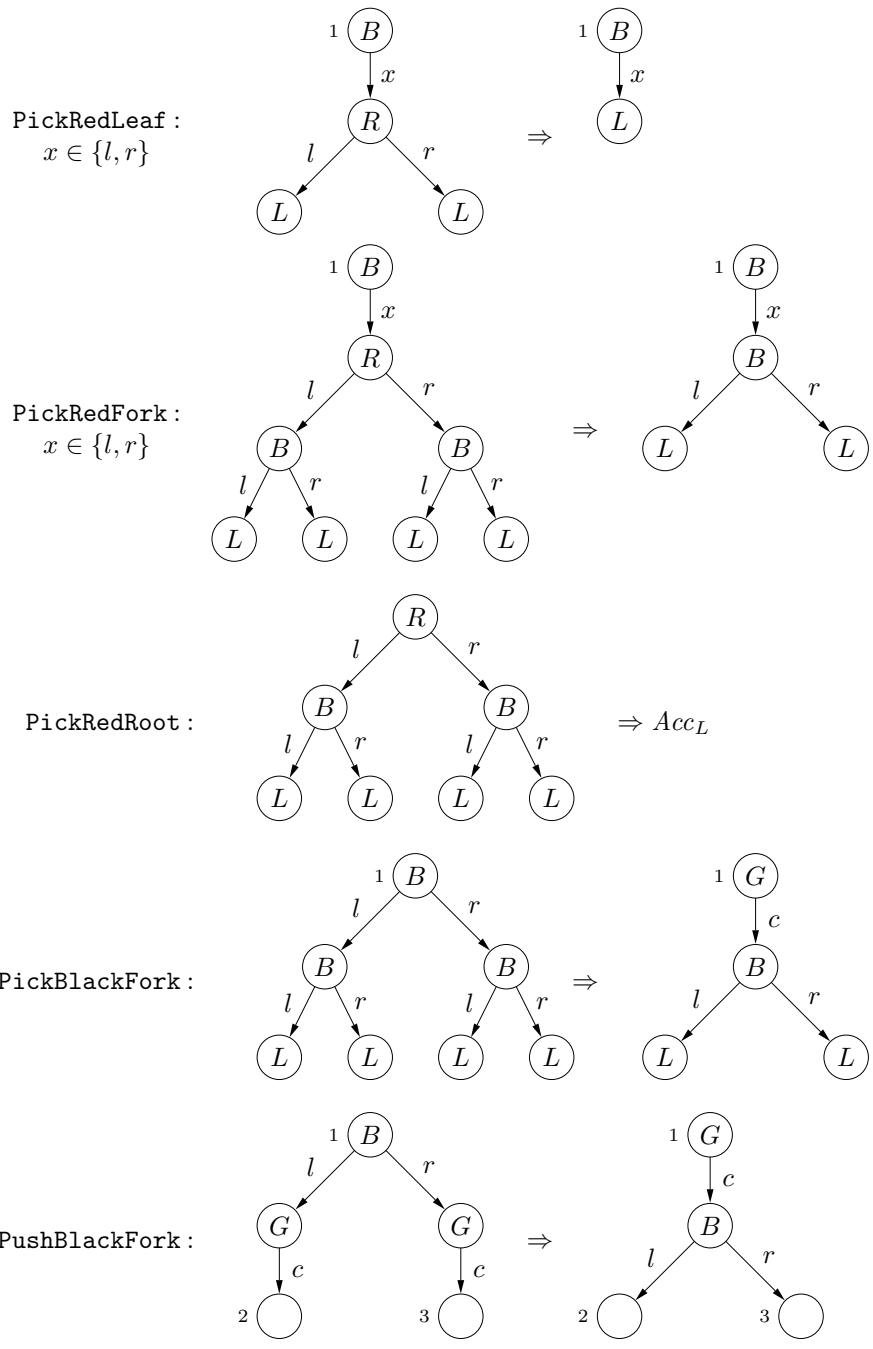


Figure 22: Size-reducing red-black tree reduction rules 1.

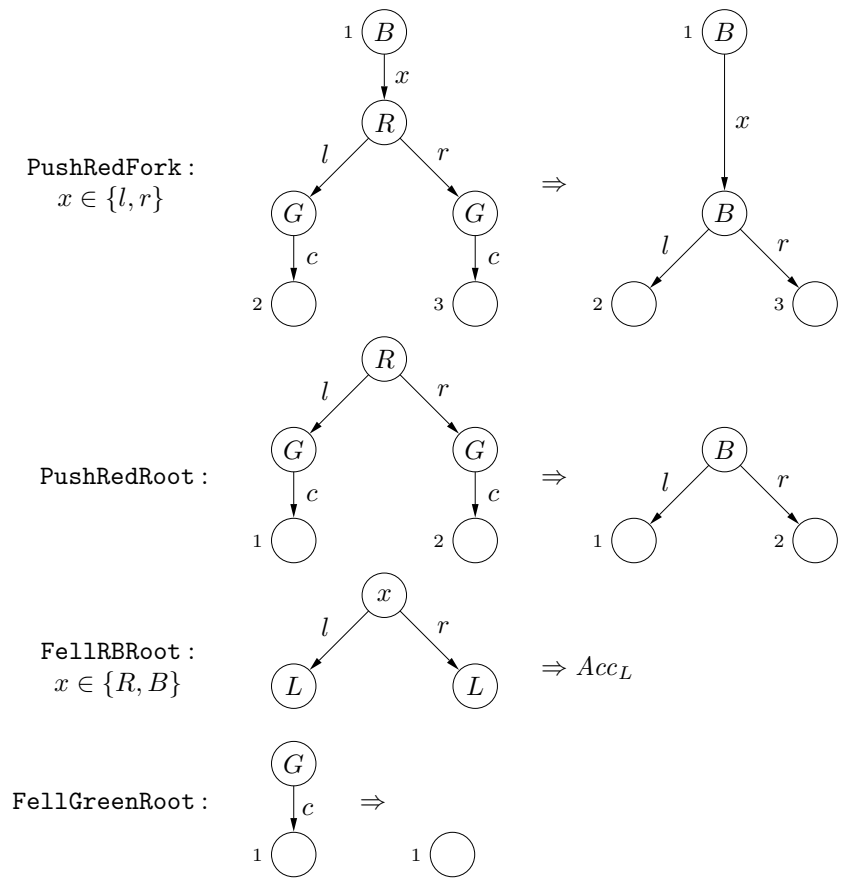


Figure 23: Size-reducing red-black tree reduction rules 2.

6.5 AVL trees

Definition 33 (AVL tree properties)

1. AVL tree nodes are binary branches or leaves.
2. Branches are balanced, left-leaning or right-leaning.
3. The subtree depths of balanced branches are equal.
4. Left subtree depth of a left-leaning node is one plus right subtree depth.
5. Right subtree depth of a right-leaning node is one plus left subtree depth. \square

Theorem 13 (An AVL specification needs non-terminals)

AVL trees cannot be specified by an NT-free GRS.

Proof

Using Lemma 2. 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

Definition 34 (PGRS of AVL trees)

$$\begin{aligned} \Sigma_{AVL} = \langle & \{B, L, R, N, S, B?, L?, R?\}, \{S, B?, L?, R?\}, \{l, r, s\}, \\ & \{B, L, R \mapsto \{l, r\}, N \mapsto \{\}, S \mapsto \{s\}, B?, L?, R? \mapsto \{l, s, r\}\} \rangle \\ AVL = \langle & \Sigma_{AVL}, \mathcal{R}_{AVL}, Acc_N \rangle \end{aligned}$$

Acc_N and the rules in \mathcal{R}_{AVL} are shown in Figures 24. \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 **FellS**. There are three kinds of branches: balanced, left-leaning and right-leaning. A tree is checked bottom-up. A branch labelled B , which claims 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. **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 **Balanced** rewrites the $B?$ node to an S node. Checking branches which claim to be left or right leaning follows the same patterns, but uses **LeftLeaning** or **RightLeaning** as appropriate.

AVL is strongly confluent; it is linearly terminating as each derivation from G reduces $\#V_G + 2 \times \#\{v \in V_G \mid l_G(v) \in \{B, L, R\}\}$. It is correct by Theorem 11 where p defines all *reducing AVL trees*. These are trees which are Σ_{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 the size-reducing PGRS needs 16 rules compared to the 10 in our terminating PGRS.

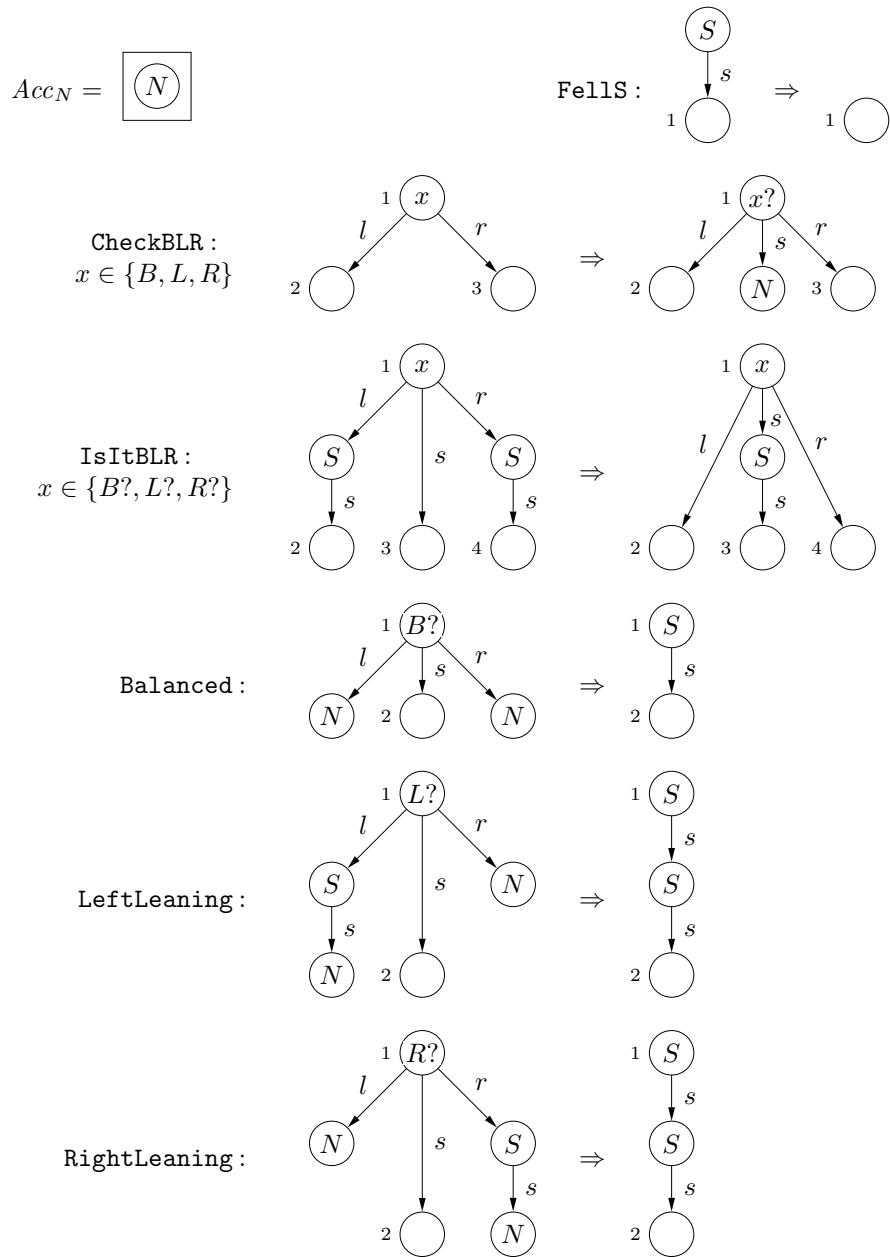


Figure 24: AVL tree accepting graph and reduction rules.

6.6 Rectangular grids

Definition 35 (Grid specification)

1. A grid has $n \times m$ nodes labelled B and $n + m$ nil nodes labelled N .
2. $n, m > 0$.
3. Each B node has a *down* and a *right* pointer.
4. Each B node is assigned a unique coordinate $(i, j) \in \{1, \dots, n\} \times \{1, \dots, m\}$.
5. Each N node is assigned a coordinate $(n + 1, j)$ or $(i, m + 1)$.
6. The *down* arc of node (i, j) points to node $(i, j + 1)$.
7. The *right* arc of node (i, j) points to node $(i + 1, j)$. □

Theorem 14 (Grids need non-terminals)

Grids cannot be specified by an NT-free GRS.

Proof

Using Lemma 2. 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. □

Definition 36 (Grid PGRS)

$$\begin{aligned} \Sigma_{GRID} &= \langle \{B, C, N\}, \{C\}, \{d, r\}, \{B, C \mapsto \{d, r\}, N \mapsto \{\}\} \rangle \\ GRID &= \langle \Sigma_{GRID}, \mathcal{R}_{GRID}, Acc_{GRID} \rangle \end{aligned}$$

Acc_{GRID} and the rules in \mathcal{R}_{GRID} are shown in figures 25 and 26. □

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**".

Theorem 15 (GRID specifies rectangular grids)

$G \in \mathcal{L}(GRID)$ iff G satisfies definition 35.

Proof

Let a *reducing grid* be a grid whose top-left node $(1, 1)$ is labelled C . 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. NT-free reducing grids are grids so the result follows by Theorem 11. □

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.

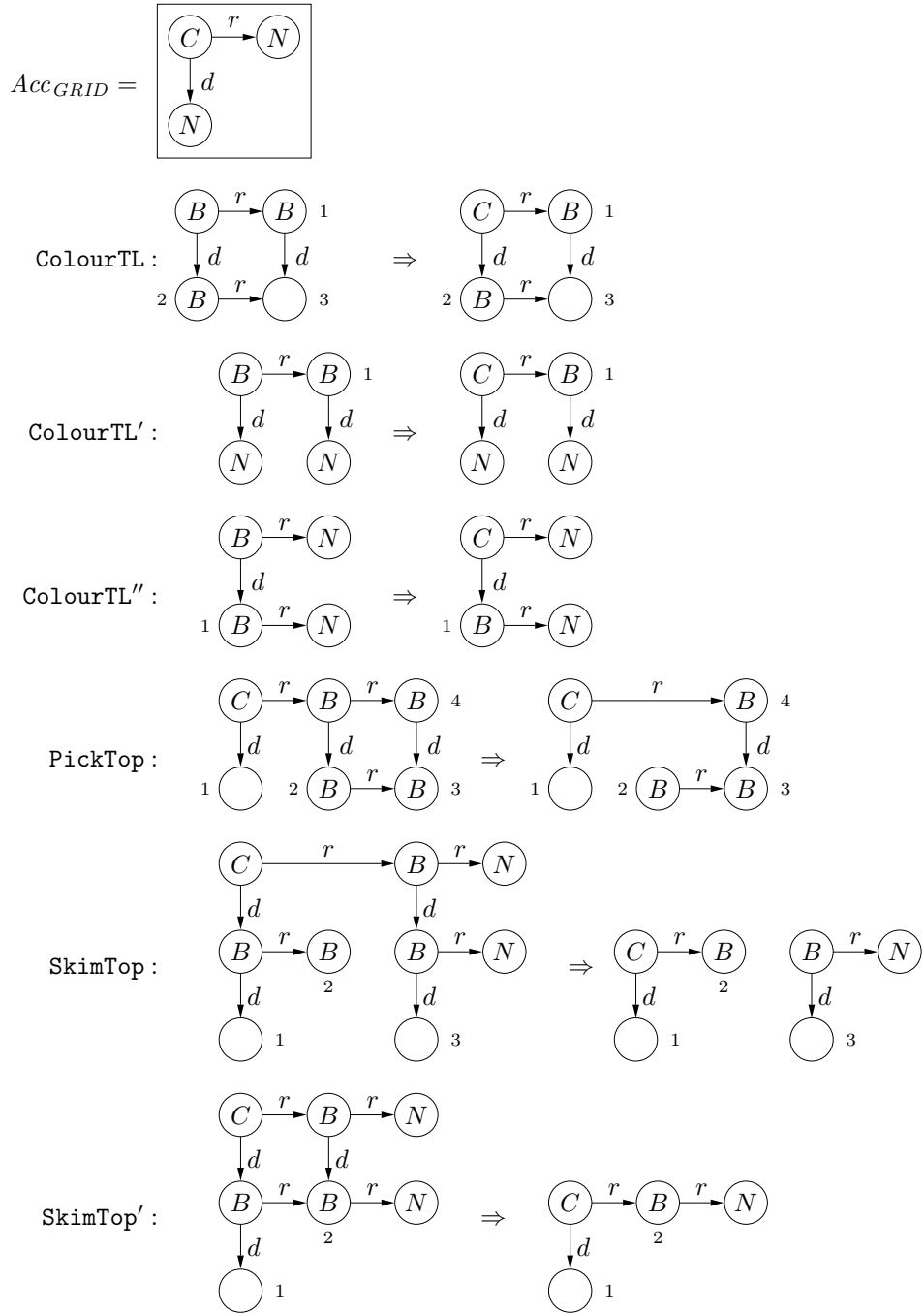


Figure 25: Grid accepting graph and reduction rules 1.

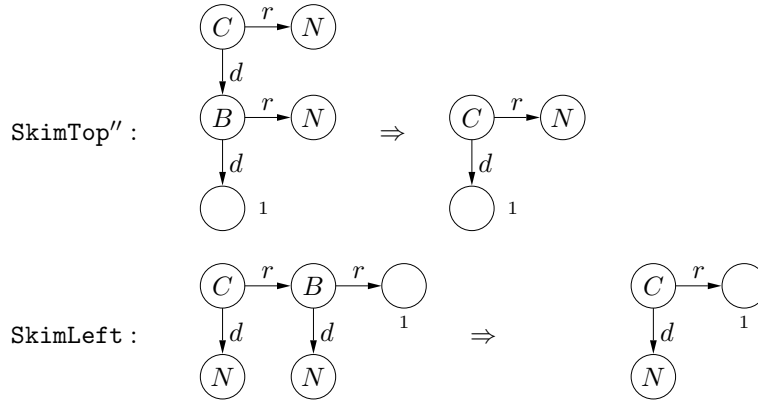


Figure 26: Grid reduction rules 2.

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 [FM96, FM97, FM98] 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 specifiable by context-free graph grammars whose membership problem is NP-complete [Dre93, DHK97], 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 [FM97].

Definition 37 (Converting hypergraphs and grammars)

Shape Types model pointer structures as hypergraphs defined as multisets of relation tuples. These multisets can be converted to graphs as follows.

$$\begin{aligned}
 & \text{convert } \{R_i a_{i,1} \cdots a_{i,n_i}\}_{i=1}^m \\
 = & \langle \bigcup \{v_i\} \cup \{a_{i,j}\}_{j=1}^{n_i}\}_{i=1}^m, \bigcup \{e_{i,j}\}_{j=1}^{n_i}\}_{i=1}^m, \\
 & \bigcup \{e_{i,j} \mapsto v_i\}_{j=1}^{n_i}\}_{i=1}^m, \bigcup \{e_{i,j} \mapsto a_{i,j}\}_{j=1}^{n_i}\}_{i=1}^m, \\
 & \bigcup \{v_i \mapsto R_i\} \cup \{a_{i,j} \mapsto O\}_{j=1}^{n_i}\}_{i=1}^m, \bigcup \{e_{i,j} \mapsto j\}_{j=1}^{n_i}\}_{i=1}^m \rangle
 \end{aligned}$$

All the hypergraph nodes become $a_{i,j}$ nodes and its edges become graph v_i nodes. There is a graph edge $e_{i,j}$ for each hypergraph tentacle, its source

is the node v_i and its target is the node $a_{i,j}$, its label is j . The v_i nodes are labelled by the relation symbol R_i and the $a_{i,j}$ nodes are all labelled O . Various optimisations to this translation are possible, but in general our graphs are a more direct model of data structures than converted hypergraphs.

A hypergraph grammar (N, T, A, P, S) , whose components are non-terminal edge labels, terminal edge labels, an arity function $A : N \cup T \rightarrow \mathbb{N}$, production rules and a start label $S \in N, A(S) = 0$ respectively, converts to a GRS as follows.

$$\begin{aligned} & \text{convert}(N, T, A, P, S) \\ = & \langle \langle \{O\} \cup N \cup T, N, \bigcup \text{Rng}(A), \{O \mapsto \emptyset\} \cup \{l \mapsto \{1, \dots, A(l)\} \mid l \in T \cup N\} \rangle, \\ & \langle \text{convert } r \Rightarrow \text{convert } l \mid l = r \in P \rangle, \\ & \text{convert } \{S\} \rangle \end{aligned}$$

That is, the GRS signature has all the terminal and non-terminal hyperedge labels plus O as its node labels, its edge labels are natural numbers, its *type* function assigns each label the edge labels from 1 to its arity. The production rules become reduction rules by converting them and reversing the direction. The start symbol becomes the accepting graph. \square

Example 18 (Converting the Shape Types list specification)

In [FM97] lists are specified by the grammar $(\{L, List\}, \{next\}, \{L \mapsto 1, List \mapsto 0, next \mapsto 2\}, P, List)$. This converts to the size-reducing and strongly confluent PGRS *STLIST* whose signature (abbreviating *next* as n and *List* as S) is:

$$\Sigma_{STLIST} = \langle \{L, S, n, O\}, \{L, S\}, \{1, 2\}, \{L \mapsto \{1\}, S \mapsto \emptyset, n \mapsto \{1, 2\}, 0 \mapsto \emptyset\} \rangle$$

The rules in P and their conversion are shown with the accepting graph in Figure 27. So in this representation a list is a chain whose nodes are labelled O and whose tail pointers are represented by n -labelled nodes which point to the location and target nodes of the tail pointer. The end node tail pointer points to itself, instead of having a separate nil node. \square

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 [Hin00]; 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 [HHN92] 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* [BRS99] 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

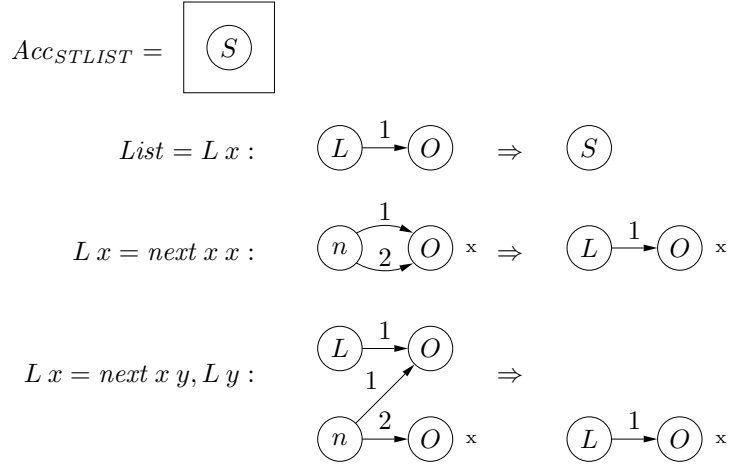


Figure 27: Reduction rules and accepting graphs of the PGRS *STLIST* obtained by converting the Shape Types list grammar.

the complexity is unclear) and the logic is closed under intersection, union and complement.

In *role analysis* [KLR02] 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 [KS93] are recursive data types extended with routing expressions which allow the target of a pointer to be specified relative to its source. In [MS01], 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.

Alias types are an advanced pseudo-linear type system for specifying store shapes with strictly controlled sharing [WM01].

7.3 Checking pointer manipulations

This section briefly demonstrates our method for verifying the shape safety of an algorithm.

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 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.

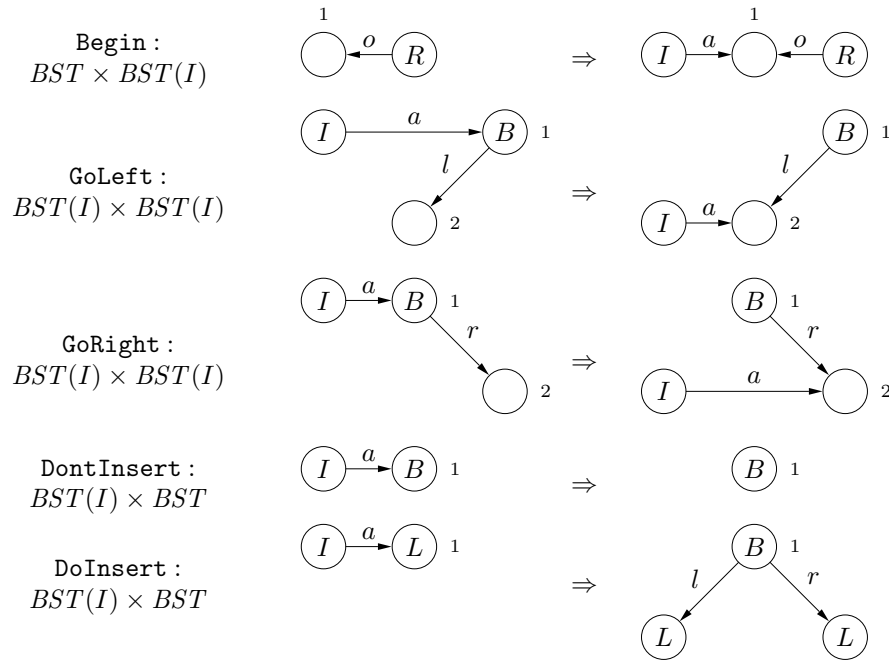


Figure 28: Transformation rules for insertion into binary search trees.

Example 19 (Insertion in binary search trees)

Figure 28 gives five rules which model insertion into binary search trees (BSTs) as a transformation on their shape. BSTs (see Definition 38) have a single *root node* labelled R whose o -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 $BST(I)$ in Definition 38.

Insertion then applies the other rules in Figure 28 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. \square

Definition 38 (Binary search trees and insertion states)

Binary search trees (BSTs) are rooted full binary trees defined by

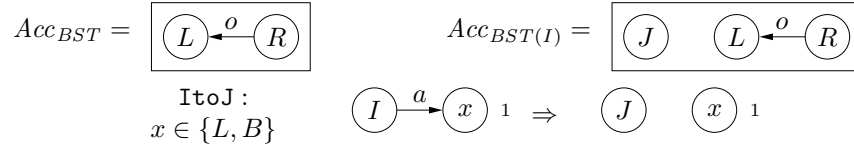


Figure 29: BST and $BST(I)$ accepting graphs and reduction rules.

$\Sigma_{BST} = \Sigma_{BT} + \langle \{R, I, J\}, \{\}, \{o, a\}, \{R \mapsto \{o\}, I \mapsto \{a\}, J \mapsto \{\}\} \rangle$ and $BST = \langle \Sigma_{BST}, \{\text{BtoL}\}, Acc_{BST} \rangle$ (see figures 1 and 29). BSTs with an auxiliary are rooted full binary trees with an I -node whose arc a may point anywhere in the tree: $BST(I) = \langle \Sigma_{BST}, \{\text{BtoL}, \text{BtoLl}, \text{BtoLr}\}, Acc_{BST(I)} \rangle$ (see figures 1 and 29). \square

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 39 (Shape-safe transformation rule)

Transformation rule $t : S \times T$ is shape safe if $G \in \mathcal{L}(S) \wedge G \Rightarrow_t H \Rightarrow H \in \mathcal{L}(T)$. \square

The safety checker (available from [SPG]) 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.

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 is a useful taming of the universal power of non-context-free graph grammars.

Section 8.1 summarises the examples we have specified and classifies them according to their termination, confluence and use of non-terminals, intersection or union. Section 8.2 outlines future work. The GRS tool available from [SPG] implements GRS checking including confluence, membership and operation checks.

8.1 Summary of GRSs

Language	GRS	Classification			Definition	Page	
2-3 trees	<i>23</i>	nt	S	s	Definition 27	29	
2-3-4 trees	<i>234</i>	f	S	s	Definition 29	30	
$A^n B^n$	<i>AB</i>	f	S	c	Example 15	21	
AVL trees	<i>AVL</i>	nt	L	s	Definition 34	37	
Balanced binary trees	<i>BBT</i>	f	S	s	Example 7	10	
Binary DAGs	<i>BDAG</i>	f	L	c	Example 9	11	
Binary search trees	<i>BST</i>	f	S	s	Definition 38	44	
BSTs with auxiliary	<i>BST(I)</i>	f	S	s	Definition 38	44	
Binary trees	<i>BT</i>	f	S	s	Example 1	3	
Complete binary trees	<i>CBT</i>	\cap	f	S	s	Example 14	16
Complete binary trees	<i>CBT</i>	nt	S	s	Example 13	16	
Cyclic lists	<i>CLIST</i>	f	S	c	Example 2	4	
Discrete $2n$ or $3n$ -nodes	<i>D23</i>	\cup	f	S	c	Theorem 8	19
Doubly-linked lists	<i>DLIST</i>	f	S	s	Definition 18	26	
Full binary trees	<i>FBT</i>	f	S	s	Example 1	3	
FBTs (shared leaf)	<i>SFBT</i>	f	S	c	Example 16	22	
Last-element lists	<i>LAST</i>	f	S	c	Definition 17	26	
Linked-leaf trees	<i>TLEAF</i>	f	S	m	Definition 23	27	
Linked lists	<i>LIST</i>	f	S	s	Definition 16	26	
Partial full binary trees	<i>PFBT</i>	f	N	c	Example 17	24	
Rectangular grids	<i>GRID</i>	nt	L	m	Definition 36	39	
Red-black trees	<i>RBT</i>	f	L	c	Definition 31	32	
Red-black trees	<i>SRBT</i>	nt	S	c	Definition 32	34	
Root-connected trees	<i>TROOT</i>	f	S	s	Definition 25	28	
Singly threaded trees	<i>TT</i>	f	S	m	Definition 21	26	
Shape Types lists	<i>STLIST</i>	nt	S	c	Example 18	42	
Skip lists	<i>SKIP</i>	f	S	c	Definition 19	26	
Classification notes							
\cap intersection of PGRSs		nt	uses non-terminals				
\cup union of PGRSs		f	NT-free				
S size-reducing		m	closed (conjectured)				
L linearly terminating		c	confluent with critical pairs				
N non-terminating (not a PGRS)		s	strongly confluent				

8.2 Future work

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 [BPR03a] is a starting point. For operations like insertion into

red-black trees [CLR90] 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

- [BPR03a] A Bakewell, D Plump, and C Runciman. Checking the shape safety of pointer manipulations — extended abstract. In *Participant's Proceedings of the 7th International Seminar on Relational Methods in Computer Science (RelMiCS 7), Malente, Germany*, pages 144–151. University of Kiel, 2003. Available from [SPG].
- [BPR03b] A Bakewell, D Plump, and C Runciman. Specifying pointer structures by graph reduction (shortened version). In *Proc. International Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '03), Charlottesville, USA*, 2003. Available from [SPG].
- [BRS99] M Benedikt, T Reps, and M Sagiv. A decidable logic for describing linked data structures. In *Proc. European Symposium on Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 2–19. Springer-Verlag, 1999.
- [CLR90] T H Cormen, C E Leiserson, and R L Rivest. *Introduction to algorithms*. MIT Press, 1990.
- [DHK97] F Drewes, A Habel, and H-J Kreowski. Hyperedge replacement graph grammars. In G Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Volume I: Foundations*, chapter 2, pages 95–162. World Scientific, 1997.
- [Dre93] F Drewes. NP-completeness of k -connected hyperedge-replacement languages of order k . *Information Processing Letters*, 45(2):89–94, February 1993.
- [FM96] P Fradet and D Le Métayer. Type checking for a multiset rewriting language. In *Proc. Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *LNCS*, pages 126–140. Springer-Verlag, 1996.
- [FM97] P Fradet and D Le Métayer. Shape types. In *Proc. Principles of Programming Languages (POPL '97)*, pages 27–39. ACM Press, 1997.

- [FM98] P Fradet and D Le Métayer. Structured gamma. *Science of Computer Programming*, 31(2–3):263–289, 1998.
- [HHN92] 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 Language Design and Implementation (PLDI '92)*, pages 249–260. ACM Press, 1992.
- [Hin00] R Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 10(3):305–317, May 2000.
- [HMP01] A Habel, J Müller, and D Plump. Double-pushout graph transformation revisited. *Math. Struct. in Comp. Science*, 11:637–688, 2001.
- [HP02] 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.
- [KLR02] V Kuncak, P Lam, and M Rinard. Role analysis. In *Proc. Principles of Programming Languages (POPL '02)*, pages 17–32. ACM Press, 2002.
- [KS93] N Klarlund and M I Schwartzbach. Graph types. In *Proc. Principles of Programming Languages (POPL '93)*, pages 196–205. ACM Press, 1993.
- [MS01] A Møller and M I Schwartzbach. The pointer assertion logic engine. In *Proc. ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM Press, 2001.
- [Plu93] 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.
- [Rea92] C M P Reade. Balanced trees with removals: an exercise in rewriting and proof. *Science of Computer Programming*, 18:181–204, 1992.
- [SPG] Safe Pointers by Graph Transformation, project webpage.
<http://www-users.cs.york.ac.uk/~ajb/spgt/>.
- [Ues78] T Uesu. A system of graph grammars which generates all recursively enumerable sets of labelled graphs. *Tsukuba J. Math.*, 2:11–26, 1978.
- [WM01] D Walker and G Morrisett. Alias types for recursive data structures. In *Types in Compilation (TIC '00), Selected Papers*, volume 2071 of *LNCS*, pages 177–206. Springer-Verlag, 2001.