VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Program Derivation in the Refinement Calculus: An Introduction

Lindsay Groves*

Technical Report CS-TR-94/21
December 1994

Abstract

The refinement calculus is a formal calculus for deriving imperative programs from logical specifications, which formalises program construction by stepwise refinement. This paper introduces the basic concepts underlying the refinement calculus: a wide-spectrum language including non-executable constructs for expressing specifications, a refinement ordering on specifications/programs in this language, and a set of rules embodying laws about certain refinements that are always valid. The paper also shows how data refinement is formalised within the refinement calculus, presents some important theoretical results, and illustrates the techniques by deriving an algorithm to find the minimum and maximum elements of a set, using only $3n/2$ comparisons, and then data refining this to an array representation.

Publishing Information

Appeared in: Steve Reeves (Ed), *Proceedings of the First New Zealand Formal Program Development Colloquium*, Waikato University, November 1994, pp 25–42.

*Email: `lindsay@comp.vuw.ac.nz`.

1 Introduction

The refinement calculus is a calculus for deriving imperative programs from logical specifications. Programs are derived in a wide-spectrum language including both executable and non-executable constructs. A derivation typically begins with a non-executable specification, and progresses via a number of intermediate forms combining both executable and non-executable constructs, ending up with a program containing only executable constructs.

In this way, the refinement calculus can be seen as a formalisation of stepwise refinement [30]. At each step in the derivation, some decision is made which takes the program nearer to an executable form, possibly giving rise to certain proof obligations. These proof obligations can be discharged as they arise, so that the program and proof are developed hand-in-hand, as advocated by Dijkstra [8], rather than the proof being attempted once the program has been completed. In fact, the refinement calculus doesn't have to be used in this way, and can also be used effectively to verify existing programs.

The refinement calculus has its origins in the early work of Ralph Back [1, 2], which was not widely known at the time, and later work by Carroll Morgan and others [17, 24, 25], and Joe Morris [26]. It can be seen as an extension of Dijkstra's program calculus [9, 10, 12], in which the partial programs corresponding to steps in the derivation, and the relationships between them, are made explicit. The refinement calculus is also related to formal development methods used with VDM [16] and Z [31].

As well as formalising stepwise refinement, the refinement calculus provides a framework within which a number of aspects of software development can be investigated. For example, data refinement, which was first formalised by Hoare [14] and explored within VDM [15], has flourished in the context of the refinement calculus [5, 18, 19, 23, 11, 27, 7, 32]. Furthermore, the refinement calculus allows simpler proof obligations than are found, for example, with VDM and Z, and admits a more "calculational" style of development. The refinement calculus lends itself well to mechanical support, and the development of interactive refinement tools is now an active field. The refinement calculus also opens up new avenues for theoretical investigation, such as the lattice theoretic properties investigated by Back and von Wright [4].

In this paper we give a brief introduction to the refinement calculus. We begin by introducing a wide-spectrum language, along with its semantics, and discuss healthiness conditions. We then introduce the refinement relation which relates successive steps in a derivation, and explore some of its properties. Next, we introduce the idea of refinement rules, and illustrate these ideas by presenting a sample derivation. We then discuss data refinement briefly, and end with a few concluding remarks.

2 A Wide-Spectrum Language

The wide-spectrum language consists of an ordinary programming language, augmented with constructs for expressing specifications, which are (in general) not executable. The executable subset of the language is usually a variant of Dijkstra's guarded commands language, with semantics based on weakest precondition predicate transformers [9]. We refer to all constructs in this language as *programs*, though we sometimes call them *specifications* when we expect them to be non-executable; executable programs are called *code*.

For any program S and predicate R, $wp(S,\ R)$ is the *weakest precondition* for S to establish R, i.e. the weakest condition that must hold in the initial state to ensure that S will terminate in a state satisfying R. A program which establishes *true* always terminates (i.e.

can terminate in any state); no program can establish *false* (i.e. terminate in no state), except by magic (see Section 2.2). If $wp(S,\ R) = \mathit{false}$, then S cannot be guaranteed to terminate in a state satisfying R, so for any initial state, S may do anything, including not terminate. Note that wp cannot distinguish between a program which always fails to terminate and one that only sometimes fails to terminate.

2.1 Executable constructs

The executable sublanguage we use is essentially Dijkstra's guarded commands language, with the addition of (untyped) local variable declarations. The weakest precondition semantics for these constructs are given in Figure 1.

$$
\begin{array}{l}
wp(\mathbf{skip},\ R) \ \widehat{=}\ R \\
wp(\mathbf{abort},\ R) \ \widehat{=}\ \mathit{false} \\
wp(\tilde{x} := \tilde{e},\ R) \ \widehat{=}\ R_{[\tilde{x} \setminus \tilde{e}]} \\
\quad \text{where } R_{[\tilde{x} \setminus \tilde{e}]} \text{ is } R \text{ with all free } x\text{s replaced by the corresponding } e\text{s} \\
wp(S_1\ ;\ S_2,\ R) \ \widehat{=}\ wp(S_1,\ wp(S_2,\ R)) \\
wp(\mathbf{if}\ \, [\!]_i\ B_i \rightarrow S_i\ \mathbf{fi},\ R) \ \widehat{=}\ (\bigvee_i B_i) \wedge \bigwedge_i (B_i \Rightarrow wp(S_i,\ R)) \\
wp(\mathbf{do}\ \, [\!]_i\ B_i \rightarrow S_i\ \mathbf{od},\ R) \ \widehat{=}\ \mu\, X \bullet (\bigwedge_i (B_i \Rightarrow wp(S_i,\ X))) \wedge\ (\bigvee_i B_i \vee R) \\
wp([\![\ \mathbf{var}\ \ \tilde{x} \bullet S]\!],\ R) \ \widehat{=}\ \forall \tilde{x}\ \bullet\ wp(S,\ R)
\end{array}
$$

Figure 1: *wp* definitions for executable constructs

There are three primitive statements: **skip** (do nothing), multiple assignment $\tilde{x} := \tilde{e}$ (where $\tilde{x}$ is a list of distinct variables, and $\tilde{e}$ is a list of expressions with the same length as $\tilde{x}$), and **abort** (do anything, including not terminate — usually to be avoided!). The composite statements are: sequential composition (**;**), alternation (**if**), repetition (**do**), and block with local variables.

As in Dijkstra's language, the alternation and repetition statements are nondeterministic. The statement $\mathbf{if}\ [\!]_i\ B_i \rightarrow S_i\ \mathbf{fi}$[1] is executed by choosing any true guard B_j and executing the corresponding statement S_j. The choice is nondeterministic if more than one guard is true; the statement is equivalent to **abort** if no guard is true. The **do** statement is similar, except that after executing S_j it returns to choose another guard, and exits (i.e. is equivalent to **skip**) if no guard is true.

The semantics of a block simply says that the meaning of the body of the block does not depend on the initial values of the new variables. This simple language can be extended to include constructs such as procedures, parameters, recursion and modules (e.g. see [20], [25]).

We are deliberately vague about what data types and operations are supported in the executable sublanguage. We generally assume that integers (or naturals), Booleans, and arrays of integers or Booleans, and the usual operations upon them, are available, but make other assumptions as appropriate. For example, when deriving an abstract algorithm, we may assume that certain abstract types (e.g. sets) and suitable operations upon them (e.g.

[1] $\mathbf{if}\ [\!]_i\ B_i \rightarrow S_i\ \mathbf{fi}$ is an abbreviation for $\mathbf{if}\ B_1 \rightarrow S_1\ [\!]\ \cdots\ [\!]\ B_n \rightarrow S_n\ \mathbf{fi}$.

union, intersection, etc.) are available, and perhaps additional primitive statements. We might then remove these in a subsequent data refinement (see Section 6). Defining functions and data types can be seen as extending the base language in which the program is to be expressed, rather than as part of constructing that program.

The definitions given in Figure 1 assume that expressions (and guards) are always defined, i.e. all variables have values and all functions used are total. In the absence of this assumption, we would need to introduce additional conjuncts to ensure that expressions and guards are defined. To ensure that the resulting predicates are well-defined when these conjuncts are false, we would also need to introduce conditional connectives, such as **cand** [10, 12], or use a three-valued logic (e.g. [6]. Assuming that expressions are always defined simplifies the semantics, and also simplifies refinement rules and the resulting proof obligations. This assumption is quite plausible when discussing specifications, but can lead to some odd anomalies, and does not capture the semantics of most real programming languages, where evaluating a division by zero or an array access with an out-of-bounds index will cause the program to terminate abnormally.

2.2 Non-executable constructs

The wide-spectrum language also provides a number of constructs which are useful for expressing specifications, but are (in general) non-executable. A variety of such constructs are used in different versions of the refinement calculus. We show some of the variations below, but generally we use the version described by Morgan [22]. The weakest preconditions for these constructs are given in Figure 2.

$$\begin{aligned}
&wp(\tilde{w}\colon \left[P \;\middle/\; Q \right],\ R) \;\widehat{=}\; P \wedge (\forall \tilde{w} \bullet Q \Rightarrow R)\\
&wp(\tilde{w}\colon \left[Q \right],\ R) \;\widehat{=}\; (\forall \tilde{w} \bullet Q \Rightarrow R)\\
&wp(\mathbf{magic},\ R) \;\widehat{=}\; \mathit{true}\\
&wp(\{P\},\ R) \;\widehat{=}\; P \wedge R\\
&wp([P],\ R) \;\widehat{=}\; P \Rightarrow R\\
&wp(\llbracket\ \mathbf{con}\ \ \tilde{x} \bullet S \rrbracket,\ R) \;\widehat{=}\; \exists \tilde{x} \bullet wp(S,\ R)
\end{aligned}$$

Figure 2: *wp* definitions for non-executable constructs

- Specification statement: $\tilde{w}\colon \left[P \;\middle/\; Q \right]$
  Establish Q, changing only variables in $\tilde{w}$, provided P holds initially. The precondition, P, and postcondition, Q, are predicates over state variables; the frame, $\tilde{w}$, is a list of distinct variables.
- Nondeterministic assignment: $\tilde{w}\colon \left[Q \right]$
  Assign values to variables in $\tilde{w}$ so as to establish Q, if such values exist[2].

[2]The weakest precondition for $\tilde{w}\colon \left[Q \right]$ is sometimes given as $(\exists \tilde{w} \bullet Q) \wedge (\forall \tilde{w} \bullet Q \Rightarrow R)$, which makes $\tilde{w}\colon \left[Q \right]$ abort when there are no values for $\tilde{w}$ which satisfy Q, whereas the definition given here makes it miraculous.

- Magic: **magic**
  Do magic — makes any postcondition true!
- Assertion: $\{P\}$
  Assert that P holds; abort if it doesn't.
- Coercion: $[P]$
  Force P to be true, doing magic if necessary.
- Logical constant: $[\![\ \mathbf{con}\ \ \tilde{x} \bullet S]\!]$
  Execute S with values for variables in $\tilde{x}$ chosen so to make any preconditions, including implicit ones, in S true (if possible).

Expressions in assignment statements and guards may also be non-executable because they involve data types and/or operators which are not supported by the executable sublanguage. For example, guards may contain quantifiers. As with the non-executable constructs above, these must be removed in order to obtain an implementation.

These constructs allow us to write specifications that cannot possibly be satisfied, for example, the specification $x\colon \left[\, \mathit{true} \;\middle/\; \mathit{false} \,\right]$, which is equivalent to **magic**. Such specifications are said to be miraculous or infeasible, where feasibility is defined as follows:

Definition A program S is *feasible* if $wp(S, \mathit{false}) = \mathit{false}$.

All of the executable constructs (i.e. code) are feasible, but some of the non-executable ones are not. Allowing infeasible statements, such as **magic**, is convenient in developing the theory of program refinement. These statements have some interesting properties, and can also be useful as an intermediate step in program derivations [18]. Part of the task of program derivation is to show that the specification is feasible, i.e. that it does not require magic.

It is often necessary for the postcondition of a specification statement to refer to the initial value of a variable as well as its final value. This can be expressed using the convention that zero-subscripted variables refer to initial values. Thus, $x\colon \left[\, \mathit{true} \;\middle/\; x > x_0 \,\right]$ specifies a program that will increase the value of x by an arbitrary amount. This convention can be formalised by the use of logical constants; the above specification is equivalent to $[\![\ \mathbf{con}\ \ c \bullet x\colon \left[\, x = c \;\middle/\; x > c \,\right]]\!]$. Our weakest precondition definitions all ignore the possibility of zero-subscripted variables, since they can always be removed in this way.

2.3 Other extensions

Some versions of the refinement calculus also split Dijkstra's **if** and **do** statements into simpler and more general constructs by treating $\rightarrow$, $[\!]$, **if**–**fi** and **do**–**od** as separate constructors. We may also add the angelic choice operator $\diamond$. The weakest preconditions for these constructs are given in Figure 3.

The construct $B \rightarrow S$ is sometimes called a "naked guarded command". It says behave like S when B holds, otherwise do magic. Since it doesn't always require magic, this is sometimes also called a "minor miracle".

The construct $S \;[\!]\; T$ says choose either S or T to execute; either choice must then establish the required postcondition. Because we must be prepared for the worst, this is called "demonic choice".

By contrast, $S \diamond T$ says choose either S or T to execute, but only one of them needs to establish the required postcondition and the right one will be chosen. This is called "angelic

$$
\begin{aligned}
&wp(B \rightarrow S,\ R) \ \widehat{=}\ B \Rightarrow wp(S,\ R)\\
&wp(S \mathbin{[\!]} T,\ R) \ \widehat{=}\ wp(S,\ R) \wedge wp(T,\ R)\\
&wp(S \Diamond T,\ R) \ \widehat{=}\ wp(S,\ R) \vee wp(T,\ R)\\
&wp(\mathbf{if}\ \ S\ \mathbf{fi},\ R) \ \widehat{=}\ grd(S) \wedge wp(S,\ R)\\
&wp(\mathbf{do}\ S\ \mathbf{od},\ R) \ \widehat{=}\ \mu X \bullet wp(S,\ X) \wedge (grd(S) \vee R)\\
&\quad \text{where } grd(S) \mathrel{\widehat{=}} \neg wp(S,\ false)
\end{aligned}
$$

Figure 3: *wp* definitions for generalized program constructors

choice"[3].

The effect of wrapping **if–fi** around a statement S is a statement which will execute S if it is feasible, and abort if S is miraculous. Similarly, **do** S **od** is a statement which repeats S as long as it is feasible, stopping when S becomes miraculous.

Breaking the language down into simpler constructors like this allows a number of very elegant lattice-theoretic properties to be obtained (see Section 3).

2.4 Healthiness conditions

Dijkstra [10] proposed a number of "healthiness conditions" that any computational mechanism should obey. These can be expressed in terms of the following properties[4]:

Strictness: S is *strict* if $wp(S,\ false) \Longleftrightarrow false$.

Monotonicity: S is *monotonic* (with respect to $\Longrightarrow$) if $Q \Longrightarrow R$ implies $wp(S,\ Q) \Longrightarrow wp(S,\ R)$.

Disjunctivity: S is *disjunctive* if $wp(S,\ Q \vee R) \Longleftrightarrow wp(S,\ Q) \vee wp(S,\ R)$.

Conjunctivity: S is *conjunctive* if $wp(S,\ Q \wedge R) \Longleftrightarrow wp(S,\ Q) \wedge wp(S,\ R)$.

Continuity: S is *continuous* if for any infinite chain of predicates $R_0, R_1, \cdots$, such that $R_i \Longrightarrow R_{i+1}$, $wp(S,\ \bigvee_i R_i) \Longleftrightarrow \bigvee_i wp(S,\ R_i)$.

Dijkstra's healthiness conditions state that any computational mechanism S must be strict (this is known as the "Law of the Excluded Miracle"), monotonic, conjunctive and continuous. A mechanism that is nondeterministic is not disjunctive, so Dijkstra used a weaker version of disjunctivity ($wp(S,\ Q) \vee wp(S,\ R) \Longrightarrow wp(S,\ Q \vee R)$), which follows from monotonicity. Continuity was required in order to avoid unbounded nondeterminism; this has since been discarded by most authors.

The constructs added in the refinement calculus further challenge these healthiness conditions. For example, specification statements can be non-strict (since they can have *false* as their postconditiona, which is equivalent to **magic**), and logical constants are not conjunctive. This leaves monotonicity as the only healthiness condition intact. The executable constructs, however, are still strict and conjunctive.

[3] Angelic and demonic choice are rather like "don't know" and "don't care" nondeterminism, respectively, as discussed in the logic programming literature.

[4] We write $\Phi \Longleftrightarrow \Psi$ to mean that predicates Φ and Ψ are equivalent for all states, i.e. $(\forall \sigma \bullet \Phi\sigma \equiv \Psi\sigma)$, and $\Phi \Longrightarrow \Psi$ to mean that Φ implies Ψ for all states, i.e. $(\forall \sigma \bullet \Phi\sigma \Rightarrow \Psi\sigma)$.

3 The Refinement Relation

A key component of the refinement calculus is the refinement ordering, written $\sqsubseteq$, on programs. This relation is defined so that $S \sqsubseteq S'$, read "S is refined by S'", if a client who asks for S will be happy if provided with code satisfying S'.

We define "satisfies" in the usual way (following Dijkstra):

Definition A program S *satisfies* a specification with precondition P and postcondition R iff $P \Rrightarrow wp(S,\ R)$.

This still applies when S is any construct in the wide-spectrum language. We now define $\sqsubseteq$ as follows:

Definition $S \sqsubseteq S'$ iff for any postcondition R, $wp(S,\ R) \Rrightarrow wp(S',\ R)$.

It is easy to see that with this definition, S' satisfies any specification that S satisfies, and any implementation of S' is also an implementation of S.

If $S \sqsubseteq S'$, then S' can differ from S in either (or both) of two ways: S' may terminate more often than S (have a weaker precondition), or S' may be less deterministic than S (have a stronger postcondition). Note that the richness of this relationship is a consequence of the class of programs being considered: if programs were always total, it would not be possible to weaken preconditions; if programs were always deterministic (i.e. functions), it would not be possible to strengthen postconditions.

To derive a program T from a specification S, we construct a sequence of programs $S_0, \cdots, S_n$, such that $S_0 = S$, $S_{i-1} \sqsubseteq S_i$ for $i = 1, \cdots, n$, and $S_n = T$ is executable (i.e. code).

In order for this stepwise approach to work, we must have the following:

Theorem Transitivity of $\sqsubseteq$. If $S \sqsubseteq S'$ and $S' \sqsubseteq S''$, then $S \sqsubseteq S''$.

This allows us to construct a program via a sequence of steps, each of which is a refinement of the previous one, and be assured that the final program is a refinement of the initial one.

In order to be able to refine components of a program individually, we also require the following:

Theorem Monotonicity of program constructors (with respect to $\sqsubseteq$). Let $\mathcal{F}[S]$ be a program with S as a component, and $\mathcal{F}[S']$ be the result of replacing the designated occurrence of S in $\mathcal{F}[S]$ by S'. If $S \sqsubseteq S'$, then $\mathcal{F}[S] \sqsubseteq \mathcal{F}[S']$. More specifically:

If $S \sqsubseteq S'$ and $T \sqsubseteq T'$ then:

$$\begin{array}{rcl}
S\ ;\ T & \sqsubseteq & S'\ ;\ T' \\
\textbf{if}\ \ A \rightarrow S\ [\!]\ B \rightarrow T\ \textbf{fi} & \sqsubseteq & \textbf{if}\ \ A \rightarrow S'\ [\!]\ B \rightarrow T'\ \textbf{fi} \\
\textbf{do}\ A \rightarrow S\ [\!]\ B \rightarrow T\ \textbf{od} & \sqsubseteq & \textbf{do}\ A \rightarrow S'\ [\!]\ B \rightarrow T'\ \textbf{od} \\
[\![\ \textbf{var}\ \ \tilde{x} \bullet S]\!] & \sqsubseteq & [\![\ \textbf{var}\ \ \tilde{x} \bullet S']\!] \\
[\![\ \textbf{con}\ \ \tilde{x} \bullet S]\!] & \sqsubseteq & [\![\ \textbf{con}\ \ \tilde{x} \bullet S']\!]
\end{array}$$

When treated as separate constructors, $\rightarrow$, $[\!]$ and $\Diamond$ are monotonic; but **if**–**fi** and **do**–**od**, as defined above, are not — they must therefore be used with great care.

The refinement ordering has a number of other interesting properties. In particular, it forms a complete lattice with **abort** and **magic** as extreme elements, and $[\!]$ and $\Diamond$ as the meet and join operations.

Theorem For any program S, **abort** $\sqsubseteq S \sqsubseteq$ **magic**.

Theorem For all programs S, T and U, if $S \sqsubseteq T$ and $S \sqsubseteq U$, then $S \sqsubseteq T\ [\!]\ U$.

Theorem For all programs S, T and U, if $S \sqsubseteq U$ and $T \sqsubseteq U$, then $S \Diamond T \sqsubseteq U$.

Now we see why it is important in deriving programs that we find T such that $S \sqsubseteq T$ and T is feasible — otherwise we could just refine everything to **magic**!

We can also find analogues of many other algebraic and lattice theoretic concepts, such as duals, inverses and adjoints (see [4]).

4 Refinement rules

Another important feature of the refinement calculus is a collection of rules embodying laws showing that certain refinements are always valid. These allow us to derive programs without having to reason directly about weakest preconditions (except where it is convenient to do so). We write refinement rules as inference rules with the proof obligations or applicability conditions above the line, and the inferred refinement below.

As a basis, we need a rule to turn a specification statement into each of the executable constructs in the language:

Introduce SKIP:

$$\frac{P \Rrightarrow Q}{\tilde{w}\colon \left[P \,\middle/\, Q \right] \sqsubseteq \mathbf{skip}}$$

Introduce Assignment:

$$\frac{\text{Every variable in } \tilde{x} \text{ occurs in } \tilde{w} \qquad P \Rrightarrow Q_{[\tilde{x} \setminus \tilde{e}]}}{\tilde{w}\colon \left[P \,\middle/\, Q \right] \sqsubseteq \tilde{x} := \tilde{e}}$$

Split Specification:

$$\tilde{w}\colon \left[P \,\middle/\, Q \right] \sqsubseteq \tilde{w}\colon \left[P \,\middle/\, M \right] \mathbin{;} \tilde{w}\colon \left[M \,\middle/\, Q \right]$$

Introduce IF:

$$\frac{P \Rrightarrow \bigvee_i B_i}{\tilde{w}\colon \left[P \,\middle/\, Q \right] \sqsubseteq \mathbf{if} \quad [\!]_i\, B_i \rightarrow \tilde{w}\colon \left[P \wedge B_i \,\middle/\, Q \right] \;\; \mathbf{fi}}$$

Introduce DO:

$$\frac{P \Rrightarrow I \qquad I \wedge \neg(\bigvee_i B_i) \Rrightarrow Q}{\tilde{w}\colon \left[P \,\middle/\, Q \right] \sqsubseteq \mathbf{do} \quad [\!]_i\, B_i \rightarrow \tilde{w}\colon \left[I \wedge B_i \,\middle/\, I \wedge 0 \leq t < t_0 \right] \;\; \mathbf{od}}$$

where I is the loop invariant, t is the variant function, and t_0 is obtained from t by replacing each variable x in $\tilde{w}$ by x_0.

Introduce Local Variables:

$$\frac{\tilde{x} \text{ are fresh names}}{\tilde{w}\colon \left[P \,\middle/\, Q \right] \sqsubseteq [\![\;\mathbf{var}\;\; \tilde{x} \bullet \tilde{x}, \tilde{w}\colon \left[P \,\middle/\, Q \right]]\!]}$$

We also have three rules for manipulating specification statements:

Contract Frame:

$$\frac{\text{All variables in } \tilde{w} \text{ are in } \tilde{w}}{\tilde{w}\colon \left[P \,\middle/\, Q \right] \sqsubseteq \tilde{w'}\colon \left[P \,\middle/\, Q \right]}$$

Weaken Precondition:

$$\frac{P \Rrightarrow P'}{\tilde{w}\colon \left[P \;\middle/\; Q \right] \sqsubseteq \tilde{w}\colon \left[P' \;\middle/\; Q \right]}$$

Strengthen Postcondition:

$$\frac{Q' \Rrightarrow Q}{\tilde{w}\colon \left[P \;\middle/\; Q \right] \sqsubseteq \tilde{w}\colon \left[P \;\middle/\; Q' \right]}$$

It is often useful to have a rule which splits a specification into two parts, where we know what the second part is and can used *wp* to calculate the intermediate assertion. For example, if the second component is an assignment, we have:

Following Assignment:

$$\tilde{w}\colon \left[P \;\middle/\; Q \right] \sqsubseteq \tilde{w}\colon \left[P \;\middle/\; Q_{[\tilde{x} \setminus \tilde{e}]} \right] ; \tilde{x} := \tilde{e}$$

We also need rules for removing other non-executable constructs. For example, the following rule allows logical constants to be removed:

Remove Logical Constant:

$$\frac{\text{No variable in } \tilde{x} \text{ is free in } S}{[\![\ \mathbf{con} \ \ \tilde{x} \bullet S]\!] \sqsubseteq S}$$

Most of these rules are closely related to corresponding rules in Hoare's logic (or a total correctness version thereof), and to Dijkstra's *wp* rules and associated techniques. If functions are not assumed to be total, we need additional proof obligations in the rules for assignment, **if** and **do**, to ensure that expressions and guards are defined.

The *Weaken Precondition* and *Strengthen Postcondition* rules are only required when the new precondition (postcondition) is strictly weaker (stronger) — we allow equivalence preserving transformations without the explicit application of a rule.

Many variations of these rules are possible. For example, the laws given in [24] typically have fewer proof obligations, but apply to more restrictive forms of specification statement. For instance, their rule for *Introduce Assignment* (ignoring zero-subscripts) is:

$$\tilde{w}\colon \left[Q_{[\tilde{w} \setminus \tilde{e}]} \;\middle/\; Q \right] \sqsubseteq \tilde{w} := \tilde{e}$$

With those rules, there is frequent need for *Weaken Precondition*, *Strengthen Postcondition* and *Contract Frame*. Morgan [22] also gives different versions of some laws according to whether zero-subscripted variables are allowed and where. We omit zero-subscripted variables as they can be handled using logical constants.

We can also define a large number of rules which perform various kinds of transformations (thus refinement encompasses program transformation). In a sense, such rules are strictly unnecessary, since any program that can be derived using them can also be derived without them. These rules do, however, become important in conjunction with data refinement (see Section 6). Morgan [22] lists many more rules — far more than anyone would want to remember! In performing derivations, we may wish to introduce new rules corresponding to particular derivation patterns that occur frequently, or to introduce new primitive statements that are used.

5 Example derivation

To illustrate the way programs are derived in the refinement calculus, we will derive an algorithm to find the minimum and maximum elements in a non-empty set, S. The algorithm we derive is one which requires $3n/2$ comparisons, rather than the $2n$ required by the "obvious" algorithm. To avoid a few messy details, we assume that S has an odd number of elements.

To simplify the notation, we will write $u \leq S \leq v$ to mean that u and v are the minimum and maximum, respectively, of S, i.e.

$$u \leq S \leq v \;\widehat{=}\; \{u, v\} \subseteq S \wedge (\forall x \in S \bullet u \leq x \wedge x \leq v)$$

The specification can now be written as a specification statement:

$$lo, hi\colon \left[odd(\,|S|\,) \;\middle/\; lo \leq S \leq hi \right] \tag{1}$$

Since the problem involves sets, we will assume that certain operations on sets are available. In particular, we will freely use set union and difference[5]. We will also assume a statement, written $x :\in S$, which selects an arbitrary element from a set S and assigns it to x. This statement is equivalent to the nondeterministic assignment $x\colon \left[x \in S \right]$, which in turn is equivalent to $x\colon \left[S \neq \varnothing \;\middle/\; x \in S \right]$, so a suitable refinement rule is:

Introduce Set Selection:

$$\frac{\begin{array}{c} x \text{ occurs in } w \\ P \Rrightarrow S \neq \varnothing \\ (\forall x \bullet x \in S \Rightarrow Q) \end{array}}{w\colon \left[P \;\middle/\; Q \right] \sqsubseteq x :\in S}$$

We will clearly need a loop, so we first look for a loop invariant. We can obtain a loop invariant by weakening the postcondition [12], so that lo and hi are the minimum and maximum of some (non-empty) subset of S, i.e.

$$Inv \;\widehat{=}\; T \subseteq S \wedge lo \leq T \leq hi$$

Since we replaced S by T to obtain Inv, we will take $T \neq S$ as the loop guard, and $|S - T|$ as the variant function. As S cannot change, we will not include the precondition $odd(\,|S|\,)$ explicitly in the loop invariant or other assertions, but will assume it to hold globally and draw on it as required; this can be formalised by the use of invariants (see [21]).

We first introduce T as a local variable:

$$\begin{array}{lll} (1) & \sqsubseteq & (\textit{Introduce Local Variables}) \\ & & \textbf{var } T \bullet \\ & & lo, hi, T\colon \left[odd(\,|S|\,) \;\middle/\; lo \leq S \leq hi \right] \end{array} \tag{2}$$

Since the loop invariant must be established, we split (2) into a sequence, with Inv as the intermediate assertion:

[5] We won't assume that set minimum and set maximum are available — that would defeat the purpose of the exercise!

$$(2) \sqsubseteq (\textit{Split Specification})$$

$$lo, hi, T\colon \left[odd(\,|S|\,) \;\big/\; Inv \right]; \qquad (3)$$

$$lo, hi, T\colon \left[Inv \;\big/\; lo \le S \le hi \right] \qquad (4)$$

We can easily establish *Inv* by taking T to be any singleton subset of S, and its single member as both minimum and maximum:

$$(3) \sqsubseteq (\textit{Strengthen Postcondition})$$

$$lo, hi, T\colon \left[odd(\,|S|\,) \;\big/\; \exists x \in S \bullet lo = hi = x \wedge T = \{x\} \right] \qquad (5)$$

This step has a simple proof obligation:

$$(\exists x \in S \bullet lo = hi = x \wedge T = \{x\}) \Rrightarrow T \subseteq S \wedge lo \le T \le hi$$

In a similar manner, we can refine (5) to:

$$\begin{array}{l} |[\ \mathbf{var}\ x \bullet \\ \quad x :\in S; \\ \quad lo, hi, T := x, x, \{x\} \\ ]| \end{array}$$

The proof obligation for this refinement reduces to $S \neq \emptyset$, which follows from the precondition, $odd(\,|S|\,)$.

We now turn our attention to (4), which we refine to a loop. As noted above, we take $T \neq S$ as the guard, and $|S - T|$ as the variant:

$$(4) \sqsubseteq (\textit{Introduce DO})$$

$$\begin{array}{l} \mathbf{do}\ T \neq S \rightarrow \\ \quad lo, hi, T\colon \left[Inv \wedge T \neq S \;\big/\; Inv \wedge 0 \le |S - T| < |S - T_0| \right] \qquad (6) \\ \mathbf{od} \end{array}$$

The proof obligations for this refinement are also trivial:

$$Inv \Rrightarrow Inv$$

$$Inv \wedge \neg(T \neq S) \Rrightarrow lo \le S \le hi$$

Now, in order for the loop to "make progress", the variant must be reduced, i.e. the size of T must be increased. An obvious choice would be to increase the size of T by one, which would lead to the "obvious" algorithm referred to above. We will do something a little less obvious, however, and increase the size of T by two.

We simply need to select two distinct elements from $S - T$, and compare the smaller with lo and the larger with hi. This will require two local variables:

$$(6) \sqsubseteq (\textit{Introduce Local Variables})$$

$$\begin{array}{l} \mathbf{var}\ u, v \bullet \\ lo, hi, T, u, v\colon \left[Inv \wedge T \neq S \;\big/\; Inv \wedge 0 \le |S - T| < |S - T_0| \right] \qquad (7) \end{array}$$

We now split (7), so as to first select u and v so that $u < v$, adjust lo and hi if necessary, and then update T:

$$(7) \;\sqsubseteq\; (\textit{Split Specification} \text{ (twice) and } \textit{Contract Frame})$$

$$u, v\colon \left[\mathit{Inv} \wedge T \neq S \;\middle/\; \mathit{Inv} \wedge \{u, v\} \subseteq S - T \wedge u < v \right]; \tag{8}$$

$$lo, hi\colon \left[\mathit{Inv} \wedge \{u, v\} \subseteq S - T \wedge u < v \;\middle/\; \begin{array}{c} T \subseteq S \wedge \{u, v\} \subseteq S - T \wedge \\ lo \leq T \cup \{u, v\} \leq hi \end{array} \right]; \tag{9}$$

$$T\colon \left[\begin{array}{c} T \subseteq S \wedge \{u, v\} \subseteq S - T \wedge \\ lo \leq T \cup \{u, v\} \leq hi \end{array} \;\middle/\; \mathit{Inv} \wedge 0 \leq |S - T| < |S - T_0| \right] \tag{10}$$

We wish to refine (8) as follows, assuming binary operators **min** and **max**:

$$(8) \;\sqsubseteq\; (\textit{Split Specification}, \textit{Introduce Set Selection} \text{ and } \textit{Introduce Assignment})$$

$$\begin{array}{l} u :\in S - T; \\ v :\in S - T - \{v\}; \\ u, v := u \;\mathbf{min}\; v, u \;\mathbf{max}\; v \end{array}$$

The proof obligation for this reduces to $T \subset S \Rrightarrow |S - T| \geq 2$. Unfortunately, we cannot prove this!

At this point we observe that, because of the way we have chosen to implement (6), we can strengthen the loop invariant, adding a further conjunct to indicate that the size of T is odd. From this it follows (since $|S|$ is odd) that $|S - T|$ is even, so $T \subset S \Rrightarrow |S - T| \geq 2$ holds.

Thus, we modify the definition of *Inv* to:

$$\mathit{Inv} \;\widehat{=}\; T \subseteq S \wedge \mathit{odd}(\,|T|\,) \wedge lo \leq T \leq hi$$

We can easily check that the previous steps are not affected by this change, and the outstanding proof obligation can be discharged. This kind of revision is quite common in program derivations, and points to the need for tool support [28, 13]; an alternative approach would be to defer the choice of *Inv* until we knew precisely what constraints it needed to satisfy (see [29]).

Next, we refine (9) using the **min** and **max** operators:

$$(9) \;\sqsubseteq\; (\textit{Introduce Assignment})$$

$$lo, hi := lo \;\mathbf{min}\; u, hi \;\mathbf{max}\; v$$

This refinement has a simple proof obligation:

$$lo \leq T \leq hi \wedge u < v \Rrightarrow (lo \;\mathbf{min}\; u) \leq T \cup \{u, v\} \leq (hi \;\mathbf{max}\; v)$$

Finally, we refine (10) to an assignment:

$$(10) \;\sqsubseteq\; (\textit{Introduce Assignment})$$

$$T := T \cup \{u, v\}$$

Again, the proof obligation is straightforward.

This completes the derivation. We have shown that (1) refines to:

$$
\begin{array}{l}
[\![\ \mathbf{var}\ T \bullet \\
\quad [\![\ \mathbf{var}\ x \bullet \\
\qquad x :\in S; \\
\qquad lo, hi, T := x, x, \{x\} \\
\quad]\!]; \\
\quad \mathbf{do}\ T \neq S \rightarrow \\
\qquad\quad [\![\ \mathbf{var}\ u, v \bullet \\
\qquad\qquad u :\in S - T; \\
\qquad\qquad v :\in S - T - \{u\}; \\
\qquad\qquad u, v := u\ \mathbf{min}\ v, u\ \mathbf{max}\ v; \\
\qquad\qquad lo, hi := lo\ \mathbf{min}\ u, hi\ \mathbf{max}\ v; \\
\qquad\qquad T := T \cup \{u, v\} \\
\qquad\quad]\!] \\
\quad \mathbf{od} \\
]\!]
\end{array}
$$

If **min** and **max** are not considered to be executable, the two statements involving these operators can be further refined to **if** statements. For example:

$$
\begin{array}{ll}
 & u, v := u\ \mathbf{min}\ v, u\ \mathbf{max}\ v \\
\sqsubseteq & \mathbf{if}\ \ u < v \rightarrow \mathbf{skip} \\
 & [\!]\ \ u > v \rightarrow u, v := v, u \\
 & \mathbf{fi}
\end{array}
$$

Since we know that $u \neq v$, the two branches are mutually exclusive (and even if $u = v$ was possible, it wouldn't matter which branch was executed). Therefore, in a language with an **if–then–else** construct, this could be implemented using a single test; the assignment $lo, hi := lo\ \mathbf{min}\ u, hi\ \mathbf{max}\ v$, however, would still require two tests. Thus, the algorithm only requires $3n/2$ comparisons.

6 Data refinement

An important aspect of program derivation is the replacement of "abstract" variables by "concrete" representations, and expressions involving abstract variables by equivalents involving the concrete ones. This process, known as data refinement, can be formalised in the refinement calculus in a number of ways [7]. The version we present here is that of Morgan and Gardiner [23] and Morris [27]. More general accounts can be found in [11] and [32].

When abstract variables are replaced by concrete ones, we must indicate in what sense the resulting program is "equivalent" to the original. This is done by specifying a relationship between concrete and abstract variables, known as a *coupling invariant* or *abstraction invariant*, showing how the concrete variables are used to represent the abstract ones. In early work on data refinement, this relationship was assumed to be a function from concrete values to abstract (called the "retrieve function" in VDM literature [16]), and was also required to be "adequate" (i.e. onto); both requirements are relaxed in the current treatment.

We write $S \preceq_{\tilde{a},I,\tilde{c}} S'$ to mean that S' is obtained by data refining S, replacing abstract variables $\tilde{a}$ by concrete variables $\tilde{c}$, with coupling invariant I. This is defined formally as follows:

Definition $S \preceq_{\tilde{a},I,\tilde{c}} S'$ iff $(\exists \tilde{a} \bullet I \wedge wp(S,\ R)) \Rrightarrow wp(S',\ (\exists \tilde{a} \bullet I \wedge R))$, for all postconditions R not containing any variables in $\tilde{c}$.

Informally, this can be understood as saying that if an abstract state $\mathcal{A}'$ could be reached by executing S, starting in an abstract state $\mathcal{A}$ corresponding to concrete state $\mathcal{C}$, then $\mathcal{A}'$ corresponds to some state $\mathcal{C}'$ that could be reached by executing S' starting in $\mathcal{C}$.

With this definition, we get the following important relationship between data refinement and procedural refinement:

Theorem If $S \preceq_{\tilde{a},I,\tilde{c}} S'$ then $\llbracket\ \mathbf{var}\ \ \tilde{a} \bullet S \rrbracket \sqsubseteq \llbracket\ \mathbf{var}\ \ \tilde{c} \bullet S' \rrbracket$

This shows that we can refine a block by replacing the variables it declares by new ones and data refining the body using a suitable coupling invariant.

It is easy to see that any statement which does not contain any abstract variables data refines itself. In particular, we get:

$$\begin{aligned}
\mathbf{abort} &\preceq_{\tilde{a},I,\tilde{c}} \mathbf{abort} \\
\mathbf{magic} &\preceq_{\tilde{a},I,\tilde{c}} \mathbf{magic} \\
\mathbf{skip} &\preceq_{\tilde{a},I,\tilde{c}} \mathbf{skip} \\
\tilde{x} := \tilde{e} &\preceq_{\tilde{a},I,\tilde{c}} \tilde{x} := \tilde{e}, \quad \text{if } \tilde{x} \text{ and } \tilde{e} \text{ contain no } a\text{'s}
\end{aligned}$$

We also get a number of laws showing how data refinement distributes through various constructs:

If $S \preceq_{\tilde{a},I,\tilde{c}} S'$, $S_i \preceq_{\tilde{a},I,\tilde{c}} S_i'$ for $i = 1, \cdots$, and $\tilde{x}$ contains no a's, then:

$$\begin{aligned}
S_1;\ S_2 &\preceq_{\tilde{a},I,\tilde{c}} S_1';\ S_2' \\
\llbracket\ \mathbf{var}\ \ \tilde{x} \bullet S \rrbracket &\preceq_{\tilde{a},I,\tilde{c}} \llbracket\ \mathbf{var}\ \ \tilde{x} \bullet S' \rrbracket \\
\llbracket\ \mathbf{con}\ \ \tilde{x} \bullet S \rrbracket &\preceq_{\tilde{a},I,\tilde{c}} \llbracket\ \mathbf{con}\ \ \tilde{x} \bullet S' \rrbracket
\end{aligned}$$

We data refine a specification statement as follows:

$$\tilde{a}, \tilde{x}\colon \left[P \ \middle/\ Q \right] \ \preceq_{\tilde{a},I,\tilde{c}} \ \tilde{c}, \tilde{x}\colon \left[(\exists \tilde{a} \ \bullet \ I \wedge P) \ \middle/\ (\exists \tilde{a} \ \bullet \ I \wedge Q) \right]$$

When the coupling invariant is functional from concrete to abstract, i.e. I can be written as $\tilde{a} = F(\tilde{x}) \wedge H(\tilde{c})$, where H is an invariant on $\tilde{c}$ (sometimes called the "representation invariant"), this simplifies to:

$$\tilde{a}, \tilde{x}\colon \left[P \ \middle/\ Q \right] \ \preceq_{\tilde{a},I,\tilde{c}} \ \tilde{c}, \tilde{x}\colon \left[P[\tilde{a} := F(\tilde{c})] \wedge H(\tilde{c}) \ \middle/\ Q[\tilde{a} := F(\tilde{c})] \wedge H(\tilde{c}) \right]$$

Assignment statements, assertions and coercions can be treated as special cases of specification statements.

To data refine **if** and **do** statements, we need to replace the guards containing a's by ones involving c's, and data refine the component statements. A guard B_i is replaced by the new guard $(\forall \tilde{a} \ \bullet \ I \Rightarrow B_i)$. We must also ensure that whenever one of the abstract guards would have been true, one of the concrete guards is true. Thus, we get:

If $S_i \preceq_{\tilde{a},I,\tilde{c}} S_i'$, and $B_i' = (\forall \tilde{a} \bullet I \Rightarrow B_i)$ and $(\exists \tilde{a} \bullet I \wedge \bigvee_i B_i) \Rightarrow (\bigvee_i B_i')$, then:

$$\textbf{if} \ \textstyle[\!]_i \ B_i \rightarrow S_i \ \textbf{fi} \ \preceq_{a,I,c} \ \textbf{if} \ \textstyle[\!]_i \ B_i' \rightarrow S_i' \ \textbf{fi}$$

$$\textbf{do} \ \textstyle[\!]_i \ B_i \rightarrow S_i \ \textbf{od} \ \preceq_{a,I,c} \ \textbf{do} \ \textstyle[\!]_i \ B_i' \rightarrow S_i' \ \textbf{od}$$

Again, these can be simplified when I is functional, giving $B_i' = B_i[a := F(c)] \wedge H(c)$ as the new guards.

It can be shown that the above rules give the "weakest" data refinement in each case. [23]. That is, if the rules give S' as the data refinement of S, then for any program S'' such that $S \preceq_{a,I,c} S''$, we also have $S' \sqsubseteq S''$. Thus, we can treat these rules as "data refinement calculators", and perform data refinement by first applying the calculators and then performing ordinary refinement on the result.

7 Data Refinement Example

Suppose we wish to data refine the algorithm derived in Section 5, so that S is represented by an array A of size n, where $n = |S|$.

We can describe the relationship between S and A as:

$$S = \mathit{Elts}(A)$$

where *Elts* is a function which returns the set of elements in an array (or array segment).

Since T is always a subset of S, we also represent T using the same array, so long as its elements are contiguous. We will assume that the elements of T occupy an initial segment of A, described by a variable k, so that the following holds:

$$T = \mathit{Elts}(A[1..k]) \wedge 1 \leq k \leq n$$

where $A[1..k]$ is the array segment from $A[1]$ to $A[k]$, inclusive. It follows that $S - T = \mathit{Elts}(A[k+1..n])$.

Thus, our coupling invariant is:

$$I \ \widehat{=} \ S = \mathit{Elts}(A) \wedge T = \mathit{Elts}(A[1..k]) \wedge 1 \leq k \leq n$$

To perform the data refinement, we need to replace the declaration of T by a declaration of k. The declaration of S (somewhere global) will be replaced by the declaration of A. We then replace constructs involving S and T by equivalents involving A and k, as outlined in Section 6.

The set selection $x :\in S$ data refines to $x\colon \left[x \in \mathit{Elts}(A) \right]$, which can be refined by selecting any element of A and assigning it to x. At this point it is not clear which element of A we should choose.

The assignment $T := \{x\}$, however, data refines to $k\colon \left[\mathit{Elts}(A[1..k]) = \{x\} \right]$. We can refine this to $k := 1$, provided that $x = A[1]$.

Thus, we should refine $x\colon \left[x \in \mathit{Elts}(A) \right]$ to $x := A[1]$. To justify this formally, we need to propagate the fact that $x = A[1]$ using assertions and laws for pushing them through other statements, or using some other technique such as those described in [18] and [7].

We replace the guard $T \neq S$ by $\mathit{Elts}(A[1..k]) = \mathit{Elts}(A)$, which simplifies to $k \neq n$.

In a similar fashion, we can data refine the remaining statements involving S and/or T, making the following replacements:

$$\begin{array}{lcl} u :\in S - T & \longrightarrow & u := A[k+1] \\ v :\in S - T - \{u\} & \longrightarrow & v := A[k+2] \\ T := T \cup \{u, v\} & \longrightarrow & k := k+2 \end{array}$$

The resulting program is then:

$$\begin{array}{l}
[\![\ \mathbf{var}\ k \bullet \\
\quad [\![\ \mathbf{var}\ x \bullet \\
\qquad x := A[1]; \\
\qquad lo, hi, k := x, x, 1 \\
\quad]\!]; \\
\quad \mathbf{do}\ k \neq n \rightarrow \\
\qquad\quad [\![\ \mathbf{var}\ u, v \bullet \\
\qquad\qquad u := A[k+1]; \\
\qquad\qquad v := A[k+2]; \\
\qquad\qquad u, v := u\ \mathbf{min}\ v, u\ \mathbf{max}\ v; \\
\qquad\qquad lo, hi := lo\ \mathbf{min}\ u, hi\ \mathbf{max}\ v; \\
\qquad\qquad k := k+2 \\
\qquad\quad]\!] \\
\quad \mathbf{od} \\
]\!]
\end{array}$$

8 Conclusions

This paper has presented the basic elements of the refinement calculus, including data refinement, and illustrated its use in deriving a fairly simple algorithm. The use of nondeterminism in an abstract algorithm allows a number of different representations to be considered during data refinement. More details of the underlying theory and more extensive examples can be found in various sources listed in the bibliography.

Acknowledgements

Thanks to Peter Andreae, Gill Dobbie and Ray Nickson for their helpful comments on earlier versions of the paper.

References

[1] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.* Mathematical Centre Tracts 131, Mathematisch Centrum, Amsterdam, 1980.

[2] R. J. R. Back. "On correct refinements of programs". *Jl. Computer and System Sciences* **23** (1981), pp49–68.

[3] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica* **25** (1988), pp593–624.

[4] R. J. R. Back and J. von Wright. "Duality in specification languages: a lattice-theoretical approach". *Acta Informatica* **27** (1990), pp583–625.

[5] R. J. R. Back. *Data Refinement in the Refinement Calculus.* Technical Report Series A, No. 68, Inst. för Informationsbehandling, Åbo Akademi, Turku, Finland, 1988.

[6] H. Barringer *et al.* "A logic covering undefinedness in program proofs". *Acta Informatica* **21** (1984), pp251–269.

[7] W. Chen and J. Y. Udding. Towards a calculus of data refinement. *Mathematics of Program Construction'*, J. L. A. van de Snepscheut (Ed.), Springer-Verlag, Lecture Notes in Computer Science 375, 1989, pp197–218.

[8] E. W. Dijkstra. "A constructive approach to the problem of program correctness". *BIT* **8** (1968), pp174–186.

[9] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs", *Communications of the ACM*, **18** (August 1975), pp453–457.

[10] E. W. Dijkstra, *A Discipline of Programming*, Academic Press, 1976.

[11] P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science* **87** (1991), pp143–162. Also in [25], pp86–102.

[12] David Gries, *The Science of Programming*, Springer-Verlag, 1981.

[13] Lindsay Groves, Raymond Nickson and Mark Utting. "A Tactic Driven Refinement Tool". *Proc. 5th Refinement Workshop*, Springer-Verlag, 1992.

[14] C. A. R. Hoare. "Proof of correctness of data representations". *Acta Informatica* **1** (1972), pp271–281.

[15] Cliff Jones. *Software Development: A Rigorous Approach.* Prentice-Hall, 1981.

[16] Cliff Jones. *Systematic software development using VDM.* Prentice Hall, 1990.

[17] Carroll Morgan. The specification statement. *TOPLAS* **10**, 3 (July 1988). Also in [25], pp7–30.

[18] C. C. Morgan. Data refinement by miracles. *Inf. Proc. Lett.* **26**, 5 (Jan. 1988), pp243–246. Also in [25], pp72–78.

[19] Carroll Morgan. Auxiliary variables in data refinement. *Information Processing Letters* **29** (1988), pp293–296. Also in [25], pp79–85.

[20] Carroll Morgan. Procedures, parameters, and abstraction: separate concerns. *Sci. Comp. Prog.* **11** (1988). Also in [25], pp58–71.

[21] Carroll Morgan. Types and invariants in the refinement calculus. *Mathematics of Program Construction*, J. L. A. van de Snepscheut (Ed.), Lecture Notes in Computer Science, Vol. 375, Springer-Verlag, 1989.

[22] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[23] Carroll Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica* **27** (1990), pp481–503. Also in [25], pp103–134.

[24] Carroll Morgan and Ken Robinson. Specification statements and refinement. *IBM Jnl. Res. Dev.* **31**,5 (Sept. 1987). Also in [25], pp31–57.

[25] Carroll Morgan, Ken Robinson and Paul Gardiner. *On the Refinement Calculus*, Technical Monograph PRG-70, Oxfrod University, 1988.

[26] J. M. Morris. "A theoretical basis for stepwise refinement and the programming calculus". *Science of Computer Programming* **9** (1987), pp287–306.

[27] J. M. Morris. Laws of data refinement. *Acta Informatica* **26** (1989), 287–308.

[28] R. G. Nickson. *Tool Support for the Refinement Calculus.* PhD thesis, Victoria University of Wellington, 1994.

[29] Raymond G. Nickson and Lindsay J. Groves. "Metavariables and Conditional Refinements in the Refinement Calculus" *Proc. 6th Refinement Workshop*, British Computer Society, London, January 1994.

[30] Niklaus With. *Program development by stepwise refinement. C.A.C.M* **14** (April 1971), pp221–227.

[31] J. B. Wordsworth. *Software development with Z : a practical approach to formal methods in software engineering.* Addison-Wesley, 1992.

[32] J. von Wright. "The lattice of data refinements". *Acta Informatica* **31** (1994), pp105–135.