Safe Pointers by Graph Transformation

Detlef Plump and Colin Runciman
Department of Computer Science
University of York

This document is a shortened version of the project proposal submitted to EPSRC. Sections with personal or financial details, for example, have been omitted.

Summary: Pointers in imperative programming languages are indispensable for the efficient implementation of many algorithms, and are ubiquitous in software at both applications and systems level. However, manipulating pointers is known to be an error-prone activity, constituting a weak point in software development. Although advances in static checks in programming systems have improved software reliability, as whole classes of fault never reach the execution stage, the type systems of current programming languages are incapable of detecting most pointer errors. The problem is that a programmer has no means to specify the intended shape of a linked data stucture, be it a list, a circular list or a tree, to mention a few examples. Consequently the type system cannot check whether pointer manipulations in a program will always preserve the intended shape.

Mathematically pointer structures are graphs, so it is natural to describe classes of pointer structures by graph grammars or graph reduction systems. These are collections of graph transformation rules which generate all intended structures from a start graph or reduce them to an accepting graph. Graph transformation rules can also specify arbitrary operations on pointer structures. Our aim is to use graph transformation rules to uniformly describe pointer-based data structures, and operations upon them, in such a way that the safety of the operations can be checked automatically. Moreover, we intend to define an extension of the C programming language for graph transformation rules. The extension will enable programmers to specify pointer structures and operations upon them, and will be translated to ordinary C by a pre-processor which also checks the safety of the specified operations.

The literature contains a large number of approaches for reasoning about pointer structures. These either aim at inferring shape information automatically from programs or rely on shape specifications of programmers to verify or type-check programs. Most of the work in the latter direction employs some kind of logic to describe pointer structures. We intend to use graph transformation rules instead, for two reasons: Firstly, their operational nature should be more appealing to programmers than logical formulas. Secondly, graph reduction systems can specify important structures, like balanced trees, that cannot be expressed, to our knowledge, in any other shape description formalism proposed in the literature.

Keywords: pointer manipulation; software reliability; program correctness; specification of linked data structures; data structure invariants; type checking; static analysis; graph transformation.

1 Background

Pointers in imperative programming languages provide a powerful and versatile programming concept which is crucial to the efficiency of many data types at both applications and systems level. On the other hand, pointer programming is very error-prone and often results in software faults. This situation has triggered a large amount of research, creating several methods for improving the safety of pointer-based programs.

Hoare-style program proving. The classical approach to program reliability is Hoare-style program proving with pre- and postconditions for statements. The challenge is to find suitable assertion languages and logics for expressing properties of linked data structures and reasoning about them [20, 18, 2, 13]. Current tools in this area do not support pointer programming.

Compile-time methods. Another line of research aims at analysing and checking programs at compile time. These approaches can be roughly divided into two categories by distinguishing whether the programmer is involved or not:

Fully automatic analysis. Many papers describe techniques for the fully automatic analysis of programs to infer information about heap-allocated data at all program points. This kind of static analysis—coined *shape analysis* in [23]—attempts to determine, for example, whether two pointer expressions may denote the same heap cell, whether a specific heap cell may be shared, whether a heap cell may occur in a cycle or, more generally, what will be the shape of the heap contents. Work along this line includes [14, 3, 16, 5, 6, 8, 9, 22, 23]. These approaches are too limiting for checking the safety of pointer manipulations, though, because users cannot supply the intended shape of a data structure.

Shape specifications. The second category of papers employs programming language extensions which enable programmers to specify the shape of intended pointer structures. Shape specifications can then be used to check statically whether pointer manipulations in a program will preserve the intended class of structures. This can be seen as an advanced form of type checking or as a light form of program verification. Moreover, shape specifications can be exploited by optimising compilers. The ADDS annotation formalism of [12] uses *dimensions* and *directions* to describe properties of pointer structures. ADDS is quite limited because it provides only a form of loose specification which cannot define the exact shape of a structure. *Graph types* as introduced in [15] are spanning trees with additional pointers defined by path expressions. In [19] graph types form the basis of *pointer assertion logic*, a monadic second-order logic which is used to express properties of pointer structures in program annotations. This approach has the drawback that a user has to learn a quite sophisticated logic—but the formalism is still too weak to express some important data structures, like balanced trees. This drawback also applies to the TVLA system [17], which requires three-valued logic for specifying properties of linked data structures and employs shape analysis algorithms from [22] to verify invariants. Finally, context-free *graph grammars* can be used to specify so-called *shape types* [7]. This approach is closest to the method we propose for this project. We will address the advantages of our method when we discuss timeliness and novelty.

2 Programme and methodology

Overall aim To use graph transformation rules to uniformly describe pointer-based data structures, and operations upon them, in such a way that the safety of the operations can be checked automatically.

Objectives

1. A new method for specifying pointer structures and operations upon them, using graph reduction systems and general graph transformation rules;

2. an algorithm for checking the safety of the specified operations;
3. theoretical results about the relative power of the new method with respect to previous approaches;
4. a range of example data structures and operations upon them specified by the new method, including examples of practical relevance that go beyond the scope of previous methods;
5. a prototype implementation with a pre-processor for the C programming language, including a language-independent checking algorithm.

In addition, we will explore how the method can be extended to cover cases where operations temporarily violate data structure invariants.

Timeliness and novelty The quest for an adequate treatment of pointers when reasoning formally about programs is as old as the idea of program verification. Despite the substantial progress made in recent years, a satisfying solution is still outstanding. The most advanced systems for verifying properties of pointer programs automatically, TVLA [17] and PALE [19], require user specifications in rather sophisticated logics. TVLA needs a specification in three-valued logic of the effect of each statement in the programming language on the data structure in question, and PALE requires program annotations in monadic second order logic on graph types.

In contrast, the method proposed here abandons logical formulas and uniformly describes pointer structures and their operations by graph transformation rules. We expect that the operational and direct nature of such rules will make them more appealing to programmers than formulas in some sophisticated logic. (Another advantage, not central to this project, is that graph transformation rules can be visually represented. This may help to popularize the style of programming advocated by our method.) The main advantage of our approach, though, is that graph reduction systems can specify important data structures, like balanced binary trees, that go beyond the expressiveness of previously proposed formalisms. This applies not only to the logics of [1, 17, 19], but also to the context-free graph grammars of [7]. Despite their expressive power, graph reduction systems enjoy a decidable model-checking problem: membership of a graph in the graph class specified by a system is decidable. Compared with [7], an additional advantage is that model-checking for confluent graph reduction systems—which still capture balanced binary trees—requires only polynomial time in the size of an input graph, whereas model-checking for context-free graph grammars is NP-complete.

Methodology Pointer structures can be directly modelled as graphs, so it is natural to describe classes of pointer structures by graph grammars or graph reduction systems. These are collections of graph transformation rules which generate all intended structures from a start graph or reduce them to an accepting graph. For example, Figure 1 shows a graph reduction system specifying circular linked lists. The rules r_1 and r_2 reduce every circular list to the accepting graph shown on the left, and every other structure to some graph different from the accepting graph.[1] In this example the reduction system becomes a context-free graph grammar by switching the left- and right-hand sides of the rules, but this is not the case in general.

We will use the so-called *double-pushout approach* to graph transformation [4, 10] as an underlying formal framework. This approach comes not only with a comprehensive theory but allows to specify arbitrary computable functions on graphs [11].

[1]The formalism underlying the rules prevents their application in situations where the unnumbered nodes in a left-hand side are incident to edges that are not part of the left-hand side.

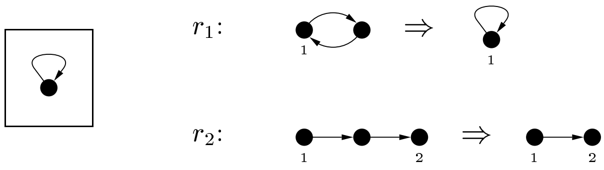

Figure 1: A graph reduction specification of circular linked lists

The graph reduction systems we will use for specifying pointer structures are collections of size-reducing rules together with an accepting graph. Particularly interesting are *confluent* reduction systems where any sequence of rule applications ends in the same graph. These systems have a simple model-checking algorithm which runs in polynomial time (in the size of the input graph): The reduction rules are applied non-deterministically as long as possible; the input graph belongs to the specified class if and only if the resulting graph is the accepting graph. Confluence of graph reduction systems can be checked by an automatic analysis of so-called *critical-pairs* [21].

Operations on pointer structures will be specified by general graph transformation rules. For example, the two reduction rules of Figure 1 happen to specify the deletion of elements in a circular list, and their inverses (obtained by switching left- and right-hand sides) specify the insertion of elements. While in this case it is obvious that the operations preserve the specified data structure, this is not so in general. Therefore an algorithm will be developed, based on critical-pair techniques, for checking the safety of the specified operations. In most cases the algorithm will either report the safety of the operations or will detect a violation of the specified data structure. In the latter case a counterexample will be automatically generated to help the user debugging the definition of the operation (or the data structure). Because of the universal power of general graph transformation rules, there will be cases where the checking algorithm has to report "don't know". We will investigate whether the algorithm can be extended to a semi-decision procedure that eventually detects every unsafe operation but does not terminate for certain safe operations (see point 4 in workplan).

Programme of work

1. *Familiarization.* The Research Associate has to become familiar with both graph transformation and the relevant literature on automatic checking of pointer-based programs.

2. *Examples.* A range of example data structures with associated operations, taken from textbooks, will be specified by the new method. Particular emphasis will be given to data structures, like balanced binary trees, that cannot be handled by previous approaches.

3. *Safety checking algorithm.* An algorithm will be developed that checks whether the specified operations of a data structure will always preserve this structure. The algorithm will internally generate and analyse so-called critical pairs of graph transformation rules.

4. *Semi-deciding and deciding safety.* We will investigate whether the safety checking algorithm can be extended to a semi-decision procedure that eventually detects every unsafe operation but does not terminate for certain safe operations. Parallel to this investigation we hope to develop conditions on specifications of data structures and operations under which the checking algorithm becomes a decision procedure for safety.

5. *Power of the method.* We shall seek theoretical results characterizing the power of our specification method for pointer structures. This includes results comparing the new method with previous approaches.

6. *Pre-processor for C.* An extension of C for graph transformation rules will be designed and implemented. The implementation will have the form of a pre-processor for C which both provides a language-independent implementation of the safety checking algorithm and generates ordinary C code.
7. *Confluence test.* A confluence test for graph reduction systems will be added to the pre-processing tool. This will be implemented by checking the convergence of all critical pairs of the rules.
8. *Extension.* We will explore how the method can be extended to cover cases where operations temporarily violate data structure invariants. Here *programs* over transformation rules, as introduced in [11], may yield a solution.
9. *Evaluation.* To evaluate the practical worth of our approach we shall pursue a small number of illustrative applications. These might include, for example, storage structures in implementations of programming languages with automatic memory management.
10. *Release and documentation.* An implementation of the pre-processor for C, together with explanatory documents, will be made available on the internet. Another task in the closing stages will be to finish writing up results of the project work for submission to conferences and journals.

3 Relevance to beneficiaries

The Technology Foresight report marked out Information Technology as a high-growth industry, underpinning many other fields of work. The means of software development, including programming languages and associated tools, are clearly of great importance for productivity and quality in Information Technology.

There is widespread use of pointer-based data structures in software systems, currently with little or no computer-supported validation of the program components that manipulate such structures. Tools for the detection and elimination of pointer-based errors at an early stage in implementation have the potential both to shorten development time and to improve confidence in the final product.

References

[1] Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for describing linked data structures. In *Proc. European Symposium on Programming Languages and Systems (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 2–19. Springer-Verlag, 1999.

[2] Richard Bornat. Proving pointer programs in Hoare logic. In *Proc. Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[3] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. Programming Language Design and Implementation (PLDI '90)*, pages 296–310, 1990.

[4] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation — Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, chapter 3, pages 163–245. World Scientific, 1997.

[5] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proc. Programming Language Design and Implementation (PLDI '94)*, pages 230–241. ACM Press, 1994.

[6] Pascal Fradet, Ronan Gaugne, and Daniel Le Métayer. Static detection of pointer errors: An axiomatisation and a checking algorithm. In *Proc. European Symposium on Programming Languages and Systems (ESOP '96)*, volume 1058 of *Lecture Notes in Computer Science*, pages 125–140, 1996.

[7] Pascal Fradet and Daniel Le Métayer. Shape types. In *Proc. Principles of Programming Languages (POPL '97)*, pages 27–39. ACM Press, 1997.

[8] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proc. Principles of Programming Languages (POPL' 96)*, pages 1–15. ACM Press, 1996.

[9] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proc. Principles of Programming Languages (POPL '98)*, pages 121–133. ACM Press, 1998.

[10] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

[11] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS '01)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2001.

[12] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3):243–260, 1992.

[13] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. Principles of Programming Languages (POPL '01)*. ACM Press, 2001.

[14] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, 1981.

[15] Nils Klarlund and Michael Schwartzbach. Graph types. In *Proc. Principles of Programming Languages (POPL '93)*, pages 196–205. ACM Press, 1993.

[16] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem classification. In *Proc. Principles of Programming Languages (POPL '91)*, pages 93–103. ACM Press, 1991.

[17] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *Proc. International Symposium on Software Testing and Analysis (ISSTA '00)*, pages 26–38. ACM Press, 2000.

[18] David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, 1979.

[19] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proc. Programming Language Design and Implementation (PLDI '01)*, pages 221–231. ACM Press, 2001.

[20] Derek C. Oppen and Stephen A. Cook. Proving assertions about programs that manipulate data structures. In *Proc. Symposium on Theory of Computation (STOC '75)*, pages 107–116. ACM Press, 1975.

[21] Detlef Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 15, pages 201–213. John Wiley, 1993.

[22] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. Principles of Programming Languages (POPL '99)*, pages 105–118. ACM Press, 1999.

[23] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In *Proc. Compiler Construction (CC '00)*, volume 1781 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2000.