



Extending C for Checking Shape Safety

Mike Dodds¹ and Detlef Plump²

The University of York, UK

Abstract

The project Safe Pointers by Graph Transformation at the University of York has developed a method for specifying the shape of pointer-data structures by graph reduction, and a static checking algorithm for proving the shape safety of graph transformation rules modelling operations on pointer structures. In this paper, we outline how to apply this approach to the C programming language. We extend ANSI C with so-called transformers which model graph transformation rules, and with shape specifications for pointer structures. For the resulting language C-GRS, we present both a translation to C and an abstraction to graph transformation. Our main result is that the abstraction of transformers to graph transformation rules is correct in that the C code implementing transformers is compatible with the semantics of graph transformation.

Keywords: Pointer programming; shape safety; C; graph transformation.

1 Introduction

Pointers in imperative programming languages are indispensable for the efficient implementation of many algorithms at both applications and systems level, but pointer programming is notoriously prone to undetected errors. This is because the type systems of current programming languages are too weak to detect ill-shaped pointer structures.

To improve this situation, the project Safe Pointers by Graph Transformation³ (SPGT) at the University of York has developed a method to specify the intended shape of a family of pointer data-structures by *graph reduction specifications* (GRSs). A GRS consists of a signature of admissible node and

¹ Email: miked@cs.york.ac.uk

² Email: det@cs.york.ac.uk

³ <http://cs-people.bu.edu/bake/spgt/>

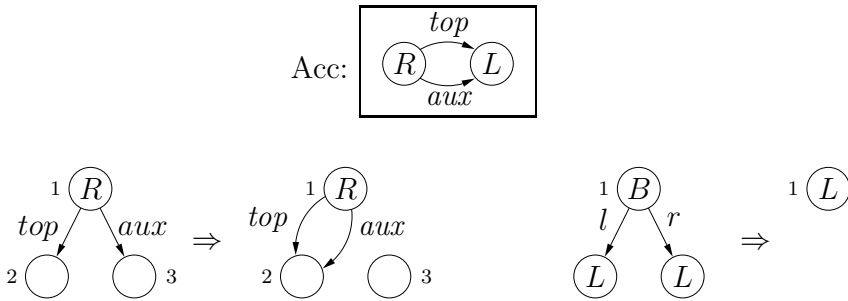


Fig. 1. Graph reduction specification of binary trees with an auxiliary pointer

edge labels, a set of graph reduction rules, and a so-called accepting graph. The shape specified by a GRS contains all graphs that can be reduced to the accepting graph by some series of rule applications [1,3].

For example, Figure 1 shows a GRS for full binary trees with an auxiliary pointer. Tree nodes are either *L*-labelled leaves or *B*-labelled branch nodes with outgoing pointers *l* and *r*, and there is a unique *R*-labelled node with pointers *top* and *aux* which point to the root of the tree and to an arbitrary tree node, respectively. The accepting graph, *Acc*, is the smallest graph of this kind. The left reduction rule redirects the auxiliary pointer to the top of the tree (regardless of the labels of nodes 2 and 3), the right rule deletes two leaves with the same father and relabels their parent node as a leaf. All and only graphs that are full binary trees with an auxiliary pointer can be reduced to *Acc* by these two rules.

Operations on pointer data-structures are also modelled by graph transformation rules. A static checking algorithm for proving that such operations are shape preserving is presented in [2] (generalizing a similar algorithm for context-free shapes given in [4,5]). Figure 7 shows an operation on the shape of Figure 1 that replaces a leaf destination of the auxiliary pointer with a branch node and two new leaves. This is an example of a shape preserving operation: when applied to a full binary tree with an auxiliary pointer, it will always produce a graph of the same shape.

In what follows, we outline how to apply the SPGT approach to the C programming language. The next section summarises how shapes are defined by graph reduction and sketches the checking algorithm for shape-preservation. Section 3 describes constructs which allow C programmers to write shape specifications and operations on shapes, Section 4 indicates how to translate the extended language—called C-GRS—to standard C, Section 5 discusses the correctness of an abstraction of C-GRS shape-specifications and operations to GRSs and graph transformation rules, and Section 6 concludes with a brief discussion of related work.

2 Safe Pointers by Graph Transformation

This section summarises our method of specifying shapes [1,3] and briefly discusses the shape-checking method of [2].

A *graph* $G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle$ consists of a finite set of nodes (or vertices) V_G , a finite set of edges E_G , functions $s_G, t_G: E_G \rightarrow V_G$ assigning a source and a target node to each edge, a partial node labelling function $l_G: V_G \rightarrow \mathcal{L}_V$, and an edge labelling function $m_G: E_G \rightarrow \mathcal{L}_E$. Graph G models a pointer-data structure by retaining only the pointer fields of records and abstracting from other values. Each node models a tagged record of pointers where the node label, drawn from the node-label alphabet \mathcal{L}_V , is the tag. Each edge leaving a node corresponds to a pointer field where the edge label, drawn from the edge-label alphabet \mathcal{L}_E , is the name of the pointer field. We use a function type: $\mathcal{L}_V \rightarrow \wp(\mathcal{L}_E)$ to associate with each record tag its set of field names: if node v is labelled l and has an outgoing edge e , then the label of e must be in $\text{type}(l)$ and no other edge leaving v must have this label. The triple $\Sigma = \langle \mathcal{L}_V, \mathcal{L}_E, \text{type} \rangle$ is called a *signature* and graphs conforming to the above constraints are called Σ -graphs. A Σ -graph is Σ -total if every node v is labelled and for each label in $\text{type}(l_G(v))$ there is an outgoing edge with that label. A *shape* is a set of Σ -total graphs. So shape members model pointer structures with no missing or dangling pointers.

A *graph morphism* $g: G \rightarrow H$ between Σ -graphs G and H consists of a node mapping $g_V: V_G \rightarrow V_H$ and an edge mapping $g_E: E_G \rightarrow E_H$ such that sources, targets and labels are preserved: $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $m_H \circ g_E = m_G$, and $l_H(g_V(v)) = l_G(v)$ for all nodes v where $l_G(v)$ is defined. Morphism g is an *inclusion* if $g(x) = x$ for all nodes and edges x . An *isomorphism* is a graph morphism that is injective and surjective in both components and maps unlabelled nodes to unlabelled nodes. If g is an isomorphism then G and H are *isomorphic*, denoted by $G \cong H$.

A *rule* $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of three Σ -graphs L , K and R , and inclusions $K \rightarrow L$ and $K \rightarrow R$. Graph K is the *interface* of r . Intuitively, a rule deletes the nodes and edges in $L - K$, preserves those in K and allocates those in $R - K$. Our pictures of rules show only the left- and right-hand graphs, the interface always consists just of the numbered nodes of the left- and right-hand graphs. Σ -graphs in rules need not be Σ -total, they can contain nodes with an incomplete set of outgoing edges or unlabelled nodes with no outgoing edges. We refer to [1] for conditions on unlabelled nodes and outgoing edges in rules which ensure that rule applications preserve both Σ -graphs and Σ -total graphs. Rules satisfying these conditions are called Σ -total rules.

Graph G *directly derives* graph H through rule $r = \langle L \leftarrow K \rightarrow R \rangle$ and

$$\begin{array}{ccc}
 L & \leftarrow K & \rightarrow R \\
 g \downarrow (1) & \downarrow (2) & \downarrow \\
 G & \leftarrow D & \rightarrow H
 \end{array}$$

Fig. 2. A double-pushout diagram

injective morphism g , denoted by $G \Rightarrow_{r,g} H$ or $G \Rightarrow_r H$ or just $G \Rightarrow H$, if squares (1) and (2) in Figure 2 are natural pushouts. (See [6] for the definition of natural pushouts.)

Operationally, graph D is obtained from G by deleting the nodes and edges in $g(L) - g(K)$, and making those nodes $g(v)$ unlabelled for which v is an unlabelled node in K that is labelled in L . By the pushout property of square (1), deleted nodes cannot be incident to any edges in $G - (g(L) - g(K))$; this is called the *dangling condition*. Graph H is obtained from D by adding all items in $R - K$, and labelling unlabelled nodes with the labels of their counterparts in R . We write $G \Rightarrow_{\mathcal{R}}^* H$ if there a sequence $G = G_0 \Rightarrow \dots \Rightarrow G_n \cong H$, $n \geq 0$, where each direct derivation uses a rule from the set \mathcal{R} . If no graph can be directly derived from G through a rule in \mathcal{R} , we say that G is \mathcal{R} -irreducible.

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

A GRS can be turned into an equivalent graph grammar by swapping left- and right-hand sides of the rules and using the accepting graph as a start graph. But we insist on the reduction-rule view as we usually impose conditions such as termination and closedness to ensure that shape membership can be efficiently checked (see below). In addition to the above definition, *nonterminal* labels can be allowed, see [1]. Because the rules in \mathcal{R} are Σ -total, we have for every step $G \Rightarrow_{\mathcal{R}} H$ that G is a Σ -total if and only if H is Σ -total. So the graphs defined by GRSs are Σ -total and $L(\mathcal{S})$ is a shape.

A GRS \mathcal{S} is *polynomially terminating* if there is a polynomial p such that for every reduction $G_0 \Rightarrow_{\mathcal{R}} \dots \Rightarrow_{\mathcal{R}} G_n$ on Σ -total graphs, $n \leq p(|V_{G_0}| + |E_{G_0}|)$. It is *closed* if for all $G \in L(\mathcal{S})$, $G \Rightarrow_{\mathcal{R}} H$ implies $H \in L(\mathcal{S})$. A *polynomial graph reduction specification*, PGRS for short, is a polynomially terminating and closed GRS. Membership of PGRS shapes is decidable in polynomial time—see [1], where also sufficient conditions for closedness and polynomial termination are discussed.

Unrestricted GRSs are universally powerful in that they can define every recursively enumerable shape, but their membership problem is undecidable in general. The power of PGRSs goes beyond the reach of context-free graph

grammars (used by Fradet and Le Métayer to specify shapes [4,5]). For example, [1] contains PGRSs for various forms of balanced trees, including red-black trees. Balance is known to be not context-free specifiable.

To illustrate the above notions, consider again the GRS of Figure 1. Its signature is given by $\mathcal{L}_V = \{R, L, B\}$, $\mathcal{L}_E = \{top, aux, l, r\}$, $\text{type}(R) = \{top, aux\}$, $\text{type}(B) = \{l, r\}$ and $\text{type}(L) = \emptyset$. Every full binary tree with an auxiliary pointer can be reduced to the accepting graph: using the left rule in Figure 1 one first redirects the *aux*-edge to the target of the *top*-edge (if the *aux*-edge points to some other node), and then repeatedly applies the other rule which removes two leaves and relabels their common parent node as a leaf. To see that the rules cannot reduce ill-shaped graphs to *Acc*, consider their inverses (which are obtained by swapping left- and right-hand sides): these rules clearly preserve full binary trees with an auxiliary pointer which implies that the specified shape cannot contain other graphs. The GRS is polynomially terminating—actually linearly terminating—because for every step $G \Rightarrow H$ on Σ -graphs, the number of nodes without outgoing parallel edges is reduced. The GRS is also *non-overlapping*, meaning that for each pair of steps $H_1 \Leftarrow G \Rightarrow H_2$ on Σ -graphs, either $H_1 \cong H_2$ or there is a Σ -graph M such that $H_1 \Rightarrow M \Leftarrow H_2$. This property implies closedness and hence the GRS is a PGRS.

Operations on pointer structures—such as the replacement of a leaf in a tree shown in Figure 7—are also modelled by graph-transformation rules (which need not obey the restrictions of PGRSs). A graph-transformation rule r is *safe* with respect to a shape $L(\mathcal{S})$ if for all G in $L(\mathcal{S})$, $G \Rightarrow_r H$ implies $H \in L(\mathcal{S})$.⁴ The static checking algorithm for shape safety developed in the SPGT project is described in [2]. Briefly, given a graph-transformation rule r and a GRS \mathcal{S} , the algorithm constructs two *abstract reduction graphs* (ARGs) which represent all contexts of r 's left- and right-hand side in members of $L(\mathcal{S})$. The rule is safe if the right-hand ARG includes the left-hand ARG. Some ARGs are infinite and hence their construction does not terminate, but in many practical cases the algorithm produces finite ARGs representing all left- and right-hand contexts so that inclusion can be checked. The general shape-safety problem is undecidable even for context-free shapes [5] and hence every checking method is necessarily incomplete.

```

bt *insert(int i, bt *b) {
  int t;
  bt_auxreset(b);
  while ( bt_getval(b, &t) ) {
    if ( t == i ) return b;
    else if ( t > i ) bt_goleft(b);
    else bt_goright(b);
  }
  bt_insert(b, &i);
  return(b);
}

```

Fig. 3. C-GRS function to insert a value into a binary search tree

3 C-GRS – An Extension to C

The language C-GRS is a small extension to ANSI C which is intended to implement the approach to shape specification and shape checking described above. The main idea (adopted from [4]) is that pointers are only manipulated by *transformers* which correspond to graph transformation rules. For example, Figure 3 shows a C-GRS function which inserts an integer value i into a binary search tree b whose shape bt corresponds to the GRS of Figure 1. This function uses the transformer `bt_auxreset` to first move the auxiliary pointer to the root of the tree. Then the tree is traversed by repeatedly comparing the integer values in branch nodes (retrieved by `bt_getval`) with the integer i and following either the left or the right pointer, using the transformers `bt_goleft` and `bt_goright`. If the search ends at a leaf, then `bt_insert` transforms the leaf into a branch node and inserts i into that node. (The transformer `bt_insert` is discussed in Subsection 3.2.)

3.1 Signatures and Shapes

The types of nodes used in a C-GRS shape are declared in a signature. C-GRS signatures are defined separately from shapes, meaning that they can be shared between several shapes. Signature node-types have a similar syntax to C structures, but in addition to values of normal C data-types, nodes can contain pointers to other nodes, declared by the keyword `edge`. Unlike C pointers, edges are defined without stating the type of the objects they are pointing to—edges can point to any (non-root) node in the signature. For example, the type of a branch node of the binary-tree shape is declared as follows:

```

nodetype branchnode {
  edge l, r;
}

```

⁴ For simplicity, this paper assumes that rules have the same input and output shape. The shape-checking method of [2] can also handle shape-changing rules.

```

signature bintree {
  root nodetype btroot {
    edge top, aux;
  }
  nodetype branchnode {
    edge l, r;
    int val;
  }
  nodetype leafnode {}
}

shape bt bintree {
  accept {
    btroot rt;
    leafnode leaf;
    rt.top => leaf;
    rt.aux => leaf;
  }
  rules {
    moveaux2root;
    branch2leaf;
  }
}

```

Fig. 4. Signature and shape declaration for the GRS of Figure 1

```

transformer
bt_insert
bintree ( bt *tree,
          int *inval ) {
  left (rt, n1) {
    btroot rt;
    leafnode n1;
    rt.aux => n1;
  }
  right (rt, n1, l1, l2) {
    branchnode n1;
    leafnode l1, l2;
    rt.aux => n1;
    n1.l => l1;
    n1.r => l2;
    n1.val = *inval;
  }
}

reducer
branch2leaf bintree {
  left (br, l1, l2) {
    branchnode br;
    leafnode l1, l2;
    br.left => l1;
    br.right => l2;
  }
  right (br) {
    leafnode br;
  }
}

```

Fig. 5. Left: transformer to replace a leaf with a branch and insert a value into it. Right: reducer for the shape `bt` of Figure 4

```

  int val;
}

```

C-GRS shape declarations consist of an accepting graph and a set of *reducers* which correspond to the reduction rules of a GRS. The accepting graph of a C-GRS shape is defined after the keyword `accept`, using the same syntax as for the left- and right-hand sides of a reducer. Reducers have a similar syntax to transformers, and are discussed in Subsection 3.2.

Figure 4 shows the declaration of the signature `bintree` and the shape `bt` which corresponds to the GRS of Figure 1. (The node types `btroot`, `branchnode` and `leafnode` correspond to the node labels R , B and L .) Observe that nodes in a C-GRS shape can contain values such as the integer `val` which do not occur in the graphs specified by a GRS.

3.2 Transformers and Reducers

Transformers are the mechanism by which pointer data-structures are manipulated in C-GRS programs. To ensure shape safety, all manipulations of shape members must be written as transformers. Like graph transformation rules, transformers consist of a left- and right-hand graph. For example, consider the transformer `bt_insert` of Figure 5 which replaces a leaf with a value-carrying branch. To apply this transformer to a tree, its left-hand node `rt` must match the source node of the auxiliary pointer `aux` in the tree and node `n1` must match a leaf in the tree that is pointed to by the auxiliary pointer.

The constituent nodes of the left- and right-hand sides of a transformer are declared in a list, as follows:

```
left(rt,n1)
```

Nodes are assigned a type from the transformer's signature (`bintree` in our example) using a syntax similar to C variable declarations:

```
btroot rt;
leafnode n1;
```

It is important to note that transformers can alter node types. The transformer `bt_insert` retypes a leaf node into a branch node rather than creating a new branch node. One can also declare nodes without assigning a type: these nodes correspond to unlabelled nodes in graph transformation rules and will match nodes of every type.

In our example, the target of the `aux`-edge leaving `rt` is specified as follows:

```
rt.aux => n1;
```

Edges must point to nodes, as null edges are not allowed. The right-hand side of `bt_insert` allocates the leaves `l1` and `l2` as children of `n1`. The types of the new leaves are assigned in the same way as shown for the left-hand side. Leaf `n1` is retyped as a branch node by the following type-assignment on the right-hand side:

```
branchnode n1;
```

Transformers can overwrite the values held in nodes or return them through transformer parameters. Only dereferenced pointer variables are allowed on the right-hand side of these assignments, rather than full C expressions. This reduces the complexity of the translation to C, as issues such as side-effects and variable scope can be ignored. The example transformer `bt_insert` inserts an integer value into the branchnode on its right-hand side as follows:

```
n1.val = *inval;
```

A transformer such as `bt_insert` is used in the same way as an ordinary C function, as shown in the search-tree insertion example of Figure 3. Arguments to a transformer function are passed by reference to ensure that several values can be manipulated simultaneously, as C makes it difficult for a function to return several values.

Reducers correspond to the reduction rules of a GRS, and have a similar syntax to transformers. They do not have arguments, and so cannot be used as functions in the program. Neither can reducers override the values held in nodes. Figure 5 shows `branch2leaf`, a reducer for the binary tree example.

The EBNF syntax definition of C-GRS is given in Appendix A, together with some (but not all) context conditions. The definition extends the syntax of ANSI C by adding transformer and shape declarations to the categories `fun-def` and `type-def`. Also, there are new syntactic categories for signatures and reducers as these do not have counterparts in C.

3.3 Rootedness

Adding graph transformation rules to C presents two problems. Rules can be applied in a graph wherever their left-hand sides match, which does not fit with C's deterministic world. Moreover, the search for a match of a (fixed) rule requires polynomial time which is too expensive. We solve both problems by requiring that C-GRS shape-structures and left-hand sides of transformers contain unique *roots*, and that all nodes in the left-hand sides of transformers are reachable from the roots. (These requirements do not apply to reducers.)

Roots are nodes of distinguished types. These special node types are declared in signatures with the keyword `root`. In general, we require that every shape structure has at least one root and that different roots in the same structure must have different root types. For example, the node `rt` in the declaration of `bt` in Figure 4 is a root, and so is an entry point for every binary-tree structure. The same restriction applies to the left-hand sides of transformers where, in addition, each node must be reachable from some root by a directed path of edges. Also, transformers must not delete or add roots, which ensures that every shape structure contains the same number of roots.

Under these conditions the application of a transformer to a shape structure is a deterministic process: the roots of the left-hand side occur in unique places in the structure and whenever a node of the left-hand side has been matched, it is checked if all its outgoing edges are among the outgoing edges of the corresponding node in the structure. The matching of the transformer fails as soon as one of the edge comparisons fails. The matching is successful if all edges of the left-hand side have been found, if the sharing of target nodes in the left-hand side corresponds to the sharing in the structure, and if the

dangling condition for direct derivations (see Section 2) is satisfied.

It is not difficult to see that for a fixed transformer, the matching process requires only constant time. This is because every member of a shape comes with a fixed selection of roots (which can be found in constant time) and because the number of outgoing edges of each node is bounded (as shape structures correspond to Σ -total graphs).

4 Translating C-GRS to C

For execution, C-GRS is translated into ANSI C by the translation function \mathcal{C} given in Appendix B (shape translation) and Appendix C (transformer translation). Only signatures, shapes and transformers result in modified code, while the pure C portions of a C-GRS program are left unmodified by the translation. Reducers are not translated to C code, as they are used for checking shape safety rather than for execution.

The translation of signatures transforms node types into C structures with the same name. For example, the type `btroot` from the signature `bintree` of Figure 4 is translated into the following C structure:

```
struct btroot {
    bintree_node *top, *aux;
}
```

All types of a signature S are wrapped into a single C union `S_node` and edges become pointers to `S_node`. This allows transformer functions to retype nodes in-place in memory.

A shape definition is translated into a root structure for the shape with pointers to each of the root nodes. As only one root of each type can exist in a shape member, the fields are distinctly named after the types of the root nodes. The shape is also used to define a constructor function which creates the shape member corresponding to the accepting graph, as this is the unique irreducible member of any shape.

Appendix C shows the translation of transformers into C functions which can be applied to shape members. The application of such a function proceeds in two major phases: first, the transformer's left-hand side is matched against nodes in the shape member, and second the image of the left-hand side is transformed into the right-hand side by deleting and adding nodes and altering the contents of preserved nodes.

The matching phase of a transformer uses *matching variables* which correspond to nodes in the left-hand graph. These variables hold pointers to nodes in the shape member such that a variable holds a pointer to a node if and only

```

if ( ! typeeq( (rt->btrout).aux, "leafnode" )
    return False;
else if (n1 == NULL) n1 = (rt->btrout).aux;
else if (n1 != (rt->btrout).aux) return False;

```

Fig. 6. Matching code for the edge `aux` in the left-hand side of `bt_insert`

if the left-hand node corresponding to the variable has been matched with the shape-member node.

Root nodes correspond to pointer fields in the C structure `S` representing a shape member, so they can be easily assigned to matching variables. For example, the following code in the translation of the transformer `bt_insert`,

```
rt = tree->btrout;
```

assigns the pointer to the root of a binary-tree to the matching variable `rt`, where `tree` is the parameter of `bt_insert` holding a pointer to the tree.

When a root has been matched, the matching process proceeds by following the edges outgoing from matched nodes. To keep the description of the translation simple, we assume that edge statements are ordered in a way such that non-root nodes never occur as the source of an edge before they occur as a target of some edge. This ensures that nodes are matched in the correct order. As all nodes in the left-hand side of a transformer are reachable from some root, each node will eventually be matched. For example, Figure 6 shows the matching code produced for the edge `aux` from `rt` to `n1` in the left-hand side of `bt_insert`.

This code first checks that the node found by following the `aux`-edge is a leaf. Then, if the matching variable `n1` is null, it is assigned a pointer to the target of the `aux`-edge. This gives `n1` an initial value if it is the first time it has been reached. If `n1` already holds a non-null value, then node `n1` on the left-hand side of the transformer has more than one incoming edge and the code checks that the pointers `n1` and `aux` point to the same node. If any of the checks fail, the transformer function returns `False` without modifying the shape member.

Once all nodes of the left-hand side of a transformer have been matched, the system performs two more checks. First, it checks by comparison of pointer values that each pair of distinct transformer nodes has been matched with distinct nodes in the shape member. This is necessary because of our requirement that matches are injective. Then the dangling condition for deleted nodes (see Section 2) is checked by reference counting, using the `indegree` field of deleted nodes. Failure of the dangling condition is treated in the same way as failure in the above cases.

If the matching process has been successful, the image of the transformer's

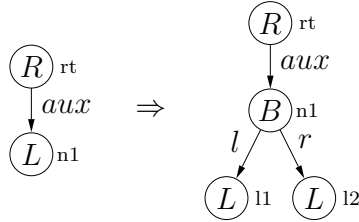


Fig. 7. Graph transformation rule produced from `bt_insert` by the abstraction \mathcal{G}

left-hand side in the shape member is modified to the right-hand side by deleting, allocating and retyping nodes, and recreating edges. Nodes are managed using the normal C memory allocation functions. Edges are recreated by assigning new values to pointer fields in nodes. For example, the edge `aux` in the right-hand side of `bt_insert` is recreated by

```
(rt->btroot).aux = n1;
```

where `rt` and `n1` are the matching variables for the nodes of the same names.

After the C-GRS code has been translated, the resulting C code can be compiled and executed in the normal way.

5 Abstracting C-GRS to Graph Transformation

Our main aim in adding shapes and transformers to C is to make it possible to statically check the shape safety of graph transformation rules corresponding to transformers, using the algorithm described in [2]. We denote by \mathcal{G} the function which abstracts C-GRS shapes and transformers to GRSs and graph transformation rules, respectively. The form of C-GRS shapes and transformers is intentionally very close to GRSs and graph transformation rules, so \mathcal{G} 's straightforward definition is omitted from this paper. As an example, Figure 7 shows the graph transformation rule produced by applying \mathcal{G} to the transformer `bt_insert` of Figure 5. The node labels *R*, *B* and *L* stand for `btroot`, `branchnode` and `leafnode`, respectively.

The translation \mathcal{G} maps C-GRS shape declarations to GRSs whose shapes consist of graphs which model pointer data-structures by abstracting from non-pointer values. Accordingly, graph transformation rules produced from transformers only model structural modifications of pointer structures and ignore value-changing operations. For instance, \mathcal{G} forgets the integer held in node `n1` when abstracting the transformer `bt_insert` to the rule in Figure 7. (Work such as [11] uses graph transformation rules that can perform calculations on labels, and in future we may employ such rules for modelling

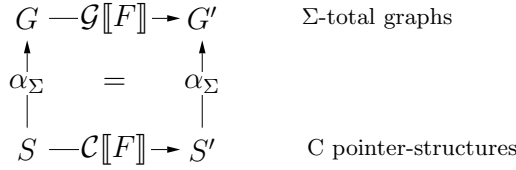


Fig. 8. Correctness of the translation \mathcal{C}

value-changing operations. But this will require an enhanced safety-checking algorithm.)

To analyse the correctness of the translation \mathcal{C} with respect to the graph model given by \mathcal{G} , we fix a few notions. By a *pointer structure* we mean a set of individual records (‘structures’ in C’s terminology) in a C program-state such that all pointers in the records point to records in the set. A pointer structure is *consistent* with a signature $\Sigma = \langle \mathcal{L}_V, \mathcal{L}_E, \text{type} \rangle$ if each record contains a field `type` holding a value $l \in \mathcal{L}_V$ such that $\text{type}(l)$ consists of the names of the pointer fields in the record. We denote by α_Σ the function that abstracts, in the obvious way, pointer structures consistent with Σ to Σ -total graphs.

Using these notions, we say that the translation \mathcal{C} is *correct* with respect to \mathcal{G} if for every transformer F and every pointer structure S that is consistent with F ’s signature Σ ,

$$\mathcal{G}\llbracket F \rrbracket(\alpha_\Sigma(S)) = \alpha_\Sigma(\mathcal{C}\llbracket F \rrbracket(S)).$$

In other words, the diagram of Figure 8 has to commute. (We assume that a failed application of the graph-transformation rule $\mathcal{G}\llbracket F \rrbracket$ returns the input graph unmodified.)

Suppose that this correctness property holds and that all pointer manipulations in a C-GRS program P happen through applications of transformers to members of the shapes associated with the transformers’ signatures. Then we can check that P is shape safe by checking the corresponding graph-transformation rules produced by \mathcal{G} .

To show that the diagram of Figure 8 commutes for every transformer F , we first show that $\mathcal{G}\llbracket F \rrbracket$ and $\mathcal{C}\llbracket F \rrbracket$ select corresponding graph elements in their matching phases. This can be proved by induction on the size of the left-hand side of F :

- (i) Roots are correctly matched, as both graphs and pointer structures can only contain a single instance of a particular root.
- (ii) The children of correctly matched nodes are correctly matched, as in both the graph and the pointer structure they are connected to their parent by a distinctly-labelled edge.

The same kind of argument shows that if matching fails, it fails for corresponding nodes processed by $\mathcal{G}[[F]]$ and $\mathcal{C}[[F]]$. Similarly, $\mathcal{G}[[F]]$ violates the dangling condition for a deleted node if and only if the C code in $\mathcal{C}[[F]]$ checking the dangling condition reports failure for the C record corresponding to that node. The proof of correctness is completed by showing that corresponding right-hand modifications are performed by $\mathcal{G}[[F]]$ and $\mathcal{C}[[F]]$.

6 Related Work

Our language C-GRS is similar to Fradet’s and Le Métayer’s Shape-C [4]. The main difference is that Shape-C is restricted to shapes specified by context-free graph grammars. The graph reduction specifications incorporated in C-GRS—even when restricted to polynomial GRSs—allow programmers to specify non-context-free data structures such as grids and various forms of balanced trees. In addition, shapes defined by polynomial GRSs come with an efficient membership test which can be used for testing and debugging shape specifications. Shapes defined by context-free graph grammars, on the other hand, are known to have an NP-complete membership problem.

Graph types [8] are spanning trees with additional pointers defined by path expressions; they form the basis of *pointer assertion logic* [10], a monadic second-order logic for expressing properties of pointer structures in program annotations. This requires programmers to use quite a sophisticated logic, but the formalism is still too weak to express some important properties such as balance in trees. These drawbacks also apply to the TVLA system [9] which demands that data structures and the effects of program statements are expressed in three-valued logic with transitive closure. TVLA employs the shape analysis method of [13] to verify invariants.

Separation logic [7,12] extends classical Hoare-style program verification so that specifications and proofs can deal with properties of linked data structures. The logic allows the heap to be divided into regions for which different logical formulas hold, making it possible to reason locally about pointers. But so far there seems to be no automatic verification method for separation logic.

Acknowledgement

We would like to thank Adam Bakewell and the anonymous referees for comments that helped to improve this paper.

References

- [1] Bakewell, A., D. Plump and C. Runciman, *Specifying pointer structures by graph reduction*, *Mathematical Structures in Computer Science*. To appear. Preliminary version available as Technical Report YCS-2003-367, University of York, 2003.
- [2] Bakewell, A., D. Plump and C. Runciman, *Checking the shape safety of pointer manipulations*, in: *Int. Seminar on Relational Methods in Computer Science (RelMiCS 7), Revised Selected Papers*, *Lecture Notes in Computer Science* **3051** (2004), pp. 48–61.
- [3] Bakewell, A., D. Plump and C. Runciman, *Specifying pointer structures by graph reduction*, in: *Int. Workshop Applications of Graph Transformations With Industrial Relevance (ACTIVE 2003), Revised Selected and Invited Papers*, *Lecture Notes in Computer Science* **3062** (2004), pp. 30–44.
- [4] Fradet, P. and D. Le Métayer, *Shape types*, in: *Proc. Principles of Programming Languages (POPL '97)* (1997), pp. 27–39.
- [5] Fradet, P. and D. Le Métayer, *Structured Gamma*, *Science of Computer Programming* **31** (1998), pp. 263–289.
- [6] Habel, A. and D. Plump, *Relabelling in graph transformation*, in: *Proc. International Conference on Graph Transformation (ICGT 2002)*, *Lecture Notes in Computer Science* **2505** (2002), pp. 135–147.
- [7] Ishtiaq, S. and P. W. O’Hearn, *BI as an assertion language for mutable data structures*, in: *Proc. Principles of Programming Languages (POPL '01)* (2001), pp. 14–26.
- [8] Klarlund, N. and M. Schwartzbach, *Graph types*, in: *Proc. Principles of Programming Languages (POPL '93)* (1993), pp. 196–205.
- [9] Lev-Ami, T. and M. Sagiv, *TVLA: A system for implementing static analyses*, in: *Proc. Static Analysis (SAS '00)*, *Lecture Notes in Computer Science* **1824** (2000), pp. 280–301.
- [10] Møller, A. and M. I. Schwartzbach, *The pointer assertion logic engine*, in: *Proc. Programming Language Design and Implementation (PLDI '01)* (2001), pp. 221–231.
- [11] Plump, D. and S. Steinert, *Towards graph programs for graph algorithms*, in: *Proc. International Conference on Graph Transformation (ICGT 2004)*, *Lecture Notes in Computer Science* **3256** (2004), pp. 128–143.
- [12] Reynolds, J., *Separation logic: A logic for shared mutable data structures*, in: *Proc. Logic in Computer Science (LICS '02)* (2002), pp. 55–74.
- [13] Sagiv, M., T. Reps and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, *ACM Transactions on Programming Languages and Systems* **24** (2002), pp. 217–298.

Appendix

A EBNF Syntax of C-GRS

```

fun-def ::= transformer trid sigid ( shid *id, [tid *id]* ) {
    left ([nid]* ) { [node-dec;]+ [nid.ed => nid;]* }
    right ([nid]* ) { [node-dec;]* [right-graph;]* } }
    | ...

type-def ::= shape shid sigid {
    accept { [node-dec;]+ [nid.ed => nid;]* }
    rules { [rdid;]* } }
    | ...

reducer-def ::= reducer rdid sigid {
    left ([nid]* ) { [node-dec;]+ [nid.ed => nid;]* }
    right ([nid]* ) { [node-dec;]* [nid.ed => nid;]* } }

sig-def ::= signature sigid{ [node-def;]+ }

node-dec ::= ntid nid [, nid]*
right-graph ::= nid.ed => nid | nid.id = *id | *id = nid.id

node-def ::= nodetype ntid { [node-cont;]+ }
            root nodetype ntid { [node-cont;]+ }
node-cont ::= edge ed [, ed]* | struct-decl-cont

```

- *id* and *tid* stand for identifiers of C variables and C types, respectively; *nid*, *ntid*, *shid*, *sigid*, *trid* and *rdid* stand for identifiers of nodes, node types, shapes, signatures, transformers and reducers, respectively; *ed* stands for edge labels.
- struct-decl-cont corresponds to the statements that can be part of a C structure-declaration.

Context Conditions

- Root nodes must not be deleted by a transformer.
- On both sides of a transformer, all nodes must be reachable from some root node.
- Nodes that are created or retyped on the right-hand side of a transformer or reducer must have all the edges for their type declared.
- All rules of a shape must be defined as reducers.

B Shape Translation

$$\begin{aligned}
 \mathcal{C} \llbracket \text{signature } C \{ N_1; \dots N_n; \} \rrbracket &= [\mathcal{N} \llbracket N \rrbracket]_{N \in \{N_1, \dots, N_n\}} \\
 &\text{typedef struct } C_node \{ \\
 &\quad \text{char *type;} \\
 &\quad \text{int indegree;} \\
 &\quad \text{union \{ } \\
 &\quad\quad [\text{struct } n;]_{n \in Defs} \\
 &\quad\quad \} \text{ node;} \\
 &\quad \} \\
 &\text{where:} \\
 &Defs = \text{node types defined in } \{N_1 \dots N_n\} \\
 \\
 \mathcal{C} \llbracket \begin{array}{l} \text{shape } S \ C \{ \\ \text{accept } \{ A_1; \dots A_n; \} \\ \text{rules } \{ P_1; \dots P_n; \} \\ \} \end{array} \rrbracket &= \\
 &\text{typedef struct } S \{ \\
 &\quad [C_node *r;]_{r \in Roots} \\
 &\quad \} \\
 &S * \text{newgraph_S } () \{ \\
 &\quad [C_node *v;]_{v \in Nodes} \\
 &\quad S *new; \\
 &\quad new = \text{malloc}(\text{sizeof}(S)); \\
 &\quad [\mathcal{I} \llbracket A \rrbracket]_{A \in \{A_1 \dots A_n\}} \\
 &\quad \text{return new;} \\
 &\quad \} \\
 &\text{where:} \\
 &Nodes = \{A_1 \dots A_n\} \\
 &Roots = \text{root nodes declared in } \{A_1 \dots A_n\} \\
 \\
 \mathcal{N} \llbracket \text{nodetype } N \{ C \} \rrbracket &= \text{struct } N \{ \mathcal{T} \llbracket C \rrbracket \} \\
 \mathcal{T} \llbracket \text{edge } E_1 \dots E_n \rrbracket &= [S_node *e;]_{e \in \{E_1 \dots E_n\}} \\
 \mathcal{T} \llbracket C \rrbracket &= C \\
 \\
 \mathcal{I} \llbracket T \ V \rrbracket &= \text{new.V} = \text{createnode}(T); \\
 T \in \text{RootTypes} &= \text{define } t_a(V) = T \\
 \\
 \mathcal{I} \llbracket T \ V_1 \dots V_N \rrbracket &= [V_i = \text{createnode}(T);]_{V_i \in \{V_1 \dots V_n\}} \\
 T \notin \text{RootTypes} &= \text{define } t_a(v) = T, \text{ for } v \in \{V_1 \dots V_N\} \\
 \\
 \mathcal{I} \llbracket S.E \Rightarrow T \rrbracket &= (S \rightarrow t_a(S)).E = T;
 \end{aligned}$$

C Transformer Translation

$$\mathcal{C} \left[\begin{array}{l} \text{transformer } F \ C \ (S \ *G, \ A) \\ \text{left}(N_l) \{ L_1; \dots L_n; \} \\ \text{right}(N_r) \{ R_1; \dots R_n; \} \\ \} \right] = \begin{array}{l} \text{bool } F \ (S \ *G, \ A) \ \{ \\ \quad [\ C_node \ *v = \text{NULL}; \]_{v \in Nodes} \\ \quad [\ \mathcal{L}[\![L]\!] \]_{L \in \{L_1 \dots L_n\}} \\ \quad [\ \text{if } (x == y) \ \text{return } \text{False}; \]_{(x,y) \in Pairs} \\ \quad [\ \text{if } (d \rightarrow \text{indegree} \neq C(d)) \\ \quad \quad \text{return } \text{False}; \]_{d \in Delete} \\ \\ \quad [\ n \rightarrow \text{indegree} \\ \quad \quad = \ n \rightarrow \text{indegree} - C(n); \]_{n \in N_l} \\ \quad [\ \text{retype}(p, \ t_r(p)); \]_{p \in Retype} \\ \quad [\ a = \text{createnode}(t_r(a)); \]_{a \in Allocate} \\ \quad [\ \mathcal{R}[\![R]\!] \]_{R \in \{R_1 \dots R_n\}} \\ \quad [\ \text{deletenode}(d); \]_{d \in Delete} \\ \quad \text{return } \text{True}; \\ \} \end{array}$$

where:

$$\begin{aligned}
 Nodes &= \{L_1 \dots L_n\} \cup \{R_1 \dots R_n\} \\
 C(i) &= \# \{ (s, e) \mid (s.e \Rightarrow i) \in \{L_1 \dots L_n\} \} \\
 Delete &= \{d \in N_l \mid d \notin N_r\} \\
 Allocate &= \{a \in N_r \mid a \notin N_l\} \\
 Retype &= \{p \in N_l \mid p \in N_r \wedge t_l(p) \neq t_r(p)\} \\
 Pairs &= \{(x, y) \mid x \in N_l \wedge y \in N_l \wedge x \neq y\}
 \end{aligned}$$

$$\begin{array}{l} \mathcal{L}[\![T \ V]\!] \\ T \in RootTypes \end{array} = \begin{array}{l} V = G \rightarrow T; \\ \text{define } t_l(V) = T \end{array}$$

$$\begin{array}{l} \mathcal{L}[\![T \ V_1 \dots V_N]\!] \\ T \notin RootTypes \end{array} = \text{define } t_l(v) = T, \text{ for } v \in \{V_1 \dots V_N\}$$

$$\begin{array}{l} \mathcal{L}[\![S.E \Rightarrow T]\!] \end{array} = \begin{array}{l} \text{if } (\ !\text{typeeq}((S \rightarrow t_l(S)).E, \ t_l(T)) \) \\ \quad \text{return } \text{False}; \\ \text{else if } (T == \text{NULL}) \\ \quad T = (S \rightarrow t_l(S)).E; \\ \text{else if } (T == (S \rightarrow t_l(S)).E) \\ \quad \text{return } \text{False}; \end{array}$$

$$\mathcal{R}[\![T \ V_1 \dots V_N]\!] = \text{define } t_r(v) = T, \text{ for } v \in \{V_1 \dots V_N\}$$

$$\mathcal{R}[\![S.V = X]\!] = (S \rightarrow t_r(S)).V = X;$$

$$\mathcal{R}[\![X = S.V]\!] = X = (S \rightarrow t_r(S)).V;$$

$$\begin{array}{l} \mathcal{R}[\![S.E \Rightarrow T]\!] \end{array} = \begin{array}{l} (S \rightarrow t_r(S)).E = T; \\ T \rightarrow \text{indegree} = T \rightarrow \text{indegree} + 1; \end{array}$$