Provably Correct Derivation of Algorithms Using FermaT

Martin Ward and Hussein Zedan
Software Technology Research Lab,
De Montfort University,
Bede Island Building,
The Gateway,
Leicester LE1 9BH, UK
`martin@gkc.org.uk` and `zedan@dmu.ac.uk`

Abstract

The transformational programming method of algorithm derivation starts with a formal specification of the result to be achieved, plus some informal ideas as to what techniques will be used in the implementation. The formal specification is then transformed into an implementation, by means of correctness-preserving refinement and transformation steps, guided by the informal ideas. The transformation process will typically include the following stages: (1) Formal specification (2) Elaboration of the specification, (3) Divide and conquer to handle the general case (4) Recursion introduction, (5) Recursion removal, if an iterative solution is desired, (6) Optimisation, if required. At any stage in the process, sub-specifications can be extracted and transformed separately. The main difference between this approach and the invariant based programming approach (and similar stepwise refinement methods) is that loops can be introduced and manipulated while maintaining program correctness and with no need to derive loop invariants. Another difference is that at every stage in the process we are working with a correct program: there is never any need for a separate "verification" step. These factors help to ensure that the method is capable of scaling up to the development of large and complex software systems. The method is applied to the derivation of a complex linked list algorithm and produces code which is over twice as fast as the code written by Donald Knuth to solve the same problem.

Contents

1 Introduction

The *waterfall model* of software development sees progress as flowing steadily downwards (like a waterfall) through the following stages:

1. Requirements Elicitation: analysing the problem domain and determining from the users what the program is required to do;

2. Design: developing the overall structure of the program;

3. Implementation: writing source code to implement the design in a particular programming language;

4. Verification: running tests and debugging;

5. Maintenance: any modifications required after delivery to correct faults, improve performance, or adapt the product to a modified environment [06]

In theory, one proceeds from one phase to the next in a purely sequential manner. But in practice, at each stage in the process, information may be uncovered which affects previous stages. For example, during implementation it may be determined that the design is unsuitable and needs to be changed, during debugging the program implementation may have to be changed to fix the bugs, and so on. So the process described above will also require feedback loops from each stage to preceding stages.

It has long been recognised that while testing may increase our confidence in the correctness of a program, no amount of testing can prove that a program is correct. As Dijkstra said: “Program testing can be used to show the presence of bugs, but never to show their absence” [Dij70]. To prove that a program is correct we need a precise mathematical specification which defines what the program is supposed to do, and a mathematical proof that the program satisfies the specification.

The *program verification* method of developing correct code involves writing code and then verifying its correctness. A simple loop, for example, can be verified using the method of “loop invariants”. This takes the following steps:

1. Determine the loop termination condition;

2. Determine the loop body;

3. Determine a suitable loop invariant;

4. Prove that the loop invariant is preserved by the loop body;

5. Determine a variant function for the loop;

6. Prove that the variant function is reduced by the loop body (thereby proving termination of the loop);

7. Prove that the combination of the invariant plus the termination condition satisfies the specification for the loop.

However, loop invariants and postconditions can be difficult to determine with sufficient precision, computing verification conditions can be tedious, and proving the correctness of verification conditions can be difficult. Even with the aid of an automated proof assistant, there may still be several hundred remaining "proof obligations" to discharge (these are theorems which need to be proved in order to verify the correctness of the development) [BiM99, JJL91, NHW89]. In addition, should the implementation happen to be incorrect (i.e. the program has a bug), then the attempt at a proof is doomed to failure.

In a paper published in 1990, Sennett [Sen90] suggested that for "real" sized programs it was impractical to discharge more than a tiny fraction of the proof obligations. He presented 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. However, since that time, there has been considerable research on the development of automated theorem provers (e.g. Event-B [ABH06]), which has led to a resurgence of interest in the program verification approach in the form of a "posit and prove" style of programming. Note that this approach still requires properties such as invariants and variants to be provided by the developers [But06]. With improvements in automated theorem provers, a large proportion of proof obligations (POs) *can* be discharged automatically: but many still require user interaction which often requires (expensive) theorem proving experts [BGL11]. For commercial application this can be thousands of proof obligations, requiring many man months or years of work. One industrial case study using Event-B generated 300 POs, of which 100 required user intervention [BGL11]. However, most of these may be handled using proof strategies leaving as few as 5%–10% of proof obligations which actually require manual proof.

An alternative to this *a posteriori* method, which was originally proposed by Dijkstra [Dij], is to control the process of program generation by constructing loop invariants in parallel with the construction of the code. Combined with *stepwise refinement* [Dij70, Wir71], this approach is claimed to scale up to fairly large programs.

A further refinement of this approach, called *Invariant based programming*, is to develop loop invariants *before* the code itself is written. The idea has been proposed from the late 70's by several researchers in different forms. Dijkstra's later work on programming [Dij76] emphasises the development of a loop invariant as an important initial step before developing the body of the loop. Figure 1 summarises this approach. Notice that in the program verification method, Dijkstra's

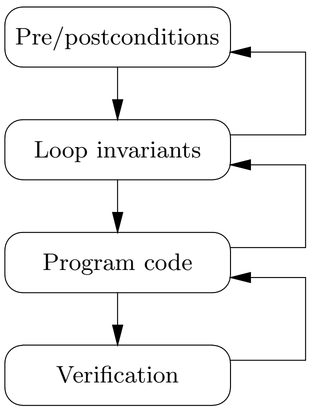

Figure 1: Invariant based programming

approach and the invariant based programming approach, the task of developing the loop invariant is moved to earlier and earlier phases in the process. Gries [Gri81] takes up the idea that a proof of correctness and a program should be developed hand in hand. Back [Bac09] presents a notation for writing invariant based programs, a method for finding invariants before writing code and methods for checking the correctness of invariant based programs. He points out that the natural structure for the code may not be the same as the natural structure for the invariants and emphasises that the program should be structured around the invariants, so that the invariants are as simple as possible, and therefore easier to manipulate: even if this results in more complicated code.

In all the development methods we have discussed so far, *verification* is the final step in development. Up until the point where verification has been completed, the programmer cannot be sure that the program is correct. Indeed, Back [Bac09] makes it clear that the program under development does not have to terminate or be free from deadlocks, and that the initial invariant is usually both incomplete and partially wrong. He stresses that it is essential to carefully check the consistency of each transition when it is introduced.

1.1 Our Approach

In this paper we present a different method of programming, called *transformational programming* or *algorithm derivation*. The method starts with a formal specification plus some informal ideas for implementation techniques which might be useful. The formal specification is *refined* into a complete program by applying a sequence of correctness-preserving refinement steps. The choice of which transformation to apply at each stage is guided by the implementation ideas. These ideas do not have to be formalised to any particular extent, since they are only used to select between different transformations. The correctness of the transformation process guarantees that the transformed program is equivalent to the original. The method is summarised in Figure 2.

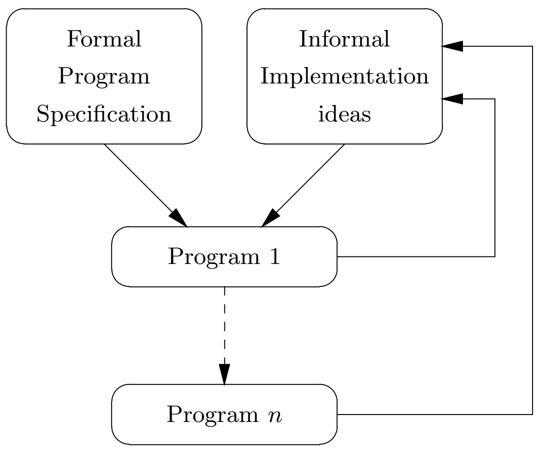

Figure 2: Algorithm Derivation

Developing a program by stepwise transformation is an idea which dates back at least to the late seventies, starting with Burstall and Darlington's transformation work [BuD77, Dar78], the project CIP (Computer-aided Intuition-guided Programming) [Bau79, BB85, BMP89, BaT87, Bro84] and continuing with the work of Morgan et al on the Refinement Calculus [Mor94, MRG88, MoV93] and the Laws of Programming [HHJ87]. However, the method presented here is very different from these. In the Refinement Calculus, the transformations for introducing and manipulating loops require that any loops introduced must be accompanied by suitable invariant conditions and variant functions. Morgan says: "The refinement law for iteration *relies on* capturing the potentially unbounded repetition in a single formula, the invariant", ([Mor94] p. 60, our emphasis). So, in order to refine a statement to a loop, or, more generally, to introduce any loop into the program, the developer still has to carry out all the steps 1–7 listed above for verifying the correctness of a loop.

In contrast with the refinement calculus, the method presented here does not require loop invariants. We have transformations to introduce, manipulate and remove loops which do not depend on the existence of loop invariants. Another key difference between Figure 2 and the other methods is that there is no Verification step. This is because at each stage in the derivation process we are working with a correct program. The program is always guaranteed to be equivalent to the original specification, because it was derived from the specification via a sequence of proven transformations and refinements.

Over the last twenty-five years we have developed a powerful wide-spectrum specification and programming language, called WSL, together with a large catalogue of proven program transformations and refinements which can be used in algorithm derivations and reverse engineering. The method has been applied to the derivation of many complex algorithms from specifications, including the Schorr-Waite graph marking algorithm [War96], a hybrid sorting algorithm (an efficient combination of Quicksort and insertion sort) [War90], various tree searching algorithms [War99a] and a program slicing transformation [WaZ10]. The latter example shows that a program transformation can itself be defined as a formal specification which can then be refined into an implementation of the transformation.

The transformation theory has also been used for reverse engineering and software migration and forms the basis for the commercial FermaT software migration technology [War99b, War01, War04, WaZ05, WZH04].

1.2 Outline of the Algorithm Derivation method

A typical algorithm derivation takes the following steps:

1. **Formal Specification:** Develop a formal specification of the program, in the form of a WSL specification statement (see Section 2). This defines precisely what the program is required to accomplish, without necessarily giving any indication as to how the task is to be accomplished. For example, a formal specification for the Quicksort algorithm for sorting the array $A[a \mathinner{..} b]$ is the statement SORT:

$$A[a \mathinner{..} b] := A'[a \mathinner{..} b].(\mathsf{sorted}(A'[a \mathinner{..} b]) \ \wedge \ \mathsf{permutation}(A[a \mathinner{..} b], A'[a \mathinner{..} b]))$$

   This states that the array is assigned a new value which is sorted and also a permutation of the original value. The formula $\mathsf{sorted}(A)$ is defined:

$$\forall 1 \leqslant i, j \leqslant \ell(A).\, i \leqslant j \Rightarrow A[i] \leqslant A[j]$$

   while $\mathsf{permutation}(A, B)$ is defined:

$$\ell(A) = \ell(B) \ \wedge \ \exists \pi \colon \{1, \dots, \ell(A)\} \rightarrowtail\!\!\!\!\twoheadrightarrow \{1, \dots, \ell(A)\}.\, \forall 1 \leqslant i \leqslant \ell(A).\, A[i] = B[\pi[i]]$$

   where $\pi \colon \{1, \dots, \ell(A)\} \rightarrowtail\!\!\!\!\twoheadrightarrow \{1, \dots, \ell(A)\}$ means π is a bijection (a 1–1 and onto function) from the set $\{1, \dots, \ell(A)\}$ to itself.

   The *form* of the specification should mirror the real-world nature of the requirements. It is a matter of constructing suitable abstractions such that local changes to the requirements involve local changes to the specification.

   The *notation* used for the specification should permit unambiguous expression of requirements and support rigorous analysis to uncover contradictions and omissions. It should then be straightforward to carry out a rigorous impact analysis of any changes.

   The two most important characteristics of a specification notation are that it should permit problem-oriented abstractions to be expressed, and that it should have rigorous semantics so that specifications can be analysed for anomalies.

In [Dij72], Dijkstra writes: "In this connection it might be worthwhile to point out that the purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise."

2. **Elaboration:** Elaborate the specification statement by taking out simple cases: for example, boundary values on the input or cases where there is no input data. These are applied by transforming the specification statement, typically using the Splitting a Tautology transformation (Transformation 1) followed by inserting assertions and then using the assertions to refine the appropriate copy of the specification to the trivial implementation. For Quicksort, an empty array or an array with a single element is already sorted, so SORT can be refined as **skip** in these cases:
$$\{a \geqslant b\};\ \mathsf{SORT}\ \approx\ \{a \geqslant b\};\ \textbf{skip}$$
Elaboration will typically generate a somewhat larger program containing one or more copies of the original specification statement.

3. **Divide and Conquer:** The general case is usually tackled via some form of "divide and conquer" strategy: this is where the informal implementation ideas come into play to direct the selection of transformations. For Quicksort the informal idea consists of two steps: (a) Partition the array around a pivot element: so that all the elements less than the pivot go on one side and the larger elements go on the other side. (b) Then the two sub-arrays are sorted using copies of the original specification statement.

   Divide and conquer will make the program still larger, often introducing more copies of the specification statement.

   At this point we still have a non-recursive program: so there are no induction proofs or invariants required for the transformations. The proofs typically consist of a simple case analysis, plus analysis of the specification under certain restricted preconditions.

4. **Recursion Introduction:** The next step is to apply the Recursive Implementation (Transformation 5) to produce a recursive program with no remaining copies of the specification. This may be carried out in several stages, with each stage leaving fewer copies of the specification statement, until all have been removed.

5. **Recursion Removal:** We now have an executable implementation of the specification. If an iterative implementation is required, then apply Recursion Removal (Transformation 8), or an appropriate special case of the transformation, to produce an iterative program. Again: this can be carried out in stages, or only partially: for example, tail recursion can be converted to iteration, but inner recursive calls left in place.

6. **Optimisation:** Apply further optimising transformations as required.

The method is *compositional* at several levels:

1. At any stage in the development, any part of the program can be worked on in isolation and the results "plugged back in" to the main program: this is due to the fact that refinement in WSL satisfies the *replacement property* [War89];

2. Different aspects of the development can often be handled separately: for example, correctness and efficiency. We can derive a provably correct program and then apply various optimising transformations to improve its efficiency;

3. At any stage in the process we may use data transformations to change the data representation using the "ghost variables" method [War94, War96, War93, War96] to convert abstract data structures to concrete data structures.

These compositionality properties are important properties that any program development method should satisfy.

It should be noted that stages 1–3 of any transformational derivation involve analysing programs which contain no recursion or iteration. This makes the analysis particularly straightforward: for example, induction arguments are not needed. This is fortunate, as it is these stages which require the most input from the informal implementation ideas. Stages 4–6 involve standard transformations for recursion introduction, recursion removal and optimisation. As the derivation progresses, the transformations involved become more generic and less domain-specific. The techniques of *calculational programming* [BiM96] may be relevant to these stages: these include rules such as fusion and tupling, generic maps, specialisations, abstract data types etc. In the later stages, the program will typically be getting larger, but the required transformations will become simpler and more susceptible to automation. At some point, an optimising compiler will take over and generate executable code, or the code will be directly executed by an interpreter, as appropriate.

An important advantage of the transformational derivation approach, over the various "code and verify" approaches is that it enables a separation of concerns between implementing the algorithm and applying various optimisation techniques. Some of the optimisation transformations, recursion removal for example (Transformation 8), produce code which is quite different in structure from the original unoptimised code. This new structure is generated automatically by the transformation sequence. Using the "posit and prove" approach, the programmer would have to work out in advance the structure of the optimised program: so that it can be "posited" as the next version. The programmer would also need to determine suitable loop invariants for the optimised code so that it could be proved correct. This is not necessarily impossible, but could be difficult to carry out correctly for a large and complex program.

2 The WSL Language

WSL is a *Wide Spectrum* language in that the language covers the whole spectrum from very high-level abstract specifications to detailed, low-level programming constructs. This means that the whole program derivation process can be carried out in a single language: we do not need separate "specification" and "implementation" languages with the consequent difficulty of proving the relationship between the two languages.

The WSL Language is constructed from a simple and tractable kernel language. New constructs are added to the language in a series of layers by means of *definitional transformations*. The main language layers are as follows, with each subsequent layer building on the previous layer:

- Kernel language;
- **if** statements, **while** loops;
- **do** ... **od** loops and action systems;
- Procedures;
- Functions and expressions with side effects.

The syntax and semantics of the kernel language and higher levels of WSL have been described in earlier papers [PrW94, War89, War04, WaZ07, YaW03] so will not be given here. In this paper we make use of the following WSL language constructs:

- **Skip:** The statement **skip** terminates immediately without changing the value of any variable;
- **Abort:** The statement **abort** never terminates;

- **Specification Statement:** For any formula $\mathbf{Q}$ and list of variables $\mathbf{x}$, let $\mathbf{x}'$ be the corresponding list of primed variables. The statement $\mathbf{x} := \mathbf{x}'.\mathbf{Q}$ assigns new values to the variables in $\mathbf{x}$ such that the condition $\mathbf{Q}$ becomes true. Within $\mathbf{Q}$, $\mathbf{x}$ represents the original values of $\mathbf{x}$ and $\mathbf{x}'$ represents the new values. For example, the statement:

  $$\langle x, y\rangle := \langle x', y'\rangle.(x' = y \ \wedge \ y' = x)$$

  swaps the values of variables x and y. The specification:

  $$\langle x\rangle := \langle x'\rangle.(x' = x^2 - 2x + 2)$$

  assigns x the value $x^2 - 2x + 2$. The specification:

  $$\langle x\rangle := \langle x'\rangle.(x' = x'^2 - 2x' + 2)$$

  will assign x the value 1 or 2 (with the particular value being chosen nondeterministically), while the specification:

  $$\langle x\rangle := \langle x'\rangle.(x = x'^2 - 2x' + 2)$$

  assigns x one of the two values: $1 \pm \sqrt{x-1}$

- **Simple Assignment:** For any variable v and expression e, the statement $v := e$ is defined:

  $$\langle v\rangle := \langle v'\rangle.(v' = e)$$

- **Deterministic Choice: if $\mathbf{B}_1$ then $\mathbf{S}_1$ elsif $\mathbf{B}_2$ then $\mathbf{S}_2 \ldots$ else $\mathbf{S}_n$ fi**

- **Nondeterministic Choice: if $\mathbf{B}_1 \ \rightarrow \ \mathbf{S}_1 \ \square \ \ldots \ \square \ \mathbf{B}_n \ \rightarrow \ \mathbf{S}_n$ fi**

- **While Loop: while B do S od**

- **Nondeterministic loop: do $\mathbf{B}_1 \ \rightarrow \ \mathbf{S}_1 \ \square \ \ldots \ \square \ \mathbf{B}_n \ \rightarrow \ \mathbf{S}_n$ od**

- **Floop: do S od** The floop is an "unbounded" loop which is terminated by the execution of a statement of the form **exit**(n) where n is a positive integer (not a variable or expression). **exit**(n) causes immediate termination of the n enclosing levels of nested floops. See [PrW94, War04] for more detailed discussion of these constructs and associated transformations.

- **Recursive Statement:** If **S** is any statement which contains occurrences of the symbol X as sub-statements, then the statement ($\mu X.\mathbf{S}$) is the recursive procedure whose body is **S** with X representing recursive calls to the procedure. For example, the while loop **while B do S od** is equivalent to the recursive statement (μX.**if B then S**; X **fi**).

- **Recursive procedures:** A **where** statement contains a main body plus a collection of (possibly mutually recursive) procedures:

```
begin
  S
where
  proc $F_1(\mathbf{x}_1) \ \equiv \ \mathbf{S}_1$.
  ...
  proc $F_n(\mathbf{x}_n) \ \equiv \ \mathbf{S}_n$.
end
```

- **Action System:** An *Action System* is a set of parameterless mutually recursive procedures together with the name of the first action to be called. There may be a special action Z (with no body): **call** Z results in the immediate termination of the whole action system with execution continuing with the next statement after the action system (if any). See [PrW94, War04] for more detailed discussion of action systems and associated transformations.

  An action system has this syntax:

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

  where, in this case, A_1 is the starting action: so $\mathbf{S}_1$ is the first statement to be executed. A statement of the form **call** A_i is a call to action A_i.

  If the execution of any action body must lead to an action call, then the action system is *regular.* In a regular action system, no call ever returns and the system can only be terminated by a **call** Z. A program written using labels and jumps translates directly into an action system, provided all the labels appear at the top level (not inside a structure). Labels can be promoted to the top level by introducing extra calls and actions, for example:

  A : **if B then** L : $\mathbf{S}_1$ **else** $\mathbf{S}_2$ **fi**; $\mathbf{S}_3$

  can be translated to the action system:

  actions A :
  $A \equiv$ **if B then call** L **else call** L_2 **fi.**
  $L \equiv \mathbf{S}_1$; **call** L_3.
  $L_2 \equiv \mathbf{S}_2$; **call** L_3.
  $L_3 \equiv \mathbf{S}_3$; **call** Z. **endactions**

  So, using this "promotion" method, *any* program written using labels and jumps can be translated directly into an action system. Any structured program can also be translated directly into an "unstructured" action system. For example, the while loop **while B do S od** is equivalent to the regular action system:

  actions A :
  $A \equiv$ **if B then** S; **call** A **else call** Z **fi. endactions**

Recursive procedures and action systems are similar in several ways, the differences are:

- There is nothing in a **where** statement which corresponds to the Z action: all procedures must terminate normally (and thus a "regular" set of recursive procedures could never terminate);
- Procedure calls can occur anywhere in a program, for example in the body of a **while** loop: action calls cannot occur as components of statements other than **if** statements and **do** ... **od** loops.

An action system which does not contain calls to Z can be translated to a **where** clause (the converse is only true provided no procedure call is a component of a simple statement).

The nondeterministic programming constructs are based on Dijkstra's *guarded command language* [Dij76].

3 Transformations Used in Derivations

In this section we list some of the main transformations which are used in algorithm derivations. The success of transformational programming is in a large part due to the development of a substantial catalogue of proven transformations. These transformations are very widely applicable and can be used in many different situations.

There are various methods used to prove the correctness of a transformation, these include:

- A direct proof of the equivalence of the denotational semantics for the two programs;
- A proof based on the logical equivalence, or implication, between the corresponding *weakest preconditions*. Given any program and a formula which defines a condition on the final state, the weakest precondition is the corresponding formula on the initial state such that if the program starts executing in a state satisfying the weakest precondition it is guaranteed to terminate in a state satisfying the given postcondition. In [War89] transformations are proved correct by using a "generic" postcondition in an extension of the logic. In [War04] we show that two specific postconditions are sufficient to completely capture the semantics of a program.
- A proof based on induction over the set of *truncations* of the iterative or recursive constructs in the programs: see [War04] for details of the induction rules.
- A proof based on a sequence of existing transformations.

Many proofs use a combination of these methods. Although there has been some work on formalising the semantics and transformation proofs using the Coq Proof Assistant [ZMH02a, ZMH02b] most transformation proofs are manual. However, the work involved in proving a transformation only has to be carried out once, while the transformation can be re-used in a huge number of program derivations.

The first four transformations can be proved directly from the weakest preconditions (see [War89] Chapter One for the details).

Transformation 1 *Splitting a Tautology*
If $\mathbf{B}_1 \lor \mathbf{B}_2$ is true then:

$$\mathbf{S} \approx \textsf{\textbf{if}}\ \mathbf{B}_1 \to \mathbf{S} \ \square\ \mathbf{B}_2 \to \mathbf{S}\ \textsf{\textbf{fi}}$$

For any formula $\mathbf{B}$ we have:

$$\mathbf{S} \approx \textsf{\textbf{if}}\ \mathbf{B}\ \textsf{\textbf{then}}\ \mathbf{S}\ \textsf{\textbf{else}}\ \mathbf{S}\ \textsf{\textbf{fi}}$$

These can be proved directly from the corresponding weakest preconditions: see [War89] Chapter One.

Transformation 2 *Introduce assertions*

$$\textsf{\textbf{if}}\ \mathbf{B}\ \textsf{\textbf{then}}\ \mathbf{S}_1\ \textsf{\textbf{else}}\ \mathbf{S}_2\ \textsf{\textbf{fi}} \approx \textsf{\textbf{if}}\ \mathbf{B}\ \textsf{\textbf{then}}\ \{\mathbf{B}\};\ \mathbf{S}_1\ \textsf{\textbf{else}}\ \{\neg\mathbf{B}\};\ \mathbf{S}_2\ \textsf{\textbf{fi}}$$

$$\textsf{\textbf{if}}\ \mathbf{B}_1 \to \mathbf{S}_1 \ \square\ \mathbf{B}_2 \to \mathbf{S}_2\ \textsf{\textbf{fi}} \approx \textsf{\textbf{if}}\ \mathbf{B}_1 \to \{\mathbf{B}_1\};\ \mathbf{S}_1 \ \square\ \mathbf{B}_2 \to \{\mathbf{B}_2\};\ \mathbf{S}_2\ \textsf{\textbf{fi}}$$

Recall that an assertion in WSL is a statement, not an annotation of the program. Any transformation which introduces an assertion is therefore guaranteeing that the condition in the assertion is always true at the point where the assertion is added.

Transformation 3 *Append assertion*
If the variables in $\mathbf{x}$ do not appear free in $\mathbf{Q}$ (i.e. the new value of $\mathbf{x}$ does not depend on the old value) then:

$$\mathbf{x} := \mathbf{x}'.\mathbf{Q} \approx \mathbf{x} := \mathbf{x}'.\mathbf{Q};\ \{\mathbf{Q}[\mathbf{x}/\mathbf{x}']\}$$

where $\mathbf{Q}[\mathbf{x}/\mathbf{x}']$ is the formula $\mathbf{Q}$ with every occurrence of a variable in $\mathbf{x}'$ replaced by the corresponding variable in $\mathbf{x}$. In particular, if x does not appear in the expression e, then:

$$x := e \;\approx\; x := e;\ \{x = e\}$$

Transformation 4 *Assignment Merging*
For any variable x and expressions e_1 and e_2:

$$x := e_1;\ x := e_2 \;\approx\; x := e_2[e_1/x]$$

Transformation 5 *Recursive Implementation*
Suppose we have a statement $\mathbf{S}'$ which we wish to transform into the recursive procedure $(\mu X.\mathbf{S})$. We claim that this is possible whenever:

1. The statement $\mathbf{S}'$ is refined by $\mathbf{S}[\mathbf{S}'/X]$ (which denotes $\mathbf{S}$ with all occurrences of X 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 $\mathbf{S}'$;

2. We can find an expression $\mathbf{t}$ (called the *variant function*) whose value is reduced before each occurrence of $\mathbf{S}'$ in $\mathbf{S}[\mathbf{S}'/X]$.

Note that a *refinement* of a program is any program which is *more defined* (i.e. defined on at least as many initial states as the original) and *more deterministic* (i.e. for each initial state on which the original program is defined, the refinement is also defined and has a smaller set of final states: therefore each of the final states for the refinement is an allowed final state for the original). If two programs are each a refinement of the other, then the programs are semantically equivalent.

The expression $\mathbf{t}$ need not be integer valued: any set Γ which has a well-founded order $\preccurlyeq$ is suitable. 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 $\mathbf{S}[\mathbf{S}'/X]$. The theorem combines these two requirements into a single condition:

Theorem 3.1 *The Recursive Implementation Theorem*
If $\preccurlyeq$ is a well-founded partial order on some set Γ and $\mathbf{t}$ is a term giving values in Γ and t_0 is a variable which does not occur in $\mathbf{S}$ then if

$$\{\mathbf{P} \wedge \mathbf{t} \preccurlyeq t_0\};\ \mathbf{S}' \;\leq\; \mathbf{S}[\{\mathbf{P} \wedge \mathbf{t} \prec t_0\};\ \mathbf{S}'/X]$$

then $\{\mathbf{P}\};\ \mathbf{S}' \;\leq\; (\mu X.\mathbf{S})$

See [War89] Chapter One for the proof.

Transformation 6 *Action call fold/unfold*
If $\mathbf{S}$ is any statement in an action system, one of whose actions is $A_i \;\equiv\; \mathbf{S}_i.$, then:

$$\mathbf{S} \;\approx\; \mathbf{S}[\mathbf{S}_i/\textsf{\textbf{call}}\ A_i]$$

This transformation can be used in either direction: to replace an action call by a copy of the action body, or to replace a statement by a call to a suitable action body. In particular, let $\mathbf{S}$ be any statement in an action system (which is in a suitable position for a **call** to appear). Let A_n be any new action name which is not already used in the system. Then we can add a new action $A_n \;\equiv\; \mathbf{S}.$ to the system: this transformation is trivial since the new action is unreachable. Now apply Transformation 6 to replace $\mathbf{S}$ by **call** A_n.

See [War89] Chapter Four for the proof.

Transformation 7 *Action Recursion Removal*
Let $A_i \;\equiv\; \mathbf{S}_i.$ be any action in a regular action system. Then we can remove "recursive" calls **call** A_i which appear in the action $A_i \;\equiv\; \mathbf{S}_i.$ by the following process:

1. First enclose $\mathbf{S}_i$ in a double-nested loop: **do do** $\mathbf{S}_i$; **exit**(2) **od od**
2. Next, replace each **call** A_i which appears in n nested loops by the statement **exit**$(n+1)$: so a **call** in $\mathbf{S}_i$ which is not in any loops is replaced by **exit**(1), and so on.

Each call is therefore replaced by an **exit** which terminates the inner loop surrounding $\mathbf{S}_i$ and re-iterates the outer loop, so that $\mathbf{S}_i$ is executed again as required.

See [War89] Chapter Four for the proof.

The double loop is not always necessary: if all the recursive calls are in tail positions, then a single loop is sufficient with the calls replaced by **skip** statements.

Transformation 8 *Recursion Removal*
Suppose we have a recursive procedure whose body is a regular action system in the following form:

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

where $\mathbf{S}_{j1}, \ldots, \mathbf{S}_{jn_j}$ 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 $\mathbf{S}_{j0}$, $\mathbf{S}_{j1}$, …,$\mathbf{S}_{jn_j-1}$ contain no action calls. Note that, since the action system is regular, each of the statements $\mathbf{S}_{jn_j}$ must contain one or more action calls: in fact, they can only terminate by calling an action. There are $M+N$ actions in total: A_1,…, A_M, B_1, …, B_N. Note that the since the action system is regular, it can only be terminated by executing **call** Z which will terminate the current invocation of the procedure.

Our aim is to remove the recursion by introducing a local stack L which records "postponed" operations: When a recursive call to $F(e)$ is required we "postpone" it by pushing the pair $\langle 0, e\rangle$ onto L (where e is the parameter required for the recursive call). Execution of the statements $\mathbf{S}_{jk}$ also has to be postponed (since they occur between recursive calls), we record the postponement of $\mathbf{S}_{jk}$ and the current value of x, by pushing $\langle\langle j, k\rangle, x\rangle$ onto L. Where the procedure body would normally terminate (by calling Z) we instead call a new action $\hat{F}$ which pops the top item off L and carries out the postponed operation. If we call $\hat{F}$ with the stack empty then all postponed operations have been completed and the procedure terminates by calling Z.

A recursive procedure in the form given above is equivalent to the following iterative procedure which uses a new local stack L and a new local variable m:

```
proc $F'(x)$ $\equiv$
  var $\langle L := \langle\rangle, m := 0\rangle$:
    actions $A_1$:
    $A_1$ $\equiv$ $\mathbf{S}_1$[call $\hat{F}$/call $Z$].
    ...$A_i$ $\equiv$ $\mathbf{S}_i$[call $\hat{F}$/call $Z$].
    ...$B_j$ $\equiv$ $\mathbf{S}_{j0}$;
                 $L := \langle\langle 0, g_{j1}(x)\rangle, \langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle,$
                      $\ldots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L$;
                 call $\hat{F}$.
    ...$\hat{F}$ $\equiv$ if $L = \langle\rangle$
                   then call $Z$
                   else $\langle m, x\rangle \xleftarrow{\text{pop}} L$;
                        if $m = 0 \rightarrow$ call $A_1$
```

```
        $\square \dots \square\ m = \langle j, k\rangle$
          $\rightarrow \mathbf{S}_{jk}[$call $\hat{F}/$call $Z]$; call $\hat{F}$
        ... fi fi. endactions end.
```

where the substitutions $\mathbf{S}_i[$**call** $\hat{F}/$**call** $Z]$ are, of course, not applied to nested action systems which are components of the $\mathbf{S}_i$.

See [War99a] for the proof, and some applications.

Within each B_j, at the point where $\hat{F}$ is called the top of the stack L is $\langle 0, g_{j1}(x)\rangle$. So this $\hat{F}$ will set x to $g_{j1}(x)$ and call A_j. We can therefore unfold $\hat{F}$ into B_j and simplify to get:

```
$B_j\ \equiv \mathbf{S}_{j0}$;
    $L := \langle\langle\langle j, 1\rangle, x\rangle, \langle 0, g_{j2}(x)\rangle,$
            $\dots, \langle 0, g_{jn_j}(x)\rangle, \langle\langle j, n_j\rangle, x\rangle\rangle \mathbin{+\!\!+} L$;
    $x := g_{j1}$;
    call $A_j$.
```

If $n_j = 1$ for all j, then a value of the form $\langle 0, v\rangle$ will never be pushed onto the stack, and each B_j takes this form:

```
$B_j\ \equiv \mathbf{S}_{j0}$;
    $L := \langle\langle\langle j, 1\rangle, x\rangle\rangle \mathbin{+\!\!+} L$;
    $x := g_{j1}$;
    call $A_j$.
```

If, in addition, the procedure is parameterless and there is only one B type action, then the *only* value pushed into the stack is $\langle\langle 1, 1\rangle\rangle$. So the stack can be replaced by a simple integer (which records how many values were pushed onto the stack). So, for the special case of a parameterless, linear recursion we have:

```
proc $F\ \equiv$
  actions $A_1$:
  $A_1\ \equiv \mathbf{S}_1$.
  $\dots A_i\ \equiv \mathbf{S}_i$.
  $B_1\ \equiv \mathbf{S}_0$; $F$; $\mathbf{S}_{11}$. endactions.
```

is equivalent to:

```
proc $F'\ \equiv$
  var $\langle L := 0\rangle$:
    actions $A_1$:
    $A_1\ \equiv \mathbf{S}_1[$call $\hat{F}/$call $Z]$.
    $\dots A_i\ \equiv \mathbf{S}_i[$call $\hat{F}/$call $Z]$.
    $\dots B_1\ \equiv \mathbf{S}_{j0}$; $L := L + 1$; call $A_1$.
    $\dots \hat{F}\ \equiv$ if $L = 0$
                  then call $Z$
                  else $L := L - 1$;
                       $\mathbf{S}_{11}[$call $\hat{F}/$call $Z]$; call $\hat{F}$ fi.
    endactions end.
```

For example:

```
proc $F\ \equiv$
  if B then $\mathbf{S}_1$ else $\mathbf{S}_2$; $F$; $\mathbf{S}_3$ fi.
```

is equivalent to the iterative program:

```
proc $F' \equiv$
  var $\langle L := 0 \rangle$:
    actions $A_1$:
    $A_1 \equiv$ if B then $\mathbf{S}_1$; call $\hat{F}$ else call $B_1$ fi.
    $B_1 \equiv \mathbf{S}_2$; $L := L + 1$; call $A_1$.
    $\hat{F} \equiv$ if $L = 0$
              then call $Z$
               else $L := L - 1$;
                     $\mathbf{S}_3$; call $\hat{F}$ fi. endactions end.
```

Remove the recursion in $\hat{F}$, unfold into A_1, unfold B_1 into A_1 and remove the recursion to give:

```
proc $F' \equiv$
  var $\langle L := 0 \rangle$:
    while $\neg$B do $\mathbf{S}_2$; $L := L + 1$ od;
    $\mathbf{S}_1$;
    while $L \neq 0$ do $L := L - 1$; $\mathbf{S}_3$ od.
```

This restructuring is carried out automatically by FermaT's Collapse_Action_System transformation.

Transformation 9 *Loop Unrolling*

$$\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od} \approx \textbf{if } \mathbf{B} \textbf{ then } \mathbf{S};\ \textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od fi}$$

Selective unrolling of **while** loops. For any formula **Q** we have:

$$\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od} \approx \textbf{while } \mathbf{B} \textbf{ do } \mathbf{S};\ \textbf{if } \mathbf{B} \wedge \mathbf{Q} \textbf{ then } \mathbf{S} \textbf{ fi od}$$

See [War89] Chapter Two for the proof.

Transformation 10 *Entire Loop Unrolling.* If $\mathbf{B}_1 \Rightarrow \mathbf{B}_2$ then:

$$\textbf{while } \mathbf{B}_2 \textbf{ do } \mathbf{S} \textbf{ od} \approx \textbf{while } \mathbf{B}_2 \textbf{ do } \mathbf{S};\ \textbf{if } \mathbf{Q} \textbf{ then while } \mathbf{B}_1 \textbf{ do } \mathbf{S} \textbf{ od fi od}$$

This is true for *any* formula **Q**. See [War89] Chapter Two for the proof.

Each of these transformations has a generalisation in which instead of inserting the "unrolled" part after **S** it is copied after an arbitrary selection of the terminal statements in **S**.

Transformation 11 *Loop Merging*

$$\textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od} \ \approx\ \textbf{while } \mathbf{B} \wedge \mathbf{Q} \textbf{ do } \mathbf{S} \textbf{ od};\ \textbf{while } \mathbf{B} \textbf{ do } \mathbf{S} \textbf{ od}$$

This transformation is valid for *any* **while** loop and *any* condition **Q**.
See [War89] Chapter Two for the proof.

An equivalent definition of the transformation is: for any statement **S** and formulae $\mathbf{B}_1$ and $\mathbf{B}_2$ such that $\mathbf{B}_1 \Rightarrow \mathbf{B}_2$:

$$\textbf{while } \mathbf{B}_1 \textbf{ do } \mathbf{S} \textbf{ od};\ \textbf{while } \mathbf{B}_2 \textbf{ do } \mathbf{S} \textbf{ od} \ \approx\ \textbf{while } \mathbf{B}_2 \textbf{ do } \mathbf{S} \textbf{ od}$$

Transformation 12 *Abstracting from a Program to a Specification*
Let $\mathbf{S} : V \to W$, be any WSL statement and let $\mathbf{x}$ be a list of all the variables in W. Then **S** is equivalent to:

$$\mathbf{x} := \mathbf{x}'.(\neg \mathrm{WP}(\mathbf{S}, \mathbf{x} \neq \mathbf{x}') \wedge \mathrm{WP}(\mathbf{S}, \textbf{true}))$$

where for any program $\mathbf{S}$ and formula $\mathbf{Q}$, the formula $\mathrm{WP}(\mathbf{S}, \mathbf{Q})$ is the *weakest precondition* of $\mathbf{S}$ on postcondition $\mathbf{Q}$. This is the weakest condition on the initial state such that if $\mathbf{S}$ is started in a state which satisfies $\mathrm{WP}(\mathbf{S}, \mathbf{Q})$ then it is guaranteed to terminate and the final state is guaranteed to satisfy $\mathbf{Q}$. If $\mathbf{S}$ has any loops or recursion, then $\mathrm{WP}(\mathbf{S}, \mathbf{Q})$ is defined as an infinite formula, so the specification statement will usually be infinitely large. But the transformation is still very useful during the "elaboration" and "divide and conquer" stages in the development process, and also for analysing fragments of larger programs. See [War04] for the proof of this transformation.

In theory, this transformation solves all reverse engineering problems: since it defines an abstract specification for any program [War04]. In practice, it is less useful because the specifications for programs containing loops or recursion are infinite. But there are many programs and program fragments which do not contain loops or recursion: in one study [WZL08] over 40% of the modules from a collection taken at random from several large assembler systems contained no loops.

Transformation 13 *Refining a Specification*
Generally, programmers find that a compound statement with assertions, **if** statements and simple assignments to be easier to read and understand than the equivalent single specification statement. So we have implemented another transformation Refine_Spec which analyses a specification statement and carries out the following operations:

1. Factor out any assertions: for example, if no variable in $\mathbf{x}'$ appears free in $\mathbf{P}$, then:

$$\mathbf{x} := \mathbf{x}'.(\mathbf{Q} \wedge \mathbf{P}) \approx \{\mathbf{P}\};\ \mathbf{x} := \mathbf{x}'.\mathbf{Q}$$

   conversely, if no variable in $\mathbf{x}$ appears free in $\mathbf{P}$ then:

$$\mathbf{x} := \mathbf{x}'.(\mathbf{Q} \wedge \mathbf{P}) \approx \mathbf{x} := \mathbf{x}'.\mathbf{Q};\ \{\mathbf{P}[\mathbf{x}/\mathbf{x}']\}$$

2. Expand into an **if** statement: for example, the specification $\mathbf{x} := \mathbf{x}'.(\mathbf{Q} \vee (\mathbf{B} \wedge \mathbf{P}))$ where $\mathbf{B}$ does not contain any variables $\mathbf{x}'$, is equivalent to

   if $\mathbf{B}$ **then** $\mathbf{x} := \mathbf{x}'.(\mathbf{Q}' \vee \mathbf{P}')$ **else** $\mathbf{x} := \mathbf{x}'.(\mathbf{Q}'')$ **fi**

   where $\mathbf{Q}'$ and $\mathbf{P}'$ are the result of simplifying $\mathbf{Q}$ and $\mathbf{P}$ under the assumption that $\mathbf{B}$ is true, and $\mathbf{Q}''$ is the result of simplifying $\mathbf{Q}$ under the assumption that $\mathbf{B}$ is false. These sub-specifications are then recursively refined;

3. Finally, any simple assignments or parallel assignments are extracted.

These transformations are proved in [War89] Chapter One.

4 Examples of Transformational Programming

A simple example of an algorithm derivation will demonstrate how the various transformations introduced in the previous section "fit together" to provide a complete derivation path from abstract specification to efficient implementation. This example also shows that different informal ideas can drive the derivation process in different directions: resulting in a substantially different final implementation.

4.1 Greatest Common Divisor

The Greatest Common Divisor (GCD) of two numbers is the largest number which divides both of the numbers with no remainder. A specification for a program which computes the GCD is the following:

$$r := \mathsf{GCD}(x, y)$$

where:

$$\mathsf{GCD}(x, y) = \max \left\{ n \in \mathbb{N} \,\middle|\, n|x \,\land\, n|y \right\}$$

and $n|x$ means "n divides x". (Note that $\mathsf{GCD}(x, y)$ is undefined when both x and y are zero).

It is easy to prove the following facts about GCD:

1. $\mathsf{GCD}(0, y) = y$
2. $\mathsf{GCD}(x, 0) = x$
3. $\mathsf{GCD}(x, y) = \mathsf{GCD}(y, x)$
4. $\mathsf{GCD}(x, y) = \mathsf{GCD}(-x, y) = \mathsf{GCD}(x, -y)$
5. $\mathsf{GCD}(x, y) = \mathsf{GCD}(x - y, y) = \mathsf{GCD}(x, y - x)$ etc.

4.1.1 Initial Program Derivation

To refine our specification into a program, the obvious first step is to split a tautology on the conditions $x = 0$ and $y = 0$, using Fact (1) and Fact (2) respectively:

```
r := GCD(x, y)  ≈  if x = 0 then r := y
                   elsif y = 0
                         then r := x
                          else r := GCD(x, y) fi
```

Fact (3) does not appear to make much progress. If we restrict attention to non-negative integers, then Fact (4) does not apply. So we are left with Fact (5). If we take as our variant function the expression $x + y$, then, since we are restricted to non-negative integers, we can only transform $r := \mathsf{GCD}(x, y)$ to $r := \mathsf{GCD}(x - y, y)$ under the condition $x \geqslant y$. Similarly, for the condition $y \geqslant x$ we can transform $r := \mathsf{GCD}(x, y)$ to $r := \mathsf{GCD}(x, y - x)$. So we have the following elaboration of the specification:

```
if x = 0
  then r := y
elsif y = 0
      then r := x
elsif x ⩾ y then r := GCD(x − y, y)
             else r := GCD(x, y − x) fi
```

If $x \geqslant y$ then $x - y + y < x + y$ (since x and y are positive at this point) and similarly, if $x < y$ then $x + y - x < x + y$, so our variant function is reduced before each copy of the specification in the elaborated program. Applying Recursive Implementation (Transformation 5), we get the following recursive program:

```
proc gcd(x, y) ≡
  if x = 0
    then r := y
  elsif y = 0
        then r := x
  elsif x ⩾ y then r := gcd(x − y, y)
               else r := gcd(x, y − x) fi end
```

This is a simple tail-recursion, so recursion removal (Transformation 8) produces this iterative program:

```
proc gcd($x, y$) $\equiv$
  while $x \neq 0 \wedge y \neq 0$ do
        if $x \geqslant y$ then $x := x - y$
                else $y := y - x$ fi od;
  if $x = 0$ then $r := y$ else $r := x$ fi end
```

This is essentially the same algorithm as Dijkstra produces by the method of invariants: except that we have not needed to prove any invariants.

4.1.2 Optimisation Transformations

There is a problem with the algorithm in that, although it is correct, it is very inefficient when x and y are very different in size. For example, if $x = 1$ and $y = 2^{31}$ then the program will take $2^{31} - 1$ steps.

One solution would be to look for some other properties of GCD and generate a new, hopefully more efficient, program which uses these properties. This requires throwing away the current program: not a big issue in this case, since the program is so small. But in the case of a very large program, the suggestion that we throw it away and start again from scratch in order to solve a small efficiency problem is unlikely to be well received! Unfortunately, this is the *only* option offered by the "Invariant Based Programming" approach.

With the transformational programming approach, we have another option: attempt to transform the program in order to improve its efficiency. Consider the case where x is much larger than y. Then the statement $x := x - y$ will be executed many times in succession. This suggests that we apply Entire Loop Unrolling (Transformation 10) to the program at the point just after this assignment with the condition $x \geqslant y$. The result is:

```
proc gcd($x, y$) $\equiv$
  while $x \neq 0 \wedge y \neq 0$ do
        if $x \geqslant y$ then $x := x - y$;
                     while $x \geqslant y$ do
                           if $x \geqslant y$
                             then $x := x - y$ fi od
                else $y := y - x$ fi od;
  $r := x$ end
```

This simplifies to:

```
proc gcd($x, y$) $\equiv$
  while $x \neq 0 \wedge y \neq 0$ do
        if $x \geqslant y$ then while $x \geqslant y$ do $x := x - y$ od
                else $y := y - x$ fi od;
  if $x = 0$ then $r := y$ else $r := x$ fi end
```

Consider the inner **while** loop. This repeatedly subtracts y from x. Suppose the loop iterates q times (we know $q > 0$ since $x \geqslant y$ initially). Then the final value of x is $x = x_0 - q.y$ where x_0 was the initial value of x. We also have $0 \leqslant x < y$. These two facts show that the final value of x is in fact the remainder obtained when x is divided by y. In other words:

$$\textbf{while } x \geqslant y \textbf{ do } x := x - y \textbf{ od} \approx x := x \bmod y$$

when $x \geqslant y$ initially.

Similarly, entire loop unrolling can be applied after the assignment $y := y - x$ and the same optimisation applied to give:

```
proc gcd(x, y) ≡
  while $x \neq 0 \wedge y \neq 0$ do
        if $x \geqslant y$ then $x := x \bmod y$
                 else $y := y \bmod x$ fi od;
  if $x = 0$ then $r := y$ else $r := x$ fi end
```

We have transformed a program which was $O(n)$ in execution time into an equivalent program which is $O(\log n)$

4.1.3 Alternate Program Derivation

With a different informal idea and/or different constraints on the algorithm, the same derivation process will often produce a different result. For example, suppose the target machine does not have a native integer division instruction, but does have efficient binary shift instructions. Our informal idea is to make use of the following additional information about the GCD function:

1. $\mathsf{GCD}(x, y) = 2.\mathsf{GCD}(x/2, y/2)$ when x and y are both even;
2. $\mathsf{GCD}(x, y) = \mathsf{GCD}(x/2, y)$ when x is even and y is odd;
3. $\mathsf{GCD}(x, y) = \mathsf{GCD}(x, y/2)$ when x is odd and y is even;
4. $\mathsf{GCD}(x, y) = \mathsf{GCD}((x - y)/2, y)$ when x and y are odd and $x \geqslant y$;
5. $\mathsf{GCD}(x, y) = \mathsf{GCD}(x, (y - x)/2)$ when x and y are odd and $y \geqslant x$.

Applying Fact (1) above produces the following elaborated specification:

```
if $x = 0$ then $r := y$
elsif $y = 0$ then $r := x$
elsif $2|x \wedge 2|y$
     then $r := 2.\mathsf{GCD}(x/2, y/2)$
      else $r := \mathsf{GCD}(x, y)$ fi
```

Applying Recursive Implementation (Transformation 5) plus Recursion Removal (Transformation 8) to the first occurrence only of GCD produces:

```
if $x = 0$ then $r := y$
elsif $y = 0$ then $r := x$
        else var $\langle L := 0 \rangle$ :
              while $2|x \wedge 2|y$ do
                    $L := L + 1$;
                    $x := x/2$; $y := y/2$ od;
              $r := \mathsf{GCD}(x, y)$;
              $r := 2^L.r$ end fi
```

Applying Fact (2) above, followed by Recursive Implementation (Transformation 5) and Recursion Removal (Transformation 8) produces the following result:

```
if $x = 0$ then $r := y$
elsif $y = 0$ then $r := x$
        else var $\langle L := 0 \rangle$ :
              while $2|x \wedge 2|y$ do
                    $L := L + 1$;
                    $x := x/2$; $y := y/2$ od;
              while $2|x$ do $x := x/2$ od;
              $\{x \neq 0 \wedge y \neq 0 \wedge \neg 2|x\}$;
              $r := \mathsf{GCD}(x, y)$;
              $r := 2^L.r$ end fi
```

We now focus attention on the case where x is known to be odd, and y is non-zero. Define:

$$\mathsf{GCDx}(x, y) \ =_{\text{DF}} \ \{y \neq 0 \ \wedge \ \neg 2|x\}; \ r := \mathsf{GCD}(x, y)$$

By Fact (3) we show that $\mathsf{GCDx}(x, y)$ is equivalent to:

```
while $2|y$ do $y := y/2$ od;
$\mathsf{GCDx}(x, y)$
```

Now apply Fact (4), and also Fact (3) from the first set of facts, to ensure that x is odd in every occurrence of GCDx:

```
while $2|y$ do $y := y/2$ od;
if $x = y$ then $r := x$
elsif $x > y$ then $\mathsf{GCDx}(y, (x - y)/2)$
              else $\mathsf{GCDx}(x, (y - x)/2)$ fi
```

Apply Recursive Implementation (Transformation 5) and Recursion Removal (Transformation 8):

```
do while $2|y$ do $y := y/2$ od;
   if $x = y$ then $r := x$; exit fi;
   if $x > y$
     then $\langle x, y\rangle := \langle y, x - y\rangle$
      else $y := y - x$ fi;
   $y := y/2$ od
```

The final program is therefore:

```
if $x = 0$
  then $r := y$
elsif $y = 0$
     then $r := x$
      else var $\langle L := 0\rangle$ :
             while $2|x \ \wedge \ 2|y$ do
                   $L := L + 1$;
                   $x := x/2$; $y := y/2$ od;
             while $2|x$ do $x := x/2$ od;
             do while $2|y$ do $y := y/2$ od;
                 if $x = y$ then $r := x$; exit fi;
                 if $x > y$
                   then $\langle x, y\rangle := \langle y, x - y\rangle$
                    else $y := y - x$ fi;
                 $y := y/2$ od
             $r := 2^L.r$ end fi
```

If the CPU has an instruction which computes the ntz function (returning the number of terminating zeros in the binary representation of the argument), and also efficient instructions for shifting binary numbers, then three **while** loops can be implemented in straight-line code. For example, the first **while** loop can be implemented as:

```
$L := \mathsf{min}(\mathsf{ntz}(x), \mathsf{ntz}(y))$;
$x := x/2^L$; $y := y/2^L$;
```

An efficient method of computing the ntz function using de Bruijn sequences, was described in [LPR98].

5 Knuth's Polynomial Addition Algorithm

In the introduction to Chapter Two of "Fundamental Algorithms" [Knu68] Knuth writes "Although List processing systems are useful in a large number of situations, they impose constraints on the programmer that are often unnecessary; it is usually better to use the methods of this chapter directly in one's own programs, tailoring the data format and the processing algorithms to the particular application. ... We will see that there is nothing magic, mysterious, or difficult about the methods for dealing with complex structures; these techniques are an important part of every programmer's repertoire, and he can use them easily whether he is writing a program in assembly language or in a compiler language like FORTRAN or ALGOL."

He goes on to describe a data structure to implement multivariate polynomials using a four-way circular-linked list structure. Using this data structure he presents an algorithm for polynomial addition: given two polynomials, represented by pointers P and Q, which do not share any nodes, the algorithm adds polynomial P to polynomial Q, updating the structure for Q while leaving P unchanged. Despite Knuth's confident assertion that "there is nothing... difficult about the methods", the four-way linked list structure he used to implement polynomial addition turned out to be rather difficult to work with in practice. The algorithm is very complex and difficult to get right: there were at least three bugs in the version published in the first edition [Knu74].

We will use Knuth's polynomial addition problem as a testbed for the transformational programming method applied to the derivation of complex linked-list algorithms. We will not make any use of Knuth's code (except to compare its efficiency with our generated code): instead we show that simply following the derivation method leads to highly efficient code. Our derived algorithm turns out to be over twice as fast as Knuth's in all test cases: and nearly four times faster in some cases.

A polynomial is either a constant or has the form:

$$\sum_{0 \leqslant j \leqslant n} g_j x^{e_j}$$

where x is a variable, $n > 0$, $0 = e_0 < e_1 < \cdots < e_j$ and g_j are polynomials involving only variables alphabetically less than x. Also, $g_1, \dots, g_n$ are not zero.

5.1 Abstract Polynomials

For our first implementation of polynomial addition, we decided to suffer the constraints imposed by a list processing system by developing a WSL algorithm for polynomial addition which uses WSL sequences to implement polynomials. The WSL is compiled into Scheme code which in turn is compiled into C using the Hobbit Scheme compiler [Tam95].

A constant polynomial is represented as the singleton list $\langle c \rangle$ where c is an integer. Otherwise, a polynomial is represented as the list of three or more elements:

$$p = \langle x, \langle g_0, e_0 \rangle, \langle g_1, e_1 \rangle, \dots, \langle g_n, e_n \rangle \rangle$$

Here, x is a string (the variable name), $n > 0$, $0 = e_0 < e_1 < \cdots < e_n$ are the integer exponents and g_i are lists representing polynomials whose variables are all lexicographically less than x. Also, $g_j \neq \langle 0 \rangle$ for all $j > 0$. The abstraction function $\mathsf{abs}(p)$ returns the polynomial represented by the list p:

$$\mathsf{abs}(p) =_{\text{DF}} \begin{cases} p[1] & \text{if } \ell(p) = 1 \\ \sum_{0 \leqslant j \leqslant \ell(p)-2} \mathsf{abs}(p[j+2][1]).p[1]^{p[j+2][2]} & \text{otherwise} \end{cases}$$

So, for example, for the list $p = \langle x, \langle \langle 0 \rangle, 0 \rangle, \langle \langle 4 \rangle, 1 \rangle, \langle \langle 1 \rangle, 2 \rangle \rangle$:

$$\mathsf{abs}(p) = 0.x^0 + 4.x^1 + 1.x^2$$

i.e.:

$$\mathsf{abs}(p) = x^2 + 4.x$$

We call polynomials implemented as lists "abstract polynomials" and polynomials implemented as pointers "concrete polynomials". The lists representing abstract polynomials have to satisfy the condition $I(p)$ where:

$$\begin{aligned} I(p) =_{\mathrm{DF}} & (\ell(p) = 1 \ \wedge\ p[1] \in \mathbb{Z}) \\ & \vee \big(\ell(p) \geqslant 3 \ \wedge\ p[1] \in \mathsf{Strings} \ \wedge\ \forall v \in \mathsf{vars}(p).\,(p[1] > v) \\ & \quad \wedge\ \ell(p[2]) = 2 \ \wedge\ I(p[2][1]) \ \wedge\ p[2][2] = 0 \\ & \quad \wedge\ \forall j : 3 \leqslant j \leqslant \ell(p).\,(\ell(p[j]) = 2 \ \wedge\ I(p[j][1]) \ \wedge\ p[j][2] \in \mathbb{N} \\ & \qquad\qquad \wedge\ p[j][2] > p[j-1][2] \ \wedge\ p[j][1] \neq \langle 0 \rangle)\big) \end{aligned}$$

where $\mathsf{Strings}$ is the set of strings, and $\mathsf{vars}(p)$ is the set of variables used in the terms of polynomial p (i.e. all variables except the base variable):

$$\mathsf{vars}(p) =_{\mathrm{DF}} \begin{cases} \varnothing & \text{if } \ell(p) = 1 \\ \bigcup_{2 \leqslant j \leqslant \ell(p)} \mathsf{allvars}(p[j][1]) & \text{otherwise} \end{cases}$$

where:

$$\mathsf{allvars}(p) =_{\mathrm{DF}} \begin{cases} \varnothing & \text{if } \ell(p) = 1 \\ \{p[1]\} \cup \bigcup_{2 \leqslant j \leqslant \ell(p)} \mathsf{allvars}(p[j][1]) & \text{otherwise} \end{cases}$$

With these definitions, the formal specification for a program which assigns r to the value of a list which represents a polynomial equal to $\mathsf{abs}(p) + \mathsf{abs}(q)$ is simply:

$$\mathsf{add}(r, p, q) =_{\mathrm{DF}} r := r'.(I(r') \ \wedge\ \mathsf{abs}(r') = \mathsf{abs}(p) + \mathsf{abs}(q))$$

5.2 Abstract Algorithm Derivation

With this data structure it is a simple task to derive a suitable recursive algorithm for adding two polynomials. First, take out special cases: the obvious cases are when p and/or q are constant polynomials (in which case, they are sequences of length 1):

```
add(r, p, q)  ≈  if $\ell(p) = 1$ then if $\ell(q) = 1$ then $r := \langle p[1] + q[1] \rangle$
                                               else add(r, p, q[2][1]); $r := \langle q[1], \langle r, 0 \rangle \rangle +\!\!+\ q[3\,..]$ fi
                 elsif $\ell(q) = 1$ then add(r, p[2][1], q); $r := \langle q[1], \langle r, 0 \rangle \rangle +\!\!+\ p[3\,..]$
                          else add(r, p, q) fi
```

Next, if p and q are polynomials in the same variable, then we add corresponding terms in the two sequences of terms. One slight complication is that if all the non-constant terms cancel, we must make sure that we return the constant term rather than returning a constant polynomial (this ensures that the condition $I(r')$ is satisfied). Similarly, when adding two lists of terms, we must take care not to generate a zero term when the exponent is non-zero.

After taking out all these cases, Recursive Implementation (Transformation 5) can be repeatedly applied to produce a set of mutually recursive procedures. These can be transformed into recursive functions using the definitional transformation for WSL functions:

```
funct Abs_Add_Poly(p, q) ≡
  (if $\ell(p) = 1$
     then if $\ell(q) = 1$
            then $\langle p[1] + q[1] \rangle$
             else $\langle q[1], \langle$Abs_Add_Poly(p, q[2][1]), 0$\rangle\rangle +\!\!+\ q[3\,..]$ fi
      else if $\ell(q) = 1$
            then $\langle p[1], \langle$Abs_Add_Poly(p[2][1], q), 0$\rangle\rangle +\!\!+\ p[3\,..]$
```

```
                    else if $p[1] = q[1]$
                            then Abs_Simplify($\langle p[1]\rangle$ ++ Abs_Add_Terms($p[2\,.\,.], q[2\,.\,.]$))
                             else if String_Less?($p[1], q[1]$)
                                     then $\langle q[1], \langle$Abs_Add_Poly($p, q[2][1]$)$, 0\rangle\rangle$ ++ $q[3\,.\,.]$
                                      else $\langle p[1], \langle$Abs_Add_Poly($q, p[2][1]$)$, 0\rangle\rangle$ ++ $p[3\,.\,.]$ fi fi fi fi).;
```

The function Abs_Add_Terms takes two lists of terms (which are assumed to be in the same variable) and returns the result of adding the terms together:

```
funct Abs_Add_Terms($p, q$) $\equiv$
  (if $p = \langle\rangle$
      then $q$
       else if $q = \langle\rangle$
               then $p$
                else if $p[1][2] = q[1][2]$
                        then Abs_Simplify_Term($\langle$Abs_Add_Poly($p[1][1], q[1][1]$)$, q[1][2]\rangle$)
                               ++ Abs_Add_Terms($p[2\,.\,.], q[2\,.\,.]$)
                         else if $p[1][2] < q[1][2]$
                                 then $\langle p[1]\rangle$ ++ Abs_Add_Terms($p[2\,.\,.], q$)
                                  else $\langle q[1]\rangle$ ++ Abs_Add_Terms($p, q[2\,.\,.]$) fi fi fi fi).;
```

The function Abs_Simplify simplifies a polynomial which has no terms apart from the constant term by returning the value in the constant term:

```
funct Abs_Simplify($p$) $\equiv$
  (if $\ell(p) = 2$
      then $p[2][1]$
       else $p$ fi).;
```

The function Abs_Simplify_Term returns an empty list if the term is zero, and otherwise returns a singleton list containing the term:

```
funct Abs_Simplify_Term(term) $\equiv$
  (if term$[2] > 0 \wedge$ term$[1] = \langle 0\rangle$
      then $\langle\rangle$
       else $\langle$term$\rangle$ fi).;
```

The code listed above was generated directly from the ASCII WSL code. After fixing a few typos, this code worked first time: this is in contrast to Knuth's polynomial addition algorithm which had at least three bugs in the published version [Knu74]. The abstract program was tested as follows:

1. First a few sample polynomials were used as test data and the results examined;
2. Then a small number of random polynomials were generated and used as test data and the results examined;
3. Finally a huge number of polynomials were generated and tested by evaluating the polynomials with random values for the variables. The sum of the values for the two input polynomials was compared with the value of the output polynomial.

5.3 Concrete Polynomials

Knuth's algorithm for polynomial addition represents a polynomial using a complex linked structure of *nodes* or records. Each record has six fields:

1. UP is a pointer to the parent node in the tree (or the null pointer value Λ if this node is the root of the whole polynomial);

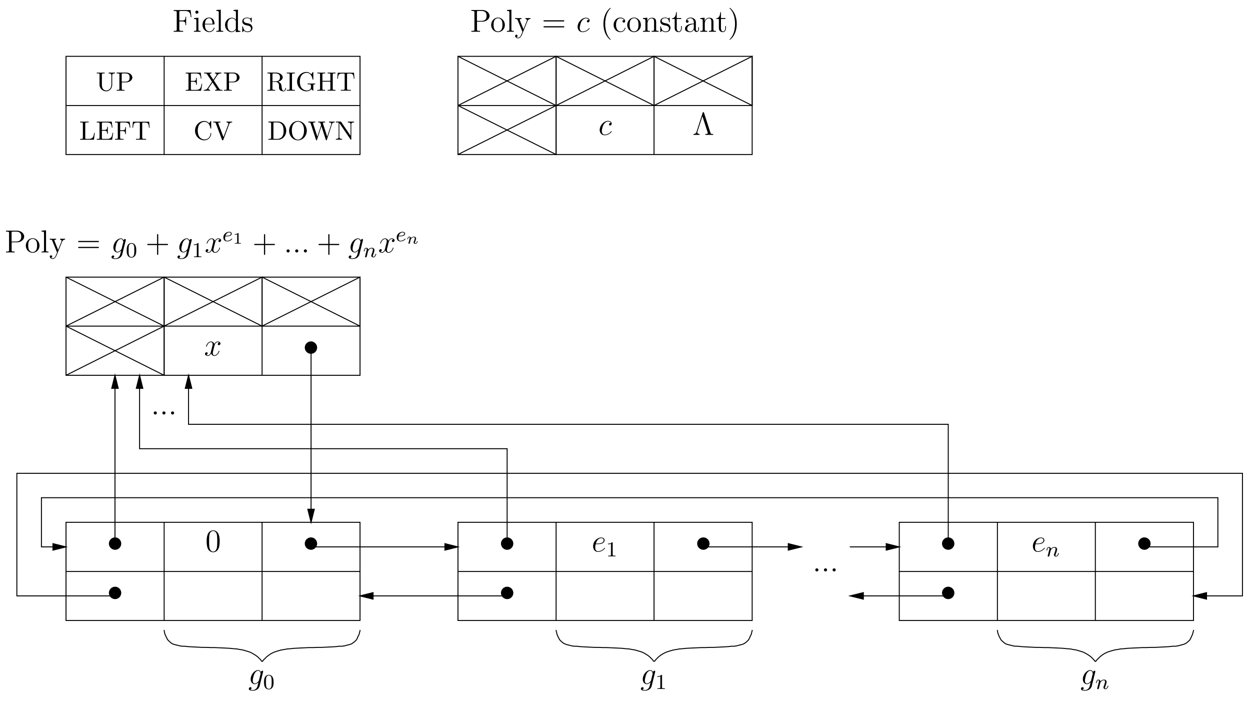

Figure 3: Representation of polynomials using four-directional links

2. DOWN is Λ for a constant polynomial, and otherwise a pointer to the first node in the list of terms for this polynomial: this will be the term with a zero exponent;

3. LEFT and RIGHT are the pointers which form a doubly-linked circular list of terms. The terms are linked in order of exponent, there is always a term with a zero exponent, and the term with the highest exponent links back to the zero exponent term via its RIGHT pointer. Each list contains at least two terms (otherwise we would have a constant polynomial: and this variable would be redundant);

4. CV is either the constant value for a constant polynomial (which has DOWN set to Λ) or the name of the variable for a non-constant polynomial (in which case DOWN points to the term with a zero exponent in the list of terms);

5. EXP is the exponent value for this term. Note that the variable which has this exponent is given by the CV value of the node pointed at by UP: the CV value for *this* node is the constant value or variable name for this term's coefficient.

Figure 3 illustrates how the data structure is used to represent both a constant and a non-constant polynomial. The crossed-out values are irrelevant.

We have implemented Knuth's algorithm in WSL, using a set of arrays U, D, L, R, C and E to implement the fields UP, DOWN, LEFT, RIGHT, CV and EXP respectively in the records, with pointers represented as indices in these arrays. The special value Λ represents a "null pointer", as in Knuth's code. For a constant polynomial (where $D[P] = \Lambda$) $C[P]$ contains the integer value and the other array values are irrelevant. Otherwise, $C[P]$ contains the variable name, and $D[P]$ points to the head of a circular-linked list of terms. The first term $D[P]$ has $E[D[P]] = 0$ and $C[D[P]]$ is the constant polynomial, $R[D[P]]$ points to the next term (which has an E value greater than zero), and so on. The L fields point back to the previous term in the list, creating a doubly-linked list, and the U fields point to the parent polynomial, also creating a doubly-linked list. The R field in the last term points to the first term, while the L field in the first term points to the last term, so the lists of terms are also circular-linked.

The abstraction function $\mathsf{Abs}(P)$ maps the concrete polynomial P (an index into the arrays) to the corresponding abstract polynomial and is defined as follows:

$$\mathsf{Abs}(P) \;=_{\text{DF}}\; \begin{cases} \langle C[P]\rangle & \text{if } D[P] = \Lambda \\ \langle C[P]\rangle \mathbin{+\!\!+} \mathsf{Abs\_Terms}(D[P]) & \text{otherwise} \end{cases}$$

where $\mathsf{Abs\_Terms}$ returns the appropriate list of abstract polynomial terms:

$$\mathsf{Abs\_Terms}(P) \;=_{\text{DF}}\; \begin{cases} \langle\langle C[P], E[P]\rangle\rangle & \text{if } E[R[P]] = 0 \\ \langle\langle C[P], E[P]\rangle\rangle \mathbin{+\!\!+} \mathsf{Abs\_Terms}(R[P]) & \text{otherwise} \end{cases}$$

We define an invariant $J(P)$ which captures the constraints on the array values for all nodes reachable from P by following D and L links: for example that the L and R links are the inverse of each other and form a doubly-linked circular list of terms, and that $U[D[P]] = P$ for each reachable node in which $D[P]$ is defined, and similarly $U[L[\dots L[D[P]]\dots]] = P$ and $U[R[\dots R[D[P]]\dots]] = P$ and so on. The definition is somewhat lengthy and not very informative, so has been omitted. Essentially, Figure 3 describes these constraints informally.

When $P \neq Q$, the conditions $I(\mathsf{Abs}(P))$, $I(\mathsf{Abs}(Q))$, $J(P)$ and $J(Q)$ together imply that the set of nodes reachable from P does not intersect with the set of nodes reachable from Q.

The WSL code for Knuth's algorithm, using these data structures, is as follows:

```
proc Knuth_Add($P, Q$) $\equiv$
  actions ADD :
  ADD $\equiv$ [Test type of poynomial.]
          if $D[P] = \Lambda$
            then while $D[Q] \neq \Lambda$ do $Q := D[Q]$ od; call A3
             else if $D[Q] = \Lambda \vee$ String_Less?($C[Q], C[P]$)
                    then call A2
                  elsif $C[Q] = C[P]$
                       then $\langle P := D[P], Q := D[Q]\rangle$; call ADD
                        else $Q := D[Q]$; call ADD fi fi.
  A2 $\equiv$ [Downward insertion.]
        $R_1 \overset{\text{pop}}{\longleftarrow}$ Avail;
        $S := D[Q]$;
        if $S \neq \Lambda$
          then do $U[S] := R_1$; $S := R[S]$;
                  if $E[S] = 0$ then exit(1) fi od fi;
        $\langle U[R_1] := Q, D[R_1] := D[Q], L[R_1] := R_1, R[R_1] := R_1, C[R_1] := C[Q], E[R_1] := 0\rangle$;
        $\langle C[Q] := C[P], D[Q] := R_1\rangle$;
        call ADD.
  A3 $\equiv$ [Match found.]
        $C[Q] := C[Q] + C[P]$;
        if $C[Q] = 0 \wedge E[Q] \neq 0$
          then call A8 fi;
        if $E[Q] = 0$ then call A7 fi;
        call A4.
  A4 $\equiv$ [Advance to left.]
        $P := L[P]$;
        if $E[P] = 0$ then call A6
                    else do $Q := L[Q]$;
                           if $E[Q] \leqslant E[P]$
                        then exit(1) fi od;
                    if $E[P] = E[Q]$ then call ADD fi fi;
        call A5.
A5  ≡ [Insert to right.]
        $R_1 \xleftarrow{\text{pop}}$ Avail;
        $\langle U[R_1] := U[Q], D[R_1] := \Lambda, L[R_1] := Q, R[R_1] := R[Q]\rangle$;
        $L[R[R_1]] := R_1$; $R[Q] := R_1$;
        $\langle E[R_1] := E[P], C[R_1] := 0\rangle$; $Q := R_1$;
        call ADD.
A6  ≡ [Return upward.]
        $P := U[P]$; call A7.
A7  ≡ [Move $Q$ up to right level.]
        if $U[P] = \Lambda$
          then call A11
          else while $C[U[Q]] \neq C[U[P]]$ do $Q := U[Q]$ od;
                 call A4 fi.
A8  ≡ [Delete zero term.]
        $R_1 := Q$; $Q := R[R_1]$; $S := L[R_1]$; $R[S] := Q$; $L[Q] := S$;
        Avail $\xleftarrow{\text{push}} R_1$;
        if $E[L[P]] = 0 \wedge Q = S$
          then call A9 else call A4 fi.
A9  ≡ [Delete constant polynomial.]
        $R_1 := Q$;
        $Q := U[Q]$;
        $\langle D[Q] := D[R_1], C[Q] := C[R_1]\rangle$;
        Avail $\xleftarrow{\text{push}} R_1$;
        $S := D[Q]$;
        if $S \neq \Lambda$ then do $U[S] := Q$; $S := R[S]$;
                         if $E[S] = 0$ then exit(1) fi od fi;
        call A10.
A10 ≡ [Zero detected?]
        if $D[Q] = \Lambda \wedge C[Q] = 0 \wedge E[Q] \neq 0$
          then $P := U[P]$; call A8 else call A6 fi.
A11 ≡ [Terminate.]
        while $U[Q] \neq \Lambda$ do $Q := U[Q]$ od; call $Z$. endactions.;
```

Here, Avail is the list of free nodes, $R_1 \xleftarrow{\text{pop}}$ Avail takes a free node from the list and returns its pointer in R_1, and Avail $\xleftarrow{\text{push}} R_1$ returns the node pointed at by R_1 to the free list.

Later, in [Knu74] Knuth wrote: "I also know of places where I have myself used a complicated structure with excessively unrestrained **goto** statements, especially the notorious Algorithm 2.3.3A for multivariate polynomial addition [Knu68]. The original program had at least three bugs; exercise 2.3.3–14, 'Give a formal proof (or disproof) of the validity of Algorithm A,' was therefore unexpectedly easy. Now, in the second edition, I believe that the revised algorithm is correct, but I still don't know any good way to prove it; I've had to raise the difficulty rating of exercise 2.3.3–14, and I hope someday to see the algorithm cleaned up without loss of its efficiency."

In a previous paper [War94] we proved the correctness of the above algorithm by transforming it into a suitable recursive procedure, applying our recursion removal theorem in reverse (in order to introduce recursion) and then changing the data representation from concrete polynomials to the corresponding abstract polynomials. So we were not surprised to find that the above program is indeed correct!

We were hopeful that the abstract algorithm would not be *too* inefficient, compared to Knuth's algorithm: in fact we suspected that the abstract algorithm might even be more efficient in some cases. Extensive testing with various sizes of random polynomials demonstrated that the abstract algorithm was in reality more efficient than Knuth's algorithm for *every* case tested!

With small polynomials (up to two variables and up to four terms at each level), the abstract algorithm was only 14% faster than Knuth's algorithm. As the polynomial size increased, the abstract algorithm became increasingly more efficient: with up to seven variables and up to four terms at each level the abstract algorithm was three times faster. For very large polynomials (up to seven variables and up to 20 terms at each level) the abstract algorithm is about 750 times faster! See Section 5.5 for detailed timing results.

The dramatic speed advantage of the abstract algorithm on large polynomials is due to the fact that the abstract algorithm can "share" data structures, or parts of data structures. Knuth's algorithm, although it apparently saves space by updating Q in place, cannot allow the result Q to share any nodes with polynomial P, due to the various "reverse pointers" (the "up" and "left" pointers in the linked lists). So, for example, adding a large polynomial P to a constant polynomial Q results in the creation of a complete copy of P: while the result for the abstract algorithm can share sub-structures with the parameters. Knuth's algorithm destroys the original polynomial in Q, while for the abstract algorithm, the original data structure is available via the original pointer. If it is not needed then any unshared nodes will be garbage-collected.

With regard to memory consumption: the abstract algorithm requires additional memory to store the recursion stack. In the worst case, the stack length is proportional to the size of the polynomial. However, each node in the abstract polynomial contains only two pointers, while the nodes in Knuth's polynomials contain four pointers. So the abstract algorithm will actually use less memory in most, if not all, cases: especially in cases where sub-trees can be shared. For example, in the abstract representation of the polynomial:

$$(x^2 + 2x + 3)z^2 + (x^2 + 2x + 3)z + 4$$

the pointers to the coefficients for z^2 and z could point to the same location: which contains a single copy of the representation of $x^2 + 2x + 3$.

To be fair to Knuth, he does not claim that his algorithm is the best: "No claim is being made here that the representation shown in Fig. 28 is the "best" for polynomials in several variables; ... Our main interest in Algorithm A is the way it typifies manipulations on trees with many links." [Knu68]. On the other hand, if this algorithm really "typifies manipulations on trees with many links", perhaps the conclusion to be drawn from these empirical results is that trees with many links (in particular: doubly-linked circular lists) should be avoided wherever possible! By using standard list structures in a language with garbage collection, many link manipulations can be avoided and components of larger data structures can be shared. This sharing of data structures would be especially valuable in programs which manipulate a large number of polynomials.

5.4 Algorithm Derivation

In the rest of this section we will apply the transformational programming method to derive an implementation of polynomial addition which uses Knuth's data structures. Our specification is therefore:

$$\begin{aligned}\mathsf{ADD}(P, Q) \ =_{\text{DF}} \ \langle U, D, L, R, E, C\rangle := \langle U', D', L', R', E', C'\rangle.&\\ \mathsf{Abs}'(P) = \mathsf{Abs}(P) \ \wedge \ \mathsf{abs}(\mathsf{Abs}'(Q)) = \mathsf{abs}(\mathsf{Abs}(P)) + \mathsf{abs}(\mathsf{Abs}(Q))&\\ \wedge \ I(\mathsf{Abs}(P)) \ \wedge \ I(\mathsf{Abs}(Q)) \ \wedge \ I(\mathsf{Abs}'(Q))&\\ \wedge \ J(P) \ \wedge \ J(Q) \ \wedge \ J'(Q)&\end{aligned}$$

where Abs' is the analogue of the Abs function defined on U', D', ... etc., and J' is the corresponding analogue of J.

With this specification, we start the derivation process in the usual way by taking out the special cases. If $D[P] = \Lambda$ then $\mathsf{Abs}(P)$ is a constant polynomial:

$$\mathsf{ADD}(P,Q) \;\approx\; \textbf{if } D[P] = \Lambda \textbf{ then } \mathsf{ADD\_CONST}(C[P],Q) \textbf{ else } \mathsf{ADD}(P,Q) \textbf{ fi}$$

where: $\mathsf{ADD\_CONST}$ adds a constant to a polynomial. The specification for $\mathsf{ADD\_CONST}$ can be elaborated as:

```
$\mathsf{ADD\_CONST}(c,Q)$ $\approx$ if $D[Q] = \Lambda$
                             then $C[Q] := C[Q] + c$
                             else $Q := D[Q]$; $\mathsf{ADD\_CONST}(c,Q)$; $Q := U[Q]$ fi
```

This can be converted to a recursive procedure: but will need a stack, or at least a recursion depth counter, to remove the recursion, since the inner copy of $\mathsf{ADD\_CONST}$ is not in a tail position, due to the need to restore the original value of Q. However, if we make c and Q parameters, then we do not need to restore their values. Applying the recursion introduction theorem to $\mathsf{ADD\_CONST}$ gives a tail-recursive procedure which can be converted to a loop:

```
proc $\mathsf{add\_const}(c,Q)$ $\equiv$
  while $D[Q] \neq \Lambda$ do $Q := D[Q]$ od;
  $C[Q] := C[Q] + c$.
```

We can therefore refine ADD as follows:

```
$\mathsf{ADD}(P,Q)$ $\leq$ if $D[P] = \Lambda$ then $\mathsf{add\_const}(C[P],Q)$
                                else $\mathsf{ADD}(P,Q)$ fi
```

If $D[P] \neq \Lambda$ but $D[Q] = \Lambda$ then we need to create a copy of the entire polynomial P, and then add the original constant value of Q to the copy of P. The specification for this copy operation is:

$$\begin{aligned}\mathsf{COPY}(P,Q) \;=_{\text{DF}}\; & \langle U,D,L,R,E,C\rangle := \langle U',D',L',R',E',C'\rangle.\\ & \mathsf{Abs}'(P) = \mathsf{Abs}(P) \,\wedge\, \mathsf{Abs}'(Q)) = \mathsf{Abs}(P))\\ & \wedge\, I(\mathsf{Abs}(P)) \,\wedge\, I(\mathsf{Abs}(Q)) \,\wedge\, I(\mathsf{Abs}'(Q))\\ & \wedge\, J(P) \,\wedge\, J(Q) \,\wedge\, J'(Q)\end{aligned}$$

We will now elaborate on this specification.

If P is a constant polynomial, then we just copy the constant and exponent to Q. Otherwise, we create a suitable doubly-linked list of nodes and copy each of the children of P into these nodes:

```
$\mathsf{COPY}(P,Q)$ $\leq$ $C[Q] := C[P]$; $E[Q] := E[P]$;
                  if $D[P] = \Lambda$
                    then $D[Q] := \Lambda$
                    else $P := D[P]$; $R_1 \xleftarrow{\text{pop}} \mathsf{Avail}$; $D[Q] := R_1$; $U[R_1] := Q$; $Q := R_1$;
                         do $\mathsf{COPY}(P,Q)$;
                            if $E[R[P]] = 0$ then $S := D[U[Q]]$; $R[Q] := S$; $L[S] := Q$; exit fi;
                            $P := R[P]$;
                            $R_1 \xleftarrow{\text{pop}} \mathsf{Avail}$; $U[R_1] := U[Q]$; $L[R_1] := Q$; $R[Q] := R_1$; $Q := R_1$ od;
                         $Q := U[Q]$; $P := U[P]$ fi
```

At the copy of the specification in the elaborated version (on the right), the size of P has been reduced. So we can apply Recursive Implementation (Transformation 5) to get a recursive procedure. To remove the recursion using Recursion Removal (Transformation 8), first restructure the body of the recursive procedure as a regular action system:

```
**proc** copy$(P, Q)$ $\equiv$
  **actions** $A$ :
  $A$ $\equiv C[Q] := C[P];\ E[Q] := E[P];$
        **if** $D[P] = \Lambda$
          **then** $D[Q] := \Lambda;$ **call** $Z$
           **else** $P := D[P];\ R_1 \xleftarrow{\text{pop}}$ Avail; $D[Q] := R_1;\ U[R_1] := Q;\ Q := R_1;$ **call** $B_1$ **fi.**
  $B_1$ $\equiv$ copy$(P, Q)$; **call** $A_1$.
  $A_1$ $\equiv$ **if** $E[R[P]] = 0$
           **then var** $\langle S := D[U[Q]]\rangle : R[Q] := S;\ L[S] := Q$ **end**; $Q := U[Q];\ P := U[P];$ **call** $Z$
            **else** $P := R[P];$
                 $R_1 \xleftarrow{\text{pop}}$ Avail; $U[R_1] := U[Q];\ L[R_1] := Q;\ R[Q] := R_1;\ Q := R_1;$
                 **call** $B_1$ **fi. endactions.**
```

For the recursion removal, we can avoid using a stack by noting that:

- The values of P and Q are preserved over the body of COPY(P, Q); and
- On returning from a call to COPY, if P has its original value, then this is the initial call, otherwise it is a recursive call.

Using these facts, and Recursion Removal (Transformation 8) we derive the following iterative procedure:

```
**proc** copy$(P, Q)$ $\equiv$
  **var** $\langle P_0 := P\rangle$ :
    **actions** $A$ :
    $A$ $\equiv C[Q] := C[P];\ E[Q] := E[P];$
          **if** $D[P] = \Lambda$
            **then** $D[Q] := \Lambda;$ **call** $\hat{A}$
             **else** $P := D[P];\ R_1 \xleftarrow{\text{pop}}$ Avail; $D[Q] := R_1;\ U[R_1] := Q;\ Q := R_1;$ **call** $B_1$ **fi.**
    $B_1$ $\equiv$ **call** $A$.
    $\hat{A}$ $\equiv$ **if** $P = P_0$ **then call** $Z$
                    **else call** $A_1$ **fi.**
    $A_1$ $\equiv$ **if** $E[R[P]] = 0$
             **then var** $\langle S := D[U[Q]]\rangle : R[Q] := S;\ L[S] := Q$ **end**; $Q := U[Q];\ P := U[P];$ **call** $\hat{A}$
              **else** $P := R[P];$
                   $R_1 \xleftarrow{\text{pop}}$ Avail; $U[R_1] := U[Q];\ L[R_1] := Q;\ R[Q] := R_1;\ Q := R_1;$
                   **call** $B_1$ **fi. endactions end.**
```

The iterative algorithm was restructured automatically, using the FermaT Maintenance Environment (see Section 6) to produce the following structured code:

```
**proc** copy$(P, Q)$ $\equiv$
  **var** $\langle P_0 := P\rangle$ :
    **do do** $C[Q] := C[P];\ E[Q] := E[P];$
          **if** $D[P] = \Lambda$ **then exit**$(1)$ **fi**;
          $P := D[P];\ R_1 \xleftarrow{\text{pop}}$ Avail; $D[Q] := R_1;\ U[R_1] := Q;\ Q := R_1$ **od**;
       $D[Q] := \Lambda;$
       **do if** $P = P_0$ **then exit**$(2)$ **fi**;
          **if** $E[R[P]] \neq 0$ **then exit**$(1)$ **fi**;
          **var** $\langle S := D[U[Q]]\rangle : R[Q] := S;\ L[S] := Q$ **end**;
          $Q := U[Q];\ P := U[P]$ **od**;
       $P := R[P];\ R_1 \xleftarrow{\text{pop}}$ Avail; $U[R_1] := U[Q];\ L[R_1] := Q;\ R[Q] := R_1;\ Q := R_1$ **od end end**
```

This code worked first time when it was tested.

We therefore have the following refinement of ADD:

```
ADD(P, Q)  ≤ if $D[P] = \Lambda$ then add_const($C[P], Q$)
             elsif $D[Q] = \Lambda$ then var $\langle c := C[Q]\rangle$ : COPY($P, Q$); add_const($c, Q$) end
             elsif $C[Q] > C[P]$ then ADD($P, Q$)
             elsif $C[Q] < C[P]$ then ADD($P, Q$)
                                  else ADD($P, Q$) fi
```

For the case where $C[Q] > C[P]$ we simply need to add Q to the constant element of P:

$$\{C[Q] > C[P]\};\ \mathsf{ADD}(P, Q)\ \leq\ \{C[Q] > C[P]\};\ Q := D[Q];\ \mathsf{ADD}(P, Q);\ Q := U[Q]$$

For the case where $C[Q] = C[P]$, we simply need to add the two lists of terms together as follows:

$P := D[P];\ Q := D[Q];\ \mathsf{ADD\_TERMS}(P, Q);\ P := U[P];\ Q := U[Q]$

where ADD_TERMS adds the corresponding terms in the two polynomials, given that P and Q point to the first term in the list. Its implementation will be discussed below.

If $C[Q] < C[P]$ we need to convert Q to a constant polynomial with the same variable as P by inserting a new parent node, after which we can add the terms of P to the terms of this new polynomial:

$R_1 \overset{\text{pop}}{\longleftarrow} \mathsf{Avail};\ C[R_1] := C[P];\ D[R_1] := Q;\ U[R_1] := U[Q];$
$E[R_1] := E[Q];\ E[Q] := 0;\ L[R_1] := L[Q];\ R[R_1] := R[Q];$
$U[Q] := R_1;\ L[Q] := Q;\ R[Q] := Q;$
$P := D[P];\ \mathsf{ADD\_TERMS}(P, Q);\ P := U[P];\ Q := U[Q]$

We can implement ADD_TERMS as a simple loop which uses ADD to add each term. After adding two terms, we need to check if the result has a zero coefficient with non-zero exponent. These terms are not allowed by the assertion I on the abstract version of the polynomial. Similarly, after adding all the terms, we need to check if the result is a constant polynomial (i.e. all the terms with non-zero exponent ended up with a zero coefficient and therefore were deleted). In this case, we need to replace the polynomial by the constant (its first term). The specification ADD_TERMS can assume that P and Q have the same exponent:

```
ADD_TERMS(P, Q)  ≈ ADD($P, Q$);
                   C : Check for a zero term with non-zero exponent;
                   if $E[Q] \neq 0\ \wedge\ D[Q] = \Lambda\ \wedge\ C[Q] = 0$
                     then var $\langle R_1 := Q, S := R[Q]\rangle$ :
                            $Q := L[Q];\ L[S] := Q;\ R[Q] := S;$
                            Avail $\overset{\text{push}}{\longleftarrow} R_1$ end fi;
                   ADD_REST($P, Q$)
```

The statement ADD_REST(P, Q) will move Q to the appropriate term, and then continue adding terms. If there is no appropriate term (a term with the same exponent as $E[P]$) then we need to insert a node in the list of terms and copy P over this node. We can then add the rest of the terms:

```
ADD_REST(P, Q)  ≈ $P := R[P]$;
                  if $E[P] \neq 0$
                    then $Q := R[Q]$;
                         while $E[Q] > 0\ \wedge\ E[Q] < E[P]$ do $Q := R[Q]$ od;
                         if $E[Q] = E[P]$
                           then ADD_TERMS($P, Q$)
                    else C :  Insert a copy of P to the left of Q;
                        var $\langle R_1 := 0\rangle$ :
                          $R_1 \overset{\text{pop}}{\longleftarrow}$ Avail;
                          $L[R_1] := L[Q];\ R[R_1] := Q;\ U[R_1] := U[Q];\ E[R_1] := E[P]$;
                          $R[L[Q]] := R_1;\ L[Q] := R_1;\ Q := R_1$ end;
                        copy$(P, Q)$;
                        ADD_REST$(P, Q)$ fi
```

This specification for ADD_REST is tail-recursive, so it can be converted to a loop. Unfolding this loop into the specification for ADD_TERMS results in another tail-recursive specification, which again can be converted to a loop:

```
proc add_terms$(P, Q)$ $\equiv$
  do ADD$(P, Q)$;
     C :  Check for a zero term with non-zero exponent;
     if $E[Q] \neq 0 \wedge D[Q] = \Lambda \wedge C[Q] = 0$
       then var $\langle R_1 := Q, S := R[Q]\rangle$ :
               $Q := L[Q];\ L[S] := Q;\ R[Q] := S$;
               Avail $\overset{\text{push}}{\longleftarrow} R_1$ end fi;
     do $P := R[P]$;
        if $E[P] = 0$ then exit$(2)$ fi;
        $Q := R[Q]$;
        while $E[Q] > 0 \wedge E[Q] < E[P]$ do $Q := R[Q]$ od;
        if $E[Q] = E[P]$ then exit$(1)$ fi;
        C :  Insert a copy of P to the left of Q;
        var $\langle R_1 := 0\rangle$ :
          $R_1 \overset{\text{pop}}{\longleftarrow}$ Avail;
          $L[R_1] := L[Q];\ R[R_1] := Q;\ U[R_1] := U[Q];\ E[R_1] := E[P]$;
          $R[L[Q]] := R_1;\ L[Q] := R_1;\ Q := R_1$ end;
        copy$(P, Q)$ od od.
```

Putting all these implementations together, and applying Recursive Implementation (Transformation 5), we derive the following recursive implementation of ADD:

```
proc add$(P, Q)$ $\equiv$
  if $D[P] = \Lambda$
    then add_const$(C[P], Q)$
  elsif $D[Q] = \Lambda$
       then var $\langle c := C[Q]\rangle$ : copy$(P, Q)$; add_const$(c, Q)$ end
  elsif $C[Q] > C[P]$
       then $Q := D[Q]$; add$(P, Q)$
        else insert_below_Q;
             $P := D[P];\ Q := D[Q]$;
             do add$(P, Q)$;
                check_for_zero_term;
                do $P := R[P]$;
                   if $E[P] = 0$ then exit$(2)$ fi;
                   $Q := R[Q]$;
                   while $E[Q] > 0 \wedge E[Q] < E[P]$ do $Q := R[Q]$ od;
                   if $E[Q] = E[P]$ then exit fi;
                   insert_copy od od;
             $P := U[P]$;
             check_for_const_poly fi.
```

```
proc add_const$(c, Q)$ $\equiv$
  while $D[Q] \neq \Lambda$ do $Q := D[Q]$ od;
  $C[Q] := C[Q] + c$.
proc insert_below_Q $\equiv$
  if $C[Q] < C[P]$
    then C : Insert a node below $Q$ and convert $Q$ to a constant poly;
         var $\langle R_1 := 0, S := D[Q]\rangle$ :
           $R_1 \overset{\text{pop}}{\longleftarrow}$ Avail; ;
           do $U[S] := R_1$; $S := R[S]$;
              if $E[S] = 0$ then exit$(1)$ fi od;
           $U[R_1] := Q$; $D[R_1] := S$; $L[R_1] := R_1$; $R[R_1] := R_1$;
           $C[R_1] := C[Q]$; $E[R_1] := 0$; $C[Q] := C[P]$; $D[Q] := R_1$ end fi.
proc check_for_zero_term $\equiv$
  C : Check for a zero term with non-zero exponent;
  if $E[Q] \neq 0 \wedge D[Q] = \Lambda \wedge C[Q] = 0$
    then var $\langle R_1 := Q, S := R[Q]\rangle$ :
           $Q := L[Q]$; $L[S] := Q$; $R[Q] := S$;
           Avail $\overset{\text{push}}{\longleftarrow}$ $R_1$ end fi.
proc insert_copy $\equiv$
  C : Insert a copy of P to the left of Q;
  var $\langle R_1 := 0\rangle$ :
    $R_1 \overset{\text{pop}}{\longleftarrow}$ Avail;
    $L[R_1] := L[Q]$; $R[R_1] := Q$; $U[R_1] := U[Q]$; $E[R_1] := E[P]$;
    $R[L[Q]] := R_1$; $L[Q] := R_1$; $Q := R_1$ end;
  copy$(P, Q)$.
proc check_for_const_poly $\equiv$
  C : Check for a constant poly;
  if $R[Q] = Q$
    then var $\langle R_1 := Q, S := 0\rangle$ :
           $Q := U[Q]$; $D[Q] := D[R_1]$; $C[Q] := C[R_1]$;
           Avail $\overset{\text{push}}{\longleftarrow}$ $R_1$;
           $S := D[Q]$;
           if $S \neq \Lambda$
             then do $U[S] := Q$; $S := R[S]$;
                     if $E[S] = 0$ then exit fi od fi end fi.
```

This recursive program was tested, and worked first time.

In order to apply the recursion removal theorem, the first step is to restructure the body of the procedure as a regular action system. The first recursive call to add is in a tail position, so it can be replaced by an action call. At the point of the call, we know that $D[P] = \Lambda$ so we can unfold and simplify the action. The resulting action system contains a single recursive call in action B_1:

```
proc add$(P, Q)$ $\equiv$
  actions $A$ :
  $A$ $\equiv$ if $D[P] = \Lambda$
         then add_const; call $Z$
         else call $A_1$ fi.
  $A_1$ $\equiv$ if $D[Q] = \Lambda$
          then copy_and_add; call $Z$
        elsif $C[Q] > C[P]$
          then $Q := D[Q]$; call $A_1$
          else insert_below_Q;
```

```
                    $P := D[P]$; $Q := D[Q]$;
                    **call** $B_1$ **fi.**
  $B_1$ $\equiv$ add$(P, Q)$; check_for_zero_term; **call** $A_3$.
  $A_3$ $\equiv$ $P := R[P]$;
        **if** $E[P] = 0$
          **then** $P := U[P]$; check_for_const_poly; **call** $Z$
           **else** $Q := R[Q]$;
                 **while** $E[Q] > 0 \wedge E[Q] < E[P]$ **do** $Q := R[Q]$ **od**;
                 **if** $E[Q] = E[P]$ **then call** $B_1$ **fi**;
                 insert_copy;
                 **call** $A_3$ **fi. endactions.**
```

In order to derive an equivalent efficient iterative program, we first wish to eliminate the parameters. The operations on parameter P restore its original value at the end of the procedure body: so the parameter can be converted to a global variable. For Q, the various operations will leave Q pointing at some element of the tree under the required final value: so Q can be restored by moving up a certain number of nodes in the tree. For the external call to add we can restore Q by moving up until $U[Q] = \Lambda$. For the recursive call, we can restore Q after the call by moving up until $C[U[Q]] = C[U[P]]$. So, both parameters can be replaced by global variables.

The recursive call to add is not in a tail position, so recursion removal would normally require a stack. However, we know that for any external call $U[P] = \Lambda$ while for any internal call $U[P] \neq \Lambda$, so we can use these conditions to eliminate the stack (since the stack is empty if and only if $U[P] = \Lambda$). The recursion removal theorem thus produces the following iterative algorithm:

```
**proc** add $\equiv$
  **actions** $A$ :
  $A$ $\equiv$ **if** $D[P] = \Lambda$
          **then** add_const; **call** $\hat{A}$
           **else call** $A_1$ **fi.**
  $A_1$ $\equiv$ **if** $D[Q] = \Lambda$
           **then** copy_and_add; **call** $\hat{A}$
        **elsif** $C[Q] > C[P]$
              **then** $Q := D[Q]$; **call** $A_1$
               **else** insert_below_Q;
                    $P := D[P]$; $Q := D[Q]$;
                    **call** $B_1$ **fi.**
  $B_1$ $\equiv$ **call** $A$.
  $A_3$ $\equiv$ $P := R[P]$;
        **if** $E[P] = 0$
          **then** $P := U[P]$; check_for_const_poly; **call** $\hat{A}$
           **else** $Q := R[Q]$;
                 **while** $E[Q] > 0 \wedge E[Q] < E[P]$ **do** $Q := R[Q]$ **od**;
                 **if** $E[Q] = E[P]$ **then call** $B_1$ **fi**;
                 insert_copy;
                 **call** $A_3$ **fi.**
  $\hat{A}$ $\equiv$ **if** $U[P] = \Lambda$
          **then call** $Z$
           **else while** $C[U[Q]] \neq C[U[P]]$ **do** $Q := U[Q]$ **od**;
                check_for_zero_term; **call** $A_3$ **fi. endactions.**
```

The iterative algorithm was restructured automatically, using the FermaT Maintenance Environment (see Section 6) to produce the following structured code:

```
proc add( var ) ≡
  do do if $D[P] = \Lambda$ then add_const; exit(1) fi;
        do if $D[Q] = \Lambda$
               then copy_and_add; exit(2)
           elsif $C[P] < C[Q]$
                 then $Q := D[Q]$
                 else exit(1) fi od;
        insert_below_Q;
        $P := D[P]$;
        $Q := D[Q]$ od;
     do if $U[P] = \Lambda$ then exit(2) fi;
        while $C[U[Q]] \neq C[U[P]]$ do $Q := U[Q]$ od;
        check_for_zero_term;
        do $P := R[P]$;
           if $E[P] = 0$ then exit(1) fi;
           $Q := R[Q]$;
           while $E[Q] > 0 \land E[Q] < E[P]$ do $Q := R[Q]$ od;
           if $E[Q] = E[P]$ then exit(2) fi;
           insert_copy od;
        $P := U[P]$;
        check_for_const_poly od od end
```

The iterative program was tested and after fixing a few typos, it worked first time. The program was tested against the abstract algorithm by generating a huge number of random polynomials and checking that the abstract version of the output of the algorithm was the same as the abstract sum of the abstract versions of the input polynomials.

It would be a very challenging task to verify the correctness of the iterative algorithm using loop invariants: the loops are nested four deep, with **exit**s which terminate up to two enclosing nested loops. It would be difficult to give meaningful loop invariants for these loops.

The method of "Invariant Based Programming" [Bac09] requires one to derive the invariants and then use them to develop the code. I challenge any proponents of these methods to apply them successfully to this programming task!

5.5 Execution Timings

Vars	Terms	Count	Abstract	Concrete	Knuth	K/A	K/C
7	4	1,000,000	7.68s	10.77s	22.64s	2.95	2.10
7	4	10,000,000	78.02s	109.14s	229.57s	2.94	2.10
2	4	100,000,000	112.64s	34.46s	130.76s	1.16	3.79
2	20	10,000,000	24.48s	14.38s	31.66s	1.29	2.20
7	10	10,00,000	11.02s	158.68s	332.80s	30	2.10
7	20	400,000	4.47s	1,271.43s	3,129.33s	700	2.46

Table 1: Execution times for the algorithms

Table 1 shows the timings for the various algorithms applied to different sizes of polynomials. "Vars" is the maximum number of variables in the polynomial (and hence the maximum depth of nesting of the tree structure). "Terms" is the maximum number of terms in the polynomials at each level. "K/A" is the Knuth algorithm time divided by the abstract algorithm time, while "K/C" is the Knuth algorithm time divided by the concrete algorithm time: these are the two algorithms which use Knuth's four-way linked data structure.

"Count" is the total number of executions: the program generated Count/100 different random polynomials for each run and executed the algorithms 100 times for each polynomial. This gave more accurate timings since the overhead for generating a random polynomial is quite large. The whole set of test runs were repeated several times to check the accuracy of the timings: using the same random number seed for each run, to ensure that the same sequence of polynomials was generated.

As we can see from this table, the abstract algorithm is always faster than Knuth's algorithm (though not always faster than our algorithm using Knuth's data structures). For large polynomials, the abstract algorithm is considerably faster: with the speed ration increasing as the polynomials get larger. This is due to the fact that the abstract algorithm can "share" large sub-polynomials by setting up pointers to the same data structure.

The experimental setup did not allow us to measure the memory usage with any accuracy. In general, memory requirements for the two algorithms will differ depending on the size and type of data being processed. As discussed above, the abstract algorithm requires additional memory to store the recursion stack, but less memory for the nodes, so may be expected to use less memory than the concrete algorithm. In cases where sub-trees can be shared, the abstract algorithm will have considerably smaller memory requirements along with much smaller processing time.

Our algorithm is consistently at least 2.1 times faster than Knuth's algorithm. For very small polynomials (up to two variables and up to four terms per polynomial), our algorithm is nearly four times faster. Memory requirements for our concrete algorithm are identical to Knuth's.

We had assumed that Knuth's algorithm was probably close to optimal, for the data structures he used, since Knuth is a very experienced and knowledgeable programmer, and the code had been subjected to a number of reviews by other programmers. We had hoped that the code produced by our transformational programming method would be at least comparable to Knuth's in efficiency and that any minor inefficiencies would be compensated for by the greater transparency between the specification and the code. The results (a speed improvement by a factor of at least 2.1) were a pleasant surprise which demanded some explanation: especially given that various other applications of transformational programming also produced highly efficient code. Naturally, the efficiency of the code produced by this method is subject to the limitations inherent in the ideas used to guide the transformation process: but even when these ideas produce inefficient code, there are cases where optimising transformations can be applied to improve the result (see Section 4.1 for an example).

One possible explanation for these efficiency results is that in the specification elaboration and divide and conquer stages, executable code is only inserted when it is necessary to achieve a particular object. All code, therefore, makes a useful contribution to the final result. In contrast, the invariant based programming method (in common with any "write and verify" programming method) does not prevent the programmer from computing superfluous results, or inserting superfluous tests. As long as the required invariants can be proved from the introduced code, the verification proof will go through and the resulting program will be accepted: there is nothing to guide the programmer to the most efficient solution. For example, if it is possible to write a single **while** loop which maintains the invariants and reduces the variant function, then this is likely to be selected for verification even when a more efficient solution using multiply-nested loops with exits from the middle of the loops is possible.

With the transformational programming method, the biggest impact on the efficiency of the derived algorithm is, of course, the quality of the informal implementation ideas which are brought into the process. However, given a good initial idea, an efficient implementation will be produced without unnecessary extra computation. This is because at each stage in the derivation process we are working with a complete and correct program which *only* computes the required output. Multiply-nested loops with **exit**s from the middle "fall out" of the derivation process in a natural way when such loops are the most efficient implementation method.

5.6 Polynomial Multiplication

Exercise 2.3.3–15 of [Knu68] asks: "Design an algorithm to compute the product of two polynomials represented as in Fig. 28.".

The specification uses the same abstraction functions and invariants as the specification for polynomial addition:

$$\begin{aligned}\mathsf{MULT}(P, Q, R_1) \;=_{\text{DF}}\; \langle U, D, L, R, E, C\rangle := \langle U', D', L', R', E', C'\rangle.\;& \mathsf{Abs}'(P) = \mathsf{Abs}(P) \;\wedge\; \mathsf{Abs}'(Q) = \mathsf{Abs}(Q)\\ &\wedge\; \mathsf{abs}(\mathsf{Abs}'(R_1)) = \mathsf{abs}(\mathsf{Abs}(P)) * \mathsf{abs}(\mathsf{Abs}(Q))\\ &\wedge\; I(\mathsf{Abs}(P)) \;\wedge\; I(\mathsf{Abs}(Q)) \;\wedge\; I(\mathsf{Abs}'(R_1))\\ &\wedge\; J(P) \;\wedge\; J(Q) \;\wedge\; J'(R_1)\end{aligned}$$

The only differences are that the addition operation $(+)$ has been replaced by the multiplication operation $(*)$, and instead of updating Q with the result, we return the result in R_1. A node pointer is provided in parameter R_1 and this node is used as the root of the result. (Since each term of P has to be multiplied by each term of Q it would needlessly complicate matters to attempt to update Q in place.)

The informal ideas which drive the derivation process are as follows:

1. Multiplying a constant by a polynomial is a special case which is treated separately (as we did with polynomial addition);
2. To multiply two polynomials which are in the same variable: Initialise the result to zero, then for each term of p:
   (a) Multiply this term by each term of q to create a list of terms;
   (b) Convert this list of terms to a polynomial;
   (c) Add this polynomial to the total;
3. If the variable for p is larger than the variable for q then multiply each term of p by q
4. Otherwise, multiply each term of q by p

The derivation follows broadly the same stages as for polynomial addition:

1. Take out multiplication by a constant as a special case: this is implemented as Mult_Const below;
2. Otherwise, split into cases depending on the variables for the two polynomials;
3. Use Recursive Implementation (Transformation 5) to derive an abstract polynomial multiplication algorithm;
4. Convert this recursive algorithm to a recursive algorithm which operates on concrete polynomials;
5. Convert parameters and local variables to global variables;
6. Remove the recursion: this step uses the fact that we can determine the return point by examining the nodes at P, Q and R_1 to avoid the need for a control stack.

The details are omitted for brevity, but should be straightforward to anyone familiar with the method. The resulting implementation worked first time when tested:

```
proc Mult($P, Q, R_1$) $\equiv$
  var $\langle L_1 := 0, L_2 := 0, S := 0, c := 0 \rangle$ :
    do do if $D[P] = \Lambda$
            then if $C[P] = 0$
                   then $C[R_1] := 0$; $D[R_1] := \Lambda$
                   else Mult_Const($C[P], Q, R_1$) fi;
                 exit(1)
          elsif $D[Q] = \Lambda$
            then if $C[Q] = 0$
                   then $C[R_1] := 0$; $D[R_1] := \Lambda$
                   else Mult_Const($C[Q], P, R_1$) fi;
                 exit(1)
          elsif $C[P] = C[Q]$
            then $C[R_1] := 0$; $D[R_1] := \Lambda$;
                 Pop_Avail( var $L_1$); $U[L_1] := S$; $S := L_1$;
                 $L[S] := R_1$; $P := D[P]$; Pop_Avail( var $R_1$);
                 $L[R_1] := R_1$; $R[R_1] := R_1$; $Q := D[Q]$
          elsif String_Less?($C[P], C[Q]$)
            then $C[R_1] := C[Q]$; $Q := D[Q]$; Pop_Avail( var $L_1$);
                 $D[R_1] := L_1$; $U[L_1] := R_1$; $L[L_1] := L_1$; $R[L_1] := L_1$; $R_1 := L_1$
            else $C[R_1] := C[P]$; $P := D[P]$; Pop_Avail( var $L_1$);
                 $D[R_1] := L_1$; $U[L_1] := R_1$; $L[L_1] := L_1$; $R[L_1] := L_1$; $R_1 := L_1$ fi od;
       do if $U[P] = \Lambda$
            then if $U[Q] = \Lambda$
                   then $c := 0$
                 elsif $C[U[Q]] = C[U[R_1]]$
                   then $c := 2$
                   else $c := 3$ fi
          elsif $U[Q] = \Lambda$
            then if $C[U[P]] = C[U[R_1]]$ then $c := 3$ else $c := 2$ fi
          elsif $C[U[P]] = C[U[Q]]$
            then $c := 1$
          elsif $C[U[Q]] = C[U[R_1]]$
            then $c := 2$
            else $c := 3$ fi;
          if $c = 0$ then exit(2)
          elsif $c = 1$
            then $E[R_1] := E[P] + E[Q]$; $Q := R[Q]$;
                 if $E[Q] = 0$
                   then $Q := U[Q]$; $R_1 := R[R_1]$;
                        if $E[R_1] \neq 0$
                          then Pop_Avail( var $L_2$);
                               $R[L_2] := R_1$; $R[L[R_1]] := L_2$; $L[L_2] := L[R_1]$; $L[R_1] := L_2$;
                               $D[L_2] := \Lambda$; $E[L_2] := 0$; $C[L_2] := 0$; $R_1 := L_2$ fi;
                        Pop_Avail( var $L_2$);
                        $U[R_1] := L_2$; $D[L_2] := R_1$; $U[L_2] := \Lambda$;
                        $L[L_2] := L_2$; $R[L_2] := L_2$; $C[L_2] := C[Q]$; $E[L_2] := 0$; $R_1 := R[R_1]$;
                        while $E[R_1] \neq 0$ do
                               $U[R_1] := L_2$; $R_1 := R[R_1]$ od;
                        add($L_2, L[S]$);
                        Free_Poly($L_2$); $P := R[P]$;
                        if $E[P] = 0$
```

```
                                   then $R_1 := L[S]$; Avail $\xleftarrow{\text{push}}$ $S$; $S := U[S]$; $P := U[P]$
                                    else Pop_Avail( var $R_1$);
                                        $L[R_1] := R_1$; $R[R_1] := R_1$; $Q := D[Q]$; exit(1) fi
                             else Pop_Avail( var $L_2$);
                                 $L[L_2] := R_1$; $L[R[R_1]] := L_2$; $R[L_2] := R[R_1]$; $R[R_1] := L_2$;
                                 $D[L_2] := \Lambda$; $R_1 := L_2$; exit(1) fi
        elsif $c = 2$
            then $E[R_1] := E[Q]$; $Q := R[Q]$;
                if $E[Q] = 0$
                  then $R_1 := U[R_1]$; $Q := U[Q]$
                   else Pop_Avail( var $L_2$); $U[L_2] := U[R_1]$; $L[L_2] := R_1$; $L[R[R_1]] := L_2$;
                       $R[L_2] := R[R_1]$; $R[R_1] := L_2$; $D[L_2] := \Lambda$; $R_1 := L_2$; exit(1) fi
             else $E[R_1] := E[P]$; $P := R[P]$;
                if $E[P] = 0$
                  then $R_1 := U[R_1]$; $P := U[P]$
                   else Pop_Avail( var $L_2$);
                       $U[L_2] := U[R_1]$; $L[L_2] := R_1$; $L[R[R_1]] := L_2$; $R[L_2] := R[R_1]$;
                       $R[R_1] := L_2$; $D[L_2] := \Lambda$; $R_1 := L_2$; exit(1) fi fi od od.;

proc Free_Poly($P$) $\equiv$
  $U[P] := \Lambda$;
  do do Avail $\xleftarrow{\text{push}}$ $P$;
        if $D[P] = \Lambda$ then exit(1) fi;
        $P := D[P]$ od;
     do $P := R[P]$;
        if $E[P] = 0$ then $P := U[P]$; if $U[P] = \Lambda$ then exit(2) fi
                    else exit(1) fi od od.;

proc Mult_Const($c, P, R_1$) $\equiv$
  var $\langle P_0 := P, R_2 := 0, S := 0\rangle$ :
    do do $E[R_1] := E[P]$;
          if $D[P] = \Lambda$
            then $D[R_1] := \Lambda$; $C[R_1] := c * C[P]$; exit(1)
             else $C[R_1] := C[P]$; Pop_Avail( var $R_2$); $D[R_1] := R_2$;
                 $U[R_2] := R_1$; $R_1 := R_2$; $P := D[P]$ fi od;
       do if $P = P_0$ then exit(2) fi;
          if $E[R[P]] \neq 0$ then exit(1) fi;
          $S := D[U[R_1]]$; $R[R_1] := S$; $L[S] := R_1$; $P := U[P]$; $R_1 := U[R_1]$ od;
       $P := R[P]$; Pop_Avail( var $R_2$); $U[R_2] := U[R_1]$; $L[R_2] := R_1$;
       $R[R_1] := R_2$; $R_1 := R_2$ od end.
```

6 Conclusion

In this paper we have outlined the transformational programming method for program development and compared it with some popular formal methods for program development which have been proposed. In this section we will reiterate the main advantages of transformational programming.

With transformational programming we are working with a correct program at every stage in the development process. This is in contrast to the invariant-based program development methods, which all require a verification stage in which the correctness of the proposed implementation is checked against the proposed invariants and, ultimately, against the original specification. If there happens to be an error in the proposed code, then the only indication of this error is an inability to carry through one or more proof obligations. This may not be a problem for a small, "toy" program, but for a large program with perhaps thousands of lines of code in the body of the loop, it may prove extremely difficult to track down the source of the error.

Transformational programming also allows the developer to treat correctness and efficiency separately: correctness is guaranteed by the derivation process, so the selection of transformations can be dominated by efficiency considerations. This is especially useful when a single data structure in the final program is used for two separate purposes. For example, all the "verification" style proofs of the Schorr-Waite graph marking algorithm need to prove that the graph is correctly marked, *and* that the pointers are restored to their original values, in a single proof. The transformational derivation of this algorithm [War96] first produces a recursive algorithm, then applies the "pointer switching" strategy to eliminate the need for a stack, then removes the recursion.

A problem with any "write and verify" programming method is that the method does not prevent the programmer from computing superfluous results. As long as the required result is available somewhere among all the computed data, the verification proof will go through and the resulting program will be accepted. Our experience is that, given a good initial idea, transformational programming will usually produce an efficient implementation of that idea, without unnecessary extra computation. This may be because at each stage in the derivation process we are working with a complete and correct program which *only* computes the required output.

This point is illustrated by the derivation of the polynomial addition algorithm: our derived algorithm is twice as fast as the one presented by Knuth, even though it is a structured program which operates on the same data structures and produces the same result. Knuth's wish ("I hope someday to see the algorithm cleaned up without loss of its efficiency" [Knu74]) has therefore been granted. The hypothesis that the authors are simply better programmers than Donald Knuth and his colleagues cannot be seriously entertained, therefore the efficiency improvement must be attributed to the methodology.

The most important advantage of transformational programming is that it is capable of scaling up to large programs: any method which requires loop invariants will encounter difficulties when the body of the loop extends to hundreds, or perhaps thousands, of lines of code.

The source code for all the polynomial addition algorithms is available as part of the FermaT Transformation System, and the FermaT Maintenance Environment and can be downloaded from the following web sites:

http://www.gkc.org.uk/fermat.html
http://www.cse.dmu.ac.uk/~mward/fermat.html

6.1 Software Evolution

In [Tho03] Martyn Thomas writes:

> Software change is the *most important step* in the software lifecycle: most software costs far more after delivery than before (and most current "software maintenance" destroys more value than it preserves in your software assets).
>
> When requirements change, it is important to be able to make controlled changes to the specification. (In these circumstances, modifying software by going directly to the detailed design or code is vandalism). The specification therefore needs to be expressed in such a way that the nature, scope and impact of any change can be assessed and accommodated.

None of the "code and verify" development methods are particularly adept at handling changes to the specification. If the original specification is changed, then it is possible that many invariants will change. It is certain that all the proofs will have to be re-done (or at least checked): and given that many proofs require human intervention, this could be a lot of work.

With the transformational programming approach, it is likely that many of the informal ideas are still valid for the new specification, and therefore that much of the basic transformation sequence generated by these ideas can still be applied to the new specification (generating modified code at each stage).

One example, which actually involves a small but significant change to the specification, is the development of the polynomial multiplication algorithm. The change in the specification simply involves replacing "+" with "$*$", and adding a new variable for the result — but this is quite a significant change! However, many of the implementation ideas and transformation sub-sequences used in the derivation of the addition program could also be used in the derivation of the multiplication program: so that, even though the final multiplication algorithm is much more complex than the addition algorithm, it took significantly *less* time to develop.

It would appear that with a properly-written specification (see Section 1.2), a small change to the requirements is likely to result in a small change to the specification. Any informal implementation ideas may still be valid: in which case, the derivation process can repeat many of the steps from the original derivation. This is because the implementation ideas are used to select the sequence of transformations to be applied: if the ideas are still valid then it is likely that the sequence is still valid and can be applied to the modified specification with only minimal changes. This process can be streamlined even further with the aid of suitable tool support, such as the FermaT Maintenance Environment.

In order to accommodate future changes to the specification or implementation environment (and to support software evolution in general) it is recommended that the original formal specification and development history be recorded and maintained: not just the final code. With proper tools, there is no need to keep a copy of the final code: since it can be reconstructed automatically from the specification and development history.

This idea is a natural extension of the current practice with high level language compilers: nobody would dream of compiling a program and then throwing away the source code and maintaining the generated object code!

References

[06] ISO JTC 1/SC 7, "Software Engineering – Software Life Cycle Processes – Maintenance," ISO/IEC 14764:2006, 2006.

[ABH06] Jean-raymond Abrial, Michael Butler, Stefan Hallerstede & Laurent Voisin, "An open extensible tool environment for Event-B," *ICFEM*, New York–Heidelberg–Berlin (2006).

[Bac09] Ralph-Johan Back, "Invariant Based Programming: Basic Approach and Teaching Experiences," *Formal Aspects of Computing* 21 #3 (May, 2009), 227–244.

[Bau79] F. L. Bauer, "Program Development By Stepwise Transformations—the Project CIP," in *Program Construction*, G. Goos & H. Hartmanis, eds., Lect. Notes in Comp. Sci. #69, Springer-Verlag, New York–Heidelberg–Berlin, 1979, 237–266.

[BB85] F. L. Bauer, R. Berghammer, et. al. & The CIP Language Group, *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*, Lect. Notes in Comp. Sci. #183, Springer-Verlag, New York–Heidelberg–Berlin, 1985.

[BMP89] F. L. Bauer, B. Moller, H. Partsch & P. Pepper, "Formal Construction by Transformation—Computer Aided Intuition Guided Programming," *IEEE Trans. Software Eng.* 15 #2 (Feb., 1989).

[BaT87] F. L. Bauer & The CIP System Group, *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, Lect. Notes in Comp. Sci. #292, Springer-Verlag, New York–Heidelberg–Berlin, 1987.

[BiM99] Juan C. Bicarregui & Brian M. Matthews, "Proof and Refutation in Formal Software Development ," *In 3rd Irish Workshop on Formal Software Development* (July, 1999).

[BiM96] Richard Bird & Oege de Moor, *The Algebra of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

[Bro84] M. Broy, "Algebraic Methods for Program Construction: the Project CIP," 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, 199–222.

[BGL11] Alan Bundy, Gudmund Grov & Yuhui Lin, "Productive use of failure in top-down formal methods," *Automated Reasoning Workshop* (2011).

[BuD77] R. M. Burstall & J. A. Darlington, "A Transformation System for Developing Recursive Programs," *J. Assoc. Comput. Mach.* 24 #1 (Jan., 1977), 44–67.

[But06] Michael Butler, "On the Verified-by-Construction Approach," BCS FACS, Electronics & Computer Science EPrints Service, 2006, oai:eprints.soton.ac.uk:265110.

[Dar78] J. Darlington, "A Synthesis of Several Sort Programs," *Acta Informatica* 11 #1 (1978), 1–30.

[Dij] E. W. Dijkstra, "A Constructive Approach to the Problem of Program Correctness.," Technische Hogeschool Eindhoven, EWD209, ⟨ http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD209.PDF⟩.

[Dij76] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Dij70] E. W. Dijkstra, "Notes On Structured Programming," Technische Hogeschool Eindhoven, EWD249, Apr., 1970, ⟨http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF⟩.

[Dij72] E. W. Dijkstra, "The Humble Programmer," *Comm. ACM* 15 #10 (Oct., 1972), 859–866.

[Gri81] David Gries, *The Science of Programming*, Springer-Verlag, New York–Heidelberg–Berlin, 1981.

[HHJ87] C. A. R. Hoare, I. J. Hayes, H. E. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey & B. A. Sufrin, "Laws of Programming," *Comm. ACM* 30 #8 (Aug., 1987), 672–686.

[JJL91] C. B. Jones, K. D. Jones, P. A. Lindsay & R. Moore, *mural: A Formal Development Support System*, Springer-Verlag, New York–Heidelberg–Berlin, 1991.

[Knu74] D. E. Knuth, "Structured Programming with the GOTO Statement," *Comput. Surveys* 6 #4 (1974), 261–301.

[Knu68] D. K. Knuth, *Fundamental Algorithms*, The Art of Computer Programming #1, Addison Wesley, Reading, MA, 1968.

[LPR98] Charles E. Leiserson, Harald Prokop & Keith H. Randall, "Using de Bruijn Sequences to Index a 1 in a Computer Word," CiteSeer, 1998, ⟨http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.8562⟩.

[Mor94] C. C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1994, Second Edition.

[MRG88] C. C. Morgan, K. Robinson & Paul Gardiner, "On the Refinement Calculus," Oxford University, Technical Monograph PRG-70, Oct., 1988.

[MoV93] C. C. Morgan & T. Vickers, *On the Refinement Calculus*, Springer-Verlag, New York–Heidelberg–Berlin, 1993.

[NHW89] M. Neilson, K. Havelund, K. R. Wagner & E. Saaman, "The RAISE Language, Method and Tools," *Formal Aspects of Computing* 1 (1989), 85–114 .

[PrW94] H. A. Priestley & M. Ward, "A Multipurpose Backtracking Algorithm," *J. Symb. Comput.* 18 (1994), 1–40, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/backtr-t.ps.gz⟩ doi:10.1006/jsco.1994.1035.

[Sen90] C. T. Sennett, "Using Refinement to Convince: Lessons Learned from a Case Study," *Refinement Workshop, 8th–11th January, Hursley Park, Winchester* (Jan., 1990).

[Tam95] T. Tammet, "Lambda lifting as an optimization for compiling Scheme to C," Chalmers University of Technology, Department of Computer Sciences, Goteborg, Sweden, 1995, ⟨ftp://ftp.cs.chalmers.se/pub/users/tammet/hobbit.ps⟩.

[Tho03] Martyn Thomas, "The Modest Software Engineer," *The Sixth International Symposium on Autonomous Decentralized Systems, ISADS* (2003).

[War89] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989, ⟨http://www.cse.dmu.ac.uk/~mward/martin/thesis⟩.

[War90] M. Ward, “Derivation of a Sorting Algorithm,” Durham University, Technical Report, 1990, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/sorting-t.ps.gz⟩.

[War94] M. Ward, “Reverse Engineering through Formal Transformation Knuths “Polynomial Addition” Algorithm,” *Comput. J.* 37 #9 (1994), 795–813, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/poly-t.ps.gz⟩ doi:10.1093/comjnl/37.9.795.

[War96] M. Ward, “Program Analysis by Formal Transformation,” *Comput. J.* 39 #7 (1996), ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/topsort-t.ps.gz⟩ doi:10.1093/comjnl/39.7.598.

[War99a] M. Ward, “Recursion Removal/Introduction by Formal Transformation: An Aid to Program Development and Program Comprehension,” *Comput. J.* 42 #8 (1999), 650–673, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/recursion-t.ps.gz⟩ doi:10.1093/comjnl/42.8.650.

[War99b] M. Ward, “Assembler to C Migration using the FermaT Transformation System,” *International Conference on Software Maintenance, 30th Aug–3rd Sept 1999, Oxford, England* (1999).

[War93] M. Ward, “Abstracting a Specification from Code,” *J. Software Maintenance: Research and Practice* 5 #2 (June, 1993), 101–122, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/prog-spec.ps.gz⟩.

[War96] M. Ward, “Derivation of Data Intensive Algorithms by Formal Transformation,” *IEEE Trans. Software Eng.* 22 #9 (Sept., 1996), 665–686, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/sw-alg.ps.gz⟩ doi:doi.ieeecomputersociety.org/10.1109/32.541437.

[War01] Martin Ward, “The FermaT Assembler Re-engineering Workbench,” *International Conference on Software Maintenance (ICSM), 6th–9th November 2001, Florence, Italy* (2001).

[War04] Martin Ward, “Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations,” *Science of Computer Programming, Special Issue on Program Transformation* 52 #1–3 (2004), 213–255, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/migration-t.ps.gz⟩ doi:dx.doi.org/10.1016/j.scico.2004.03.007.

[WaZ05] Martin Ward & Hussein Zedan, “MetaWSL and Meta-Transformations in the FermaT Transformation System,” *29th Annual International Computer Software and Applications Conference, Edinburgh, UK, November 2005* (2005).

[WaZ07] Martin Ward & Hussein Zedan, “Slicing as a Program Transformation,” *Trans. Programming Lang. and Syst.* 29 #2 (Apr., 2007), 1–52, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/slicing-t.ps.gz⟩ doi:doi.acm.org/10.1145/1216374.1216375.

[WaZ10] Martin Ward & Hussein Zedan, “Deriving a Slicing Algorithm via FermaT Transformations,” *IEEE Trans. Software Eng.*, IEEE computer Society Digital Library (Jan., 2010), ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/derivation2-a4-t.pdf⟩ doi:doi.ieeecomputersociety.org/10.1109/TSE.2010.13.

[WZH04] Martin Ward, Hussein Zedan & Tim Hardcastle, “Legacy Assembler Reengineering and Migration,” *20th IEEE International Conference on Software Maintenance, 11th-17th Sept Chicago Illinois, USA.* (2004).

[WZL08] Martin Ward, Hussein Zedan, Matthias Ladkau & Stefan Natelberg, “Conditioned Semantic Slicing for Abstraction; Industrial Experiment,” *Software Practice and Experience* 38 #12 (Oct., 2008), 1273–1304, ⟨http://www.cse.dmu.ac.uk/~mward/martin/papers/slicing-paper-final.pdf⟩ doi:doi.wiley.com/10.1002/spe.869.

[Wir71] N. Wirth, “Program Development by Stepwise Refinement,” *Comm. ACM* 14 #4 (1971), 221–227.

[YaW03] H. Yang & M. Ward, *Successful Evolution of Software Systems*, Artech House, Boston, London, 2003, ISBN-10 1-58053-349-3 ISBN-13 978-1580533492.

[ZMH02a] Xingyuan Zhang, Malcolm Munro, Mark Harman & Lin Hu, “Mechanized Operational Semantics of WSL,” *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, Los Alamitos, California, USA (2002).

[ZMH02b] Xingyuan Zhang, Malcolm Munro, Mark Harman & Lin Hu, *Weakest Precondition for General Recursive Programs Formalized in Coq*, Lect. Notes in Comp. Sci., Springer-Verlag, New York–Heidelberg–Berlin, 2002, Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs).