

# Shape Types

Pascal Fradet and Daniel Le Métayer

IRISA/INRIA

Campus de Beaulieu,

35042 Rennes, France

[fradet, lemetayer]@irisa.fr

## Abstract

Type systems currently available for imperative languages are too weak to detect a significant class of programming errors. For example, they cannot express the property that a list is doubly-linked or circular. We propose a solution to this problem based on a notion of *shape types* defined as context-free graph grammars. We define graphs in set-theoretic terms, and graph modifications as multiset rewrite rules. These rules can be checked statically to ensure that they preserve the structure of the graph specified by the grammar. We provide a syntax for a smooth integration of shape types in C. The programmer can still express pointer manipulations with the expected constant time execution and benefits from the additional guarantee that the property specified by the shape type is an invariant of the program.

## 1 Motivation and approach

Facilities for explicit pointer manipulation are useful for certain classes of applications, but they may lead to a very error-prone style of programming. It is well-known that static type checking is one of the most effective ways to improve program robustness. Unfortunately, the expressiveness of type systems currently available for imperative languages is too weak and a significant class of programming errors falls outside their scope. The main reason is that they fail to capture properties about the sharing which is inherent in many data structures used in efficient imperative programs. As an illustration, it is impossible to express the property that a list is doubly-linked or circular in existing type systems.

The work described here is an effort to provide a solution to this problem which is both sound and realistic. The contribution of the paper is twofold:

- We introduce a notion of *shape* defined in terms of graph grammar and an algorithm for the static shape checking of graph *transformers*. Most useful data structures can be expressed as shapes in a precise and natural manner.

---

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.

POPL 97, Paris, France

© 1997 ACM 0-89791-853-3/96/01 ..\$3.50

- We propose a notation for the introduction of *shape types*<sup>1</sup> and *transformers* in C. This notation can be translated into pure C without loss of efficiency, and the previously defined shape checking algorithm can be used to check extended C programs.

Let us stress that the use of shape types does not impose a drastic change in programming practices: the more that traditional pointer types are integrated within shape types, the more static verifications will be performed. So, the programmer can adapt his use of shape types to the level of confidence required for his program. Shape types can also be used to improve the accuracy of program analyses (and enable optimizing transformations), but this application is not described in this paper.

We believe that the following qualities of shape types should favor their adoption in realistic programming environments:

- They can express data structures with complex sharing patterns in a natural way.
- They can be implemented into a language with explicit pointer manipulation without loss of efficiency.
- They are not limited to one style of programming language. We have chosen to present their integration into C here, but the general framework is independent of the host programming language.

We review related work in the next section. For the sake of clarity, we present shape types in two stages. First, we introduce the notion of shape in a programming language independent way (Section 3); we propose a model of graph transformer and an algorithm for static “shape checking” of transformers (Section 4). Then, we show how shapes and transformers can be used as a basis for linguistic extensions of C (Section 5). In Section 6, we assess the proposal described in the paper and we suggest avenues for further research.

## 2 Related work

A large amount of work has been devoted to the design of methods for reasoning about the “shape” (in a broader sense than the one adopted in this paper) of heap-allocated structures. The contributions can be classified in two categories,

---

<sup>1</sup>We use the expression “shape types” for the notion of types introduced here, keeping the denomination “graph types” to refer to [15]

depending on the level of cooperation required from the programmer:

- In the “fully automatic approach”, no help is expected from the programmer. An analyzer automatically infers properties about shapes at all program points. Most storage analyses and alias analyses belong to this class [3, 7, 9, 10, 14, 17, 21]. These analyses are based on various models of “shapes” ( $k$ -limited graphs, regular tree grammars, access path matrices, points-to relationships, ...). A short survey of this trend of work can be found in [7].
- In the “programming language” approach, the programmer can specify the properties of shapes; these properties can then be checked, either statically or dynamically, and used by an optimizing compiler. This approach has been less popular until recently. It involves programming language extensions to describe properties of shapes. These extensions are usually based on traditional (tree-like) recursive data structures enhanced with properties on pointers. ADDS [12, 13] associates directions (*forward*, *backward*) with pointers, making it possible to distinguish, for instance, trees and doubly-linked lists. Graph types [15] are spanning trees augmented with extra links defined using regular routing expressions. The class of graphs considered in [16] is also based on spanning trees, but auxiliary edges are specified by constraints in monadic second-order logic. A quite different formalism is proposed in [20] to specify checkable interfaces as constraints on scalars, sets and multisets. Graph-like data structures are also supported by [11], but the formalism used is akin to more traditional tree grammars.

It should be clear that both approaches are in fact complementary since the shape information provided by language extensions can be used to increase the accurateness of automatic alias analyses [13] (or to make them more efficient). The work described in this paper falls into the second category. We believe that the programming language approach is worthwhile because it makes it possible to get accurate information about the shape of the store at a reasonable cost. Furthermore, it should not necessarily be seen as a compromise, but rather as a step in the right direction, favoring the integration of a better style of programming within existing languages.

The main difference between this work and ADDS is that we specify the links in a shape very precisely (a data structure conforming to a shape must include exactly the links specified by the shape, and no more) whereas the *forward* and *backward* attributes of [13] characterize the authorized links in a less constrained way. This difference reflects the intended application of the description, which is mainly program optimization in [13], whereas our work on shape types is first directed towards a more robust style of programming through type checking.

The *graph types* introduced in [15] are defined as traditional recursive data types enhanced with a notation for expressing the sharing between subterms through auxiliary pointers. Although this work is close in spirit to the approach followed here, we believe that the notion of graph types suffers from two weaknesses which may limit their use:

- The first, and most important, shortcoming is the fact that basic operations on values of a graph type may

involve an implicit walk through the whole structure. Although the worst-case complexity of this walk is linear, this hidden cost can be a serious obstacle to the integration of graph types in languages which are typically used by programmers requiring a very fine grain control over the efficiency of their code.

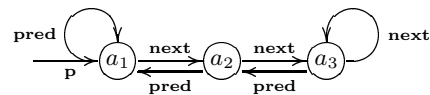
- The second, and more subjective, weakness is the lack of naturalness of the definition of the types. The destination of extra-pointers has to be expressed by regular expressions which characterize paths in the structure. These paths can include a mixture of upward and downward moves leading to quite complex specifications.

We believe that the origin of these difficulties lies in the separation of pointer links into two classes, the spanning tree pointers and the auxiliary pointers, which are defined using two heterogeneous techniques. For example, it does not seem natural to distinguish one particular pointer in a circular list, neither from the perspective of program reasoning nor from the implementation point of view. Shapes are also more expressive because the extra edges of [15] depend functionally on the backbone, which makes it impossible, for instance, to specify a list with an extra link from the head to a random element. This limitation is lifted in [16] which proposes a more general way of specifying classes of graphs as spanning forests enhanced with auxiliary edge constraints expressed in monadic second-order logic. The expressive power of this new formalism and the context-free graph grammars are incomparable.

### 3 Shapes

Our notion of shape is inspired by previous work on the chemical reaction model [2, 8] and set-theoretic graph rewriting [19]. Formally, a graph is defined as a multiset of relation tuples noted  $R a_1 \dots a_n$  where  $R$  is a  $n$ -ary relation name and  $a_i \in V$  with  $V$  a countable set of variables. In the sequel, we use the words “graph” and “multiset” interchangeably.

As an illustration, the following graph represents an example of doubly-linked list with a pointer to the first element:



As it is common in C-like languages, terminal values point to themselves. The list involves three variables  $a_1$ ,  $a_2$  and  $a_3$ . It is formally defined as the multiset  $\Delta$ :

$$\{ \mathbf{p} a_1, \mathbf{pred} a_1 a_1, \mathbf{next} a_1 a_2, \mathbf{pred} a_2 a_1, \\ \mathbf{next} a_2 a_3, \mathbf{pred} a_3 a_2, \mathbf{next} a_3 a_3 \}$$

It should be clear that this graph is just one representative of a class of graphs following the same pattern. We specify such a class as a context-free graph grammar and we call it a *shape*. Different notions of context-free graph grammars have been studied in the literature. They are defined either in terms of node replacement [6] or in terms of hyper-edge replacement [5]. Our definition of graphs as multisets allows us to express hyper-edge replacement in a very natural

way. A grammar is a four-tuple  $\langle NT, T, PR, O \rangle$  where  $NT$  and  $T$  are sets of, respectively, ranked non-terminal and ranked terminal symbols,  $PR$  is a set of production rules and  $O$  is the origin of the derivation. The multisets considered in this paper contain terms built from the symbols of  $NT \cup T$  and variables of  $V$ . A multiset is said to be terminal if it contains only terms built from  $T$  and  $V$ . The production rules of  $PR$  are pairs  $l = r$  where  $l$  is a term  $A x_1 \dots x_n$  (with  $A$  a non-terminal of arity  $n$ ) and  $r$  is a collection of terms.

Continuing our example, the shape representing doubly-linked lists with a pointer to the first element is defined as:

$$H_{DL} = \langle \{Doubly, L\}, \{\mathbf{next}, \mathbf{pred}, \mathbf{p}\}, R_{Doubly}, Doubly \rangle$$

with  $R_{Doubly}$  the following set of rules:

$$\begin{aligned} Doubly &= \mathbf{p} x, \mathbf{pred} x x, L x \\ L x &= \mathbf{next} x y, \mathbf{pred} y x, L y \\ L x &= \mathbf{next} x x \end{aligned}$$

In the following, we use the symbols  $+$  and  $-$  to denote the sum and difference on multisets. We use Greek letters  $\sigma, \tau$  to represent *injective* substitutions (mapping variables to variables).

**DEFINITION 1** *Let  $H$  be the grammar  $\langle NT, T, PR, O \rangle$ . The shape defined by  $H$  is the set:*

$$Shape(H) = \{M \mid M \xrightarrow{*PR} \{O\} \text{ and } M \text{ terminal}\} \text{ with}$$

$$X + (\sigma r) \rightarrow_{PR} X + (\sigma l) \Leftrightarrow$$

$$l = r \in PR \text{ and } (Var(\sigma r) - Var(\sigma l)) \cap Var(X) = \emptyset$$

A multiset belongs to the shape if it rewrites by  $\rightarrow_{PR}$  to the origin  $O$  of the shape. We could alternatively have defined  $Shape(H)$  as the set of the terminal multisets generated from the origin  $O$ , but the definition in terms of reductions makes the subsequent developments easier.

The multiset rewrite system  $\rightarrow_{PR}$  is derived as a “right to left” reading of the rules  $l = r$  of  $PR$ .  $M_0 \rightarrow_{PR} M_1$  if  $M_0$  contains an instantiation  $(\sigma r)$  of a right-hand side of  $PR$  and  $M_1$  is obtained by replacing  $(\sigma r)$  by the corresponding left-hand side  $(\sigma l)$ . It is important to note that in the rewriting

$$X + (\sigma r) \rightarrow_{PR} X + (\sigma l)$$

$X + (\sigma r)$  represents the entire multiset. In other words, the rewrite rules of  $\rightarrow_{PR}$  are global.

The last condition in Definition 1 ensures that new variables occurring on the right-hand side of a rule of the grammar are instantiated with variables which are distinct from all other existing variables. This constraint, which is usual in graph rewriting [19], is necessary to avoid unexpected variable sharing.

The rewrite system associated with *Doubly* is:

$$\begin{aligned} \mathbf{p} x, \mathbf{pred} x x, L x &\rightarrow_{R_{Doubly}} Doubly \\ \mathbf{next} x y, \mathbf{pred} y x, L y, X &\rightarrow_{R_{Doubly}} L x, X \quad y \notin X \\ \mathbf{next} x x, X &\rightarrow_{R_{Doubly}} L x, X \end{aligned}$$

The variable  $X$  stands for the rest of multiset (the *context* of the reduction) and  $y \notin X$  expresses the last condition in Definition 1.

It is easy to check that the multiset  $\Delta$  defined above belongs to  $Shape(H_{DL})$ . But the multiset  $\Delta'$ :

$$\{\mathbf{p} a_1, \mathbf{pred} a_1 a_1, \mathbf{next} a_1 a_2, \mathbf{pred} a_2 a_1, \mathbf{next} a_2 a_1, \mathbf{pred} a_1 a_2, \mathbf{next} a_1 a_1\}$$

which is obtained by confusing  $a_3$  and  $a_1$ , does not belong to  $Shape(H_{DL})$ . Applying the last rule of  $R_{Doubly}$ , it reduces to

$$\{\mathbf{p} a_1, \mathbf{pred} a_1 a_1, \mathbf{next} a_1 a_2, \mathbf{pred} a_2 a_1, \mathbf{next} a_2 a_1, \mathbf{pred} a_1 a_2, L a_1\}$$

But the second rule of  $R_{Doubly}$  cannot be applied to this term because the variable instantiating  $y$  ( $a_1$  here) must not occur in the rest of the multiset.

In order to enhance the intuition about shapes, Figure 1 gathers a few examples illustrating their use to describe pointer structures. Skip lists are used as an alternative to balanced trees for more efficient data insertions and deletions [18]. Red-black trees are binary search trees whose links are either “black” or “red” [22]. A property of red-black trees is that there are never two successive red links along a path from the root to a leaf (red links are represented as dotted lines in the figure). This property is expressed in the shape. The left-child, right-sibling trees (Lcrs-trees) are binary trees used to represent trees with unbounded branching [4]. Note, that each node has a parent pointer and a pointer (**leftc**) to its leftmost child and a pointer (**rights**) to its sibling immediately to the right. The grammars can be intuitively explained by attaching a meaning to each non-terminal. For example, in the last grammar,  $N x y$  denotes a Lcrs-tree whose root is  $x$  and parent  $y$ .  $L x y$  denotes a list of Lcrs-trees whose parent is  $y$ ; the first tree of a list  $L x y$  has root  $x$ .

## 4 Shape invariance

### Transformers

We consider a simple model of program  $P = (C \Rightarrow A)$ , called a *transformer*, whose semantics is defined as a “single step” rewriting:

$$X + (\sigma C) \rightarrow X + (\sigma A) \Leftrightarrow$$

$$(Var(\sigma A) - Var(\sigma C)) \cap Var(X) = \emptyset$$

A transformer replaces an instantiation of its left-hand side (the *condition*  $C$ ) by an instantiation of its right-hand side (the *action*  $A$ ). Again, the condition ensures that new variables occurring on the right-hand side are really fresh.

As an illustration, the following transformers respectively add an element at the front of a doubly-linked list and remove an intermediate element from a doubly-linked list:

$$P_1 = \mathbf{p} a, \mathbf{next} a b, \mathbf{pred} b a \Rightarrow \mathbf{p} a, \mathbf{next} a a', \mathbf{pred} a' a, \mathbf{next} a' b, \mathbf{pred} b a'$$

$$P_2 = \mathbf{next} a b, \mathbf{pred} b a, \mathbf{next} b c, \mathbf{pred} c b \Rightarrow \mathbf{next} a c, \mathbf{pred} c a$$

Because of the condition on new variables, the variable  $a'$  in the first program must be fresh (it must not occur in the context  $X$  of the reduction).

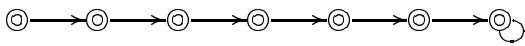
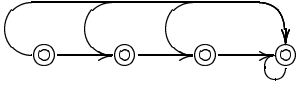
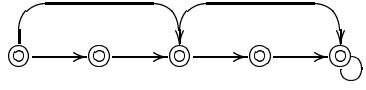
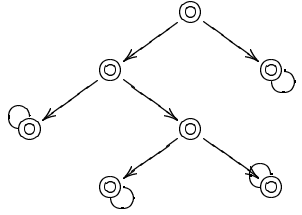
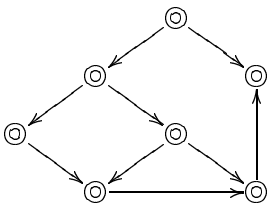
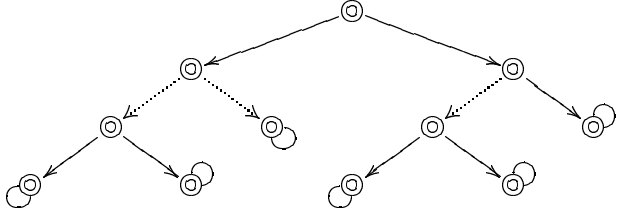
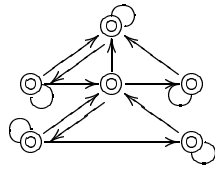
<p>Simple lists:</p> <p><math>List = L x</math>  <math>L x = \mathbf{next} x y, L y</math>  <math>L x = \mathbf{next} x x</math></p>	
<p>Lists with connections to the last element:</p> <p><math>Listlast = L x z</math>  <math>L x z = \mathbf{next} x y, \mathbf{last} x z, L y z</math>  <math>L x z = \mathbf{next} x z, \mathbf{last} x z, \mathbf{next} z z</math></p>	
<p>Skip lists of level 2:</p> <p><math>Skip = S x x</math>  <math>S x y = \mathbf{next} x z, S z y</math>  <math>S x y = \mathbf{next} x z, \mathbf{skip} y z, S z z</math>  <math>S x y = \mathbf{next} x x, \mathbf{skip} y x</math></p>	
<p>Binary trees:</p> <p><math>Bintree = B x</math>  <math>B x = \mathbf{left} x y, \mathbf{right} x z, B y, B z</math>  <math>B x = \mathbf{leaf} x x</math></p>	
<p>Binary trees with linked leaves:</p> <p><math>Binlink = L x y z</math>  <math>L x y z = \mathbf{left} x u, L u y v, R x v z</math>  <math>L x y z = \mathbf{left} x y, R x y z</math>  <math>R x y z = \mathbf{right} x u, \mathbf{next} y v, L u v z</math>  <math>R x y z = \mathbf{right} x z, \mathbf{next} y z</math></p>	
<p>Red-black trees:</p> <p><math>Redblack = L x</math>  <math>L x = \mathbf{leaf} x x</math>  <math>L x = \mathbf{leftb} x y, R x, L y</math>  <math>L x = \mathbf{leftr} x y, R x, B y</math>  <math>R x = \mathbf{rightb} x y, L y</math>  <math>R x = \mathbf{rightr} x y, B y</math>  <math>B x = \mathbf{leftb} x y, \mathbf{rightb} x z, L y, L z</math></p>	
<p>Left-child, right-sibling trees:</p> <p><math>Lcrs = N x x</math>  <math>N x y = \mathbf{leftc} x z, \mathbf{parent} x y, N z x, L x y</math>  <math>N x y = \mathbf{leftc} x x, \mathbf{parent} x y, L x y</math>  <math>L x y = \mathbf{rightc} x z, N z y, L z y</math>  <math>L x y = \mathbf{rightc} x x</math></p>	

Figure 1: Examples of shapes

$Check_{C,A}(PR, O) = Verify_A (Build_C(PR, O))$  where:

- $Build_C(PR, O)$  returns the tree with root  $C$  and all the edges  $C_i \xrightarrow{X_i} C_{i+1}$  such that  
 $\exists l = r \in PR, \exists \sigma \in MGU(C_i, (l, r))$  and  
 $X_i = (\sigma r) - C_i$   
 $C_{i+1} = (C_i - (\sigma r)) + (\sigma l)$   
and  $C_{i+1}$  is not isomorphic to one of its ancestors  $C_j$  in the tree.
- $Verify_A (Tree)$  returns *true* if and only if  
 $\forall C_1 \xrightarrow{X_1} C_2 \dots \xrightarrow{X_{k-1}} C_k$  complete path in *Tree* ( $C_1 = C$  and  $C_k$  is a leaf),  
 $A + X_1 + \dots + X_{k-1} \xrightarrow{*} PR C_k$
- $MGU(C, (l, r))$  is the set of all substitutions (modulo renaming)  $\sigma$  of variables of  $l$  and  $r$  such that:

$$C \cap (\sigma r) \neq \emptyset \text{ and } (Var(\sigma r) - Var(\sigma l)) \cap Var(C - (\sigma r)) = \emptyset$$

Figure 2: A simple shape checking algorithm

### A simple shape checking algorithm

Let us consider a shape  $H = \langle NT, T, PR, O \rangle$  and a given transformer  $P = (C \Rightarrow A)$ . The natural question at this stage concerns the possibility of verifying that  $P$  is correct with respect to  $H$ . A static “shape checking” amounts to a proof of invariance: if a multiset  $M$  belongs to the shape  $H$  and  $M$  can be rewritten into  $M'$  by  $P$ , then  $M'$  must also belong to the shape  $H$ . So, what is needed is an algorithm  $Check_{C,A}$  satisfying the following property:

PROPOSITION 2

If  $Check_{C,A}(PR, O)$  then  $\forall X, \forall \sigma,$

$$X + (\sigma C) \xrightarrow{*} PR \{O\} \Rightarrow X + (\sigma A) \xrightarrow{*} PR \{O\}$$

We describe such an algorithm in Figure 2. Its termination and correctness proofs can be found in the appendix. In order to convey the intuition, we devote the rest of this section to an informal presentation of the algorithm. Let us consider the verification of the transformers  $P_1$  and  $P_2$  above with respect to the shape *Doubly* defined in Section 3.  $Build_C$  returns the following tree for  $P_1$  (with the root at the top):

$$\begin{array}{c} \mathbf{p} a, \mathbf{next} a b, \mathbf{pred} b a \\ \downarrow L b \\ \mathbf{p} a, L a \\ \downarrow \mathbf{pred} a a \\ \mathbf{Doubly} \end{array}$$

The root of the tree is the left-hand side

$$C = \mathbf{p} a, \mathbf{next} a b, \mathbf{pred} b a$$

of the transformer to be checked.  $MGU$  computes the substitutions matching  $C$  with a subset of the left-hand side of a  $\rightarrow_{R_{Doubly}}$  rule. There is only one possibility here, namely the second rule of  $\rightarrow_{R_{Doubly}}$  and  $\sigma = \{(x, a), (y, b), (X, \mathbf{p} a)\}$ .

The label of the corresponding edge is  $X_1 = \{L b\}$  which is the context required for the reduction. The reduced term is  $C_2 = \mathbf{p} a, L a$ . The only possible matching of  $C_2$  is with the left-hand side of the first rule of  $\rightarrow_{R_{Doubly}}$ . The label of the second edge is the context  $X_2 = \mathbf{pred} a a$  and the result of the derivation is the origin *Doubly*. Note that  $C_2$  does not match the left-hand side of the second rule of  $\rightarrow_{R_{Doubly}}$  due to the side condition  $y \notin X$  (because of the presence of  $\mathbf{p} a$ ). Indeed, a context built from this rule would not be valid since it would add an element at the front of  $\mathbf{p} a$ .

In a second stage,  $Verify_A$  is applied to this tree, with  $A = \mathbf{p} a, \mathbf{next} a a', \mathbf{pred} a' a, \mathbf{next} a' b, \mathbf{pred} b a'$ .  $Verify_A$  checks that  $A + \{L b, \mathbf{pred} a a\} \xrightarrow{*} R_{Doubly} \mathbf{Doubly}$ , which is straightforward. It should be clear that this step would have failed if we had inadvertently misnamed a variable, swapped two variables, or forgotten any link in the definition of  $A$ .

The tree constructed by  $Build_C$  for  $P_2$  is the following:

$$\begin{array}{c} \mathbf{next} a b, \mathbf{pred} b a, \mathbf{next} b c, \mathbf{pred} c b \\ \downarrow L c \\ \mathbf{next} a b, \mathbf{pred} b a, L b \\ \downarrow \emptyset \\ L a \end{array}$$

$L a$  is a leaf of the tree because the derivation

$$L a, \mathbf{next} a' a, \mathbf{pred} a a' \rightarrow_{R_{Doubly}} L a'$$

would lead to an isomorphic term. This stopping condition is necessary to avoid infinite unrolling of the tree. As usual in static program analysis, this condition could be weakened to get more precise results at the price of the construction of a larger tree.

Again,  $Verify_A$  checks that the action of the transformer ( $\mathbf{next} a c, \mathbf{pred} c a$ ) in the same context  $L c$  derives to the same term  $L a$ .

## Improvements of the checking algorithm

For the sake of clarity, we have presented here a simplified version of the algorithm. Several optimizations can be considered. The most important ones concern the intermediate structure: it can be represented as a graph rather than a tree and it can be pruned to remove all the nodes which cannot lead to the origin  $O$  (they represent contexts which cannot occur in a multiset of the given shape). Also, the condition checked by  $Verify_A$  for non-terminal leaves can be weakened for a better precision. The basic idea is to consider nodes up to isomorphisms and to build the complete reduction graph (with all paths leading to the origin of the shape). This reduction graph can be represented by a graph grammar whose language is the set of possible contexts, that is to say, the quotient language  $L(O)/C = \{X \mid X+C \xrightarrow{*PR} \{O\}\}$ . Shape checking amounts to proving  $L(O)/C \subseteq L(O)/A$ , which can be done using classical techniques for (word) grammar inclusion. This technique improves the precision of the simple algorithm considerably.

## Completeness issues

Context-free graph grammars are a very flexible and powerful formalism. The price to pay for this generality is, not surprisingly, that the grammar equivalence and inclusion problems are undecidable in this framework. Since shape checking reduces to proving the inclusion of graph grammars, it is also undecidable. So, no complete shape checking algorithm can be expected for unrestricted grammars and transformers. Even if we believe that a sophisticated algorithm can deal with most common situations, this theoretical result is annoying. As it is, the programmer would remain helpless when a plausible transformer is rejected by the checker. In the following, we define a subclass of shape grammars and transformers for which a complete (and practical) checking algorithm exists.

If the shape grammar  $H = \langle NT, T, PR, O \rangle$  and the transformer  $C \Rightarrow A$  are such that:

- the rewriting system  $\xrightarrow{*PR}$  is confluent and
- the set of contexts of  $C$  (i.e.  $\{X \mid C + X \xrightarrow{*PR} \{O\}\}$ ) can be represented as a finite collection of multisets of the form  $\{X_1, \dots, X_n\}$  with  $X_i \in T \cup NT$ ,

then a simple extension of the previous naïve algorithm is enough to decide whether the transformer  $C \Rightarrow A$  is correct with respect to  $H$ .

The idea is to compute only irreducible contexts and to find a minimal representation of the quotient language  $L(O)/C$ . Confluence ensures that considering only irreducible contexts is sufficient. The algorithm checks that any irreducible context  $X$  satisfies  $A + X \xrightarrow{*PR} \{O\}$ . The second condition ensures that the number of such contexts is finite, thus the checking process terminates and is complete.

It seems that most practical transformers can be checked without these restrictions and therefore we do not intend to impose them. However, when a (supposedly) valid transformer cannot be checked, these two conditions can provide guidance to re-express the problem in a tractable way.

The confluence can be statically checked using the standard method based on overlapping terms. Unjoinable critical pairs constitute useful feedback for the programmer to change his grammar. The second condition can be rephrased intuitively as follows: the shape after removal of  $C$  can be described finitely in terms of terminals and nonterminals of

the grammar. This provides guidance to the programmer to modify the reaction (e.g. by making the context more precise) or the grammar (e.g. by introducing new nonterminals).

## 5 Shapes within C

We describe now Shape-C, an extension of C which integrates the notions of shapes and transformers. The design of Shape-C is guided by the following criteria:

- The extensions should be blended with other C features and be natural enough for C programmers.
- The result of the translation of Shape-C into simple C should be efficient.
- The checking algorithm of Section 4 should be applicable to ensure shape invariance.

Space limitations prevent us from describing all the details of Shape-C. Instead, we present the extensions and their translation into C through an example: the Josephus program. This program, borrowed from ([22], pp. 22), first builds a circular list of  $n$  integers; then it proceeds through the list, counting through  $m - 1$  items and deleting the next one, until only one is left (which points to itself). Figure 3 displays the program in Shape-C and its translation into C. The complete syntax and translation rules of Shape-C are described in Figure 4 and Figure 5 in the appendix.

### Declaration and representation of shapes

The Josephus program first declares a shape `cir` denoting a circular list of integers with a pointer `pt`.

```
shape int cir { pt x, L x x;
                L x y = L x z, L z y;
                L x y = next x y;    };
```

Besides cosmetic differences, the definition of shapes is similar to the context free grammars presented in Section 3. The variables of  $V$  in the previous section are now interpreted as addresses. They possess a value whose type must be declared (here `int`). This addition is essential for programming purposes but it can be ignored during shape checking. Values can be tested or updated but cannot refer to addresses. They do not have any impact on shape types.

Intuitively, unary relations (here `pt`) correspond to roots whereas binary relations (here `next`) represent pointer fields. The shape `cir` is translated into

```
struct ad {int val ; struct ad *next;};
struct cir {struct ad *pt;};
```

An address is represented by a structure (`struct ad`) with a value field (`val`) and as many fields (of type pointer to `struct ad`) as the shape has binary relations (here just one). The shape itself is represented by a structure (called *root structure*) with as many fields (of type `struct ad *`) as the shape has unary relations. In the following, if  $f x y$  belongs to the shape, we say that  $x$  (resp.  $y$ ) is a *source* (resp. *destination*) of the binary relation  $f$ .

Shape-C uses only a subset of shapes which corresponds to the rooted pointer structures manipulated in imperative languages. This subset is defined by the following properties:

<pre> /* Integer circular list                                     */ shape int cir { pt x, L x x;                 L x y = L x z, L z y;                 L x y = next x y;    };  main() {   int i, n, m;    /*initialization to a one element circular list*/   cir s = [  =&gt; pt x; next x x; \$x=1;  ];    scanf("%d%d", &amp;n, &amp;m);   /* Building the circular list 1-&gt;2-&gt;...-&gt;n-&gt;1 */   for (i = n; i &gt; 1; i--)     s:[  pt x; next x y; =&gt;       pt x; next x z; next z y; \$z=i;  ];    /* Printing and deleting the m th element   until only one is left                                     */   while (s:[  pt x; next x y; x != y; =&gt;  ])   {     for (i = 1; i &lt; m-1; ++i)       s:[  pt x; next x y; =&gt;         pt y; next x y;  ];      s:[  pt x; next x y; next y z; =&gt;       pt z; next x z; printf("%d ",\$y);  ];   }   /* Printing the last element                               */   s:[  pt x =&gt; pt x; printf("%d\n", \$x);  ]; } </pre> <p style="text-align: center;">(a) in Shape-C</p>	<pre> struct ad {int val ; struct ad *next;}; struct cir {struct ad *pt };  main() {struct cir s; struct ad * x, *y, *z;   int i, n, m;    x = (struct ad *) malloc(sizeof (struct ad)),   s.pt = x, x-&gt;next = x, x-&gt;val = 1;    scanf("%d%d", &amp;n, &amp;m);    for (i = n; i &gt; 1; i--)     if (x = s.pt, y = x-&gt;next, 1)       {z = (struct ad *) malloc(sizeof (struct ad)),        s.pt = x, x-&gt;next = z, z-&gt;next = y, z-&gt;val = i;}    while (x = s.pt, y = x-&gt;next, x != y)   {     for (i = 1; i &lt; m-1; ++i)       if (x = s.pt, y = x-&gt;next, 1)         {s.pt = y, x-&gt;next = y; }      if (x = s.pt, y = x-&gt;next, z = y-&gt;next, 1)       {s.pt = z, x-&gt;next = z, printf("%d ",y-&gt;val),        free(y);}   }   if (x = s.pt, 1)     {s.pt = x, printf("%d\n", x-&gt;val);}   deallocate(s,Cir); } </pre> <p style="text-align: center;">(b) after translation into C (without optimizations)</p>
--	--

Figure 3: Josephus Program

- (S1) *Relations are either unary or binary.*
- (S2) *Each unary relation is satisfied by exactly one address in the shape.*
- (S3) *Binary relations are functions.*
- (S4) *The whole shape can be traversed starting from its roots.*
- (S5) *An address is a source for all binary relations.*

The first four conditions correspond directly to properties of rooted pointer structures. The last one is used to keep the issue of uninitialized pointers separate. The conditions (S2) and (S5) ensure that roots and pointers in the shape are always valid. Null pointers will be represented by elements pointing to themselves, as it is common in C-like languages.

These conditions can be enforced by analyzing the definition of grammars. Except (S1) which is purely syntactic, checking the other conditions amounts to a simple data-flow-like analysis. Let us point out that these constraints do not weaken the expressive power of graph grammars. It is always possible to transform any shape grammar to meet the conditions above (e.g. by adding new binary relations to represent n-ary relations or to make the shape fully connected).

### Manipulation of shapes

The reaction, noted [ $C \Rightarrow A$ ], is the main operation on shapes and corresponds to the transformers presented in Section 4. Two specialized versions of reactions are also provided: initializers, with only an action, noted [ $\Rightarrow A$ ] and tests, with only a condition, noted [ $C \Rightarrow$ ].

The Josephus program declares a local variable  $s$  of shape  $cir$  and initializes it to a one element circular list.

```
cir s = [| => pt x; next x x; $x = 1; |];
```

The value of address  $x$  is noted  $\$x$  and is initialized to 1. In general, actions may include arbitrary C-expressions involving values. The for-loop builds a  $n$  element circular list using the reaction

```
s:[| pt x; next x y; =>
  pt x; next x z; next z y; $z=i; |];
```

The condition selects the address  $x$  pointed to by  $pt$  and its successor. The action inserts a new address  $z$  and initializes it to  $i$ . The interpretation of actions as transformers is almost straightforward. The only subtlety concerns variable name confusion. For programming purposes, we have found it more convenient to allow two different variable names in the condition to denote the same address. For example, the reaction above corresponds to the two transformers:

`pt x , next x y ⇒ ...` and `pt x , next x x ⇒ ...`

The user can make equality or difference explicit using expressions of the form `x == y` or `x != y`. So, conditions may include boolean expressions on values or simple comparisons of addresses. For example, the while-loop specifies a deletion of the *m*th element until only one is left. This condition is implemented by the test

```
s:[| pt x; next x y; x != y; => |]
```

which yields false if `x` points to itself.

### Translation

The translation process is local and applied to each shape operation of the program. Firstly, in order to manipulate the addresses, fresh local variables are declared as

```
struct ad *x, *y, *z;
```

in our example. Conditions are translated into a comma expression, such as

```
x = s.pt, y = x->next, x != y
```

for the while-loop test. The local C variables denoting the addresses are initialized before performing the test denoted by comparison operations and expressions of the condition. If no test occurs in the condition, initializations are followed by `1` (i.e. “true” in C).

The translation of an action is made of assignments of addresses and C expressions where values `$z` are replaced by the selection of the `val` field of the node pointed to by `z`. For example, the translation of the initializer of `s` is

```
z = (struct ad *) malloc(sizeof (struct ad));
s.pt = x, x->next = z, z->next = y, z->val = i;
```

This efficient (after local optimizations) implementation of reactions would not be possible with the general definition of transformers. Shape-C uses a variation of transformers such as:

- (R1) *Two variables can denote the same address.*
- (R2) *In a condition, an address variable occurs at most once as a destination of a relation.*
- (R3) *Any relation  $f_i x y$  in the condition is preceded by a relation  $f_j z x$  or  $p_j x$ .*

The first two requirements suppress implicit tests that conditions would have to make otherwise. Without (R1) and (R2), a condition `next x y , next y y` would entail the tests `x!=y` and `y==y->next`. The programmer must instead state explicitly

```
next x y; next y z; x!=y; y==z;
```

The last condition makes it possible to translate a relation `f x y` into `y = x->f`. Because of (R3), we know that `x` has been initialized. Furthermore, the properties (S2) and (S5) ensure that the dereferences in the translation are valid.

### Memory management

We have expressed the declaration of shapes as local variable declarations. On block exit, local shapes are deallocated using the function `deallocate(1,T)`. This function relies on the type to traverse and to free the shape starting from its roots. Constraint (S4) ensures that the traversal is feasible. Actually, Shape-C also includes dynamic allocation of shape objects with the instructions `(shape tid *) newshape([| => A |])` and `freeshape(id)`.

One benefit of Shape-C is to relieve the programmer of memory management within shapes. Allocation is performed implicitly when new addresses occur in actions (as in the first for-loop in our example). As far as deallocation is concerned, recall that relations are always removed *explicitly* by reactions. So, an address which occurs as the source of binary relations in the condition and does not occur in the action is freed. This sole syntactic criterion is sufficient to compile garbage collection. In our example, this case is illustrated by `y` in the second reaction of the while-loop. The translation makes its deallocation explicit.

### Interaction with C

We have striven to provide a reasonably intimate integration of shapes within C. For example, values can be of any C type, C expressions may appear in reactions, the type “pointer on shape” is allowed, etc ... However, Shape-C requires a few restrictions and we present them here.

An important property that shapes should possess is *independence*. That is to say, shape addresses should not be pointed from another shape or using a regular C pointer but only from the shape itself. By construction, addresses can appear only in the relations and comparisons of a reaction. The only direct way to modify the structure of a shape object is to use the reaction construct. Still, undisciplined pointer arithmetic or wild casts (such as `(int *)intexp`) might ruin this property. Such practices are highly risky and commonly discouraged; we cannot provide any guarantee in these cases.

We have chosen to represent a shape by a structure of roots. This structure contains pointers which can be modified and we must therefore disallow the copy of root structures. The needed restriction can be stated as follows:

- (C1) *The shape type is submitted to the same restrictions as the type “function returning ...” in C.*

In particular, shapes cannot be assigned (except using initializers) and cannot be passed as parameters or yielded as function result. However, the programmer may use shape pointers e.g. to pass shapes to functions or to return them as results.

It is also crucial to ensure that reactions can be seen as atomic operations. So, a second restriction is:

- (C2) *Nested reactions on the same shape are banned.*

A simple solution is to disallow function calls in reactions but there also exists more flexible options.

### Shape checking

Shape checking amounts to verify that initializations and reactions preserve the shape of objects. First, let us point out that values and expressions on values are not relevant for shape checking purposes. The conditions and actions considered here are restricted to their relations and addresses comparisons.



For an initialization  $T\ i = [|\Rightarrow A\ |]$ , we just have to check that the action  $A$  can be rewritten into the origin  $T$ , that is,  $A \xrightarrow{*}_{PR_T} \{T\}$ .

Checking reactions is achieved through a translation into transformers and application of the algorithm of Section 4. Due to our convention for name confusion, a reaction is translated into a set of transformers which correspond to every possibility of variable equality and difference (in accordance with explicit constraints  $x=y$ ,  $x!=y$  in the condition).

The proof that shape invariance is guaranteed in Shape-C (up to independence) is sketched in the annex.

## 6 Conclusion

In order to assess the proposal described in this paper, let us consider in turn the efficiency of the translation, the complexity of the checking algorithm and the expressive power of shape types.

- The translation into C described here is naïve and the code may seem inefficient. Fortunately, most of the requisite optimizations are local and within the reach of a standard C compiler. A source of inefficiency is condition (S5) which may lead to a waste of memory space. For example, the translation of shapes would produce four field nodes to represent red-black trees (cf. Figure 1) whereas the standard representation uses two fields along with two booleans. A solution to this nuisance is to add syntactic features (or analysis) to declare (or detect) disjoint relations (such as **leftf** and **leftb** in red-black trees). Such relations can be implemented by a single tagged node. Their selection in a condition would involve checking the tag.
- The theoretical complexity of the algorithm is exponential but only in terms of the size of the grammar and transformers. In practice, it seems very unlikely that programmers would write huge grammars. As Figure 1 shows, complex data structures can be described by small grammars.
- Useful structures, such as square grids or balanced trees, cannot be described as context-free graph grammars. The extension to context-sensitive grammars would lift these limitations but is far from obvious. The main problem would be the termination of our checking algorithm.

We have undertaken an implementation which should help to assess the practicality and efficiency of Shape-C.

We are considering two other application areas for shape types:

- The first one is the integration of shapes as checkable interfaces in a programming environment for C.
- The second one is the use of shape types as a basis for more accurate (and practically feasible) alias and parallelization analyses.

We should stress that, due to their precise characterization of data structures, shape types should be a very useful facility for the construction of safe programs. Most efficient versions of algorithms are based on complex data structures which must be maintained throughout the execution of the

program [4] [22]. Ensuring the invariance of their representation is an error-prone activity. Shape types can be used to describe these invariants in a natural way (see Figure 1 for instance) and have them automatically verified. Their use as checkable interfaces should enhance their role in a distributed programming environment, possibly serving as a basis for program indexing.

The operations on a given shape type can naturally be gathered into a specialized module (or class in object-oriented languages), but it should be clear that the approach described here goes beyond the design of a fixed set of library functions, since new types can be defined by the user, with their operations automatically checked.

## Acknowledgments

This work was partly supported by Esprit Basic Research project 9102 *Coordination*. Thanks are due to Julia Lawall and Tommy Thorn for commenting on an earlier version of this paper.

## References

- [1] L. Andersen, *Program analysis and specialization for the C programming language*, Ph.D Thesis, DIKU, University of Copenhagen, May 1994.
- [2] J.-P. Banâtre and D. Le Métayer, *Programming by multiset transformation*, Communications of the ACM, Vol. 36-1, pp. 98-111, January 1993.
- [3] D. Chase, M. Wegman and F. Zadeck, *Analysis of pointers and structures*, in Proc. ACM Conf. on Programming Language Design and Implementation, Vol. 25(6) of SIGPLAN Notices, pp. 296-310, 1990.
- [4] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to algorithms*, MIT Press, 1990.
- [5] B. Courcelle, *Graph rewriting: an algebraic and logic approach*, Handbook of Theoretical Computer Science, Chapter 5, J. van Leeuwen (ed.), Elsevier Science Publishers, 1990.
- [6] P. Della Vigna and C. Ghezzi, *Context-free graph grammars*, Information and Control, Vol. 37, pp. 207-233, 1978.
- [7] A. Deutsch, *Semantic models and abstract interpretation techniques for inductive data structures and pointers*, in Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM'95, pp. 226-229, 1995.
- [8] P. Fradet and D. Le Métayer, *Structured Gamma*, Irisa Research Report PI-989, March 1996.
- [9] P. Fradet, R. Gaugne and D. Le Métayer, *Detection of pointer errors: an axiomatisation and a checking algorithm*, Proc. European Symposium on Programming, Springer Verlag, LNCS 1058, pp. 125-140, 1996.
- [10] R. Ghiya and L. J. Hendren, *Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C*, in Proc. ACM Principles of Programming Languages, pp. 1-15, 1996.

- [11] J. Grosch, *Tool support for data structures*, Structured Programming, Vol. 12, pp. 31-38, 1991.
- [12] L. J. Hendren, J. Hummel and A. Nicolau, *Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs*, in Proc. ACM Conf. on Programming Language Design and Implementation, pp. 249-260, 1992.
- [13] J. Hummel, L. J. Hendren and A. Nicolau, *Abstract description of pointer data structures: an approach for improving the analysis and optimisation of imperative programs*, ACM Letters on Programming Languages and Systems, Vol. 1, No 3, pp. 243-260, September 1992.
- [14] N. Jones and S. Muchnick, *Flow analysis and optimization of Lisp-like structures*, in Program Flow Analysis: Theory and Applications, New Jersey 1981, Prentice-Hall, pp. 102-131.
- [15] N. Klarlund and M. Schwartzbach, *Graph types*, Proc. ACM Principles of Programming Languages, pp. 196-205, 1993.
- [16] N. Klarlund and M. Schwartzbach, *Graphs and decidable transductions based on edge constraints*, in Proc. Trees in Algebra and Programming - CAAP'94, Springer Verlag, LNCS 787, pp. 187-201, 1994.
- [17] W. Landi and B. Ryder, *Pointer induced aliasing, a problem classification*, Proc. ACM Principles of Programming Languages, pp. 93-103, 1991.
- [18] W. Pugh, *Skip lists: a probabilistic alternative to balanced trees*, Communications of the ACM, Vol. 33-6, pp. 668-676, June 1990.
- [19] J.-C. Raoult and F. Voisin, *Set-theoretic graph rewriting*, Proc. int. Workshop on Graph Transformations in Computer Science, Springer Verlag, LNCS 776, pp. 312-325, 1993.
- [20] J. R. Russel, R. E. Storm and D. M. Yellin, *A checkable interface language for pointer-based structures*, Proc. Workshop on Interface Declaration Languages, ACM Sigplan Notices, Vol. 29, No. 8, August 1994.
- [21] M. Sagiv, T. Reps and R. Wilhelm, *Solving shape-analysis problems in languages with destructive updating*, Proc. ACM Principles of Programming Languages, pp. 16-31, 1996.
- [22] R. Sedgewick, *Algorithms in C*, Addison-Wesley publishing company, 1990.
- [23] J. H. Siekmann, *Unification theory*, Advances in Artificial Intelligence, II, Elsevier Science Publishers, pp. 365-400, 1987.

## Appendix

### Termination and correctness of the shape checking algorithm

The following observations allow us to prove the termination of the algorithm:

- The tree returned by  $Build_C(PR, O)$  is finite; this is because:
  - $PR$  and  $MGU(C_i, (l, r))$  are finite ( $MGU$  is a restricted form as associative-commutative unification [23]); thus each node has a finite number of sons.
  - $\forall l = r \in PR, size(l) = 1 \leq size(r)$ ; thus the sizes of all the descendants of a node are less than its own size and the number of nodes is finite since no term isomorphic to an ancestor is introduced (the set of relation symbols occurring in terms is obviously finite).

The tree can be built following a depth-first strategy. We do not go into these details here.

- The termination of the reductions

$$A + X_1 + \dots + X_{k-1} \xrightarrow{*}_{PR} C_k$$

performed by  $Verify_A$  can be shown using a well-founded ordering based on a Chomsky normal form of the grammar defined by  $PR$  (see [8] for a complete proof).

In order to establish the correctness of the algorithm, we introduce the notion of normal reduction.

PROPOSITION 3 *Let  $M, C, M'$  be multisets such that*

$$M + C \xrightarrow{*}_{PR} M'.$$

*Then,  $\exists M_0, \dots, M_n, E_1, \dots, E_n, C_1, \dots, C_{n+1}$ , with  $M_0 = M, C_1 = C$  and  $C_{n+1} = M'$ , such that  $\forall i \in [1, n]$*

$$\begin{aligned} &M_{i-1} \xrightarrow{*}_{PR} M_i + E_i \\ &C_i + E_i \xrightarrow{PR} C_{i+1} \text{ and} \\ &\exists l = r \in PR, \exists \sigma \text{ such that} \end{aligned}$$

$$\begin{aligned} &C_i \cap (\sigma r) \neq \emptyset \text{ and} \\ &E_i = (\sigma r) - C_i \text{ and} \\ &C_{i+1} = (C_i - (\sigma r)) + (\sigma l) \text{ and} \\ &(\text{Var}(\sigma r) - \text{Var}(\sigma l)) \cap \\ &(\text{Var}(C_i - (\sigma r)) + (E_{i+1} + \dots + E_n)) = \emptyset \\ &((C_1, E_1), \dots, (C_n, E_n), M') \text{ is called a normal derivation of} \\ &C \text{ in context } M. \end{aligned}$$

Normal derivations are useful because they isolate the reduction steps which are independent of  $C$  and they make explicit the local contexts  $E_i$  which are consumed by a reduction step involving  $C$  or its by-products  $C_i$ .

The following lemma can be proven par recurrence on  $n$ .

LEMMA 4 *Let  $((C_1, E_1), \dots, (C_n, E_n), \{O\})$  be a normal derivation of  $C$  in context  $M$ . Then there is a complete path  $N_1 \xrightarrow{X_1} N_2 \dots \xrightarrow{X_{k-1}} N_k$  of length  $k \leq n$  in  $Build_C(PR, O)$  and a substitution  $\sigma$  such that:*

$$\forall i \in [1, k], C_i = \sigma N_i, E_i = \sigma X_i.$$

The existence of a normal reduction is guaranteed by Proposition 3. The following two observations allow us to conclude the proof of Proposition 2:

- The reduction steps  $M_{i-1} \xrightarrow{*PR} M_i + E_i$  in Proposition 3 are not affected by the replacement of  $C$  by  $A$ .
- The reductions  $A + X_1 + \dots + X_{k-1} \xrightarrow{*PR} C_k$  in the definition of  $Verify_A$  are stable by substitution through  $\sigma$ .

### Shape invariance in Shape-C

The correctness proof relies on the dynamic semantics of C as stated in [1] (pp. 30-37). This SOS involves rules of the form

$$\frac{\mathcal{E} \vdash_{smt} \langle smt, \mathcal{S} \rangle \rightsquigarrow \mathcal{S}'}{\mathcal{E} \vdash_{smt} \langle smt', \mathcal{S} \rangle \rightsquigarrow \mathcal{S}'}$$

with  $\mathcal{E}$  and  $\mathcal{S}$  standing for the environment and the store respectively. In order to treat Shape-C, we add a rule for each new construct. For example, let  $\mathcal{T}[[\ ]]$  denote the translation into C (cf. Figure 5), then the rule for reactions is

$$\frac{\mathcal{E} \vdash_{smt} \langle \mathcal{T}[[C \Rightarrow A]], \mathcal{S} \rangle \rightsquigarrow \mathcal{S}'}{\mathcal{E} \vdash_{smt} \langle [C \Rightarrow A], \mathcal{S} \rangle \rightsquigarrow \mathcal{S}'}$$

The first property to be proven is the independence of shapes. The property is stated using a function which extracts from the store the set of locations which can be reached starting from an identifier in the environment and the set of locations of shapes. The property is simply that a shape and any other identifier have disjoint sets of reachable locations. Even if Shape-C is intended to be an extension of full C, proof of independence can only be done for a subset of C excluding union types, casts, arrays, and pointer arithmetic.

The proof of shape invariance assumes independence. Let us first define a function  $\Psi$  which extracts from the store the set of relations denoted by a shape. The result of  $\Psi$  is a graph (multiset) as defined in Section 3, except that the domain of variables  $V$  is a set of locations.  $\Psi$  takes the location  $l$  of a shape (e.g.  $\mathcal{E}(s)$  if  $s$  is a shape identifier), its shape type  $T$ , and a store  $\mathcal{S}$ . Let  $\mathbf{p}_1, \dots, \mathbf{p}_n$  be the unary relations of shape  $T$ ;  $\Psi$  is defined as

$$\begin{aligned} \Psi(l, T, \mathcal{S}) &= X^* \\ \text{with } X^* &= X_0 \cup X_1 \cup \dots \\ \text{and } X_0 &= \{\mathbf{p}_i \mathcal{S}(l + \text{Offset}(p_i))\}_{i=1, \dots, n} \\ X_{i+1} &= \{\mathbf{f} \ x \ \mathcal{S}(x + \text{Offset}(f)) \\ &\quad | \ \mathbf{f} \text{ binary relation of } T \\ &\quad \text{and } \exists (\mathbf{p} \ x) \in X_i \vee \exists (\mathbf{f}' \ z \ x) \in X_i\} \end{aligned}$$

where  $\text{Offset}(f)$  represents the offset of field  $f$  in a structure.

A store  $\mathcal{S}$  is said to be valid w.r.t. an environment if all its shape identifiers denote a structure in accordance with their shape definition. More formally,

$$\text{Valid}(\mathcal{E}, \mathcal{S}) = \forall s : \text{shape } T \in \mathcal{E} \quad \Psi(\mathcal{E}(s), T, \mathcal{S}) \xrightarrow{*RT} \{T\}$$

The proof is done by induction on the SOS. The key part is the case of reactions that we briefly describe. Assuming that the reaction has been shape checked, we must show that

$$\text{Valid}(\mathcal{E}, \mathcal{S}) \wedge \mathcal{E} \vdash_{smt} \langle [C \Rightarrow A], \mathcal{S} \rangle \rightsquigarrow \mathcal{S}' \Rightarrow \text{Valid}(\mathcal{E}, \mathcal{S}')$$

In general, a reaction  $[C \Rightarrow A]$  denotes a set of transformers (noted  $\mathcal{ST}(C, A)$ ) and shape checking has been applied to all the transformers of this set. The proof boils down to showing that the translation of a reaction modifies the store in the same way as a transformer in  $\mathcal{ST}(C, A)$ . That is to say,

$$\begin{aligned} \text{If} \quad & \mathcal{E} \vdash_{smt} \langle \mathcal{T}[[C \Rightarrow A]], \mathcal{S} \rangle \rightsquigarrow \mathcal{S}' \\ \text{then} \quad & \exists (C', A') \in \mathcal{ST}(C, A) \\ \text{such that} \quad & \Psi(\mathcal{E}(s), T, \mathcal{S}) - \sigma(C') + \sigma(A') = \Psi(\mathcal{E}(s), T, \mathcal{S}') \end{aligned}$$

with  $\sigma$  a substitution from variables to locations.

Shape checking ensures that for any multiset  $M$  of shape  $T$  (so in particular for  $\Psi(\mathcal{E}(s), T, \mathcal{S})$ ) and for any transformer  $(C', A')$  of  $\mathcal{ST}(C, A)$ ,  $M - \sigma(C') + \sigma(A')$  has shape  $T$  (so in particular  $\Psi(\mathcal{E}(s), T, \mathcal{S}')$ ).

### Syntax and translation of Shape-C

The abstract syntax of Shape-C is built upon the syntax of C presented in [1] (pp. 21-24) and Figure 4 displays only the extensions to C.

The translation of Shape-C into C is described in Figure 5 and consists in expanding the syntactic sugar added to C. In Figure 5, we assume that “name” denotes a renaming of “name” avoiding name clashes.

$id$	$\in$	$Id$	C identifiers
$tid$	$\in$	$Tid$	Shape identifiers
$ad$	$\in$	$Ad$	Address identifiers
$nt$	$\in$	$NonTerm$	Nonterminal symbols
$rel$	$\in$	$Rel$	Terminal symbols (relations)
translation-unit	::=	type-def* decl* fun-def*	
type-def	::=	<b>shape</b> type-spec $tid$ { prod ; [nonterminal=prod ; ]* }	Type definition
		...	
nonterminal	::=	$nt$ $ad^*$	
prod	::=	$rel$ $ad$   $rel$ $ad$ $ad$   nonterminal   prod , prod	
init	::=	$tid$ $id$ = [   => shapexp   ]	Declaration/Initialization
type-spec	::=	<b>shape</b> $tid$	
		...	
fun-def	::=	type-spec $id$ ([type-spec $id$ ]*) {decl* init* stmt* }	
stmt	::=	[*] $id$ : [   shapexp => shapexp   ] [ <b>else</b> stmt]	Reaction
		...	
shapexp	::=	$rel$ $ad$   $rel$ $ad$ $ad$   $ad$ eq $ad$   exp   shapexp ; shapexp	eq $\in$ {==,!=}
exp	::=	[*] $id$ : [   shapexp =>   ]	Test
		<b>newshape</b> ( [   => A   ], $tid$ )	dynamic allocation
		<b>freeshape</b> ( e, $tid$ )	and deallocation
		<b>\$ad</b>	Value
		...	

Figure 4: Syntax of Shape-C

$\mathcal{D}[\{ \text{block} \}]$	= {	[ <b>struct</b> T s;]*	for all local variable s of shape T
		<b>struct</b> T *temp;	temporary variable for dynamic allocation
		[ <b>struct</b> adT *x;]*	address variables
		block	
		[ <b>deallocate</b> (s,T);]*	for all local variable s of shape T
	}		
$\mathcal{T}[\text{shape } t \text{ T } \{ \dots \}]$	=	<b>struct</b> adT { t valT ; <b>struct</b> adT *f1, ..., *fn;};	
		<b>struct</b> T {adT *p1, ..., *pm;};	
		where $p_1, \dots, p_m$ and $f_1, \dots, f_n$ are respectively the unary and binary relations occurring in the definition	
$\mathcal{T}[\text{shape } T]$	=	<b>struct</b> T	
$\mathcal{T}[T \text{ s} = [l \Rightarrow A \ l]]$	=	([xi = ( <b>struct</b> adT *) malloc(sizeof( <b>struct</b> adT)),] $i=1, \dots, n$ $\mathcal{A}[A \ s]$ )	
		where $x_1, \dots, x_n$ are the addresses occurring in A	
$\mathcal{T}[s: [l \ C \Rightarrow \ l]]$	=	$\mathcal{C}_1[C \ s]$ , $\mathcal{C}_2[C]$	
$\mathcal{T}[s: [l \ C \Rightarrow A \ l] \ \text{[else } \ S \ ]]$	=	if ( $\mathcal{C}_1[C \ s]$ , $\mathcal{C}_2[C]$ ) { [yi = ( <b>struct</b> adT *) malloc(sizeof( <b>struct</b> adT));] $i=1, \dots, m$ $\mathcal{A}[A \ s]$ ; [free(zi);] $i=1, \dots, p$ } [else S]	
		where $y_1, \dots, y_m$ are the addresses occurring in A but not in C $z_1, \dots, z_p$ are the addresses not occurring in A but appearing as the first argument of a binary relation in C.	
$\mathcal{T}[\text{newshape}( [l \Rightarrow A \ l], T )]$	=	(temp = ( <b>struct</b> T *)malloc(sizeof( <b>struct</b> T)), $\mathcal{T}[T \ *temp \ [l \Rightarrow A \ l]]$ , temp)	
$\mathcal{T}[\text{freeshape}( *i, T )]$	=	( <b>deallocate</b> (*i,T), <b>free</b> (*i))	
$\mathcal{C}_1[E ; F] \ s$	=	$\mathcal{C}_1[E] \ s$ , $\mathcal{C}_1[F] \ s$	
$\mathcal{C}_1[p \ x] \ s$	=	x = s.p	
$\mathcal{C}_1[f \ x \ y] \ s$	=	y = x->f	
	=	skip otherwise	
$\mathcal{C}_2[E ; F]$	=	$\mathcal{C}_2[E] \ \&\& \ \mathcal{C}_2[F]$	
$\mathcal{C}_2[x \ \text{eq} \ y]$	=	x eq y eq $\in \{==, !=\}$	
$\mathcal{C}_2[e]$	=	e [xi->valT / \$xi] $i=1, \dots, n$ where $\$x_1, \dots, \$x_n$ are the values occurring in e (e $\in$ exp)	
	=	1 otherwise	
$\mathcal{A}[E ; F] \ s$	=	$\mathcal{A}[E] \ s$ , $\mathcal{A}[F] \ s$	
$\mathcal{A}[p \ x] \ s$	=	s.p = x	
$\mathcal{A}[f \ x \ y] \ s$	=	x->f = y	
$\mathcal{A}[e] \ s$	=	e [xi->valT / \$xi] $i=1, \dots, n$ where $\$x_1, \dots, \$x_n$ are the values occurring in e (e $\in$ exp)	

Figure 5: Translation of Shape-C into C