Derivation of Data Intensive Algorithms by Formal Transformation

The Schorr-Waite Graph Marking Algorithm

Martin Ward
email: Martin.Ward@durham.ac.uk
URL: http://www.dur.ac.uk/∼dcs0mpw
Computer Science Dept
Science Labs
South Rd
Durham DH1 3LE, UK

January 17, 2003

Abstract

In this paper we consider a particular class of algorithms which present certain difficulties to formal verification. These are algorithms which use a single data structure for two or more purposes, which combine program control information with other data structures or which are developed as a combination of a basic idea with an implementation technique. Our approach is based on applying proven semantics-preserving transformation rules in a wide spectrum language. Starting with a set theoretical specification of "reachability" we are able to derive iterative and recursive graph marking algorithms using the "pointer switching" idea of Schorr and Waite. There have been several proofs of correctness of the Schorr-Waite algorithm, and a small number of transformational developments of the algorithm. The great advantage of our approach is that we can derive the algorithm from its specification using only general-purpose transformational rules: without the need for complicated induction arguments. Our approach applies equally well to several more complex algorithms which make use of the pointer switching strategy, including a hybrid algorithm which uses a fixed length stack, switching to the pointer switching strategy when the stack runs out.

Keywords:

Program development, transformation, transformational development, refinement, graph marking, Schorr-Waite, ghost variables, data refinement, recursion removal, pointer switching.

1 Introduction

There are several approaches to the formal development of executable programs from abstract specifications (expressed in say First Order Logic), for instance:

1. Write the program and then attempt to formally verify its correctness against the specification. This has the problem that most informally developed programs will contain bugs, so the verification step is almost certain to fail. Also, for a large and complex program, after the fact verification is extremely difficult;
2. Develop the program and its correctness proof concurrently, this is the approach used by Gries [18] for example;
3. Starting with the specification, successively transform it into an executable program. There are two variants on this approach:
   (a) The user invents *ad hoc* refinements at each stage which must be verified after the fact. Each step will therefore carry with it a set of *proof obligations*, which are theorems which must be proved for the refinement step to be valid;
   (b) The user selects and apply a sequence of *transformation rules* from a catalogue of rules, which have been previously proven to preserve the semantics. The correctness of the resulting program is thus guaranteed *by construction.*

Approach (3a), which is generally favoured in the **Z** and VDM communities, is used in several program development systems including *μ*ral [23], RAISE [31] and the B-tool [1]. These systems thus have a much greater emphasis on proofs, rather than the selection and application of transformation rules. Discharging these proof obligations can often involve a lot of tedious work, and much effort is being exerted to apply automatic theorem provers to aid with the simpler proofs. However, Sennett in [36] indicates that for "real" sized programs it is impractical to discharge much more than a tiny fraction of the proof obligations. He presents a case study of the development of a simple algorithm, for which the implementation of one function gave rise to over one hundred theorems which required proofs. Larger programs will require many more proofs. In practice, since few if any of these proofs will be rigorously carried out, what claims to be a formal method for program development turns out to be a formal method for program specification, together with an *informal* development method. For this approach to be used as a reverse-engineering method, it would be necessary to discover suitable loop invariants for each of the loops in the given program, and this is very difficult in general, especially for programs which have not been developed according to some structured programming method.

Transformational development seems to be the most suitable for scaling up to large programs: this is because a single proof of a large program will be almost impossible to understand let alone develop, while transformational development allows the "proof" to be broken down into small manageable steps. The great advantage of method (3b) over (3a) is that the proof steps only need be carried out once for each transformation: once a transformation has been proved and its correctness conditions determined, it can be applied to many different programs without generating further proof obligations. This is particularly advantageous when the transformation process can be carried out by a computer system which takes care of checking all applicability conditions, applying the transformations, and maintaining the various program versions.

There are two types of proof theory in transformational programming: the algebraic and the model based. The algebraic approach describes programs as an algebra of expressions: a set of axioms is given which defines certain equivalences and refinements between programs. Two general programs are defined to be equivalent if there exists a sequence of these "axiomatic transformations" which will transform one program into the other. The general idea is that the "meaning" of a programming construct is defined by giving a list of properties which the construct must satisfy. Any other construct which satisfies these properties can be substituted for the given one and all proofs will still be valid. In contrast, the model based approach gives a model for the semantics of the programming language which maps each program to some mathematical object (such as a function or a relation) which is its “semantics”. Two programs are defined to be equivalent if they have the same semantics.

In the model based approach, all the required proofs are captured in a library of general purpose transformations, but if the model is weak a plethora of low-level transformations is applicable at each step (as has been reported by Balzer [6]). In contrast, experience to date with transformations defined using denotational semantics, and proved using weakest preconditions expressed in infinitary logic, suggests that the method is sufficiently powerful to form the basis of a practical transformation system. The basis for our formal transformation theory is a Wide Spectrum Language (called WSL), developed in [39,43] which includes low-level programming constructs and high-level abstract specifications within a single language. Working within a single language means that the proof that a program correctly implements a specification, or that a specification correctly captures the behaviour of a program, can be achieved by means of formal transformations in the language. We don’t have to develop transformations between the “programming” and “specification” languages. An added advantage is that different parts of the program can be expressed at different levels of abstraction, if required.

1.1 Criteria for Success

We consider the following criteria to be important for any practical wide-spectrum language and transformation theory:

1. General specifications in any “sufficiently precise” notation should be included in the language. For sufficiently precise we will mean anything which can be expressed in terms of mathematical logic with suitable notation. This will allow a wide range of forms of specification, for example **Z** specifications [20] and **VDM** [22] both use the language of mathematical logic and set theory (in different notations) to define specifications. The “Representation Theorem” (proved in [43]) proves that our specification statement is sufficient to specify *any* WSL program (and therefore any computable function, since WSL is certainly Turing complete);
2. Nondeterministic programs. Since we do not want to have to specify everything about the program we are working with (certainly not in the first versions) we need some way of specifying that some executions will not necessarily result in a particular outcome but one of an allowed range of outcomes. The implementor can then use this latitude to provide a more efficient implementation which still satisfies the specification. (Note that the example in this paper is a deterministic program derived from a deterministic specification);
3. A well-developed catalogue of proven, general-purpose transformations which do not require the user to discharge complex proof obligations before they can be applied. In particular, it should be possible to introduce, analyse and reason about imperative and recursive constructs without requiring loop invariants. It should also be possible to derive algorithms from specifications without recourse to complicated induction proofs;
4. Techniques to bridge the “abstraction gap” between specifications and programs. See Section 4.3 and [45,49] for examples;
5. Applicable to real programs—not just those in a “toy” programming language with few constructs. This is achieved by the (programming) language independence and extendibility of the notation via “definitional transformations”. See [40,42] for examples;
6. Scalable to large programs: this implies a language which is expressive enough to allow automatic translation from existing programming languages, together with the ability to cope with unstructured programs and a high degree of complexity. Obviously, space constraints preclude the detailed discussion of a really large program, instead we will focus on scaling the method from simple algorithms to complex data-intensive algorithms. See [46] for a description of the FermaT tool: a transformation system based on our theory which has been

used successfully to reverse engineer commercial Assembler modules ranging up to 40,000 lines of code.

A system which meets all these requirements would have immense practical importance in the following areas:

- Improving the maintainability (and hence extending the lifetime) of existing mission-critical software systems;
- Translating programs to modern programming languages, for example from obsolete Assembler languages to modern high-level languages;
- Developing and maintaining safety-critical applications. Such systems can be developed by transforming high-level specifications down to efficient low level code with a very high degree of confidence that the code correctly implements every part of the specification. When enhancements or modifications are required, these can be carried out on the appropriate specification, followed by "re-running" as much of the formal development as possible. Alternatively, the changes could be made at a lower level, with formal inverse engineering used to determine the impact on the formal specification;
- Extracting reusable components from current systems, deriving their specifications and storing the specification, implementation and development strategy in a repository for subsequent reuse. This is discussed in [44].

1.2 Our Approach

In developing a model based theory of semantic equivalence, we use the popular approach of defining a core "kernel" language with denotational semantics, and permitting definitional extensions in terms of the basic constructs. In contrast to other work (for example, [7,8,32]) we do not use a purely applicative kernel; instead, the concept of state is included, using a *specification statement* which also allows specifications expressed in first order logic as part of the language, thus providing a genuine wide spectrum language.

Fundamental to our approach is the use of infinitary first order logic (see [25]) both to express the weakest preconditions of programs [13] and to define assertions and guards in the kernel language. We use the logic $\mathcal{L}_{\omega_1\omega}$ which allows countably infinite disjunction and conjunction, and finite nesting of quantifiers. It is the smallest infinitary logic. Engeler [14] was the first to use infinitary logic to describe properties of programs; Back [4] used such a logic to express the weakest precondition of a program as a logical formula. His kernel language was limited to simple iterative programs. We use a different kernel language which includes recursion and guards, so that Back's language is a subset of ours. We show that the introduction of infinitary logic as part of the language (rather than just the metalanguage of weakest preconditions), together with a combination of proof methods using both denotational semantics and weakest preconditions, is a powerful theoretical tool which allows us to prove some general transformations and representation theorems.

The denotational semantics of the kernel language is based on the semantics of infinitary first order logic. Kernel language statements are interpreted as functions which map an initial state to a set of final states (the *set* of final states models the nondeterminacy in the language: for a deterministic program this set will contain a single state). A program $\mathbf{S}_1$ is a refinement of $\mathbf{S}_2$ if, for each initial state, the set of final states for $\mathbf{S}_1$ is a subset of the final states for $\mathbf{S}_2$. Back and von Wright [5] note that the refinement relation can be characterised using weakest preconditions in higher order logic (where quantification over formulae is allowed). For any two programs $\mathbf{S}_1$ and $\mathbf{S}_2$, the program $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$ if the formula $\forall \mathbf{R}.\, \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ is true. This approach to refinement has two problems:

1. It can be difficult to find a finite formula which characterises the weakest precondition of a general loop or recursive statement. Suitable invariants can sometimes provide a sufficiently

good approximation to the weakest precondition but, as already noted, these can be difficult to discover for large and complex programs;

2. Second order logic is *incomplete* in the sense that not all true statements are provable. So even if the refinement is true, there may not exist a proof of it.

In [43] we present solutions to both of these problems. Using infinitary logic allows us to give a simple definition of the weakest precondition of *any* statement (including an arbitrary loop) for any postcondition.

A program $\mathbf{S}$ is a piece of formal text, i.e. a sequence of formal symbols. There are two ways in which we interpret (give meaning to) these texts:

1. Given a structure M for the logical language $\mathcal{L}$ from which the programs are constructed, and a final state space (from which we can construct a suitable initial state space), we can interpret a program as a function f (a *state transformation*) which maps each initial state s to the set of possible final states for s. By itself therefore, we can interpret a program as a function from structures to state transformations;
2. Given any formula $\mathbf{R}$ (which represents a condition on the final state), we can construct the formula $\mathrm{WP}(\mathbf{S}, \mathbf{R})$, the *weakest precondition* of $\mathbf{S}$ on $\mathbf{R}$. This is the weakest condition on the initial state such that the program $\mathbf{S}$ is guaranteed to terminate in a state satisfying $\mathbf{R}$ if it is started in a state satisfying $\mathrm{WP}(\mathbf{S}, \mathbf{R})$. For example, the weakest precondition of the **if** statement **if** $\mathbf{B}$ **then** $\mathbf{S}_1$ **else** $\mathbf{S}_2$ is $(\mathbf{B} \Rightarrow \mathrm{WP}(\mathbf{S}_1, \mathbf{R})) \wedge (\neg\mathbf{B} \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R}))$. Using infinitary logic we can define the weakest precondition of a recursive statement as the countable disjunction of the weakest preconditions of the finite truncations:

$$\mathrm{WP}(\underline{\mathbf{proc}}\ F \ \equiv\ \mathbf{S}., \mathbf{R}) =_{\mathrm{DF}} \bigvee_{n<\omega} \mathrm{WP}(\underline{\mathbf{proc}}\ F \ \equiv\ \mathbf{S}.^n, \mathbf{R})$$

   where $\underline{\mathbf{proc}}\ F \ \equiv\ \mathbf{S}.^0 =_{\mathrm{DF}}$ **abort** and $\underline{\mathbf{proc}}\ F \ \equiv\ \mathbf{S}.^{n+1} =_{\mathrm{DF}} \mathbf{S}[\underline{\mathbf{proc}}\ F \ \equiv\ \mathbf{S}.^n/F]$ which is $\mathbf{S}$ with each occurrence of F replaced by $\underline{\mathbf{proc}}\ F \ \equiv\ \mathbf{S}.^n$.

These interpretations give rise to two different notions of refinement: *semantic refinement* and *proof-theoretic refinement.*

1.3 Semantic Refinement

A *state* is a collection of variables (the *state space*) with values assigned to them; thus a state is a function which maps from a (finite, non-empty) set of variables to a set of values. There is a special extra state $\perp$ which is used to represent nontermination or error conditions. A state transformation f maps each initial state s in one state space, to the set of possible final states $f(s)$ which may be in a different state space. If $\perp$ is in $f(s)$ then so is every other state; also $f(\perp)$ is the set of all states (including $\perp$).

Semantic refinement is defined in terms of these state transformations. A state transformation f is a refinement of a state transformation g if they have the same initial and final state spaces and $f(s) \subseteq g(s)$ for every initial state s. Note that if $\perp \in g(s)$ for some s, then $f(s)$ can be anything at all. In other words we can correctly refine an "undefined" program to do anything we please. If f is a refinement of g (equivalently, g is refined by f) we write $g \ \leq\ f$. A *structure* for a logical language $\mathcal{L}$ consists of a set of values, plus a mapping between constant symbols, function symbols and relation symbols of $\mathcal{L}$ and elements, functions and relations on the set of values. A model for a set of sentences (formulae with no free variables) is a structure for the language such that each of the sentences is interpreted as true. If the interpretation of statement $\mathbf{S}_1$ under the structure M is refined by the interpretation of statement $\mathbf{S}_2$ under the same structure, then we write $\mathbf{S}_1 \ \leq_M \mathbf{S}_2$. If this is true for every model of a countable set Δ of sentences of $\mathcal{L}$ then we write $\Delta \models \mathbf{S}_1 \ \leq\ \mathbf{S}_2$.

1.4 Proof-Theoretic Refinement

Given two statements $\mathbf{S}_1$ and $\mathbf{S}_2$, let $\mathbf{x}$ be a sequence of all the variables assigned in either $\mathbf{S}_1$ or $\mathbf{S}_2$ and let $\mathbf{x}'$ be a sequence of new variables of the same length as $\mathbf{x}$. We can construct the two (infinitary logic) formulae $\mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}')$ and $\mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}')$. If there exists a proof of the formula $\mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}') \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}')$ using a set Δ of sentences as assumptions, then we write $\Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$.

A fundamental result, proved in [43] is that these two notions of refinement are equivalent. More formally:

Theorem 1.1 *For any statements $\mathbf{S}_1$ and $\mathbf{S}_2$, and any countable set Δ of sentences of $\mathcal{L}$:*

$$\Delta \models \mathbf{S}_1 \leq \mathbf{S}_2 \iff \Delta \vdash \mathbf{S}_1 \leq \mathbf{S}_2$$

These two equivalent definitions of refinement give rise to two very different proof methods for proving the correctness of refinements. Both methods are exploited in [43]—weakest preconditions and infinitary logic are used to develop the induction rule for recursion and the recursive implementation theorem, while state transformations are used to prove the representation theorem.

The theorem illustrates the importance of using $\mathcal{L}_{\omega_1\omega}$ rather than a higher-order logic, or indeed a larger infinitary logic. Back and von Wright [5] describe an implementation of the refinement calculus, based on (finitary) higher-order logic using the refinement rule $\forall \mathbf{R}.\, \mathrm{WP}(\mathbf{S}_1, \mathbf{R}) \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{R})$ where the quantification is over all formulae. However, the completeness theorem fails for all higher-order logics. Karp [25] proved that the completeness theorem holds for $\mathcal{L}_{\omega_1\omega}$ and fails for all infinitary logics larger than $\mathcal{L}_{\omega_1\omega}$. Finitary logic is not sufficient since it is difficult to determine a finite formula giving the weakest precondition for an arbitrary recursive or iterative statement. Using $\mathcal{L}_{\omega_1\omega}$ (the smallest infinitary logic) we simply form the infinite disjunction of the weakest preconditions of all finite truncations of the recursion or iteration. We avoid the need for quantification over formulae because with our proof-theoretic refinement method the single postcondition $\mathbf{x} \neq \mathbf{x}'$ is sufficient. Thus we can be confident that the proof method is complete, in other words if $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ then there exists a proof of $\mathrm{WP}(\mathbf{S}_1, \mathbf{x} \neq \mathbf{x}') \Rightarrow \mathrm{WP}(\mathbf{S}_2, \mathbf{x} \neq \mathbf{x}')$. Basing our transformation theory on any other logic would not provide the two different proof methods we require.

2 Data Intensive Programs

In the context of deriving algorithms from specifications, a serious problem is that there may be no simple relationship between the specification structure and the program structure. A particular case is where the program modifies a complex data structure as it operates, while this data structure directs the operation of the program: an example is using a stack to implement recursion. The problem is compounded when the implementor wishes (for efficiency reasons) to combine two or more "abstract" data structures into a single "concrete" data structure. Such a refinement step does not fit into the top-down paradigm, and this is where the techniques of formal program transformation come into play. We class this sort of program as a *data intensive program.* A data intensive program satisfies one or more of the following:

- Using one data structure for two or more purposes: i.e. two or more abstract data types are implemented in a single concrete data object;
- Combining control flow information (such as a protocol stack) with other data structures used by the program;
- A program which is developed as a combination of a basic algorithm together with an implementation technique which reduces the amount of storage required for the basic algorithm.

Ideally, the derivation of the algorithm should treat the basic algorithm and implementation technique separately. This suggests a transformational approach where the different ideas are introduced in stages, starting with an abstract specification. An algorithm which exhibits all these properties is the Schorr-Waite graph marking algorithm, first published in 1967 [35]: we have therefore selected this algorithm as our main example.

2.1 Ghost Variables

The usual approach to "data reification" (a refinement which changes the representation of data in the program) involves defining an *abstraction function*—a function from concrete states to abstract states which shows precisely which abstract state is represented by a particular concrete state (see for example [12,29]). For simple cases such a stacks and sets, there is no difficulty in finding such a function, but for data intensive programs the relationship between abstract and concrete states may be very complicated. Also, the relationship may be a general relation rather than a strict function. Programs which implement several data objects in a single data structure can be particularly hard to analyse, especially if one of the data objects is also used to direct the control flow of the program. Our approach to data intensive programs is based on the use of "ghost variables" which are variables which have no effect on the computation, but which shadow the actual variables used in the program. This technique is used in [9,24,47]. We use ghost variables to define intermediate stages in the development in which both abstract and concrete data structures are present. We introduce the ghost variables in the recursive version of the program, two trivial invariants are used which are proved correct using only local information. These invariants are used to gradually replace references to abstract states by appropriate references to concrete states. Thus a formal derivation can be achieved without knowing the precise relationship between abstract and concrete data structures. The method uses the following stages:

1. Start with a formal specification and develop a recursive algorithm using the recursion implementation theorem [43]. This uses abstract data structures;
2. Introduce parallel data structures (which will ultimately be the concrete data structures) as a collection of "ghost variables". These are variables which are assigned to but never accessed (except within assignments to ghost variables). Assign values to these variables so as to maintain local invariants between the two data structures. The key idea here is that only local information is required to prove the correctness of these invariants; all the work of the algorithm is done by the abstract variables which are still present and can be referred to;
3. Gradually replace references to abstract variables by references to the new variables. Since both sets of variables are present in the program, the references can be replaced in stages, perhaps removing the variables one at a time. Keep in the assignments to the abstract variables so that their values are still the same and they can still appear in invariants;
4. Once all references to the abstract variables have been replaced, the abstract variables become "ghost variables" and can be removed from the program without affecting its operation;
5. At the last possible moment, remove the recursion using a general recursion removal theorem (see Theorem 4.5 below). This is best done as late as possible since recursive programs are generally easier to manipulate than iterative ones;
6. Finally, the iterative algorithm can be restructured by applying further transformations.

The abstract program in stage 2 acts as a "scaffolding" which ensures that the program gives the correct result. We build the "real" program, using the scaffolding as support. In stage 3 we gradually transfer the weight onto the real program. Finally, all the work is done by the real program and the scaffolding can be demolished.

The advantages of this approach are:

- The use of ghost variables means that the assertions we introduce are simple and easy to prove. We do not need to determine the precise relationship between abstract and concrete states, only that the ghost variables are preserved by the recursive calls;

- We remove the recursion at a late stage in the development, using a powerful recursion removal transformation followed by general-purpose restructuring transformations;
- The most important benefit is that we can separate the algorithm idea (depth-first searching) from the implementation technique (pointer switching). This separation of concerns provides a method for applying the pointer switching technique to many other algorithms. We use the same implementation technique to derive a "hybrid" algorithm which combines the time efficiency of the recursive algorithm with the space efficiency of the pointer switching strategy.

In the next section we introduce the wide spectrum language WSL and give some example transformations, and then go on to tackle the algorithm derivation. See [39,43] for the formal semantics of WSL and details of the kernel language from which it is developed.

3 The Wide Spectrum Language

Within expressions we use the following notation:

Sequences: $s = \langle a_1, a_2, \ldots, a_n \rangle$ is a sequence, the ith element a_i is denoted $s[i]$, $s[i\,..\,j]$ is the subsequence $\langle s[i], s[i+1], \ldots, s[j] \rangle$, where $s[i\,..\,j] = \langle\rangle$ (the empty sequence) if $i > j$. The length of sequence s is denoted $\ell(s)$, so $s[\ell(s)]$ is the last element of s. We use $s[i\,..]$ as an abbreviation for $s[i\,..\,\ell(s)]$.

Sequence concatenation: $s_1 +\!\!\!+\, s_2 = \langle s_1[1], \ldots, s_1[\ell(s_1)], s_2[1], \ldots, s_2[\ell(s_2)] \rangle$.

Stacks: Sequences are also used to implement stacks. For this purpose we have the following notation: For a sequence s and variable x: $x \xleftarrow{\text{pop}} s$ means $x := s[1]$; $s := s[2\,..]$ which pops an element of the stack into variable x. To push the value of the expression e onto stack s we use: $s \xleftarrow{\text{push}} e$ which is equivalent to: $s := \langle e \rangle +\!\!\!+\, s$.

Sets: We have the usual set operations $\cup$ (union), $\cap$ (intersection) and $-$ (set difference), $\subseteq$ (subset), $\in$ (element), $\wp$ (powerset). $\{\, x \in A \mid P(x) \,\}$ is the set of all elements in A which satisfy predicate P. For the sequence s, $\mathsf{set}(s)$ is the set of elements of the sequence, i.e. $\mathsf{set}(s) = \{\, s[i] \mid 1 \leqslant i \leqslant \ell(s) \,\}$. For a set A, $\#A$ denotes the number of elements of A.

We use the following notation for statements, where $\mathbf{S}$, $\mathbf{S}_1$, $\mathbf{S}_2$ etc. are statements and $\mathbf{B}$, $\mathbf{B}_1$, $\mathbf{B}_2$, etc. are formulae of first order logic:

Sequential composition: $\mathbf{S}_1;\ \mathbf{S}_2;\ \mathbf{S}_3;\ \ldots;\ \mathbf{S}_n$

Deterministic Choice: $\underline{\textbf{if}}\ \mathbf{B}\ \underline{\textbf{then}}\ \mathbf{S}_1\ \underline{\textbf{else}}\ \mathbf{S}_2\ \underline{\textbf{fi}}$

Assertion: $\{\mathbf{B}\}$ is an assertion: it acts as a partial **skip** statement, it aborts if the condition is false but does nothing if the condition is true. Thus inserting an assertion statement by means of a transformation is the same as proving that the condition is always true at that point.

Assignment: $\mathbf{x} := \mathbf{x}'.\mathbf{Q}$ is a general assignment which assigns new values to the list $\mathbf{x}$ of variable such that the condition $\mathbf{Q}$ is true. Unprimed variables in $\mathbf{Q}$ refer to initial values and primed variables refer to the values assigned, thus $\langle x \rangle := \langle x' \rangle.(x' = x + 1)$ increments the value of variable x

Simple Assignment: if $\mathbf{x}$ is a list of variables and $\mathbf{t}$ a list of expressions, then $\mathbf{x} := \mathbf{t}$ means $\mathbf{x} := \mathbf{x}'.(\mathbf{x}' = \mathbf{t})$

Variable Rotation: $\circlearrowleft\langle x_1, x_2, \ldots, x_n \rangle$ where the x_i are variables or array references, stands for the assignment: $\langle x_1, x_2, \ldots, x_n \rangle := \langle x'_1, x'_2, \ldots, x'_n \rangle.(x'_1 = x_2 \land x'_2 = x_3 \land \cdots \land x'_n = x_1)$. This therefore "rotates" the values of the variables or array references. For example, $\circlearrowleft\langle a[i], a[j] \rangle$ swaps the values of $a[i]$ and $a[j]$.

Nondeterministic Choice: The "guarded command" of Dijkstra [13]:

$$\begin{array}{l}\underline{\textbf{if}}\ \mathbf{B}_1 \rightarrow \mathbf{S}_1 \\ \Box\ \mathbf{B}_2 \rightarrow \mathbf{S}_2 \\ \quad \ldots \\ \Box\ \mathbf{B}_n \rightarrow \mathbf{S}_n\ \underline{\textbf{fi}}\end{array}$$

Deterministic Iteration: **while** **B** **do** **S** **od**

Initialised local Variables: **var** **x** := **t**: **S** **end**

Counted Iteration: **for** $i := b$ **to** f **step** s **do** **S** **od** Here, i is local to the body of the loop

Recursive procedure: **proc** $X \equiv$ **S.** Here X is a statement variable, an occurrence of X within **S** is a call to the procedure.

Block with local procedure: **begin** $\mathbf{S}_1$ **where** **proc** $X \equiv \mathbf{S}_2$**.** **end** This is equivalent to the statement $\mathbf{S}_1$[**proc** $X \equiv \mathbf{S}_2$**.**$/X$] which consists of $\mathbf{S}_1$ with each occurrence of X replaced by the recursive statement **proc** $X \equiv$ **S.**.

Unbounded loops and exits: Statements of the form **do** **S** **od**, where **S** is a statement, are "infinite" or "unbounded" loops which can only be terminated by the execution of a statement of the form **exit**(n) (where n is an integer, *not* a variable or expression) which causes the program to exit the n enclosing loops. To simplify the language we disallow **exit**s which leave a block or a loop other than an unbounded loop. This type of structure is described in [26] and more recently in [37].

3.1 Action Systems

A true wide spectrum language requires some notation for unstructured transfer of control (**goto** statements). We introduce the concept of an *Action System* as a set of parameterless mutually recursive procedures. A program written using labels and jumps translates directly into an action system. Note however that if the end of the body of an action is reached, then control is passed to the action which called it (or to the statement following the action system) rather than "falling through" to the next label. The exception to this is a special action called the terminating action, usually denoted **Z**, which when called results in the immediate termination of the whole action system.

Arsac [2,3] uses a restricted definition of actions together with deterministic assignments, the binary **if** statement and **do** loops with **exit**s; so there is no place for nondeterminism in his results. The main differences between our action systems and Arsac's are: (i) that we use a much more powerful language (including general specifications), (ii) we give a formal definition (ultimately in terms of denotational semantics, see [39]), and (iii) our action systems are simple statements which can form components of other constructs.

The idea of an action system as a single statement is what gives the recursion removal theorems of Section 4.5 much of their power and generality. We can restructure the body of *any* recursive procedure into a form in which the theorem applies. Often there will be several different forms for which the theorem applies: these will yield different (but equivalent) iterative versions of the procedure.

Definition 3.1 An *action* is a parameterless procedure acting on global variables (cf [2,3]). It is written in the form $A \equiv$ **S.** where A is a statement variable (the name of the action) and **S** is a statement (the action body). A set of (mutually recursive) actions is called an *action system.* There may sometimes be a special action (usually denoted Z), execution of which causes termination of the whole action system: even if there are unfinished recursive calls. An occurrence of a statement **call** X within the action body denotes a call of another action.

A collection of actions forms an action system:

```
actions $A_1$ :
$A_1 \equiv \mathbf{S}_1$.
$A_2 \equiv \mathbf{S}_2$.
...
$A_n \equiv \mathbf{S}_n$. endactions
```

where statements $\mathbf{S}_1, \dots, \mathbf{S}_n$ must have no **exit**(m) statements within less than m nested loops.

Definition 3.2 An action is *regular* if every execution of the action leads to an action call. (This is similar to a regular rule in a Post production system [33]). A regular action system can only be terminated by a call to Z.

Definition 3.3 An action system is regular if every action in the system is regular. Any algorithm defined by a flowchart, or a program which contains labels and **goto**s but no procedure calls in non-terminal positions, can be expressed as a regular action system.

3.2 Procedures and Functions with Parameters

For simplicity we only use procedures with parameters which are called by value or by value-result. Here the value of the actual parameter is copied into a local variable which replaces the formal parameter in the body of the procedure. For result parameters, the final value of this local variable is copied back into the actual parameter. In this case the actual parameter must be a variable or some other object (e.g. an array element) which can be assigned a value. Such objects are often denoted as "L-values" because they can-occur on the left of assignment statements.

The reason for concentrating on value parameters is that they avoid some of the problems caused by "aliasing" where two variable names refer to the same object. For example if a global variable of the procedure is also used as a parameter, or if the same variable is uses for two actual parameters then with other forms of parameter passing aliasing will occur but with value parameters the aliasing is avoided (unless the same variable is used for two result parameters and the procedure tries to return two different values). This means that procedures with value parameters have simpler semantics.

4 Example Transformations

In this section we describe a few of the transformations we will use later:

4.1 Re-arrange IF statement

Theorem 4.1 If the conditions $\mathbf{Q}_{ij}$ for each i and j are disjoint (i.e. $\mathbf{Q}_{ij} \wedge \mathbf{Q}_{kl} \Leftrightarrow$ **false** whenever $i \neq j$ or $k \neq l$) then the nested **if** statement:

```
if $\mathbf{B}_1 \rightarrow$ if $\mathbf{B}_{11} \rightarrow \{\mathbf{Q}_{11}\};\ \mathbf{S}_{11}$
           ...
           $\square\ \mathbf{B}_{1j} \rightarrow \{\mathbf{Q}_{1j}\};\ \mathbf{S}_{1j}$
           ... fi
...
$\square\ \mathbf{B}_i \rightarrow$ if $\mathbf{B}_{i1} \rightarrow \{\mathbf{Q}_{i1}\};\ \mathbf{S}_{i1}$
           ...
           $\square\ \mathbf{B}_{ij} \rightarrow \{\mathbf{Q}_{ij}\};\ \mathbf{S}_{ij}$
           ... fi
... fi
```

can be refined to the flattened **if** statement which tests $\mathbf{Q}_{ij}$ instead of $\mathbf{B}_{ij}$:

```
if $\mathbf{Q}_{11} \rightarrow \mathbf{S}_{11}$
...
$\square\ \mathbf{Q}_{1j} \rightarrow \mathbf{S}_{1j}$
```

...
$\Box\ \mathbf{Q}_{i1} \rightarrow \mathbf{S}_{i1}$
...
$\Box\ \mathbf{Q}_{ij} \rightarrow \mathbf{S}_{ij}$
... **fi**

The proof is by case analysis on the $\mathbf{Q}_{ij}$.

4.2 Recursive Implementation of Statements

In this section we present an important theorem on the recursive implementation of statements, and show how it can be used in a method for transforming a general specification into an equivalent recursive statement. These transformations can be used to implement recursive specifications as recursive procedures, to introduce recursion into an abstract program to get a "more concrete" program (i.e. closer to a programming language implementation), and to transform a given recursive procedure into a different form. The theorem is used in the algorithm derivations of [40] and [39].

Suppose we have a statement $\mathbf{S}'$ which we wish to transform into the recursive procedure $\underline{\textbf{proc}}\ F \ \equiv\ \mathbf{S}.$. We claim that this is possible whenever:

1. The statement $\mathbf{S}'$ is refined by $\mathbf{S}[\mathbf{S}'/F]$ (which denotes $\mathbf{S}$ with all occurrences of F replaced by $\mathbf{S}'$). In other words, if we replace recursive calls in $\mathbf{S}$ by copies of $\mathbf{S}'$ then we get a refinement of $\mathbf{S}'$; and
2. We can find an expression $\mathbf{t}$ (called the *variant function*) whose value is reduced before each occurrence of $\mathbf{S}'$ in $\mathbf{S}[\mathbf{S}'/F]$.

The expression $\mathbf{t}$ need not be integer valued: any set Φ which has a well-founded order $\preccurlyeq$ is a suitable value set for $\mathbf{t}$. To prove that the value of $\mathbf{t}$ is reduced it is sufficient to prove that if $\mathbf{t} \preccurlyeq t_0$ initially, then the assertion $\{\mathbf{t} \prec t_0\}$ can be inserted before each occurrence of $\mathbf{S}'$ in $\mathbf{S}[\mathbf{S}'/F]$. The theorem combines these two requirements into a single condition:

Theorem 4.2 If $\preccurlyeq$ is a well-founded partial order on some set Φ and $\mathbf{t}$ is an expression giving values in Φ and t_0 is a variable which does not occur in $\mathbf{S}$ then if

$$\forall t_0. ((\mathbf{P} \ \wedge\ \mathbf{t} \preccurlyeq t_0) \Rightarrow \mathbf{S}' \ \leq\ \mathbf{S}[\{\mathbf{P} \ \wedge\ \mathbf{t} \prec t_0\};\ \mathbf{S}'/F])$$

then $\mathbf{P} \Rightarrow \mathbf{S}' \ \leq\ \underline{\textbf{proc}}\ F \ \equiv\ \mathbf{S}.$

Proof: See [43] ■

4.3 A Method for Algorithm Derivation

It is frequently possible to *derive* a suitable procedure body $\mathbf{S}$ from the statement $\mathbf{S}'$ by applying transformations to $\mathbf{S}'$, splitting it into cases etc., until we get a statement of the form $\mathbf{S}[\mathbf{S}'/F]$ which is still defined in terms of $\mathbf{S}'$. If we can find a suitable variant function for $\mathbf{S}[\mathbf{S}'/F]$ then we can apply the theorem and refine $\mathbf{S}[\mathbf{S}'/F]$ to $\underline{\textbf{proc}}\ F \ \equiv\ \mathbf{S}.$ which is no longer defined in terms of $\mathbf{S}'$.

As an example we will consider the familiar factorial function. Let $\mathbf{S}'$ be the statement $r := n!$. We can transform this (by appealing to the definition of factorial) to get:

$$\mathbf{S}' \ \approx\ \underline{\textbf{if}}\ n = 0\ \underline{\textbf{then}}\ r := 1\ \underline{\textbf{else}}\ r := n.(n-1)!\ \underline{\textbf{fi}}$$

Separate the assignment:

$$\approx\ \underline{\textbf{if}}\ n = 0\ \underline{\textbf{then}}\ r := 1\ \underline{\textbf{else}}\ n := n - 1;\ r := n!;\ n := n + 1;\ r := n.r\ \underline{\textbf{fi}}$$

So we have:

$$\approx\ \underline{\textbf{if}}\ n = 0\ \underline{\textbf{then}}\ r := 1\ \underline{\textbf{else}}\ n := n - 1;\ \mathbf{S}';\ n := n + 1;\ r := n.r\ \underline{\textbf{fi}}$$

The positive integer n is decreased before the copy of $\mathbf{S}'$, so if we set $\mathbf{t}$ to be n, Φ to be $\mathbb{N}$ and $\preccurlyeq$ to be $\leqslant$ (the usual order on natural numbers), and $\mathbf{P}$ to be **true** then we can prove:

$$n \leqslant t_0 \implies \mathbf{S}' \;\leq\; \underline{\mathbf{if}}\ n = 0\ \underline{\mathbf{then}}\ r := 1\ \underline{\mathbf{else}}\ n := n-1;\ \{n < t_0\};\ \mathbf{S}';\ n := n+1;\ r := n.r\ \underline{\mathbf{fi}}$$

So we can apply Theorem 4.2 to get:

$$\mathbf{S}' \;\leq\; \underline{\mathbf{proc}}\ F \;\equiv\; \underline{\mathbf{if}}\ n = 0\ \underline{\mathbf{then}}\ r := 1\ \underline{\mathbf{else}}\ n := n-1;\ F;\ n := n+1;\ r := n.r\ \underline{\mathbf{fi}}\mathbf{.}$$

and we have derived a recursive implementation of factorial.

This theorem is a fundamental result towards the aim of a system for transforming specifications into programs since it "bridges the gap" between a recursively defined specification and a recursive procedure which implements it. It is of use even when the final program is iterative rather than recursive since many algorithms may be more easily and clearly specified as recursive functions—even if they may be more efficiently implemented as iterative procedures. This theorem may be used by the programmer to transform the recursively defined specification into a recursive procedure or function which can then be transformed into an iterative procedure.

4.4 The Induction Rule for Recursion

The nth truncation of a procedure $\underline{\mathbf{proc}}\ F \;\equiv\; \mathbf{S.}$ is defined:

$$\underline{\mathbf{proc}}\ F \;\equiv\; \mathbf{S.}^0 =_{\text{DF}} \mathbf{abort} \qquad \textit{and} \qquad \underline{\mathbf{proc}}\ F \;\equiv\; \mathbf{S.}^{n+1} =_{\text{DF}} \mathbf{S}[\underline{\mathbf{proc}}\ F \;\equiv\; \mathbf{S.}^n/F]$$

The nth truncation of any statement $\mathbf{S}^n$ is formed by replacing each recursive component by its nth truncation.

A statement has *bounded nondeterminacy* if each assignment statement has a finite set of values it can assign to the variables to satisfy the given condition.

For statements with bounded nondeterminacy we have the following induction rule:

Theorem 4.3 *The Induction Rule for Recursion:*
If $\mathbf{S}$ is any statement with bounded nondeterminacy, and $\mathbf{S}'$ is another statement such that $\Delta \vdash \mathbf{S}^n \;\leq\; \mathbf{S}'$ for all $n < \omega$, then $\Delta \vdash \mathbf{S} \;\leq\; \mathbf{S}'$.

This result is extremely valuable in proving many transformations involving recursive and iterative statements. It shows that the set of all finite truncations of a recursive statement tells us everything we need to know about the full recursion. Using this induction rule we have proved a powerful collection of general purpose transformations. These enable many algorithm derivations to be carried out by appealing to general transformation rules rather than *ad hoc* induction proofs.

An example of a transformation proved by induction is the following:

Theorem 4.4 *Invariant Maintenance*

(i) If for any statement $\mathbf{S}_1$ we can prove: $\{\mathbf{P}\};\ \mathbf{S}[\mathbf{S}_1/X] \;\leq\; \mathbf{S}[\{\mathbf{P}\};\ \mathbf{S}_1/X]$ then:

$$\{\mathbf{P}\};\ \underline{\mathbf{proc}}\ X \;\equiv\; \mathbf{S.} \;\leq\; \underline{\mathbf{proc}}\ X \;\equiv\; \{\mathbf{P}\};\ \mathbf{S.}$$

(ii) If in addition $\Delta \vdash \{\mathbf{P}\};\ \mathbf{S}_1 \;\leq\; \mathbf{S}_1;\ \{\mathbf{P}\}$ implies $\Delta \vdash \{\mathbf{P}\};\ \mathbf{S}[\mathbf{S}_1/X] \;\leq\; \mathbf{S}[\mathbf{S}_1/X];\ \{\mathbf{P}\}$ then

$$\{\mathbf{P}\};\ \underline{\mathbf{proc}}\ X \;\equiv\; \mathbf{S.} \;\leq\; \underline{\mathbf{proc}}\ X \;\equiv\; \mathbf{S.};\ \{\mathbf{P}\}$$

Proof: (i) Claim: $\{\mathbf{P}\};\ \underline{\mathbf{proc}}\ X \;\equiv\; \mathbf{S.}^n \;\leq\; \underline{\mathbf{proc}}\ X \;\equiv\; \{\mathbf{P}\};\ \mathbf{S.}^n$. If this claim is proved then the result follows from the induction rule for recursion (Theorem 4.3). We prove the claim by induction on n. For $n = 0$ both sides are **abort**, so suppose the result holds for n.

Put $\mathbf{S}_1 = \underline{\mathbf{proc}}\ X \ \equiv\ \mathbf{S}.^n$ in the premise. Then:

$$\begin{aligned}\{\mathbf{P}\};\ \underline{\mathbf{proc}}\ X \ \equiv\ \mathbf{S}.^{n+1} &\leq \{\mathbf{P}\};\ \mathbf{S}[\underline{\mathbf{proc}}\ X \ \equiv\ \mathbf{S}.^n/X]\\ &\leq (\{\mathbf{P}\};\ \mathbf{S})[\{\mathbf{P}\};\ \underline{\mathbf{proc}}\ X \ \equiv\ \mathbf{S}.^n/X]\end{aligned}$$

from the premise

$$\leq (\{\mathbf{P}\};\ \mathbf{S})[\underline{\mathbf{proc}}\ X \ \equiv\ \{\mathbf{P}\};\ \mathbf{S}.^n/X]$$

by the induction hypothesis

$$\leq \underline{\mathbf{proc}}\ X \ \equiv\ \{\mathbf{P}\};\ \mathbf{S}.^{n+1}$$

The result follows by induction on n.

(ii) Claim: $\{\mathbf{P}\};\ \underline{\mathbf{proc}}\ X \ \equiv\ \mathbf{S}.^n \ \leq\ \underline{\mathbf{proc}}\ X \ \equiv\ \{\mathbf{P}\};\ \mathbf{S}.^n;\ \{\mathbf{P}\}$ for all n. Again, we prove the claim by induction on n: For $n = 0$ both sides are **abort**, so suppose the result holds for n.

$$\begin{aligned}\{\mathbf{P}\};\ \underline{\mathbf{proc}}\ X \ \equiv\ \{\mathbf{P}\};\ \mathbf{S}.^{n+1} &\leq \{\mathbf{P}\};\ \mathbf{S}[\underline{\mathbf{proc}}\ X \ \equiv\ \mathbf{S}.^n/X]\\ &\leq \{\mathbf{P}\};\ \mathbf{S}[\{\mathbf{P}\};\ \underline{\mathbf{proc}}\ X \ \equiv\ \mathbf{S}.^n/X] \quad \text{by part (i)}\end{aligned}$$

Put $\mathbf{S}_1 = \underline{\mathbf{proc}}\ X \ \equiv\ \mathbf{S}.^n$ in the premise and use: $\{\mathbf{P}\};\ \underline{\mathbf{proc}}\ X \ \equiv\ \mathbf{S}.^n \ \leq\ \underline{\mathbf{proc}}\ X \ \equiv\ \mathbf{S}.^n;\ \{\mathbf{P}\}$

$$\leq \underline{\mathbf{proc}}\ X \ \equiv\ \mathbf{S}.^{n+1};\ \{\mathbf{P}\}$$

from premise (ii).

The result follows by induction on n. ■

4.5 General Recursion Removal

Theorem 4.5 Suppose we have a recursive procedure whose body is an action system in the following form, in which the body of the procedure is an action system. (A $\underline{\mathbf{call}}$ Z in the action system will therefore terminate only the current invocation of the procedure):

```
$\underline{\mathbf{proc}}\ F(x) \ \equiv$
  $\underline{\mathbf{actions}}\ A_1:$
  $A_1 \ \equiv \mathbf{S}_1.$
  $\ldots A_i \ \equiv \mathbf{S}_i.$
  $\ldots B_j \ \equiv \mathbf{S}_{j0};\ F(g_{j1}(x));\ \mathbf{S}_{j1};\ F(g_{j2}(x));\ \ldots;\ F(g_{jn_j}(x));\ \mathbf{S}_{jn_j}.$
  $\ldots$ $\underline{\mathbf{endactions}}.$
```

where the statements $\mathbf{S}_{j1}, \ldots, \mathbf{S}_{jn_j}$ preserve the value of x and no $\mathbf{S}$ contains a call to F (i.e. all the calls to F are listed explicitly in the B_j actions) and the statements $\mathbf{S}_{j0}, \mathbf{S}_{j1} \ldots, \mathbf{S}_{jn_j-1}$ contain no action calls. There are $M + N$ actions in total: $A_1, \ldots, A_M, B_1, \ldots, B_N$.

We claim that this is equivalent to the following iterative procedure which uses a new local stack K and a new local variable m:

```
$\underline{\mathbf{proc}}\ F(x) \ \equiv$
  $\underline{\mathbf{var}}\ K := \langle\rangle, m := 0:$
    $\underline{\mathbf{actions}}\ A_1:$
    $\ldots A_i \ \equiv \mathbf{S}_i[\underline{\mathbf{call}}\ \hat{F}/\underline{\mathbf{call}}\ Z].$
    $\ldots B_j \ \equiv \mathbf{S}_{j0};\ K := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j,1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} K;$
              $\underline{\mathbf{call}}\ \hat{F}.$
    $\ldots \hat{F} \ \equiv \underline{\mathbf{if}}\ K = \langle\rangle\ \underline{\mathbf{then}}\ \underline{\mathbf{call}}\ Z$
                     $\underline{\mathbf{else}}\ \langle m, x\rangle \xleftarrow{\text{pop}} K;$
                        $\underline{\mathbf{if}}\ m = 0 \rightarrow \underline{\mathbf{call}}\ A_1$
                        $\square \ldots \square\ m = \langle j, k\rangle \rightarrow \mathbf{S}_{jk};\ \underline{\mathbf{call}}\ \hat{F}$
                        $\ldots\ \underline{\mathbf{fi}}\ \underline{\mathbf{fi}}.\ \underline{\mathbf{endactions}}\ \underline{\mathbf{end}}.$
```

Proof: See [41]. ■

By unfolding the calls to $\hat{F}$ in B_j we can avoid pushing and popping $\langle 0, g_{j1}(x)\rangle$:

Corollary 4.6 $F(x)$ is equivalent to:

```
proc $F(x)$ $\equiv$
  var $K := \langle\rangle, m := 0$:
    actions $A_1$:
    $\ldots A_i$ $\equiv \mathbf{S}_i[$call $\hat{F}/$call $Z]$.
    $\ldots B_j$ $\equiv \mathbf{S}_{j0}$; $K := \langle\langle\langle j,1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle, \ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{++} K$;
                $x := g_{j1}(x)$; call $A_1$.
    $\ldots \hat{F}$ $\equiv$ if $K = \langle\rangle$ then call $Z$
                          else $\langle m, x\rangle \overset{\text{pop}}{\longleftarrow} K$;
                               if $m = 0 \rightarrow$ call $A_1$
                               $\square \ldots \square$ $m = \langle j,k\rangle \rightarrow \mathbf{S}_{jk}$; call $\hat{F}$
                               $\ldots$ fi fi. endactions end.
```

Note that *any* procedure $F(x)$ can be restructured into the required form; in fact there may be several different ways of structuring $F(x)$ which meet these criteria. The simplest such restructuring is to put each recursive call into its own B action (with no other statements apart from a call to the next action). Since it is always applicable, this is the method used by most compilers. See [42] for further applications of the theorem.

4.6 Recursion Removal Without a Stack

Consider the following parameterless procedure, where we have restructured the body to put each procedure call into a separate action. Suppose we have been able to insert a set of mutually disjoint assertions $\mathbf{Q}_0, \mathbf{Q}_1, \ldots, \mathbf{Q}_n$ where for $i \neq j$, $\mathbf{Q}_i \wedge \mathbf{Q}_j \Leftrightarrow \mathbf{false}$:

```
begin $F$; $\{\mathbf{Q}_0\}$
where
proc $F()$ $\equiv$
  actions $A_1$:
  $\ldots A_i$ $\equiv \mathbf{S}_i[$call $\hat{F}/$call $Z]$.
  $\ldots B_j$ $\equiv \mathbf{S}_{j0}$; $F$; $\{\mathbf{Q}_j\}$; $\mathbf{S}_{j1}$.
  $\ldots$ endactions.
end
```

By Corollary 4.6 we have:

```
begin $F$; $\{\mathbf{Q}_0\}$
where
proc $F()$ $\equiv$
  var $K := \langle\rangle, m := 0$ :
    actions $A_1$:
    $\ldots A_i$ $\equiv \mathbf{S}_i$.
    $\ldots B_j$ $\equiv \mathbf{S}_{j0}$; $K := \langle j\rangle \mathbin{++} K$; call $A_1$.
    $\ldots \hat{F}$ $\equiv$ if $K = \langle\rangle$ then call $Z$
                          else $m \overset{\text{pop}}{\longleftarrow} K$;
                               if $m = 1 \rightarrow \{\mathbf{Q}_1\}$; $\mathbf{S}_{11}$; call $\hat{F}$
                               $\square \ldots \square$ $m = j \rightarrow \{\mathbf{Q}_j\}$; $\mathbf{S}_{j1}$; call $\hat{F}$
                               $\ldots$ fi fi. endactions end.
end
```

Unfold F and push $\{\mathbf{Q}_0\}$ into the action system (so the single **call** Z is replaced by $\{\mathbf{Q}_0\}$; **call** Z). Now we can use the disjointness of the $\mathbf{Q}_i$ to refine the **if** statements in $\hat{F}$ to test $\mathbf{Q}_i$ instead of K (using Theorem 4.1). Then we can remove K and m (since we have deleted all accesses to these local variables) to prove the following theorem:

Theorem 4.7 Under these conditions, the procedure F is equivalent to:

```
actions $A_1$:
$\ldots A_i \;\equiv \mathbf{S}_i$.
$\ldots B_j \;\equiv \mathbf{S}_{j0}$; call $A_1$.
$\ldots \hat{F} \;\equiv$ if $\mathbf{Q}_0 \rightarrow$ call $Z$
          $\square \ldots \square\; \mathbf{Q}_j \rightarrow \mathbf{S}_{j1}$; call $\hat{F}$
          $\ldots$ fi. endactions
```

Thus we have removed the recursion without the need for a protocol stack.

5 Graph Marking

Definition 5.1 A *graph* is a pair $\langle \mathcal{N}, D\rangle$ where $\mathcal{N}$ is any set (the set of *nodes* of the graph), and D is any function from $\mathcal{N}$ to the set of all subsets of $\mathcal{N}$ (denoted $\wp(\mathcal{N})$). i.e. $D : \mathcal{N} \rightarrow \wp(\mathcal{N})$. For each $x \in \mathcal{N}$, $D(x)$ is the set of *daughter nodes* for x.

The purpose of a graph marking algorithm is to determine all the nodes reachable from a given node, or set of nodes. In other words, we need to determine the set of daughters, and daughters of daughters, etc. of the given node or nodes. For a given set $X \subseteq \mathcal{N}$ we define $R(X) \subseteq \mathcal{N}$ to be the set of nodes reachable from X. We can determine R by constructing the transitive closure of D, or by considering the set of paths in the graph which start at X. See Appendixes A to C which shows how to derive the following iterative graph marking algorithm:

```
proc mark $\equiv$
  while $X \setminus M \neq \varnothing$ do
        var $x \in X \setminus M$ :
          $M := M \cup \{x\}$; $X := (X \setminus \{x\}) \cup (D(x) \setminus M')$ end od
```

and the following recursive algorithm:

```
proc mark($x$) $\equiv$
  var $K := \langle\rangle, D := \langle\rangle$ :
    do $M := M \cup \{x\}$;
       $K \xleftarrow{\text{push}} D$; $D := D(x)$;
       do if $D = \varnothing$ then $D \xleftarrow{\text{pop}} K$; if $K = \langle\rangle$ then exit(2) fi
                      else $x := x'.(x' \in D)$; $D := D \setminus \{x\}$;
                           if $x \notin M$ then exit fi fi od od end.
```

6 The Schorr-Waite Algorithm

The Schorr-Waite algorithm [35] seems to have acquired the status of a standard testbed for program verification techniques applied to complex data structures. A proof of a simplified version of the algorithm using the method of invariants was given in Gries [16]. This required two rather complicated invariants to prove partial correctness, and an informally described termination function to demonstrate total correctness. An entirely informal transformational development from a standard recursive algorithm is described in Griffiths [19]. Morris [30] proves the algorithm using the axiomatic methods of Hoare [21]. Topor [38] presents a proof using the method of "intermittent assertions" which are assertions of the form: "if at some time during the execution assertion A holds at this point, then at some later time assertion B will hold at this point". Intermittent assertions are described by Manna and Waldinger in [28] where they are used to reason about some iterative algorithms for computing recursive functions. Yelowitz and Duncan, de Roever and Kowalski [27, 34,48] also give proofs for this algorithm.

The method applied by Gries to the Schorr-Waite algorithm [16] and more generally in [18] and [17] is to determine the "total situation" in one go, by calculating the precise relationship between abstract and concrete data structures and expressing this in the form of invariants. Gries recognises that such invariants can be very difficult to determine directly from the final program. Griffiths [19] uses a transformational development approach, but, in order to justify some of the development steps, he still has to appeal (informally) to global invariants which describe the total situation. This is because he removes the recursion at an early stage in the development, resulting in a less tractable iterative program.

Most of these proofs treat the problem as an exercise in program *verification*, and therefore start with a statement of the algorithm. The methods that rely on invariants give a long list of complex invariants, again with little indication of how these invariants could be developed. We are more interested in developing a *derivation* of the algorithm: starting with an abstract specification and a vague idea of the technique to be used, we want to see if our transformation catalogue is sufficiently complete so as to provide a rigorous derivation of the algorithm by transformation from the specification. We take a different route to that of Griffiths [19] since we prefer to do as much simplification as possible with the recursive form of the algorithm before removing the recursion. In particular we introduce the central idea of the algorithm while it is still in the form of a recursive procedure. This gives a much clearer development than Griffith's introduction of the central idea after he has removed the recursion.

Gerhart [15] presents a derivation-oriented proof: the information gained during the derivation phase is used to guide the subsequent (nearly automatic) verification, with the help of a system.

Broy and Pepper [9] use a transformational approach to the algorithm, based on algebraic abstract data types for binary graphs and binary graphs whose pointers can be modified. They use a combination of algebraic and algorithmic reasoning to develop a recursive marking algorithm and transform it into the Schorr-Waite algorithm. They use ghost variables in the form of additional parameters to applicative functions, but their development still requires several induction proofs. We aim to produce a transformational derivation by applying a sequence of proven transformation rules, without the need for induction arguments. The *same* transformations are used to apply the "pointer switching idea" to other algorithms, and in fact it should be possible to use the same technique with practically any graph-walking or tree-walking algorithm.

6.1 The "Pointer Switching" Idea

The problem with the iterative and recursive algorithms derived in Appendices B and C is that they both require an unbounded amount of working storage. A common application of graph marking is in garbage collection algorithms—and such algorithms are usually invoked when there is almost no free storage available! The "pointer switching" idea of Schorr and Waite [35] is a technique for walking through a graph using a small and fixed amount of extra storage.

For the rest of this section we will treat the case where each node has exactly two daughter nodes (i.e. $\#D(x) = 2$ for all $x \in \mathcal{N}$). This is usual for many applications (such as LISP implementations) where garbage collection is required. We suppose that the two daughters for each node are available through the functions $L : \mathcal{N} \to \mathcal{N}$ and $R : \mathcal{N} \to \mathcal{N}$ so $\forall x \in \mathcal{N}.\, D(x) = \{L(x), R(x)\}$. So called "null pointers" can be represented by a special node $\Lambda \in \mathcal{N}$ with $L(\Lambda) = R(\Lambda) = \Lambda$. Typically, Λ will be already marked, i.e. $\Lambda \in M$ initially, though this is not essential. Broy and Pepper [9] set $R(x) = x$ to represent a null pointer in $R(x)$ (and similarly for $L(x)$). This has the disadvantage that their algorithm cannot be used in applications which need to distinguish between null pointers and self-pointers (and garbage collection in LISP is such an application!).

Under these conditions, the recursive algorithm simplifies to:

```
proc mark($x$) ≡
  $M := M \cup \{x\}$;
  if $L(x) \notin M$ then mark($L(x)$) fi;
  if $R(x) \notin M$ then mark($R(x)$) fi.
```

In the actual implementation, the values $L(x)$ and $R(x)$ will be stored in arrays $l[x]$ and $r[x]$. The central idea behind the algorithm devised by Schorr and Waite is that when we return from having marked the left subtree of a node we know what the value of $L(x)$ is for the current node (since we just came from there). So while we are marking the left subtree, we can use the array element $l[x]$ to store something else—for instance a pointer to the node we will return to after having marked this node. Similarly, while we are marking the right subtree we can store this pointer in $r[x]$. The algorithm uses some additional storage for each node (denoted by the array $m[x]$) to record which subtrees of the current node have been marked. The next section turns this informal idea into a formal transformational development.

6.2 Deriving the Algorithm

In this section we apply the pointer switching idea to the recursive algorithm, in order to produce a formal derivation of the Schorr-Waite algorithm, for the case where each node has exactly two daughters.

The difficulty the Schorr-Waite algorithm presents to any formal analysis, is that it uses the same data structure for three different purposes: to store the original graph structure, to record the path from the current node to the root, and to record the current "state of play" at each node. The program is required to mark the graph without changing its structure, yet works by modifying that structure as it executes. This means that any proof of correctness must also demonstrate that all the pointers are restored on termination of the program. Hence Schorr-Waite is an ideal candidate for the ghost variables technique discussed in Section 2.1. Consider the recursive algorithm for binary graphs, at the end of the previous section. We can eliminate the parameter by introducing a local variable x' to save and restore x. So we have the following implementation of the specification MARK({root}):

```
begin $x :=$ root; mark
where
proc mark $\equiv$
  $M := M \cup \{x\}$;
  var $x' := x$ : if $L(x') \notin M$ then $x := L(x')$; mark; $x := x'$ fi end;
  var $x' := x$ : if $R(x') \notin M$ then $x := R(x')$; mark; $x := x'$ fi end;
```

6.2.1 Apply the Pointer Switching idea

To apply the pointer switching idea we will insert some assignments to the "ghost variables", (l, r and m). These variables are arrays with these initial values:

$$\forall x \in \mathcal{N}.\, (l[x] = L(x) \ \wedge\ r[x] = R(x) \ \wedge\ m[x] = 0)$$

we also have the variable q whose initial value is arbitrary.

There are two very simple invariants which our assignments to ghost variables will preserve:

(i) We only modify array elements which were unmarked, and which we first mark. So for any $y \notin M$ we know that the ghost arrays have their initial values;

(ii) All ghost variables are preserved by all calls to mark.

Invariant (i) is trivially proved since the assignment $M := M \cup \{x\}$ appears before any assignments to $l[x]$, $r[x]$ or $m[x]$. We preserve invariant (ii) by always pairing assignments to ghost variables in swap statements, using the transformation:

$$\{a = b\};\ \mathbf{S} \quad \approx \quad \{a = b\};\ \overset{\curvearrowright}{\leftarrow}\langle a, b\rangle;\ \mathbf{S};\ \overset{\curvearrowright}{\leftarrow}\langle a, b\rangle$$

We can insert pairs of swaps freely anywhere in the program without affecting the invariants.

XXX

We will only call mark when $x \notin M$ so by invariant (i), $l[x] = L(x)$ and $r[x] = R(X)$ at the beginning of each call to mark. To eliminate x' we need to save and restore x using either $l[x]$ or $r[x]$ and q. This is achieved by swapping values around so that x is given the value of $l[x]$ (or $r[x]$), q holds x and the original value of x is stored in $l[x]$ (or $r[x]$).

```
begin $x :=$ root; mark
where
proc mark $\equiv$
  $M := M \cup \{x\}$;
  var $x' := x$ :
    $\circlearrowleft\langle l[x], q\rangle$;
    if $L(x') \notin M$ then $\circlearrowleft\langle x, q\rangle$; $x := L(x')$; mark; $x := x'$; $\circlearrowleft\langle x, q\rangle$ fi;
    $\circlearrowleft\langle l[x], q\rangle$ end;
  var $x' := x$ :
    $\circlearrowleft\langle r[x], q\rangle$;
    if $R(x') \notin M$ then $\circlearrowleft\langle x, q\rangle$; $x := R(x')$; mark; $x := x'$; $\circlearrowleft\langle x, q\rangle$ fi;
    $\circlearrowleft\langle r[x], q\rangle$ end.
```

The simplest way to prove that this is correct is to replace the recursive calls by copies of the specification, prove the transformation for this (now non-recursive) procedure, and then apply Theorem 4.2 to re-insert the recursive calls. Note that since x is preserved over the body of the **var** clause, we can treat $l[x]$ as a simple variable and use it in the roation operator.

Now we can replace references to L, R, and x' by equivalent references to ghost variables. For instance the assignments $x := L(x')$ and $x := R(x')$ are redundant, as are the assignments $x := x'$. We get:

```
begin $x :=$ root; mark
where
proc mark $\equiv$
  $M := M \cup \{x\}$;
  $\circlearrowleft\langle l[x], q\rangle$;
  if $q \notin M$ then $\circlearrowleft\langle x, q\rangle$; mark; $\circlearrowleft\langle x, q\rangle$ fi;
  $\circlearrowleft\langle l[x], q\rangle$;
  $\circlearrowleft\langle r[x], q\rangle$;
  if $q \notin M$ then $\circlearrowleft\langle x, q\rangle$; mark; $\circlearrowleft\langle x, q\rangle$ fi;
  $\circlearrowleft\langle r[x], q\rangle$.
```

6.2.2 Remove Recursion

In order to remove the recursion without introducing a stack (via Theorem 4.7), we need three disjoint assertions. These are easily supplied by adding assignments to $m[x]$. By ensuring that we set $m[x] := 0$ at the end of the procedure we guarantee that m is preserved for all recursive calls (since we know $m[x] = 0$ at the start of the procedure):

```
begin $x :=$ root; mark
where
proc mark $\equiv$
  $M := M \cup \{x\}$;
  $m[x] := 1$;
  $\circlearrowleft\langle l[x], q\rangle$;
  if $q \notin M$ then $\circlearrowleft\langle x, q\rangle$; mark; $\circlearrowleft\langle x, q\rangle$ fi;
  $\circlearrowleft\langle l[x], q\rangle$;
  $m[x] := 2$;
  $\circlearrowleft\langle r[x], q\rangle$;
  if $q \notin M$ then $\circlearrowleft\langle x, q\rangle$; mark; $\circlearrowleft\langle x, q\rangle$ fi;
  $m[x] := 0;$
  $\circlearrowleft\langle r[x], q\rangle.$
```

The first call to mark puts root into M and sets $m[\mathsf{root}]$ to a non-zero value before each recursive call. So we know that $m[\mathsf{root}] \neq 0$ after each recursive call. Again, this proof does not require an induction argument: simply replace the recursive calls by the specification, insert the assertions and apply Theorem 4.2 to re-insert the recursive calls[1]. We have the assertions:

```
begin $x :=$ root; mark; $\{m[\mathsf{root}] = 0 \wedge \mathsf{root} \in M\}$
where
proc mark $\equiv$
  $M := M \cup \{x\};$
  $m[x] := 1;$ $\circlearrowleft\langle l[x], q\rangle;$
  if $q \notin M$ then $\circlearrowleft\langle x, q\rangle;$
                  mark; $\{m[q] = 1 \wedge m[\mathsf{root}] \neq 0\};$
                  $\circlearrowleft\langle x, q\rangle$ fi;
  $\circlearrowleft\langle l[x], q\rangle;$
  $m[x] := 2;$ $\circlearrowleft\langle r[x], q\rangle;$
  if $q \notin M$ then $\circlearrowleft\langle x, q\rangle;$
                  mark; $\{m[q] = 2 \wedge m[\mathsf{root}] \neq 0\};$
                  $\circlearrowleft\langle x, q\rangle$ fi;
  $m[x] := 0;$ $\circlearrowleft\langle r[x], q\rangle.$
```

After restructuring the body of mark as an action system and removing the recursion we get the iterative action system:

```
$x :=$ root;
actions $A_1$ :
$A_1$ $\equiv$ $M := M \cup \{x\};$ $m[x] := 1;$ $\circlearrowleft\langle l[x], q\rangle;$
       if $q \notin M$ then $\circlearrowleft\langle x, q\rangle;$ call $A_1$ fi;
       call $A_2$.
$A_2$ $\equiv$ $\circlearrowleft\langle l[x], q\rangle;$ $m[x] := 2;$ $\circlearrowleft\langle r[x], q\rangle;$
       if $q \notin M$ then $\circlearrowleft\langle x, q\rangle;$ call $A_1$ fi;
       call $A_3$.
$A_3$ $\equiv$ $m[x] := 0;$ $\circlearrowleft\langle r[x], q\rangle;$ call $\hat{F}$.
$\hat{F}$ $\equiv$ if $m[\mathsf{root}] = 0 \wedge \mathsf{root} \in M \rightarrow$ call $Z$
       $\Box$ $m[q] = 1 \wedge m[\mathsf{root}] \neq 0 \rightarrow \circlearrowleft\langle x, q\rangle;$ call $A_2$
       $\Box$ $m[q] = 2 \wedge m[\mathsf{root}] \neq 0 \rightarrow \circlearrowleft\langle x, q\rangle;$ call $A_3$ fi. endactions
```

Finally, we can replace the two variables M and m by a single variable m'. If $x \notin M$ then $m[x] = 0$ so we assign values to the new ghost variable m' as follows:

$$\begin{array}{lll}
m'[x] = 0 & \text{if} & x \notin M \quad (m[x] \text{ must be } 0) \\
m'[x] = 1 & \text{if} & x \in M \wedge m[x] = 1 \\
m'[x] = 2 & \text{if} & x \in M \wedge m[x] = 2 \\
m'[x] = 3 & \text{if} & x \in M \wedge m[x] = 0
\end{array}$$

If we initialise m' so that $m'[x] = 0$ for $x \notin M$ and $m'[x] = 3$ for $x \in M$ then the final value of M is $\{\, x \in \mathcal{N} \mid m'[x] = 3 \,\}$. We can replace references to M and m by references to the new variable m' and then rename m' to m to get:

[1] Broy and Pepper [9] use a similar recursion removal technique, but their approach introduces an artificial node virtualroot which must not be a daughter of any other node. This has the obvious drawback that it may not be possible to find such a node: for example in a machine with either a fully populated address range, or with some form of virtual memory system, any address selected for virtualroot may be either an invalid address, or already in use. Our approach avoids this problem and allows the whole address range to be used.

```
$x := \mathsf{root}$;
actions $A_1$ :
$A_1$  $\equiv m[x] := 1;\ \circlearrowleft\langle l[x], q\rangle$;
         if $m[q] = 0$ then $\circlearrowleft\langle x, q\rangle$; call $A_1$ fi;
         call $A_2$.
$A_2$  $\equiv \circlearrowleft\langle l[x], q\rangle;\ m[x] := 2;\ \circlearrowleft\langle r[x], q\rangle$;
         if $m[q] = 0$ then $\circlearrowleft\langle x, q\rangle$; call $A_1$ fi;
         call $A_3$.
$A_3$  $\equiv m[x] := 3;\ \circlearrowleft\langle r[x], q\rangle$; call $\hat{F}$.
$\hat{F}$  $\equiv$ if $m[\mathsf{root}] = 3$
           then call $Z$
            else if $m[q] = 1 \rightarrow \circlearrowleft\langle x, q\rangle$; call $A_2$
                  $\square\ m[q] = 2 \rightarrow \circlearrowleft\langle x, q\rangle$; call $A_3$ fi fi. endactions
```

6.2.3 Restructure

The algorithm at the end of the previous section is suitable for implementation as it stands. However, we can if we choose apply some further transformations in order to simplify the structure of the iterative algorithm, at the expense of introducing some redundant tests. The aim is to unfold everything into $\hat{F}$ which will become a tail recursive action, which will transform into a simple **while** loop. First unfold the calls to A_2 and A_3 in $\hat{F}$, and add an extra line to its **if** statement with the test $m[q] = 0$ containing a copy of A_1:

```
$x := \mathsf{root}$;
actions $A_1$ :
$A_1$  $\equiv \{m[\mathsf{root} \neq 3 \wedge m[x] = 0\};\ m[x] := 1;\ \circlearrowleft\langle l[x], q\rangle$;
         if $m[q] = 0$ then $\circlearrowleft\langle x, q\rangle$; call $A_1$ fi;
         call $A_2$.
$A_2$  $\equiv \{m[\mathsf{root} \neq 3 \wedge m[x] = 1\};\ \circlearrowleft\langle l[x], q\rangle;\ m[x] := 2;\ \circlearrowleft\langle r[x], q\rangle$;
         if $m[q] = 0$ then $\circlearrowleft\langle x, q\rangle$; call $A_1$ fi;
         call $A_3$.
$A_3$  $\equiv \{m[\mathsf{root} \neq 3 \wedge m[x] = 2\};\ m[x] := 3;\ \circlearrowleft\langle r[x], q\rangle$; call $\hat{F}$.
$\hat{F}$  $\equiv$ if $m[\mathsf{root}] = 3$
           then call $Z$
            else if $m[q] = 0 \rightarrow \circlearrowleft\langle x, q\rangle;\ \circlearrowleft\langle l[x], q\rangle;\ m[x] := 1$;
                                        if $m[q] = 0$ then $\circlearrowleft\langle x, q\rangle$; call $A_1$ fi;
                                        call $A_2$
                  $\square\ m[q] = 1 \rightarrow \circlearrowleft\langle x, q\rangle$;
                                        $\circlearrowleft\langle l[x], q\rangle;\ m[x] := 2;\ \circlearrowleft\langle r[x], q\rangle$;
                                        if $m[q] = 0$ then $\circlearrowleft\langle x, q\rangle$; call $A_1$ fi;
                                        call $A_3$
                  $\square\ m[q] = 2 \rightarrow \circlearrowleft\langle x, q\rangle$;
                                        $m[x] := 3;\ \circlearrowleft\langle r[x], q\rangle$; call $\hat{F}$ fi fi. endactions
```

Insert the "**skip** equivalent" statement $\circlearrowleft\langle x, q\rangle;\ \circlearrowleft\langle x, q\rangle$ before **call** Z and take $\circlearrowleft\langle x, q\rangle$ out of the nested **if** statements. Then take it out of $\hat{F}$ by replacing each **call** $\hat{F}$ by $\circlearrowleft\langle x, q\rangle$; **call** $\hat{F}$. Now, by adding some redundant tests and statements to the bodies of A_1, A_2 and A_3 (using the assertions), these are made identical to the body of $\hat{F}$. So we can replace all calls to A_i by calls to $\hat{F}$, we can also make $\hat{F}$ the initial action for the system instead of A_1. Now $\hat{F}$ becomes a tail-recursive action, which we transform into a **while** loop. Finally, the action system is removed and we have:

```
$x := \mathsf{root}$;
while $m[\mathsf{root}] \neq 3$ do
         if $m[x] = 0 \rightarrow m[x] := 1;\ \circlearrowleft\langle l[x], q\rangle$; if $m[q] = 0$ then $\circlearrowleft\langle x, q\rangle$ fi
```

```
        □ $m[x] = 1 \rightarrow m[x] := 2;\ \circlearrowleft\langle l[x], q\rangle;\ \circlearrowleft\langle r[x], q\rangle;$ if $m[q] = 0$ then $\circlearrowleft\langle x, q\rangle$ fi
        □ $m[x] = 2 \rightarrow m[x] := 3;\ \circlearrowleft\langle x, q\rangle;\ \circlearrowleft\langle r[x], q\rangle$ fi od;
$\circlearrowleft\langle x, q\rangle$
```

If we don't care about the final values of x and q, the last swap can be deleted. As it stands however, this version will preserve the value of q (provided $m[\mathsf{root}] \neq 3$ initially—one of our first assumptions). We may not need to use a new local variable for q, since its value is restored when the algorithm terminates. Any variable lying around can be used: for example any node which is marked initially, such as Λ.

For efficiency, we merge some assignments to get the final version:

```
$x := \mathsf{root}$;
while $m[\mathsf{root}] \neq 3$ do
        $m[x] := m[x] + 1$;
        if $m[x] = 1 \rightarrow$ if $m[l[x]] = 0$ then $\langle l[x], q, x\rangle := \langle q, x, l[x]\rangle$
                                            else $\circlearrowleft\langle l[x], q\rangle$ fi
        □ $m[x] = 2 \rightarrow$ if $m[r[x]] = 0$ then $\langle r[x], l[x], q, x\rangle := \langle l[x], q, x, r[x]\rangle$
                                            else $\langle r[x], l[x], q\rangle := \langle l[x], q, r[x]\rangle$ fi
        □ $m[x] = 3 \rightarrow \langle r[x], q, x\rangle := \langle x, r[x], q\rangle$ fi od; $\circlearrowleft\langle x, q\rangle$
```

6.3 Summary of the Derivation

The transformational derivation of the algorithm falls neatly into four separate stages:

1. Derive the depth-first recursive algorithm, based on properties of R and R_M;
2. Add ghost variables l and r and apply the "pointer switching" idea to save and restore the parameter x. Then replace references to the abstract variables (the functions $L()$ and $R()$ and the local variable x') by references to the ghost variables, and remove the abstract variables;
3. Add assignments to the "extra mark", variable m, so that suitable assertions are provided for recursion removal without a stack. Then remove the recursion using Theorem 4.7;
4. Finally, restructure the action system to a simple **while** loop and merge a few assignments to get the final version.

6.4 A Different Implementation

The algorithm derived on the last section is essentially the same as that devised by Schorr and Waite. We can get a more compact (though slightly less efficient) form of the algorithm by changing the assignments to the "ghost variables" l, r and q. Instead of inserting pairs of swaps we insert three "rotations". using the transformation:

$$\circlearrowleft\langle a, b, c\rangle;\ \circlearrowleft\langle a, b, c\rangle;\ \circlearrowleft\langle a, b, c\rangle \quad \approx \quad \textbf{skip}$$

The version at the beginning of Section 6.2.1 is changed to:

```
begin $x := \mathsf{root}$; mark
where
proc mark $\equiv$
   $M := M \cup \{x\}$;
   $\circlearrowleft\langle l[x], r[x], q\rangle$;
   var $x' := x$ :
      if $L(x') \notin M$ then $\circlearrowleft\langle x, q\rangle;\ x := L(x')$;
                              mark;
                              $\circlearrowleft\langle x, q\rangle;\ x := x'$ fi end;
   $\circlearrowleft\langle l[x], r[x], q\rangle$;
   var $x' := x$ :
    if $R(x') \notin M$ then $\circlearrowleft\langle x, q\rangle$; $x := R(x')$;
                             mark;
                             $\circlearrowleft\langle x, q\rangle$; $x := x'$ fi end;
  $\circlearrowleft\langle l[x], r[x], q\rangle$.
```

For the rest of the development, we can apply the same sequence of transformations. The final version is:

```
$x :=$ root;
while $m[\text{root}] \neq 3$ do
    $m[x] := m[x] + 1$;
    if $m[x] = 1 \rightarrow$ if $m[l[x]] = 0$ then $\circlearrowleft\langle l[x], r[x], q, x\rangle$
                                          else $\circlearrowleft\langle l[x], r[x], q\rangle$ fi
    $\square$ $m[x] = 2 \rightarrow$ if $m[l[x]] = 0$ then $\circlearrowleft\langle l[x], r[x], q, x\rangle$
                                          else $\circlearrowleft\langle l[x], r[x], q\rangle$ fi
    $\square$ $m[x] = 3 \rightarrow$ $\circlearrowleft\langle l[x], r[x], q, x\rangle$ fi od
```

Although the abstraction function has changed considerably, the changes needed to the development using the ghost variables method, in order to arrive at this version, are trivial. This version can be further simplified by re-arranging the **if** statements to get the final version:

```
$x :=$ root;
while $m[\text{root}] \neq 3$ do
    $m[x] := m[x] + 1$;
    if $m[x] = 3 \vee m[l[x]] = 0$ then $\circlearrowleft\langle l[x], r[x], q, x\rangle$
                                   else $\circlearrowleft\langle l[x], r[x], q\rangle$ fi od
```

7 Graphs With Arbitrary Out-Degree

Not all graphs are binary by any means. Often the nodes of a graph may have an arbitrary number of daughters: typical examples are the call graph of a set of procedures, where the daughters of a procedure are the procedures it calls, and the control dependency graph of a set of basic blocks, where the daughters of a block are its potential immediate successors.

An informal development of a version of Schorr-Waite for graphs with arbitrary out-degree was given by Derschowitz in [11]. This used a Boolean array to store the reachable nodes, and an extra array of integers. We aim to give a simple, but rigorous, transformational development of the algorithm which does not need the Boolean array.

Suppose that the "Daughter set" $D(x)$ is given in the form of a pair of functions: $C : \mathcal{N} \to \mathbb{N}$ and $E : \mathcal{N} \times \mathbb{N} \to \mathcal{N}$. C returns the number of daughters: $C(x) =_{\text{DF}} \#D(x)$. For each $x \in \mathcal{N}$ and i with $1 \leqslant i \leqslant C(x)$, $E(x, i)$ is the ith element of $D(x)$ so $D(x) = \{\, E(x, i) \mid 1 \leqslant i \leqslant C(x) \,\}$. With these data structures, the recursive algorithm for MARK({root}) may be written:

```
begin mark(root)
where
proc mark($x$) $\equiv$
  $M := M \cup \{x\}$;
  for $i := 1$ to $C(x)$ step 1 do
      if $E(x, i) \notin M$ then mark($E(x, i)$) fi od.
end
```

Although this is a quite different (and more general) algorithm to the one in the previous section, we can apply the pointer switching idea to this algorithm using the same general purpose transformations.

We add the arrays $c[x]$ and $e[x, i]$ as ghost variables with initial values $C(x)$ and $E(x, i)$ respectively. We save x in a new variable q which is swapped with $e[x, i]$:

```
begin $x$ := root; mark
where
proc mark ≡
  $M := M \cup \{x\}$;
  for $i := 1$ to $C(x)$ step 1 do
      $\langle e[x, i], q \rangle := \langle q, e[x, i] \rangle$;
      if $E(x, i) \notin M$
        then var $x' := x$; $\circlearrowleft\langle x, q \rangle$; mark; $\circlearrowleft\langle x, q \rangle$ end fi;
      $\langle e[x, i], q \rangle := \langle q, e[x, i] \rangle$ od.
end
```

To remove the local variable i we need an extra integer array $m[x]$ which is initially zero, and preserved by calls to mark. We assign i to $m[x]$ and can then remove the abstract variables from the program. $m[x]$ also provides suitable assertions for recursion removal (in this case, only two assertions are required since there is only one recursive call to mark). We get:

```
begin $x$ := root; mark; $\{m[\text{root}] = 0 \wedge \text{root} \in M\}$
where
proc mark ≡
  $M := M \cup \{x\}$;
  while $m[x] < c[x]$ do
        $m[x] := m[x] + 1$;
        $\circlearrowleft\langle e[x, m[x]], q \rangle$;
        if $q \notin M$
          then $\circlearrowleft\langle x, q \rangle$;
                 mark; $\{m[\text{root}] > 0 \wedge \text{root} \in M\}$;
                 $\circlearrowleft\langle x, q \rangle$ fi;
        $\circlearrowleft\langle e[x, m[x]], q \rangle$ od;
  $m[x] := 0$.
end
```

Remove the recursion and restructure, merging M and m as before. Marked elements will have $m[x] > 0$:

```
$x$ := root;
while $m[\text{root}] \neq c[\text{root}]$ do
      $m[x] := m[x] + 1$;
      if $m[x] = 1 \rightarrow \circlearrowleft\langle e[x, m[x]], q \rangle$;
                    if $m[q] < c[q]$ then $\circlearrowleft\langle x, q \rangle$ fi
      □ $1 < m[x] < c[x] \rightarrow \circlearrowleft\langle e[x, m[x] - 1], q \rangle$;
                         $\circlearrowleft\langle e[x, m[x]], q \rangle$;
                         if $m[q] < c[q]$ then $\circlearrowleft\langle x, q \rangle$ fi
      □ $m[x] = c[x] \rightarrow \circlearrowleft\langle e[x, m[x] - 1], q \rangle$ fi od
```

7.1 A Different Implementation

In some situations there may not be an extra integer array available to replace the local variable i, but there may be an unused bit available in each pointer (for example, there may be a sign bit which is always zero, or the nodes may be aligned in memory in such a way that all pointer addresses are even and the least significant bit of each pointer is available for use). The idea behind this implementation is to record pointers to the daughters of a node as a circular-linked list. The first element in the list is negated and the corresponding daughter marked. Then the list is rotated, marking each daughter that comes up, until the negated element reaches the front of the list again.

Suppose that the list of daughters for node x is given by the function $E : \mathcal{N} \rightarrow \mathcal{N}^*$ where for each $x \in \mathcal{N}$, $E(x)$ is a sequence of distinct nodes such that $D(x) = \text{set}(E(x))$. Then the algorithm at the beginning of Section 7 may be written:

```
begin mark(root)
where
proc mark($x$) $\equiv$
  $M := M \cup \{x\}$;
  for $i := 1$ to $\ell(E(x))$ step 1 do
      if $E(x)[i] \notin M$ then mark($E(x)[i]$) fi od.
end
```

For any non-empty sequence s of positive values, the following statements are equivalent to **skip**:

```
$s[1] := -s[1]$;
for $i := 1$ to $\ell(s)$ do
    $s :=$ rotate($s$) od;
$s[1] := -s[1]$
```

where $\mathsf{rotate}(s) \ =_{\text{DF}} \ s[2\,..] \mathbin{+\!\!+} \langle s[1] \rangle$ returns a rotated sequence, with the first element moved to the end. Add a **skip** equivalent to the algorithm (which also introduces the ghost variable $e[x]$ initialised to $E(x)$), and a test for the case $E(x) = \langle\rangle$ to get:

```
begin mark(root)
where
proc mark($x$) $\equiv$
  $M := M \cup \{x\}$;
  if $E(x) \neq \langle\rangle$
    then var $s := e[x]$ :
              for $i := 1$ to $\ell(E(x))$ step 1 do
                  if $E(x)[i] \notin M$ then mark($E(x)[i]$) fi od;
              $s[1] := -s[1]$;
              for $i := 1$ to $\ell(s)$ do
                  $s :=$ rotate($s$) od;
              $s[1] := -s[1]$ end fi.
end
```

The two **for** loops can be merged using a transformation in [39], and then converted to a **do** ... **od** loop (we can put the test at the end since we know $1 \leqslant \ell(E(x))$):

```
begin mark(root)
where
proc mark($x$) $\equiv$
  $M := M \cup \{x\}$;
  if $E(x) \neq \langle\rangle$
    then $e[x][1] := -e[x][1]$;
          var $s := e[x], i := 1$ :
            do if $E(x)[i] \notin M$ then mark($E(x)[i]$) fi;
                $e[x] :=$ rotate($e[x]$);
                $i := i + 1$;
                if $i = \ell(E(x)) + 1$ then exit fi od end;
          $e[x][1] := -e[x][1]$ fi.
end
```

At the beginning of the loop we have the assertion:

$$e[x] \ = \ \mathsf{rotate}^{i-1}(s) \ = \ s[(i-1 \bmod \ell(s)) + 1\,..] \mathbin{+\!\!+} s[1\,..\,(i-1 \bmod \ell(s))]$$

So at the end of the loop:

$$e[x][1] < 0 \iff s[(i - 1 \bmod \ell(s)) + 1] < 0$$
$$\iff (i - 1 \bmod \ell(s)) + 1 = 1$$

since $s[1]$ is the only negative element of s

$$\iff i - 1 = \ell(s)$$

since $i \geqslant 2$ at this point

$$\iff i = \ell(s) + 1$$
$$\iff i = \ell(E(x)) + 1$$

So we can replace the termination test by $e[x][1] < 0$ and remove i and E from the program:

```
begin mark(root)
where
proc mark($x$) ≡
  $M := M \cup \{x\}$;
  if $e[x] \neq \langle\rangle$
    then $e[x][1] := -e[x][1]$;
          do if $|e[x][1]| \notin M$ then mark($|e[x][1]|$) fi;
              $e[x]$ := rotate($e[x]$);
              if $e[x][1] < 0$ then exit fi od;
          $e[x][1] := -e[x][1]$ fi.
end
```

Now we have a collection of sequences $e[x]$ where the only operations we need are to access and update the first element, and rotate the sequence. These can be efficiently implemented by representing the sequences as circular-linked lists, using a standard data representation transformation.

We remove the parameter by introducing a local variable x' and then using the pointer switching idea to remove it. Note that pointer switching will set x to $e[x][1]$ rather than $|e[x][1]|$, but a trivial change to the algorithm will allow negated arguments:

```
begin $x :=$ root; mark
where
proc mark ≡
  $M := M \cup \{|x|\}$;
  if $e[|x|] \neq \langle\rangle$
    then $e[|x|][1] := -e[|x|][1]$;
          do $\overset{\curvearrowright}{\leftarrow}\langle e[|x|][1], q\rangle$;
              if $|q| \notin M$ then $\overset{\curvearrowright}{\leftarrow}\langle |x|, q\rangle$; mark; $\overset{\curvearrowright}{\leftarrow}\langle |x|, q\rangle$ fi;
              $\overset{\curvearrowright}{\leftarrow}\langle e[|x|][1], q\rangle$;
              $e[|x|]$ := rotate($e[|x|]$);
              if $e[|x|][1] < 0$ then exit fi od;
          $e[|x|][1] := -e[|x|][1]$ fi.
end
```

To remove the recursion, we only need two disjoint assertions since there are only two calls to mark. A simple solution (which could be used for any of the other algorithms) is to initialise q to root. The outermost call to mark will terminate with $x =$ root $= q$, for all the inner calls, just before the call q will be a marked node (the initial value of x) while x will be unmarked, so $q \neq x$ before (and therefore after) each inner call. So we can use the assertions $\{q = x\}$ and $\{q \neq x\}$ to remove the recursion:

```
$x :=$ root;
actions $A_1$ :
```

```
$A_1$  $\equiv M := M \cup \{|x|\}$;
        if $e[|x|] \neq \langle\rangle$ then $e[|x|][1] := -e[|x|][1]$; call $A_2$
                                   else call $\hat{F}$ fi.
$A_2$  $\equiv \overset{\curvearrowright}{\leftarrow}\langle e[|x|][1], q\rangle$;
        if $|q| \notin M$ then $\overset{\curvearrowright}{\leftarrow}\langle |x|, q\rangle$; call $A_1$ else call $A_3$ fi.
$A_3$  $\equiv \overset{\curvearrowright}{\leftarrow}\langle e[|x|][1], q\rangle$; $e[|x|] := \mathsf{rotate}(e[|x|])$;
        if $e[|x|][1] < 0$
           then $e[|x|][1] := -e[|x|][1]$; call $\hat{F}$
            else call $A_2$ fi.
$\hat{F}$  $\equiv$ if $q = x$ then call $Z$ else $\overset{\curvearrowright}{\leftarrow}\langle |x|, q\rangle$; call $A_3$ fi.
```

Finally, we restructure and represent the sequences $e[x]$ by circular-linked lists where $\mathsf{first}[x]$ is a pointer to the first element of the list (or 0 if the list is empty), $\mathsf{next}[p]$ is a pointer to the next element of the list, and $\mathsf{node}[p]$ is the node pointed at by p. So for $e[x] = 0$ we have $\mathsf{first}[x] = 0$ and for $e[x] \neq \langle\rangle$ we have:

$$e[x] \;=\; \langle \mathsf{node}[\mathsf{first}[x]], \mathsf{node}[\mathsf{next}[\mathsf{first}[x]]], \ldots \rangle$$

and $\mathsf{node}^{\ell(e[x])}[\mathsf{first}[x]] = \mathsf{first}[x]$. We also represent the set M by an array $m[x]$.

```
$x := \mathsf{root}$;
actions $A_1$ :
$A_1$  $\equiv m[|x|] := 1$;
        if $\mathsf{first}[|x|] \neq 0$ then $\mathsf{node}[\mathsf{first}[|x|]] := -\mathsf{node}[\mathsf{first}[|x|]]$; call $A_2$
                                      else call $\hat{F}$ fi.
$A_2$  $\equiv \overset{\curvearrowright}{\leftarrow}\langle \mathsf{node}[\mathsf{first}[|x|]], q\rangle$;
        if $m[|q|] = 0$ then $\overset{\curvearrowright}{\leftarrow}\langle x, q\rangle$; call $A_1$ else call $A_3$ fi.
$A_3$  $\equiv \overset{\curvearrowright}{\leftarrow}\langle \mathsf{node}[\mathsf{first}[|x|]], q\rangle$; $\mathsf{first}[|x|] := \mathsf{next}[\mathsf{first}[|x|]]$;
        if $\mathsf{node}[\mathsf{first}[|x|]] < 0$
           then $\mathsf{node}[\mathsf{first}[|x|]] := -\mathsf{node}[\mathsf{first}[|x|]]$; call $\hat{F}$
            else call $A_2$ fi.
$\hat{F}$  $\equiv$ if $q = x$ then call $Z$ else $\overset{\curvearrowright}{\leftarrow}\langle x, q\rangle$; call $A_3$ fi.
```

8 Acyclic Graph Marking

A *cycle* in a graph is a path $\langle x_1, \ldots, x_n\rangle \in P(D)$ where $n \geqslant 1$ and $x_1 = x_n$. A graph is *acyclic* if it has no cycles. If $\langle x_1, \ldots, x_n\rangle$ is a cycle then the path $\langle x_2, \ldots, x_n\rangle$ is a witness to the fact that $x_1 \in R(\{x_2\}) \subseteq R(D(x_1))$. Conversely, if $x \in R(D(x))$ then a witness p to this fact can be extended to a cycle $\langle x\rangle +\!\!\!+\, p$. Hence we have the theorem:

Theorem 8.1 A graph $\langle \mathcal{N}, D\rangle$ is acyclic iff $\forall x \in \mathcal{N}.\, x \notin R(D(x))$

For our algorithm it is sufficient for there to be no *unmarked* cycles, i.e. $\forall x \in \mathcal{N}.\, x \notin R_M(D(x))$. In particular we can still use node Λ to represent a null pointer, where $\Lambda \in M$ initially and $L(\Lambda) = R(\Lambda) = \Lambda$.

An important property of acyclic graphs is the following:

Theorem 8.2 If $\langle \mathcal{N}, D\rangle$ is acyclic then for any $x \notin M$:

$$R_{M\cup\{x\}}(D(x)) = R_M(D(x))$$

Proof: "$\subseteq$" is trivial, so suppose $y \in R_M(D(x))$ and let p be a witness to this. If some element of p equals x, say $p[i] = x$ then the path $\langle x\rangle +\!\!\!+\, p[1\,..\,i]$ is a cycle. So p must avoid $\{x\}$, so it is a witness to $y \in R_{M\cup\{x\}}(D(x))$. ■

So for an acyclic binary graph, where $D(x) = \{L(x), R(x)\}$, we have for $x \notin M$:

```
$\mathsf{MARK}(\{x\}) \approx$ $M := M \cup \{x\}$;
                 **if** $L(x) \notin M$ **then** $M := M \cup R_M(L(x))$ **fi**;
                 **if** $R(x) \notin M$ **then** $M := M \cup R_M(R(x))$ **fi**
```

By Corollary A.8.

```
$\approx$ **if** $L(x) \notin M$ **then** $M := M \cup R_M(L(x))$ **fi**;
  $M := M \cup \{x\}$;
  **if** $R(x) \notin M$ **then** $M := M \cup R_M(R(x))$ **fi**
```

By Theorem 8.2.

```
$\approx$ **if** $L(x) \notin M$ **then** $\mathsf{MARK}(\{L(x)\})$ **fi**;
  $M := M \cup \{x\}$;
  **if** $R(x) \notin M$ **then** $\mathsf{MARK}(\{R(x)\})$ **fi**
```

by Corollary A.8 again.

```
$\approx$ **proc** $\mathsf{mark}(x) \equiv$
    **if** $L(x) \notin M$ **then** $\mathsf{mark}(L(x))$ **fi**;
    $M := M \cup \{x\}$;
    **if** $R(x) \notin M$ **then** $\mathsf{mark}(R(x))$ **fi.**
```

By Theorem 4.2. We have $x \in R(\{\mathsf{root}\})$ at the top of mark which is equivalent to $x = \mathsf{root} \lor x \in R(D(\mathsf{root}))$ (by Corollary A.7). So $L(x) \in R(D(\mathsf{root}))$ and $R(x) \in R(D(\mathsf{root}))$. So, since the graph is acyclic we must have $L(x) \neq \mathsf{root}$ and $R(x) \neq \mathsf{root}$.

As in Section 6.2 we remove parameter by introducing a local variable x' and then introduce the ghost variables l, r and q. We can then replace references to L, R and x' by references to ghost variables. We get: $\mathsf{MARK}(\{\mathsf{root}\}) \approx$

```
**begin** $x := \mathsf{root}$; mark; $\{x = \mathsf{root}\}$
**where**
**proc** mark $\equiv$
  $\circlearrowleft\langle l[x], r[x], q\rangle$;
  **if** $q \notin M$ **then** $\circlearrowleft\langle x, q\rangle$;
                 mark; $\{x \neq \mathsf{root} \land x \notin M\}$;
                 $\circlearrowleft\langle x, q\rangle$ **fi**;
  $M := M \cup \{x\}$
  $\circlearrowleft\langle l[x], r[x], q\rangle$;
  **if** $q \notin M$ **then** $\circlearrowleft\langle x, q\rangle$;
                 mark; $\{x \neq \mathsf{root} \land x \in M\}$;
                 $\circlearrowleft\langle x, q\rangle$ **fi**;
  $\circlearrowleft\langle l[x], r[x], q\rangle$.
```

For the recursion removal step, we already have a set of suitable assertions for Theorem 4.7 so *no* additional storage is required. After recursion removal and restructuring we get:

```
$x := \mathsf{root}$;
**do** **do** $\circlearrowleft\langle l[x], r[x], q\rangle$;
      **if** $q \notin M$ **then** $\circlearrowleft\langle x, q\rangle$ **else** **exit** **fi** **od**;
   **do** $M := M \cup \{x\}$; $\circlearrowleft\langle l[x], r[x], q\rangle$;
      **if** $q \notin M$ **then** $\circlearrowleft\langle x, q\rangle$; **exit** **fi**;
      **do** $\circlearrowleft\langle l[x], r[x], q\rangle$;
         **if** $x = \mathsf{root}$ **then** **exit**(3) **fi**;
         **if** $x \notin M$ **then** **exit** **fi** **od** **od** **od**
```

This program has a rather more complex control flow structure than the program in Section 6.2 but it does no unnecessary tests and only requires a single mark bit instead of the two bits required by the general algorithm.

9 A Hybrid Algorithm

Although the Schorr-Waite algorithm is very efficient in the use of storage, it is less efficient than the original recursive procedure in terms of the number of assignments carried out. Also if we knew that the graph structure was similar to a "bushy tree" (with many nodes, but few long paths) then a small fixed stack would be able to deal with most, if not all, of the nodes. For example: with a binary tree in which each node had either zero or two subtrees, a stack of length 40 could deal with trees containing more than 1,000,000,000,000 nodes more efficiently than the Schorr-Waite algorithm.

> *Fancy algorithms are slow when n is small, and n is usually small.* — *Rob Pike.*

However, it is impossible to tell in advance how large a stack is required for a given tree: in the other extreme case where each node has no more than one daughter, the stack would require as many elements as the graph being marked. A typical application for graph marking is garbage collection: and in this situation it is vital to minimise the amount of memory required and know in advance how much memory will be used. This suggests using a stack to deal with the short paths and using the general "pointer switching" strategy when the stack runs out.

The algorithm we derive in this section is much more complex than previous algorithms, but we only need to apply the *same* transformations: the algorithm involves no new ideas or proofs. As one reviewer of this paper remarked: "Transformational developments (and automatic aids) are really the only hope for reasoning about such large, complex programs".

Starting with the version at the beginning of Section 6.2 we add an integer i to record the current depth of recursion nesting and insert a (redundant) test of i:

```
begin $x :=$ root; $i := 0$; $K := \langle\rangle$; mark;
where
proc mark $\equiv$
  $M := M \cup \{x\}$; $i := i + 1$;
  if $i > N$ then var $x' := x$ : if $L(x') \notin M$ then $x := L(x')$; mark; $x := x'$ fi end;
                  var $x' := x$ : if $R(x') \notin M$ then $x := R(x')$; mark; $x := x'$ fi end
             else var $x' := x$ : if $L(x') \notin M$ then $x := L(x')$; mark; $x := x'$ fi end;
                  var $x' := x$ : if $R(x') \notin M$ then $x := R(x')$; mark; $x := x'$ fi end fi;
  $i := i - 1$.
end
```

If the depth is $\leqslant N$ then we remove the local variable x' using a stack, if the depth is $> N$, we use the pointer switching idea to eliminate x'. We also add the ghost variable $m[x]$ and some assertions:

```
begin $x :=$ root; $i := 0$; $K := \langle\rangle$; mark; $\{i = 0\}$
where
proc mark $\equiv$
  $M := M \cup \{x\}$; $i := i + 1$;
  if $i > N$ then $m[x] := 1$; $\overset{\curvearrowright}{\longleftarrow}\langle l[x], q\rangle$;
                  if $q \notin M$ then $\overset{\curvearrowright}{\longleftarrow}\langle x, q\rangle$; mark; $\{m[q] = 1 \wedge i > N\}$; $\overset{\curvearrowright}{\longleftarrow}\langle x, q\rangle$ fi;
                  $m[x] := 2$; $\overset{\curvearrowright}{\longleftarrow}\langle l[x], q\rangle$; $\overset{\curvearrowright}{\longleftarrow}\langle r[x], q\rangle$;
                  if $q \notin M$ then $\overset{\curvearrowright}{\longleftarrow}\langle x, q\rangle$; mark; $\{m[q] = 2 \wedge i > N\}$; $\overset{\curvearrowright}{\longleftarrow}\langle x, q\rangle$ fi;
                  $\overset{\curvearrowright}{\longleftarrow}\langle r[x], q\rangle$
             else $m[x] := 1$;
                  if $l[x] \notin M$ then $K \xleftarrow{\text{push}} x$; $x := l[x]$;
                                          mark; $\{m[x] = 1 \wedge 0 < i \leqslant N\}$;
                                          $x \xleftarrow{\text{pop}} K$ fi;
                  $m[x] := 2$;
                  if $r[x] \notin M$ then $K \overset{\text{push}}{\longleftarrow} x$; $x := r[x]$;
                                       mark; $\{m[x] = 2 \wedge 0 < i \leqslant N\}$;
                                       $x \overset{\text{pop}}{\longleftarrow} K$ fi fi;
  $m[x] := 0$; $i := i - 1$.
end
```

Broy and Pepper claim [9] that adding a "depth counter" such as i, requires an unbounded workspace, since the counter will require up to $\lg(\#\mathcal{N})$ bits for graphs with $\#\mathcal{N}$ nodes. In practice, this is a bounded amount of storage, since even if every subatomic particle in the visible universe was pressed into service to represent a node in the graph, then 266 bits (or 34 bytes) would be sufficient to hold i. On the other hand, if one were to accept their argument, then the requirement for bounded workspace has still not been met by their algorithm: since the q pointer has to be able to hold at least $\#\mathcal{N}$ different values, so it requires at least $\lg(\#\mathcal{N})$ bits. Our version of the algorithm restores the final value of q: so any variable can be used for q, for example a variable from another part of the program, or a node which is already marked (e.g. the Λ node).

We have five calls to mark, requiring five disjoint assertions for recursion removal. But the test $i > N$ can be used to distinguish between two pairs of calls, so only three values are needed for $m[x]$. The test $i = 0$ is used to detect the outermost call. So we can restructure the body of mark as an action system and remove the recursion (we also merge M and m as usual)[2]:

```
$x :=$ root; $i := 0$; $K := \langle\rangle$;
actions $A_1$ :
$F$   $\equiv$ if $i \geqslant N$ then $A_1$ else $B_1$ fi.
$A_1$  $\equiv$ $m[x] := 1$; $i := i + 1$;
        if $m[x] \neq 0$ then $\circlearrowleft\langle l[x], q, x\rangle$; call $A_1$
                    else $\circlearrowleft\langle l[x], q\rangle$; call $A_2$ fi.
$A_2$  $\equiv$ $m[x] := 2$;
        if $m[x] \neq 0$ then $\circlearrowleft\langle r[x], l[x], q, x\rangle$; call $A_1$
                    else $\circlearrowleft\langle r[x], l[x], q\rangle$; call $A_3$ fi.
$A_3$  $\equiv$ $m[x] := 3$; $i := i - 1$; $\circlearrowleft\langle r[x], q\rangle$;
        if $i < N$ then call $\hat{B}$ else call $\hat{A}$ fi.
$B_1$  $\equiv$ $m[x] := 1$; $i := i + 1$;
        if $m[l[x]] \neq 0$ then $K \overset{\text{push}}{\longleftarrow} x$; $x := l[x]$; call $F$ else call $B_2$ fi.
$B_2$  $\equiv$ $m[x] := 2$;
        if $m[r[x]] \neq 0$ then $K \overset{\text{push}}{\longleftarrow} x$; $x := r[x]$; call $F$ else call $B_3$ fi.
$B_3$  $\equiv$ $m[x] := 3$; $i := i - 1$; if $i = 0$ then call $Z$ else call $\hat{B}$ fi.
$\hat{A}$   $\equiv$ $\circlearrowleft\langle x, q\rangle$; if $m[x] = 1$ then call $A_2$ else call $A_3$ fi.
$\hat{B}$   $\equiv$ $x \overset{\text{pop}}{\longleftarrow} K$; if $m[x] = 1$ then call $B_2$ else call $B_3$ fi. endactions
```

10 Conclusion

The method of algorithm derivation discussed here, which involves formal transformation in a wide spectrum language, together with the "ghost variable" technique for changing data representations has proved a powerful way to prove the correctness of some challenging algorithms. In this paper we discuss "pointer switching idea" first used in the Schorr-Waite graph marking algorithm [35]. This algorithm is a particularly interesting challenge in that the same data structure is used to direct the control flow and to store the original graph structure. A correctness proof for the algorithm has to show that:

1. The original graph structure is preserved by the algorithm (although it is temporarily disrupted as the algorithm proceeds);

[2] This version minimises the number of tests and assignments.

2. The algorithm achieves the correct result (all reachable nodes are marked).

Most published correctness proofs for the algorithm [16,19,27,30,34,38,48] have to treat these two problems together. The methods involving assertions (and intermittent assertions) require an understanding of the "total situation" at any point in the execution of the program. The "ghost variable" approach adopted by Broy and Pepper [9] and ourselves, makes it possible to separate the two requirements: starting with a simple abstract algorithm which is known to mark all the nodes, the concrete data structure is introduced as a collection of ghost variables which the algorithm is shown to preserve. Broy and Pepper use a completely applicative programming style and introduce a new abstract data type for binary graphs with various operations on the pointers. This means that they have to prove various properties of this new data structure, and their algorithm derivation requires several induction proofs. In contrast, we believe that the basic idea behind the algorithm can be formulated and clearly explained as a recursive procedure, representing the graph as a collection of arrays. Efficient algorithms can then be derived by the application of general-purpose recursion removal and restructuring transformations.

We derive several marking algorithms which make use of the pointer switching ides and which illustrate the advantages of transformational development using a wide spectrum language:

1. The development divides into four stages: (i) Recursive Algorithm; (ii) Apply the pointer switching idea; (iii) Recursion Removal; and (iv) Restructuring. Each stage uses general-purpose transformations with no complicated invariants or induction proofs;
2. The method easily scales up to larger programs: for example, the hybrid algorithm in Section 9 is much more complex than the simple algorithms, yet our development uses the same transformations and involves no new ideas or proofs;
3. Elimination of induction arguments: the tactics we used to introduce ghost variables can be applied over and over again in transformational developments. The technique of replacing recursive calls by the equivalent specification means that most of our reasoning of invariants on ghost variables is carried out on *non-recursive* and *non-iterative* programs. Hence there is no need for an induction argument!

Acknowledgements

This work was partly supported by grant SE-88 from the Alvey Directorate, partly by the Durham Software Engineering Ltd. and partly by the Science and Engineering Research Council.

References

[1] J. R. Abrial, S. T. Davis, M. K. O. Lee, D. S. Neilson, P. N. Scharbach & I. H. Sørensen, *The B Method*, BP Research, Sunbury Research Centre, U.K., 1991.

[2] J. Arsac, "Transformation of Recursive Procedures," in *Tools and Notations for Program Construction*, D. Neel, ed., Cambridge University Press, Cambridge, 1982, 211–265.

[3] J. Arsac, "Syntactic Source to Source Program Transformations and Program Manipulation," *Comm. ACM* 22 (Jan., 1982), 43–54.

[4] R. J. R. Back, *Correctness Preserving Program Refinements*, Mathematical Centre Tracts #131, Mathematisch Centrum, Amsterdam, 1980.

[5] R. J. R. Back & J. von Wright, "Refinement Concepts Formalised in Higher-Order Logic," *Formal Aspects of Computing* 2 (1990), 247–272.

[6] R. Balzer, "A 15 Year Perspective on Automatic Programming," *IEEE Trans. Software Eng.* SE 11 (Nov., 1985), 1257–1267.

[7] F. L. Bauer, B. Moller, H. Partsch & P. Pepper, "Formal Construction by Transformation—Computer Aided Intuition Guided Programming," *IEEE Trans. Software Eng.* 15 (Feb., 1989).

[8] R. Bird, "Lectures on Constructive Functional Programming," in *Constructive Methods in Computing Science*, M. Broy, ed., NATO ASI Series #F55, Springer-Verlag, New York–Heidelberg–Berlin, 1989, 155–218.

[9] M. Broy & P. Pepper, "Combining Algebraic and Algorithmic Reasoning: an Approach to the Schorr-Waite Algorithm," *Trans. Programming Lang. and Syst.* 4 (July, 1982), 362–381.

[10] B. A. Davey & H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, Cambridge, 1990.

[11] N. Derschowitz, "The Schorr-Waite Marking Algorithm Revisited," *Inform. Process. Lett.* 11 (1980), 141–143.

[12] R. B. K. Dewar, A. Grand, S. C. Liu & J. T. Schwartz, "Programming by Refinement as Exemplified by the SETL Representation Sublanguage," *ACM Trans. on Prog. Lang. and Systems* 1 (July, 1979).

[13] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[14] E. Engeler, *Formal Languages: Automata and Structures*, Markham, Chicago, 1968.

[15] S. L. Gerhart, "A Derivation-Oriented Proof of the Schorr-Waite Graph Marking Algorithm," in *Program Construction*, G. Goos & H. Hartmanis, eds., Lect. Notes in Comp. Sci. #69, Springer-Verlag, New York–Heidelberg–Berlin, 1979, 472–492.

[16] D. Gries, "The Schorr-Waite Graph Marking Algorithm," *Acta Inform.* 11 (1979), 223–232.

[17] D. Gries, "Is Sometimes Ever Better than Always?," in *Program Construction*, G. Goos & H. Hartmanis, eds., Lect. Notes in Comp. Sci. #69, Springer-Verlag, New York–Heidelberg–Berlin, 1979, 113–124.

[18] D. Gries, *The Science of Programming*, Springer-Verlag, New York–Heidelberg–Berlin, 1981.

[19] M. Griffiths, *Development of the Schorr-Waite Algorithm*, Lect. Notes in Comp. Sci. #69, Springer-Verlag, New York–Heidelberg–Berlin, 1979.

[20] I. J. Hayes, *Specification Case Studies*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[21] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM* (1969).

[22] C. B. Jones, *Systematic Software Development using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[23] C. B. Jones, K. D. Jones, P. A. Lindsay & R. Moore, *mural: A Formal Development Support System*, Springer-Verlag, New York–Heidelberg–Berlin, 1991.

[24] U. Jørring & W. L. Scherlis, "Deriving and Using Destructive Data Types," in *Program Specification and Transformation: Proceedings of the IFIP TC2/WG 2.1 Working Conference, Bad Tölz, FRG, 15-17 April, 1986*, L. G. L. T. Meertens, ed., North-Holland, Amsterdam, 1987.

[25] C. R. Karp, *Languages with Expressions of Infinite Length*, North-Holland, Amsterdam, 1964.

[26] D. E. Knuth, "Structured Programming with the GOTO Statement," *Comput. Surveys* 6 (1974), 261–301.

[27] R. Kowalski, "Algorithm = Logic + Control," *Comm. ACM* 22 (July, 1979), 424–436.

[28] Z. Manna & R. Waldinger, "Is 'Sometime' Sometimes Better than 'Always'? Intermittent Assertions in Proving Program Correctness," *Comm. ACM* 21 (1978), 159–172.

[29] C. C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1994, Second Edition.

[30] J. H. Morris, "A Proof of the Schorr-Waite Algorithm," in *Theoretical Foundations of Programming Methodology* Int. Summer School, Marktoberdorf 1981, M. Broy & G. Schmidt, eds., Dordrecht: Reidel, 1982.

[31] M. Neilson, K. Havelund, K. R. Wagner & E. Saaman, "The RAISE Language, Method and Tools," *Formal Aspects of Computing* 1 (1989), 85–114 .

[32] H. Partsch, "The CIP Transformation System," in *Program Transformation and Programming Environments* Report on a Workshop directed by F. L. Bauer and H. Remus, P. Pepper, ed., Springer-Verlag, New York–Heidelberg–Berlin, 1984, 305–323.

[33] E. L. Post, "Formal Reduction of the General Combinatorial Decision Problem," *Amer. J. Math.* (1943).

[34] W. P. de Roever, "On Backtracking and Greatest Fixpoints," in *Formal Description of Programming Constructs*, E. J. Neuhold, ed., North-Holland, Amsterdam, 1978, 621–636.

[35] H. Schorr & W. M. Waite, "An Efficient Machine-Independant Procedure for Garbage Collection in Various List Structures," *Comm. ACM* (Aug., 1967).

[36] C. T. Sennett, "Using Refinement to Convince: Lessons Learned from a Case Study," *Refinement Workshop, 8th–11th January, Hursley Park, Winchester* (Jan., 1990).

[37] D. Taylor, "An Alternative to Current Looping Syntax," *SIGPLAN Notices* 19 (Dec., 1984), 48–53.

[38] R. W. Topor, "The Correctness of the Schorr-Waite List Marking Algorithm," *Acta Inform.* 11 (1979), 211–221.

[39] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.

[40] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/sorting-t.ps.gz⟩.

[41] M. Ward, "A Recursion Removal Theorem—Proof and Applications," Durham University, Technical Report, 1991, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/rec-proof-t.ps.gz⟩.

[42] M. Ward, "A Recursion Removal Theorem," Springer-Verlag, Proceedings of the 5th Refinement Workshop, London, 8th–11th January, New York–Heidelberg–Berlin, 1992, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/ref-ws-5.ps.gz⟩.

[43] M. Ward, "Foundations for a Practical Theory of Program Refinement and Transformation," Durham University, Technical Report, 1994, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/foundation2-t.ps.gz⟩.

[44] M. Ward, "Using Formal Transformations to Construct a Component Repository," in *Software Reuse: the European Approach*, Springer-Verlag, New York–Heidelberg–Berlin, Feb., 1991, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/reuse.ps.gz⟩.

[45] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/prog-spec.ps.gz⟩.

[46] M. Ward & K. H. Bennett, "A Practical Program Transformation System For Reverse Engineering," *Working Conference on Reverse Engineering, May 21–23, 1993*, Baltimore MA (1993), ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/icse.ps.gz⟩.

[47] D. Wile, "Type Transformations," *IEEE Trans. Software Eng.* 7 (Jan., 1981).

[48] L. Yelowitz & A. G. Duncan, "Abstractions, Instantiations and Proofs of Marking Algorithms," *SIGPLAN Notices* 12 (1977), 13–21, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/prog-spec.ps.gz⟩.

[49] E. J. Younger & M. Ward, "Inverse Engineering a simple Real Time program," *J. Software Maintenance: Research and Practice* 6 (1993), 197–234, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/eddy-t.ps.gz⟩.

Appendices

A Graph Marking

Definition A.1 Given any set $\mathcal{N}$ and any function $D : \mathcal{N} \to \wp(\mathcal{N})$, the *transitive closure* $\mathcal{TC}(D) : \wp(\mathcal{N}) \to \wp(\mathcal{N})$ of D is defined as:

$$\mathcal{TC}(D) =_{\text{DF}} \bigcup_{n<\omega} \mathcal{TC}^n(D)$$

where ω represents infinity, so $\bigcup_{n<\omega} X_n$ means $X_0 \cup X_1 \cup X_2 \cup \dots$ and for each $X \subseteq \mathcal{N}$:

$$\mathcal{TC}^0(D)(X) =_{\text{DF}} X \qquad \text{and for all } n \geqslant 0: \qquad \mathcal{TC}^{n+1}(D)(X) =_{\text{DF}} \bigcup \{\, D(y) \mid y \in \mathcal{TC}^n(D)(X) \,\}$$

We therefore define $R =_{\text{DF}} \mathcal{TC}(D)$. The set of nodes reachable from X is $R(X)$.

Theorem A.2 For any function D, the function $R = \mathcal{TC}(D)$ is a *closure operator* on sets of nodes. For all $X, Y \subseteq \mathcal{N}$:

1. $X \subseteq R(X)$;
2. If $X \subseteq Y$ then $R(X) \subseteq R(Y)$;
3. $R(R(X)) = R(X)$.

An equivalent definition of $R(X)$ may be given in terms of the set of “closed” sets of nodes. For each D we define $\Gamma_D \subseteq \wp(\mathcal{N})$, to be the set of subsets $X \in \Gamma_D$ which are closed under the $D(\cdot)$ function, i.e.:

$$\Gamma_D =_{\text{DF}} \{\, X \subseteq \mathcal{N} \mid \forall x \in X.\, D(x) \subseteq X \,\}$$

Then for $X \subseteq \mathcal{N}$ we can define:

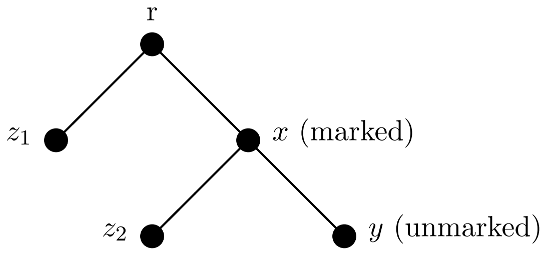
$$R(X) =_{\text{DF}} \bigcap \{\, Y \in \Gamma_D \mid X \subseteq Y \,\}$$

See [10] for the proof of this equivalence.

From this we can define a specification for a program which marks all nodes reachable from the given set X. We “mark” a node by adding it to the set M of marked nodes:

$$\mathsf{MARK}(X) =_{\text{DF}} M := M \cup R(X)$$

Our algorithm will be designed to work correctly when some nodes are already marked: most derivations assume $M = \varnothing$ initially. We will derive several implementations of this specification, each of which starts by marking a node $x \in X$ and then marking the nodes reachable from the daughters $D(x)$ of x. For efficiency reasons, we want to assume that all unmarked nodes reachable from X are reachable via unmarked nodes. To see why this is needed, consider this situation:



where $X = \{r\}$, $M = \{x\}$, $D(r) = \{z_1, x\}$, $D(x) = \{z_2, y\}$ and $D(z_1) = D(z_2) = D(y) = \varnothing$. After marking r and z_1 we reach x and find it already marked. We have no way of knowing whether there are any unmarked descendants of x, so *every* marked node would have to be explored fully. But now suppose $r = z_2$. Then exploring z_2 fully would lead to an infinite loop.

This problem is avoided if all unmarked nodes reachable from X are reachable via unmarked nodes. For each $x \in \mathcal{N}$ we define $D_M(x) =_{\text{DF}} D(x) \setminus M$ to be the set of unmarked daughters. Then the set of unmarked nodes reachable via unmarked nodes from X is $R_M(X \setminus M)$ where $R_M =_{\text{DF}} \mathcal{TC}(D_M)$. Our new specification assumes that all the unmarked reachable nodes are reachable via unmarked nodes.

$$\mathsf{MARK}(X) =_{\text{DF}} \{M \cup R(X) = M \cup R_M(X)\};\ M := M \cup R(X)$$

The following theorem follows from the definition of R_M, it shows that marked nodes can be removed from X without affecting $R_M(X)$:

Theorem A.3 If $Y \subseteq M$ then $R_M(X) = R_M(X \setminus Y)$.

If a node is reachable, then there must be a way to reach it, and this observation leads to the concept of a path:

Definition A.4 A *path from X to y* is a non-empty sequence of nodes $p = \langle x_1, x_2, \ldots, x_n \rangle$ where $n \geqslant 1$, $x_1 \in X$, $x_n = y$ and for each $1 \leqslant i < n$, $x_{i+1} \in D(x_i)$. We define $P(D)$ to be the set of all paths, and for each $n \geqslant 1$, define $P^n(D)$ to be the set of paths of length n.

The next theorem shows that the intuitive "path based" definition of reachability is the same as the transitive closure definition. A node is reachable from X if and only if there is a path to the node from X:

Theorem A.5 $y \in R(X) \iff \exists p \in P(D).\,(p[1] \in X \wedge p[\ell(p)] = y)$

Proof: It is sufficient to show by induction on n that $y \in \mathcal{TC}^n(D)(X) \iff \exists p \in P^{n+1}(D).\,(p[1] \in X \wedge p[\ell(p)] = y)$. For $n = 0$ we have $\mathcal{TC}^0(D)(X) = X$ and $P^1(D) = \{\ \langle x \rangle \mid x \in \mathcal{N}\ \}$, so the result follows.

Suppose the result holds for n and suppose $y \in \mathcal{TC}^{n+1}(D)(X)$. Then there exists $y' \in \mathcal{TC}^n(X)$ such that $y \in D(y')$. By the induction hypothesis, there exists $p' \in P^{n+1}(D)$ such that $p'[1] \in X$ and $p'[\ell(p')] = y'$. Hence $p = p' +\!\!+ \langle y \rangle$ is a path from X to y of length in $P^{n+2}(D)$.

Conversely, let $p \in P^{n+2}(D)$ be a path from X to y. Then $p' = p[1\,..\,n+1]$ is a path from X to some element $y' = p[n+1]$ where $y \in D(y')$. By the induction hypothesis, $y' \in \mathcal{TC}^n(D)(X)$ so $y \in \mathcal{TC}^{n+1}(D)(X)$ as required. ■

So for every $y \in R(X)$ there is a path to y from X which we call a *witness* to the fact that $y \in R(X)$. Conversely, every path $p \in P(D)$ whose first element is in X is a witness to the fact that its last element is in $R(X)$. In particular, every path p is a witness that $p[\ell(p)] \in R(\{p[1]\})$. If no element of a path p is in the set M then we say *p avoids M*.

Theorem A.6 *Reachability Theorem:*
Let M and X be sets of nodes such that $M \cup R(X) = M \cup R_M(X)$ and let $x \in X \setminus M$. Let A and B be any subsets of $R_M(\{x\})$ such that $R_M(\{x\}) \setminus A \subseteq R_{M \cup A}(B)$. Then:

$$M \cup R_M(X) \ = \ M \cup A \cup R_{M \cup A}((X \setminus \{x\}) \cup B) \ = \ M \cup A \cup R((X \setminus \{x\}) \cup B)$$

Proof: Let $M' = M \cup A$ and $X' = (X \setminus \{x\}) \cup B$. For convenience we define:

$$\mathsf{LHS} = M \cup R_M(X) \qquad \mathsf{CHS} = M' \cup R_{M'}(X') \qquad \mathsf{RHS} = M' \cup R(X')$$

Part 1: We claim $\mathsf{LHS} \subseteq \mathsf{CHS} \subseteq \mathsf{RHS}$. The second relation follows from $R_M(X) \subseteq R(X)$ for any M and X. So suppose $y \in \mathsf{LHS}$. If $y \in M'$ then the result is trivial, so suppose $y \in R_M(X)$.

(i) If $y \in R_M(\{x\})$ then, since $y \notin A$, we have $y \in R_M(\{x\}) \setminus A \subseteq R_{M'}(B) \subseteq \mathsf{CHS}$.

(ii) If $y \notin R_M(\{x\})$ then, since $y \in R_M(X) = R_M(X \setminus \{x\}) \cup R_M(\{x\})$, we must have $y \in R_M(X \setminus \{x\})$. Let p be a witness for this. Suppose element $p[i]$ of p is in $A \subseteq R_M(\{x\})$ and let p' be a witness of this fact. Then $p' +\!\!+ p[i+1\,..\,]$ is a witness that $y \in R_M(\{x\})$ which is a contradiction. So p must avoid A, so p is a witness that $y \in R_{M'}(X \setminus \{x\}) \subseteq \mathsf{CHS}$.

So part 1 is proved.

Part 2: We claim $\mathsf{RHS} \subseteq \mathsf{LHS}$ which together with part 1 proves the equalities. Suppose $y \in \mathsf{RHS}$. If $y \in M'$ then the result is trivial, so suppose $y \in R(X') = R(X \setminus \{x\}) \cup R(B)$.

(i) Suppose $y \notin R(B)$, so $y \in R(X \setminus \{x\}) \subseteq R(X)$. Then $y \in R_M(X)$ since $M \cup R(X) = M \cup R(X)$ and $y \notin M$. So $y \in \mathsf{CHS}$.

(ii) Suppose $y \in R(B)$. We have $B \subseteq R_M(\{x\})$ so by the properties of closure operators we have $R(B) \subseteq R(R_M(\{x\})) \subseteq R(R_M(X)) \subseteq R(R(X)) = R(X)$. So $y \in R(X)$ so $y \in R_M(X)$ as for (i).

So part 2 is proved and the theorem is proved. ■

Two obvious choices for A are $\{x\}$ and $R_M(\{x\})$. In the former case, a suitable choice for B is $D(x) \setminus (M \cup \{x\})$ and in the latter case, the only choice for B is $\varnothing$. So we have two corollaries:

Corollary A.7 If $M \cup R(X) = M \cup R_M(X)$ and $x \in X \setminus M$ then:

$$M \cup R_M(X) \;=\; M \cup \{x\} \cup R_{M \cup \{x\}}(X') \;=\; M \cup \{x\} \cup R(X')$$

where $X' = (X \setminus \{x\}) \cup (D(x) \setminus (M \cup \{x\}))$.

Corollary A.8 If $M \cup R(X) = M \cup R_M(X)$ and $x \in X \setminus M$ then:

$$M \cup R_M(X) = M \cup R(\{x\}) \cup R_{M \cup R(\{x\})}(X \setminus \{x\}) = M \cup R(\{x\}) \cup R(X \setminus \{x\})$$

B An Iterative Algorithm

By applying Corollary A.7 we can transform $\mathsf{MARK}(X)$ as follows:

```
MARK(X) ≈ {$M \cup R(X) = M \cup R_M(X)$};
          if $X \subseteq M$ then skip
                else {$X \setminus M \neq \varnothing$}; $M := M \cup R_M(X)$ fi
        ≈ {$M \cup R(X) = M \cup R_M(X)$};
          if $X \subseteq M$
            then skip
             else var $x \in X \setminus M$ :
                    $M := M \cup \{x\}$; $X := (X \setminus \{x\}) \cup (D(x) \setminus M')$;
                    {$M \cup R(X) = M \cup R_M(X)$};
                    $M := M \cup R(X)$ end fi
```

If the set $R(X)$ is finite, then the set $R(X) \setminus M$ is reduced before the copy of $\mathsf{MARK}(X)$ on the RHS. So by the recursive implementation theorem (Theorem 4.2), $\mathsf{MARK}(X)$ is equivalent to:

```
proc mark ≡
  if $X \subseteq M$ then skip
          else var $x \in X \setminus M$ :
                 $M := M \cup \{x\}$; $X := (X \setminus \{x\}) \cup (D(x) \setminus M')$;
                 mark end fi.
```

Apply the recursion removal theorem (Theorem 4.5) to get the iterative algorithm:

```
proc mark ≡
  while $X \setminus M \neq \varnothing$ do
      var $x \in X \setminus M$ :
        $M := M \cup \{x\}$; $X := (X \setminus \{x\}) \cup (D(x) \setminus M')$ end od
```

Note that if $M \cap X = \varnothing$ initially then this will always be true at the beginning of the loop, and we can replace $X \setminus M$ by X.

This algorithm maintains a set X of nodes which have not been marked completely, it repeatedly removes an element from X, marks it, and adds its unmarked daughters to X.

C A Recursive Algorithm

By applying Corollary A.8 we can transform $\mathsf{MARK}(X)$ as follows:

```
$\mathsf{MARK}(X)$ $\approx$ $\{M \cup R(X) = M \cup R_M(X)\};\ M := M \cup R_M(X)$
          $\leqslant$ if $X \neq \varnothing$ then var $x \in X$ :
                                  if $x \notin M$ then $M := M \cup R_M(\{x\})$ fi;
                                  $X := X \setminus \{x\};\ \{M \cup R(X) = M \cup R_M(X)\};$
                                  $M := M \cup R_M(X)$ end fi
          $\approx$ if $X \neq \varnothing$ then var $x \in X$ :
                                  if $x \notin M$ then $M := M \cup R_M(\{x\})$ fi;
                                  $X := X \setminus \{x\};$ $\mathsf{MARK}(X)$ end fi
```

Apply Theorem 4.2 using the fact that X is reduced to get:

```
          $\approx$ proc $F$ $\equiv$ if $X \neq \varnothing$
                        then var $x \in X$ :
                               if $x \notin M$ then $M := M \cup R_M(\{x\})$ fi;
                               $X := X \setminus \{x\};$ $F$ end fi
```

Remove the recursion:

```
          $\approx$ for $x \in X$ do
               if $x \notin M$ then $M := M \cup R_M(\{x\})$ fi od
          $\approx$ for $x \in X$ do
               if $x \notin M$ then var $X := \{x\}$ : $\mathsf{MARK}(X)$ end fi od
```

Now suppose X contains the single element $x \notin M$. Then:

```
$\mathsf{MARK}(\{x\})$ $\approx$ $\{M \cup R(\{x\}) = M \cup R_M(\{x\})\};\ M := M \cup R_M(\{x\})$
            $\approx$ $\{M \cup R(\{x\}) = M \cup R_M(\{x\})\};\ M := M \cup \{x\};$
              $M := M \cup R_M(D(x) \setminus M)$
```

Apply Corollary A.7:

```
            $\approx$ $\{M \cup R(X) = M \cup R_M(X)\};\ M := M \cup \{x\};$
              var $X := D(x) \setminus M$ :
                $\{M \cup R(X) = M \cup R_M(X)\};\ M := M \cup R_M(X)$ end
            $\approx$ $\{M \cup R(X) = M \cup R_M(X)\};\ M := M \cup \{x\};$
              var $X := D(x) \setminus M$ :
                MARK(X) end
            $\approx$ $\{M \cup R(X) = M \cup R_M(X)\};\ M := M \cup \{x\};$
              for $x \in D(x) \setminus M$ do
                  if $x \notin M$ then var $X := \{x\}$ : $\mathsf{MARK}(X)$ end fi od
```

by the result above. Now we can apply Theorem 4.2 again:

```
            $\approx$ $\{M \cup R(X) = M \cup R_M(X)\};$
              proc $F$ $\equiv$
                $M := M \cup \{x\};$
                for $x \in D(x) \setminus M$ do
                    if $x \notin M$ then $F$ fi od
```

There is no need to remove elements of M from $D(x)$ in the **for** loop since these will be tested for in the loop body. Also, the result is slightly clearer if we add a parameter to the procedure and rename the iteration variable. So we have the following recursive marking algorithm, which marks all nodes reachable from x. If $x \notin M$ and $M \cup R(\{x\}) = M \cup R_M(\{x\})$ then $\mathsf{mark}(x) \approx \mathsf{MARK}(\{x\})$ where:

```
proc mark($x$) ≡
  $M := M \cup \{x\}$;
  for $y \in D(x)$ do
      if $y \notin M$ then mark($y$) fi od.
```

We can remove the recursion for this program to get a different iterative algorithm to that of Section B (we do not need a stack for the parameter, but we *do* need a stack for the local set variable D introduced by the **for** loop):

```
proc mark($x$) ≡
  var $K := \langle\rangle, D := \langle\rangle$ :
    do $M := M \cup \{x\}$;
       $K \xleftarrow{\text{push}} D$; $D := D(x)$;
       do if $D = \varnothing$ then $D \xleftarrow{\text{pop}} K$; if $K = \langle\rangle$ then exit(2) fi
                         else $x := x'.(x' \in D)$; $D := D \setminus \{x\}$;
                              if $x \notin M$ then exit fi fi od od end.
```

It is an interesting and instructive exercise to transform this iterative program into a refinement of the program derived in Section B.