On the

Refinement Calculus

by

Carroll Morgan, Ken Robinson

and Paul Gardiner

Technical Monograph PRG-70
ISBN 0-902928-52-X

October 1988

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Programming Research Group
Oxford University Computing Laboratory
8–11 Keble Road
Oxford OX1 3QD
UK

Contents

Introduction

The refinement calculus is a notation and set of rules for deriving programs from their specifications. It is distinguished from earlier methods (though based on them) because the derivations are carried out within a single "programming" language: there is no separate language of specifications.

That does not mean that specifications are executable; it means rather that "not all programs are executable" [2]. Some are written too abstractly for any computer to execute, and they are the opposite extreme to those which, though executable, are too complex for any human to understand. *Program derivation* is the activity that transforms one into the other.

Tbis refinement calculus is distinguished from some other "wide spectrum" approaches (*e.g.*, [9, 17]) by its origins and scale: it is a simple programming language to which specifications have been added. The extension is modest and unpredjudiced, and one can sit back to see where it leads. So far, it has uncovered "miracles" [35, 41, 33], novel techniques of data refinement [3, 40, 13, 36], a simpler treatment of procedures [34, 6, 39], and "conjunction" of programs [32, 13, 36].

R.-J. Back [3] first extended Dijkstra's language of guarded commands with specifications, and is still active [4, 6, 5]. Joe Morris also does significant research in this area [41, 40]. This document collects only the work done at Oxford. The three approaches are strongly related, though not identical: Back does not have miracles; neither Back nor Morris use program conjunction; both of those authors address the calculus at a more theoretical level than we do.

Our work was motivated by the quickening interest at Oxford in developing programs from Z specifications [18, 38, 48], and it was surprising (to some) that we do it by adding Z to a programming language rather than by adding programming constructs to Z.

The specification statement introduces specification to Dijkstra's language of guarded commands, and explores the consequences: increased expressive power, the new prominence of the refinement relation, miracles, and a surprising factorisation of that language into smaller pieces.

Specification statements and refinement gives our first collection of "laws of refinement". (They appear again in *Laws of program refinement: a summary*, where they have been simplified by program conjunction [36].)

Procedures, parameters, and abstraction: separate concerns shows how specifications in a programming language allow the *copy rule* of ALGOL 60, once again, to give the meaning of procedures. A side effect of that is the parametrization of program fragments which are *not* procedures.

Data refinement using miracles and *Auxiliary variables in data refinement* describe small aspects of data refinement, independently of the refinement calculus. The former uses the Gries and Prins data refinement rule [16] in order to be self contained.. Data refinement is dealt with more generally in *Data refinement of predicate transformers* and *Data refinement by calculation*. The first gives a more theoretical, the second a more practical exposition of the way data refinement and the refinement calculus can interact.

Laws of program refinement: a summary collects for reference some laws used in practical derivations. No effort has been spent on their completeness.

A reasonable overview can be gained by reading *Specification statements and refinement* and *Data refinement by calculation*.

There is some overlap between the papers: the introduction to *Specification statements and refinement* repeats material from *The specification statement*; *Auxiliary variables in data refinement* amplifies a section of *Data refinement by calculation*; and various laws of program refinement appear in three places: in *Specification statements and refinement*, as an appendix to *Data refinement by calculation*, and finally in *Laws of program refinement: a summary*. The last of those is the latest and most comprehensive.

Carroll Morgan
July 1988

The specification statement

Carroll Morgan

Abstract

Dijkstra's programming language is extended by *specification statements*, which specify parts of a program "yet to be developed." A weakest precondition semantics is given for these statements, so that the extended language has a meaning as precise as the original.

The goal is to improve the *development* of programs, making it more as it should be: manipulations within a single calculus. The extension does this by providing one semantic framework for specifications and programs alike — developments begin with a program (a single specification statement), and end with a program (in the executable language). And the notion of *refinement* or *satisfaction*, which normally relates a specification to its possible implementations, is automatically generalised to act between specifications and between programs as well.

A surprising consequence of the extension is the appearance of *miracles:* program fragments that do not satisfy Dijkstra's *Law of the Excluded Miracle*. Uses for them are suggested.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies*; D.2.2 [**Software Engineering**]: Tools and Techniques—*Top-down programming*; D.2.4 [**Software Engineering**]: Program Verification—*Correctness proofs*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Pre- and post-conditions, Specification techniques*

General Terms: Theory, Verification

Additional Keywords and Phrases: Program refinement, procedural abstraction, development calculus, weakest preconditions, guarded commands, miracles

[0]Appeared in *TOPLAS 10*, 3 (July 1988).

1 Introduction

Dijkstra in [12] introduces the *weakest precondition* of a program P with respect to a postcondition *post*; following [20] we will write this $P < post >$. In this style, a *specification* of a program P is written

$$pre \Rightarrow P < post > .$$

This means "if activated in a state for which *pre* holds, the program P must terminate in a state for which *post* holds."

In traditional top-down developments, we build algorithmic structure around a collection of ever-decreasing program fragments "yet to be implemented," and at any stage we have specifications for those fragments. Thus one finds the dictions

$$\vdots$$
$$P;$$
$$\vdots$$

where $pre \Rightarrow P < post >$.

The letter P stands for the missing fragment, and the **where** clause gives its specification. But in our approach, we write instead

$$\vdots$$
$$[pre, post]; \qquad (1)$$
$$\vdots$$

We write the specification itself at the point to be occupied by its implementation. More significantly, by giving a weakest precondition semantics to $[pre, post]$, we make this intermediate stage (1) into a *program* — albeit an abstract one.

Program development we see as analogous to solving equations: one transforms an abstract program into a concrete one, just as one transforms a complex equation (*e.g.*, $x^2 - x - 1 = 0$) into a simple equality (*e.g.*, $x = (1+\sqrt{5})/2$). For such formulæ, the manipulations are mediated by the relation of implication: the simple equality *implies* the complex equation.

The abstract-to-concrete transformation of programs is mediated by a relation $\sqsubseteq$ of ***refinement***, which is defined so that $P \sqsubseteq Q$ means "any specification satisfied by P is satisfied by Q also." This relation can appear between abstract programs (specifications), between concrete programs, or between one and the other. As we write

$$x^2 - x - 1 = 0 \quad \Leftarrow \quad x = \tfrac{1+\sqrt{5}}{2},$$

so we will write with complete rigor

$$x{:}\ [x^2 - x - 1 = 0] \quad \sqsubseteq \quad x := \tfrac{1+\sqrt{5}}{2}.$$

An unexpected consequence of our extension is the introduction of abstract programs that do not obey Dijkstra's *Law of the excluded miracle*. These correspond to specifications that have no concrete solution, just as negative numbers stand for insoluble equations in elementary arithmetic ("3 from 2 won't go"). An example is the statement $[\mathit{true}, \mathit{false}]$; we will see that the following holds:

$$\mathit{true} \Rightarrow [\mathit{true}, \mathit{false}] < \mathit{false} > .$$

But just as negative numbers simplify arithmetic, miracles simplify program derivation.

Our overall contribution is ***uniformity***: we place program development within reach — in principle — of a single calculus. We expect this to be useful not only at the level of small intricacies, but in the larger scale also. Modules, for example, can be written using specification statements instead of concrete constructions: thus we have ***specifications*** of modules. Because of the generality of our approach, any structuring facility offered by the target programming language is offered to specifications also.

2 Specification statements

We introduce the syntax and weakest precondition semantics of specification statements, moving from simple to more general forms.

2.1 The simple form

The simple specification statement $[pre, post]$ comprises two predicates over the program variables $\vec{v}$. Informally, it means "assuming an initial state satisfying *pre*, establish a final state satisfying *post*." Its precise definition is (using $\hat{=}$ for "is defined to be")

Definition 1 $[pre, post] < R > \quad \hat{=} \quad pre \wedge (\forall \vec{v}.\ post \Rightarrow R)$ ♡

For example, assuming $\vec{v}$ is just the single variable x, we have

$$\begin{aligned} & [true, x = 1] < R > \\ = \ & true \wedge (\forall x.\ x = 1 \Rightarrow R) \\ = \ & R[x\backslash 1]. \end{aligned}$$

The substitution $[x\backslash 1]$ denotes syntactic replacement of x by 1 in the usual way.

2.2 Confining change

We allow the changing of variables to be confined to those of interest. For any subvector $\vec{w}$ of $\vec{v}$, the statement $\vec{w} : [pre, post]$ has the following informal meaning:

> assuming an initial state satisfying *pre*, establish a final state satisfying *post* while changing only variables in $\vec{w}$.

The precise definition of $\vec{w} : [pre, post]$ is

Definition 2 $\vec{w} : [pre, post] < R > \quad \hat{=} \quad pre \wedge (\forall \vec{w}.\ post \Rightarrow R)$ ♡

The only change from definition 1 is that the vector of quantified variables is now $\vec{w}$ rather than $\vec{v}$. Taking $\vec{v}$ to be "x, y", we have

$$\begin{aligned} & x : [true, x = y] < R > \\ = \ & true \wedge (\forall x.\ x = y \Rightarrow R) \\ = \ & R[x\backslash y]. \end{aligned}$$

Since $(x := y) < R >$ equals $R[x\backslash y]$ also, we have shown that $x : [true, x = y]$ and $x := y$ have the same meaning. If we allow ***both*** x and y to change, this is no longer true:

$$\begin{array}{ll} & x, y : [true, x = y] < R > \\ = & true \wedge (\forall x, y.\ x = y \Rightarrow R) \\ = & (\forall y.\ R[x\backslash y]). \end{array}$$

The statement $x, y : [true, x = y]$ can set y to x, x to y, or hoth x and y to some third value.

2.3 Referring to the initial state

Occurrences of 0-subscripted variables $\vec{v}_0$ in *post* refer to the values held by those variables ***initially***. We reserve 0-subscripts for this purpose, and assume that they do not occur as ordinary variables in programs. We now have the following informal meaning for $\vec{w} : [pre, post]$:

> assuming an initial state satisfying *pre*, change only variables in $\vec{w}$ to establish *post*, in which 0-subscripted variables refer to the values those variables held initially.

The precise definition appears below. In practice, however, we usually apply the simpler version given in lemma 1 following.

Definition 3

$$\vec{w} : [pre, post] < R > \quad \widehat{=} \quad pre \wedge (\forall \vec{w}.\ post[\vec{v}_0 \backslash \vec{f}\,] \Rightarrow R)[\vec{f}\,\backslash \vec{v}]$$

where $\vec{f}$ ***is some fresh vector of variables.***
♡

The use of fresh variables $\vec{f}$ in definition 3 is only to avoid interference with possible occurrences of $\vec{v}_0$ in R, which are rare in practice. Usually we can apply the simpler construction below:

Lemma 1 *If R contains no 0-subscripted variables,*

$$\vec{w} : [pre, post] < R > \;=\; pre \wedge (\forall \vec{w}.\ post \Rightarrow R)[\vec{v}_0 \backslash \vec{v}]$$

Proof: Immediate from definition 3.
♡

Notice that if *post* contains no $\vec{v}_0$, then both definition 3 and lemma 1 reduce to definition 2.

For example, taking $\vec{v}$ to be "x, y" as before we have from lemma 1

$$\begin{aligned} & x : [true, x = x_0 + y_0] < R > \\ = \;& true \wedge (\forall x.\ x = x_0 + y_0 \Rightarrow R)[x_0, y_0 \backslash x, y] \\ = \;& R[x \backslash x_0 + y_0][x_0, y_0 \backslash x, y] \\ = \;& R[x \backslash x + y]. \end{aligned}$$

2.4 The implicit precondition

We allow the omission of the precondition in a specification statement. The informal meaning of $\vec{w} : [post]$ is

> assuming it is possible to do so, change only variables in $\vec{w}$ to establish *post*, in which 0-subscripted variables refer to the values those variables held initially.

The meaning is given syntactically — we make the missing precondition explicit:

Definition 4 $\vec{w} : [post] \;\hat{=}\; \vec{w} : \lfloor ((\exists \vec{w} \bullet post))[\vec{v}_0 \backslash \vec{v}],\ post \rceil$ ♡

For example, we can write

$$\begin{array}{lll} & m\colon [l \le m \le h] & \text{for} \quad m\colon [l \le h\ ,\ l \le m \le h] \\ \text{and} & i\colon [a[i] = v] & \text{for} \quad i\colon [((\exists i \bullet a)[i] = v)\ ,\ a[i] = v] \end{array}$$

The first statement places m between l and h; the second locates an index i of value v in array a. If in either case the result is not achievable (*e.g.*, if l exceeds h, or v does not occur in a), the statement can abort.

2.5 Generalised assignment

We generalise assignment by giving the following meaning to the statement $x :\odot e$, for any binary relation $\odot$:

> assuming it is possible to do so, assign to x a value bearing the relation $\odot$ to the expression e, where occurrences of x in e refer to its initial value.

Ordinary assignment statements are now the special case in which $\odot$ is "$=$". But we can also write, for example,

$$\begin{array}{lllll} & x :\in s & \text{for} & \textit{if possible, choose } x \textit{ from } s \\ \text{and} & n :< n & \text{for} & \textit{decrease } n. \end{array}$$

The definition is given syntactically:

Definition 5 $x :\odot e \;\triangleq\; x : [x \odot e[x\backslash x_0]]$ ♡

With this definition, our abbreviations above become respectively

$$\begin{array}{ll} & x : [x \in s] \qquad (\text{that is, } x : [s \neq \{\}, x \in s]) \\ \text{and} & n : [n < n_0]. \end{array}$$

The syntax for generalised assignment was suggested (long ago) by Jean-Raymond Abrial.

3 The implementation ordering

For programs P and Q, we give $P \sqsubseteq Q$ the informal meaning: "every specification satisfied by P is satisfied by Q also." This means that Q is an acceptable replacement for P. Our precise definition is

Definition 6 $P \sqsubseteq Q$ *iff for all predicates* R,

$$P < R > \Rightarrow Q < R > .$$

♡

The following theorem shows definition 6 to have the property we require:

Theorem 1 *If* $pre \Rightarrow P < post >$ *and* $P \sqsubseteq Q$, *then also*

$$pre \Rightarrow Q < post > .$$

Proof: Since $P \sqsubseteq Q$, we have $P < post > \Rightarrow Q < post >$. The result follows immediately.
♡

As an example of refinement between *programs*, let P be

if $2|x \rightarrow x := x \text{ div } 2$
[] $3|x \rightarrow x := x \text{ div } 3$
fi,

and let Q be

if $2|x \rightarrow x := x \text{ div } 2$
[] $\neg(2|x) \rightarrow x := x \text{ div } 3$
fi,

where $2|x$ means "2 divides x exactly", and div denotes integer division. We have $P \sqsubseteq Q$ because

$$\begin{array}{lll} P < R > & = & (2|x \vee 3|x) \quad \wedge \\ & & (2|x \Rightarrow R[x \backslash x \text{ div } 2]) \quad \wedge \\ & & (3|x \Rightarrow R[x \backslash x \text{ div } 3]) \end{array}$$

and

$$\begin{array}{lll} Q < R > & = & (2|x \Rightarrow R[x \backslash x \text{ div } 2]) \quad \wedge \\ & & (\neg(2|x) \Rightarrow R[x \backslash x \text{ div } 3]). \end{array}$$

Thus $P < R > \Rightarrow Q < R >$ for any R. But Q differs from P in that Q will always terminate, even when $x = 7$. And Q is deterministic: if $x = 6$, Q will establish $x = 3$. In spite of these differences, Q *is* an acceptable substitute

for P, and that is why we can implement P as IF $2|x$ THEN $x := x \operatorname{div} 2$ ELSE $x := x \operatorname{div} 3$ END.

We now state the well-known but crucial fact that the program constructors are monotonic with respect to $\sqsubseteq$; only this ensures that refining a fragment (say P above) "in place," in some larger program, refines that larger program overall.

Theorem 2 *If $F(P)$ is a program containing the program fragment P, and for another program fragment Q we have $P \sqsubseteq Q$, then*

$$F(P) \sqsubseteq F(Q)$$

Proof: Structural induction, over the program constructors ";", "if", and "do".
♡

4 Suitability of the definitions

We now show the suitability of our definitions by proving that

$$pre \Rightarrow P < post > \quad \text{iff} \quad [pre\ ,\ post] \sqsubseteq P.$$

In fact, we prove a stronger result, dealing with the general form of section 2.3.

In long formulæ, we will sometimes "stack" conjunctions for clarity, writing

$$\left(\begin{array}{l} this \\ that \end{array} \right) \qquad \text{for} \qquad (this \wedge that).$$

Our theorem is a consequence of the following two lemmas.

Lemma 2 *If $\vec{u}$ and $\vec{w}$ partition the vector $\vec{v}$ of program variables, then*

$$pre \wedge \vec{v} = \vec{v}_0 \qquad \Longrightarrow \qquad \vec{w} : [pre, post] < post \wedge \vec{u} = \vec{u}_0 >$$

Proof: Here we must use definition 3 rather than lemma 1, since the post-condition contains $\vec{v}_0$. We have

$$(pre \wedge \vec{v} = \vec{v}_0) \Longrightarrow \vec{w} : [pre, post] < post \wedge \vec{u} = \vec{u}_0 >$$

if by definition 3,

$$(pre \wedge \vec{v} = \vec{v}_0) \Longrightarrow pre \wedge \left(\forall \vec{w}.\ post[\vec{v}_0 \backslash \vec{f}] \Rightarrow \begin{array}{c} post \\ \vec{u} = \vec{u}_0 \end{array} \right) [\vec{f} \backslash \vec{v}]$$

if $\vec{v} = \vec{v}_0 \Longrightarrow \left(\forall \vec{w}.\ post[\vec{v}_0 \backslash \vec{f}] \Rightarrow \begin{array}{c} post \\ \vec{u} = \vec{u}_0 \end{array} \right) [\vec{f} \backslash \vec{v}]$

if $\vec{v} = \vec{v}_0 \Longrightarrow \left(\forall \vec{w}.\ post[\vec{v}_0 \backslash \vec{f}] \Rightarrow \begin{array}{c} post \\ \vec{u} = \vec{u}_0 \end{array} \right) [\vec{f} \backslash \vec{v}_0]$

if $\vec{v} = \vec{v}_0 \Longrightarrow \left(\forall \vec{w}.\ post \Rightarrow \begin{array}{c} post \\ \vec{u} = \vec{u}_0 \end{array} \right)$

if since $\vec{u}$, $\vec{w}$ partition $\vec{v}$
true.

♡

Lemma 3 *If* $pre \wedge \vec{v} = \vec{v}_0 \Longrightarrow P < post \wedge \vec{u} = \vec{u}_0 >$ *then*

$$\vec{w} : [pre, post] \sqsubseteq P$$

where $\vec{w}$ *and* $\vec{u}$ *partition the program variables* $\vec{v}$.

Proof:

$$pre \wedge \vec{v} = \vec{v}_0 \Longrightarrow P < post \wedge \vec{u} = \vec{u}_0 >$$

hence by distributivity of $\Rightarrow$ over weakest preconditions,

$$pre \wedge \vec{v} = \vec{v}_0 \wedge \left(\forall \vec{v}. \quad \begin{matrix} post \\ \vec{u} = \vec{u}_0 \end{matrix} \Rightarrow R \right)$$
$$\Longrightarrow \quad P < R >$$

hence $pre \wedge \vec{v} = \vec{v}_0 \wedge (\forall \vec{w}.\ post \Rightarrow R)[\vec{u} \setminus \vec{u}_0] \Longrightarrow P < R >$
hence $pre \wedge \vec{v} = \vec{v}_0 \wedge (\forall \vec{w}.\ post \Rightarrow R) \Longrightarrow P < R >$

hence since pre and $P < R >$ do not contain $\vec{v}_0$,
$pre \wedge (\forall \vec{w}.\ post \Rightarrow R)[\vec{v}_0 \setminus \vec{v}] \Longrightarrow P < R >$

hence by lemma 1,
$\vec{w} : [pre, post] < R > \Longrightarrow P < R >$.

Since R was arbitrary, we conclude from definition 6 that $\vec{w} : [pre, post] \sqsubseteq P$ as required.
♡

Those two lemmas give us our theorem immediately:

Theorem 3 *If $\vec{w}$, $\vec{u}$ partition the program variables $\vec{v}$, then*

$$pre \wedge \vec{v} = \vec{v}_0 \Longrightarrow P < post \wedge \vec{u} = \vec{u}_0 >$$

if and only if

$$\vec{w} : [pre, post] \sqsubseteq P.$$

Proof: "If" follows from lemma 2 and theorem 1; "only if" is lemma 3 exactly.
♡

5 Using specification statements

For illustration we take the simplest of examples: we are given an array $a[0..N-1]$ and must find an index i at which the value v occurs. And we may assume there is such an i. The program is

$$i{:}\ \left[\begin{array}{c} 0 \le i < N \\ a[i] = v \end{array}\right] \tag{2}$$

This *is* a program, though abstract, and perhaps we can execute it directly (see further below). But for now, we assume not — and so we "solve" it, refining it to statements we can execute.

First we use definition 4, rewriting

$$i{:}\ \left[(\exists\, i.0 \le i < N \wedge a[i] = v)\ ,\ \begin{array}{c} 0 \le i < N \\ a[i] = v \end{array}\right] \tag{3}$$

We take as invariant

$$Inv \ \widehat{=}\ \begin{array}{c} 0 \le i < N \\ (\exists j.\, i \le j < N \wedge a[j] = v) \end{array}$$

The variant is $N - i$. With these and theorem 3, we can prove that (3)

$$\begin{array}{l} \sqsubseteq\ \ i{:}\ [(\exists\, i.0 \le i < N \wedge a[i] = v)\ ,\ Inv]; \\ \quad \textbf{do}\ a[i] \neq v\ \rightarrow \\ \qquad i{:}\left[\begin{array}{ccc} Inv & & i_0 < i \\ a[i] \neq v & , & Inv \end{array}\right] \\ \quad \textbf{od} \end{array}$$

Notice that the fragments "to be developed" are written in-line, and that the above mixture of abstract and concrete is still a program. The first component we can refine to $i := 0$; and the second we can refine to $i := i+1$. For illustration, we show the second refinement in more detail: by theorem 3 we need

$$\left(\begin{array}{c} 0 \le i < N \\ (\exists j.\, i \le j < N \wedge a[j] = v) \\ a[i] \neq v \\ i = i_0 \end{array}\right) \implies i := i+1 \left\langle \begin{array}{c} i_0 < i < N \\ (\exists j.\, i \le j < N \wedge a[j] = v) \end{array} \right\rangle$$

By the semantics of assignment [12], the consequent is

$$i_0 < i + 1 < N$$
$$(\exists j.\, i + 1 \le j < N \wedge a[j] = v).$$

That follows easily from the antecedent.

Having our development, we may wish to collect it and others into a small "database module," based on arrays. As is typical in modern programming languages, the implementation

$i := 0;$
do $a[i] \neq v \rightarrow i := i + 1$ **od**

would be hidden within the "implementation part" of the module. What should appear in the definition part? We suggest (using the syntax of Modula-2 [49])

module *Database*;
  export *Find*, N;
  const $N = ?$;
  var a: **array** $[0..N-1]$ **of** ?;
  procedure *Find*(v: ?; **var** i: $[0..N-1]$);
  begin
    i: $\left[\begin{array}{c} 0 \le i < N \\ a[i] = v \end{array}\right]$
  end *Find*
  ⋮
end *Database*

This is not informal. Except for the "?," the module contains only constructions whose semantics are known precisely. Now a programmer wishing to implement (2) can do so directly, using the copy rule of Algol-60 (suitably extended for modules). He just writes *Find*(v, i), whose meaning is given by substituting the procedure body from the *definition* module. This is discussed further in [34].

Thus we show that our approach applies not only to small constructions, and in particular that it supports the view that the "definition module" specifies the "implementation module."

6 Miracles

In [12] it is stated that for all programs P,

$$P < \mathit{false} >= \mathit{false}. \tag{4}$$

This is no longer true: we have for example

$$\begin{array}{ll} & [\mathit{true}, \mathit{false}] < \mathit{false} > \\ = & \mathit{true} \wedge (\forall\, \vec{v}.\ \mathit{false} \Rightarrow \mathit{false}) \\ = & \mathit{true}. \end{array}$$

The statement $[\mathit{true}, \mathit{false}]$ is called a *miracle*, because it implements anything: we have for all R that $P < R > \Rightarrow [\mathit{true}, \mathit{false}] < R >$, and so for any P whatsoever,

$$P \sqsubseteq [\mathit{true}, \mathit{false}].$$

Although $[\mathit{true}, \mathit{false}]$ implements anything, it cannot itself be implemented by anything free of miracles. This is because "P is free of miracles" implies by (4) that $P < \mathit{false} >= \mathit{false}$, and so taking $R = \mathit{false}$ in definition 6, we have $[\mathit{true}, \mathit{false}] \not\sqsubseteq P$.

A program which cannot be rid of miracles is *infeasible* in the following precise way:

Definition 7 *We say that a program P is* feasible *iff*

$$P < \mathit{false} >= \mathit{false}.$$

Otherwise it is infeasible, *or* miraculous. ♡

Clearly, all programs free of specification statements are by (4) feasible: indeed, they are "implementations" already. For specifications, however, we have the following

Theorem 4 *$\vec{w} : [\mathit{pre}, \mathit{post}]$ is feasible iff*

$$\mathit{pre} \Rightarrow ((\exists\, \vec{w} \bullet \mathit{post}))[\vec{v}_0 \backslash \vec{v}].$$

Proof: Definitions 3, 7, and predicate calculus.
♡

Miracles can arise "accidentally" in program development if we make an incorrect design step; this is discussed in more detail in [25] and [37]. For the present, we take a trivial example: we (mistakenly) want to implement $x : [x = 0]$ as a sequential composition whose second component is $x := y$. That is, we want to solve the following formula for P:

$$x : [x = 0] \quad \sqsubseteq \quad P; x := y \tag{5}$$

By theorem 3, we have (5)

$$\textit{iff} \quad x = x_0 \wedge y = y_0 \Rightarrow (P; x := y) < x = 0 \wedge y = y_0 >$$

$$\textit{iff} \quad \text{by sequential composition}$$
$$x = x_0 \wedge y = y_0 \Rightarrow P < x := y < x = 0 \wedge y = y_0 >>$$

$$\textit{iff} \quad x = x_0 \wedge y = y_0 \Rightarrow P < y = y_0 = 0 >$$

$$\textit{iff} \quad \text{by theorem 3 again}$$
$$x : [\mathit{true}, y = 0] \quad \sqsubseteq \quad P$$

We have found our solution P, showing unconditionally that

$$x{:}\ [x = 0] \ \sqsubseteq \ x{:}\ [\mathit{true}\ ,\ y = 0];\ x := y$$

In fact, the above shows that $x : [\mathit{true}, y = 0]$ is the most general solution of (5), and so we take it as representative of them all, calling it "the" solution. This development technique, in which formulae like (5) are so solved, is the subject of [25].

But, after all, the statement $x{:}\ [\mathit{true}\ ,\ y = 0]$ is infeasible; and the importance of the example is its illustration of that consequence of mistaken design steps. The formula (5) is *not* insoluble, but we cannot develop executable code from its solution.

7 Guarded commands are miracles

Miracles are a strict extension of our programming capabilities — clearly, since they cannot be executed. We now show how close miracles are, nevertheless, to being in the original language.

A guarded command has the syntactic form

$$B \rightarrow P,$$

where B is a boolean expression and P is the command guarded. Originally, these occurred only within **if** and **do** constructions. Here we give meaning to guarded commands standing alone.

Informally, we say that a "naked" guarded command *cannot* be executed unless its guard is true. More precisely, we have

Definition 8 $(B \rightarrow P) < R > \quad \hat{=} \quad B \Rightarrow P < R >$ ♡

If B is true, then $B \rightarrow P$ behaves like P. But if B is false, we consider $B \rightarrow P$ to be miraculous: we may as well, since in this case we *cannot* execute it to check.

Thus we have a compact notation for miracles: they are naked guarded commands whose guards are not identically true. For example, our first miracle $[\mathit{true}, \mathit{false}]$ can be written for *any* program P

$$\mathit{false} \rightarrow P.$$

The following theorem shows that in fact every miracle can be written this way. We have

Theorem 5 *For any program P, feasible or not, there is a guard B and a feasible program Q such that*

$$P = B \rightarrow Q$$

Proof: We take

$$\begin{aligned} B &= \neg P < \mathit{false} > \\ Q &= \textbf{if } B \rightarrow P \textbf{ fi}. \end{aligned}$$

Definition 7 shows that Q is feasible, and definition 8 shows that the equality holds.
♡

We can also define also a non-deterministic composition $[]$ and a "guardless **if**," achieving correspondence with the original meaning of these constructs. We have

Definition 9 *For any programs P and Q, the program $P \,[]\, Q$ is defined*

$$(P \,[]\, Q) < R > \;\; \hat{=} \;\; P < R > \wedge Q < R > .$$

♡

Definition 10 *For any program P, the program* **if** P **fi** *is defined*

$$\textbf{if } P \textbf{ fi} < R > \;\; \hat{=} \;\; \neg P < \mathit{false} > \wedge P < R > .$$

♡

Definition 9 is simple non-deterministic choice; in fact

$$P \,[]\, Q \;=\; \textbf{if } \mathit{true} \rightarrow P \,[]\, \mathit{true} \rightarrow Q \textbf{ fi}.$$

Definition 10 *is* an extension of Dijkstra's language (necessarily, since it is not monotonic with respect to $\sqsubseteq$; it is in fact the "+" operator of [25]). Nevertheless, the meaning that definitions 8, 9, and 10 give to the if construction **if** $([]i.\ B_i \rightarrow P_i)$ **fi** is exactly as before. We have

Theorem 6 *If P_i are feasible programs, then*

$$\textbf{if } ([]i.\ B_i \rightarrow P_i) \textbf{ fi} < R > \;=\; (\vee i.\ B_i) \wedge (\wedge i.\ B_i \Rightarrow P_i < R >).$$

Proof: Let P be $([]i.\ B_i \rightarrow P_i)$. By definitions 9 and 10,

$$P < R > \;=\; (\wedge i.\ B_i \Rightarrow P_i < R >). \tag{6}$$

Hence because the P_i are feasible,

$$\neg P < \mathit{false} > \;=\; \neg(\wedge i.\ B_i \Rightarrow \mathit{false}) \;=\; (\vee i.\ B_i). \tag{7}$$

The result now follows from (6), (7), and definition 10.
♡

Unfortunately, we must note in conclusion that because the construction **if** $\cdots$ **fi** is not monotonic, we have in general

$$P \sqsubseteq Q \qquad \textit{does not imply} \qquad \textbf{if } P \textbf{ fi} \sqsubseteq \textbf{if } Q \textbf{ fi}.$$

This limits its use in program development.

8 Positive applications of miracles

By definitions 7 and 6, miracles refine only to other miracles — and hence by Dijkstra's law never to programs. Thus if a specification *overall* is miraculous (we can check using theorem 4), the development is doomed.

In VDM, where specifications are written as predicate pairs like ours, the check for miracles is the "implementability test" [26, p. 134]. In Z [18], [38], [48], where specifications are single predicates corresponding to our implicit form of section 2.4, miracles cannot be written: definition 4 and theorem 4 show that single predicate specification statements are always feasible.

From a feasible beginning, miracles can arise through mistaken refinement tactics. As shown in section 6, the "improper division" of $x := 0$ by $x := y$ gives the miraculous x: $[true\ ,\ y = 0]$. If we recognise the miracle then, we could stop there and try some other tactic; if we don't, we'll be stuck later. But the *rules* for such division (the weakest prespecification of [25]) are simpler now that soundness has been delegated to the unimplementable miracles: there is less need for "applicability conditions."

There is other potential for the deliberate use of miracles. Consider the following assignment, in which f is some function hard to calculate but easy to invert:

$$x := f(c) \tag{8}$$

And suppose in a variable y we might have the desired answer already. We can make the following refinements, in whicb both right hand sides are miracles:

$$x := f(c) \quad \sqsubseteq \quad c = f^{-1}(y) \rightarrow x := y \tag{9}$$

$$x := f(c) \quad \sqsubseteq \quad c \neq f^{-1}(y) \rightarrow x := f(c) \tag{10}$$

Neither (9) not (10) can be implemented on its own. Case (9) can be executed only when y does contain the desired answer already; case (10) can be executed only when it doesn't. But their $[]$ combination is *not* miraculous, and can always be executed:

$$(c = f^{-1}(y) \rightarrow x := y)\ []\ (c \neq f^{-1}(y) \rightarrow x := f(c)) \tag{11}$$

Since $P \sqsubseteq Q$ and $P \sqsubseteq R$ implies $P \sqsubseteq Q[]R$ (easily shown from definitions 6 and 9), we have refined (8) to (11). Such developments are treated also in [1] and [43].

Another application is as follows. Ordinarily we limit the syntax of our concrete programming language so that miracles cannot be written in it: no specifications can appear, nor naked guarded commands. If we relax this restriction, allowing naked guarded commands, then operational reasoning suggests a ***backtracking*** implementation. For example, consider the following backtracking strategy for finding the position i of a value v in an array $a[0..N-1]$:

> Choose i at random from the range $0..N-1$, and evaluate $a[i] = v$. If equality holds, then terminate; otherwise, backtrack and try again.

We have this refinement:

$$
\begin{array}{ll}
 & i:\ [a[i] = v] \\
\sqsubseteq & \textbf{if} \\
 & \qquad i := 0 \,[\!]\cdots[\!]\, i := N-1; \\
 & \qquad a[i] = v \rightarrow \textbf{skip} \\
 & \textbf{fi}
\end{array}
$$

We are using the generalised **if**$\cdots$**fi** of section 6, which here allows abortion if its body is miraculous; and the body is miraculous *only* when no branch of the alternation can avoid the miraculous behaviour to follow. In this context **if** $\cdots$ **fi** resembles the "cut" of Prolog, allowing failure (preventing backtracking) if no solution is found within (beyond). If there is a successful branch, however, the implementation is obliged to find it: only then can it execute the second statement — which we could syntactically sugar, writing force $a[i] = v$. Note that the first statement can be written i: $[0 \leq i < N]$.

A third opportunity for exploiting miracles is in novel proof rules. We introduce for a moment the weaker relation $\leq$ between programs, which holds if for all predicates R

$$P < R > \wedge Q < \mathit{true} > \Rightarrow Q < R >$$

This is simply ***partial*** correctness. Now in the style of VDM we can consider a loop invariant to be a ***statement***, rather than an assertion: any number of iterations of the loop body must refine the invariant statement I. The

advantage is that we have easy reference to the initial state; our development law is

$$\begin{array}{lll} \text{If} & I \leq I;\ G \to S \\ \text{and} & X \leq I;\ \mathbf{force}\,\neg G \\ \\ \text{then} & X \leq I;\ \mathbf{do}\,G \to S\,\mathbf{od} \end{array}$$

We "explain" this rule as follows (but it is *proved* using weakest precondition semantics). The first condition requires preservation of the effect of I by one more execution of the body $G \to S$. If G holds, the body behaves like S; but if G fails (and therefore we should *not* execute S), the first condition still holds because $G \to S$ in that case is miraculous, refining anything (and **skip** in particular).

Similar reasoning applies to the second condition. For the result, we argue informally that

$$\begin{array}{ll} & X \\ \leq & I;\ \mathbf{force}\,\neg G \\ \\ \leq & \text{by induction over the first condition} \\ & I;\ G \to S;\ \cdots;\ G \to S;\ \mathbf{force}\,\neg G \\ \\ \leq & I;\ \mathbf{do}\ G \to S\ \mathbf{od} \end{array}$$

Take for example the following program, in which we calculate the sum s of an array a indexed by $0 \leq i < N$.

$$\begin{array}{lll} X & = & s, n:\ [s = (\sum i : 0 \leq i < N : a[i])] \\ I & = & s, n:\ [s = (\sum i : 0 \leq i < n : a[i]) \wedge 0 \leq i \leq N] \\ G & = & n \neq N \\ S & = & s, n := s + a[n], n + 1 \end{array}$$

Because $I \sqsubseteq s, n := 0, 0$ (this is the initialisation), and because we can prove the conditions hold (using definitions and theorem 3), we have by our rule above

$$\begin{array}{ll} X & \leq I;\ \mathbf{do}\ n \neq N \to s, i := s + a[n], n + 1\ \mathbf{od} \\ & \leq s, n := 0, 0; \\ & \quad \mathbf{do}\ n \neq N \to s, i := s + a[n], n + 1\ \mathbf{od} \end{array}$$

Another application is as follows. Ordinarily we limit the syntax of our concrete programming language so that miracles cannot be written in it: no specifications can appear, nor naked guarded commands. If we relax this restriction, allowing naked guarded commands, then operational reasoning suggests a ***backtracking*** implementation. For example, consider the following backtracking strategy for finding the position i of a value v in an array $a[0..N-1]$:

> Cboose i at random from the range $0..N-1$, and evaluate $a[i] = v$. If equality holds, then terminate; otherwise, backtrack and try again.

We have this refinement:

$$
\begin{array}{ll}
 & i{:}\ [a[i] = v] \\
\sqsubseteq & \textbf{if} \\
 & \qquad i := 0 \,[]\cdots[]\, i := N-1; \\
 & \qquad a[i] = v \rightarrow \textbf{skip} \\
 & \textbf{fi}
\end{array}
$$

We are using tbe generalised **if** $\cdots$ **fi** of section 6, which here allows abortion if its body is miraculous; and the body is miraculous ***only*** when no branch of the alternation can avoid the miraculous behaviour to follow. In tbis context **if** $\cdots$ **fi** resembles the "cut" of Prolog, allowing failure (preventing backtracking) if no solution is found within (beyond). If there is a successful branch, however, the implementation is ohliged to find it: only then can it execute the second statement — which we could syntactically sugar, writing force $a[i] = v$. Note that the first statement can be written $i{:}\ [0 \le i < N]$.

A third opportunity for exploiting miracles is in novel proof rules. We introduce for a moment the weaker relation $\le$ between programs, which holds if for all predicates R

$$P < R > \wedge Q < true > \Rightarrow Q < R >$$

This is simply ***partial*** correctness. Now in the style of VDM we can consider a loop invariant to be a ***statement***, rather than an assertion: any number of iterations of the loop body must refine the invariant statement I. The

advantage is that we have easy reference to the initial state; our development law is

$$\begin{array}{lll} \text{If} & I \leq I;\ G \to S \\ \text{and} & X \leq I;\ \mathbf{force}\ \neg G \\ \\ \text{then} & X \leq I;\ \mathbf{do}\ G \to S\ \mathbf{od} \end{array}$$

We "explain" this rule as follows (but it is *proved* using weakest precondition semantics). The first condition requires preservation of the effect of I by one more execution of the body $G \to S$. If G holds, the body behaves like S; but if G fails (and therefore we should *not* execute S), the first condition still holds because $G \to S$ in that case is miraculous, refining anything (and skip in particular).

Similar reasoning applies to the second condition. For the result, we argue informally that

$$\begin{array}{ll} & X \\ \leq & I;\ \mathbf{force}\ \neg G \\ \\ \leq & \text{by induction over the first condition} \\ & I;\ G \to S;\ \cdots;\ G \to S;\ \mathbf{force}\ \neg G \\ \\ \leq & I;\ \mathbf{do}\ G \to S\ \mathbf{od} \end{array}$$

Take for example the following program, in which we calculate the sum s of an array a indexed by $0 \leq i < N$.

$$\begin{array}{lcl} X & = & s, n:\ [s = (\sum i : 0 \leq i < N : a[i])] \\ I & = & s, n:\ [s = (\sum i : 0 \leq i < n : a[i]) \wedge 0 \leq i \leq N] \\ G & = & n \neq N \\ S & = & s, n := s + a[n], n + 1 \end{array}$$

Because $I \sqsubseteq s, n := 0, 0$ (this is the initialisation), and because we can prove the conditions hold (using definitions and theorem 3), we have by our rule above

$$\begin{array}{ll} X & \leq I;\ \mathbf{do}\ n \neq N \to s, i := s + a[n], n + 1\ \mathbf{od} \\ & \leq s, n := 0, 0; \\ & \quad \mathbf{do}\ n \neq N \to s, i := s + a[n], n + 1\ \mathbf{od} \end{array}$$

9 Conclusion

We have extended Dijkstra's programming language with a construct allowing abstract programs, as predicate pairs, to be written within otherwise conventional "concrete" programs. The advantages are:

- Program development takes on the character of solving equations — well-established in mathematics generally. The transformation from abstract to concrete occurs within a single semantic framework.
- As lambda-expressions allow us to write functions without names (rather than the laboured "f **where** $f(x) = \cdots$") so we can write specifications directly, avoiding "P **where** $\cdots \Rightarrow P < \cdots >$." Instead of a lambda calculus, this leads to a refinement calculus.
- We gain *miracles* as an artefact of our extension, and there is increasing evidence that they simplify the development process. In [37] it is shown that applicability conditions for refinement can be simplified — or even removed altogether — because mistaken development steps simply lead to miracles from which eventually progress must cease. Also in [1], [25], [41], and more recently [43] it is argued that miracles simplify the theory. In [33] it is shown that miracles allow proof of certain data-refinements that were not provable previously.
- The lack of distinction between abstract and concrete programs allows their treatment as procedures to be made more uniform, in the sense of ALGOL-60: a procedure call, whether abstract or not, is equivalent to its text substituted in-line. This and the resulting treatment of parameters is explored in [34].
- The programmer's repertoire is increased by providing easy access to non-constructive idioms, for example: i: $[a[i] = v]$ finds the index i of value v in array a; m: $[l \leq m \leq h]$ chooses m between l and h.
- A ready connection is made with state-based specifications such as those of Z [18], [38], [48], allowing their systematic development into code.

A refinement calculus would be a collection of *laws*, each proved directly from weakest precondition definitions. They could be used, without further

proof, in program developments — just as one uses a table of integrals in engineering. For example, one such law is

$$\text{Assignment law:} \qquad \begin{array}{ll} & w{:}\ [post[w\backslash E][v_0\backslash v]\ ,\ post] \\ \sqsubseteq & w := E \end{array}$$

It is easily proved from definitions 3 and 6. A comprehensive collection of such laws is given and demonstrated in practice in [37].

Such a development style would be very close to VDM [26], where specifications are predicate pairs just as here. But Jones does not base VDM on the weakest precondition calculus, nor does he present a general refinement relation operating uniformly between all programs whether abstract or concrete (although he could do so). Another difference is our use of classical logic rather than the logic of partial functions [26], [8]. Jones does not treat miracles.

In the Z specification technique, specifications are given as single predicates corresponding to our "implicit preconditions". Thus where we write $n{:}\ [0 \le n < n_0]$ for "decrease n, but not below 0," in Z one would write (omitting types)

$$\frac{n, n'}{0 \le n' < n}$$

In Z there is no commitment to a *fixed* state (our $\vec{v}$); deliberately not, because this gives it the flexibility needed to build large specifications from their smaller components. Examples of large-scale Z specifications can be found in [18]. But when algorithmic structures are introduced — *i.e.*, once *development* begins — this lack of commitment becomes a hindrance.

Therefore one aim of our work is to provide a development method specifically for Z, by identifying the two specifications above then using the weakest precondition calculus to reach a concrete program. Another approach to Z development — derived from ours — is given in [27].

10 Acknowledgements

Back [4] first embedded specifications within programs using the weakest precondition calculus. His specifications — like those of Z — consist of one predicate only, and so he cannot take advantage of miracles. More recently Morris [41] presents independently the same extension of Back's work as we do; we have had useful discussions since discovering each other. Our refinement relation $\sqsubseteq$ is the same as theirs.

Meertens [30] also has developed these ideas, using predicate pairs, but gave them a different meaning: (in our notation) he defines

$$[pre, post] < R > \quad \widehat{=} \quad pre \Rightarrow (\exists\, \vec{v}.\, post) \\ \wedge \quad (\forall\, \vec{v}.\, post \Rightarrow R)$$

But Meertens' definition does not have the property of lemma 2, which we consider to be fundamental; in general, for Meertens

$$[pre, post] < post > \quad \neq \quad pre.$$

Hehner [20] uses predicate pairs *for specifications* as we use specification statements, but he does not integrate the approach by giving them a weakest precondition semantics. He also uses the refinement ordering $\sqsubseteq$.

The earliest example of a formulation like ours for the weakest precondition of a specification seems to be Hoare's [22], where it is given as the axiomatic meaning of procedure calls. But he did not separate abstraction from procedure calling, as we have done (and discuss further in [34]). In [14, p. 153] also the definition can be found, again coupled to procedure calls.

The idea of using pre- and post-conditions to describe program behaviour is widespread, and its use in VDM is notable. In fact our approach is very close to VDM, and I hope identical in spirit. Jones does not however make his specifications "first-class citizens" as we do. An advantage of Jones's natural deduction style is perhaps its appeal to the wider audience of practising programmers, just as natural deduction in logic is so-called because it's more "natural." But we prefer the increased freedom of the axiomatic approach directly (in logic, too): it offers more scope to the experienced user, who can construct new laws (meta-theorems) to suit his taste and skill.

[25] provided the direct inspiration for treating specifications as programs; there similar results are obtained in the relational calculus. Miracles appear as partial relations, but are not discussed in detail.

Most recently, Nelson [43] has integrated specifications and programs, but his ordering over these objects differs from ours. In particular, it does not allow the reduction of non-determinism — an essential idea in program development. He discusses miracles at some length.

Much of this work was done in collaboration with Ken Robinson. I thank Rick Hehner, Joe Morris, Doaitse Swierstra, members of IFIP 2.1, and the referees for their very useful comments.

Specification statements and refinement

Carroll Morgan Ken Robinson

Abstract

We discuss the development of executable programs from state-based specifications written in the language of first-order predicate calculus. Notable examples of such specifications are those written using the techniques *Z* and *VDM;* but our interest will be in the rigorous derivation of the algorithms from which they deliberately abstract. This is of course the role of a *development method.*

Here we propose a development method based on *specification statements* with which specifications are embedded in programs — standing in for developments "yet to he done." We show that specification statements allow description, development, and execution to be carried out within a *single* language: programs/specifications become hybrid constructions in which hoth predicates and directly executable operations can appear.

The use of a single language — embracing both high- and low-level constructs — has a very considerable influence on the development style, and it is that influence we will discuss: the specification statement is described, its associated calculus of refinement is given, and the use of that calculns is illustrated.

1 Introduction

In the Z [18, 38, 48] and VDM [26] specification techniques, descriptions of external behaviour are given by relating the "before" and "after" values of variables in a hypothetical program state. It is conventional to assume

[0]Appeared in *IBM Jnl. Res. Dev. 31(5)* (Sept. 1987).

that the *external* aspects are treated by designating certain variables as containing initially the *input* values, and certain others as containing finally the *output* values. As development proceeds, structure is created in the program — and the specifications, at that stage more "abstract algorithms," come increasingly to refer to internal program variables as well. For example, we may at some stage wish to describe the operation of taking the square-root of some integer variable n; adopting the convention that n refers to the value of that variable *after* the operation, and n_0 to its value *before*, this description could be written:

$$n^2 = n_0 \tag{1}$$

Ordinarily, we would call the above a *specification*, because "conventional" computers do not execute (*i.e.*, find a valuation making true) arbitrary formulas of predicate logic (logic programming languages deal only with a restricted language of predicates).

Two notable features of our specification (1) above are its *non-determinism* and that it is *partial*. It is non-deterministic in the sense that for some initial values n_0 (*e.g.*, 4) there may be several appropriate final values n (± 2 in this case). It is partial in the sense that for some initial values (*e.g.*, 3) there are *no* appropriate final values. We will see below that our proposed development method makes this precise in the usual way (*e.g.*, of [12]): the non-determinism allows an implementation to return either result (either consistently or even varying from one execution to the next); and the implementor can *assume* that the initial value is a perfect square, providing a program whose behaviour is wholly arbitrary otherwise.

In presenting a development technique, we are not ignorant of the fact that VDM already has (or even *is*) one; rather we are concentrating our attention on Z, where development has been less well worked out. In this our aim is most definitely to propose a *light-weight* technique — as Z is itself — in which existing material is used as much as possible. Dijkstra's language [12] therefore was chosen as the target, because it has a mathematically attractive and above all simple semantic basis, and because it includes non-determinism naturally.

The *key* to a smooth development process — the subject of this paper — is we believe the integration of description and execution in one language. This is not achieved, as is so often proposed, by restricting our language to those specifications which are executable, and thus treating specifications

as programs; instead we extend the language to allow ourselves to write programs which we cannot see at first how to execute: in effect we treat programs as specifications. It is precisely the lack of semantic distinction between the two that allows finally our smooth transition from abstract description to executable algorithm.

We will assume some familiarity with Dijkstra's weakest pre-condition concept and its associated guarded command programming language [12].

1.1 Weakest pre-conditions and specifications

In [12], Dijkstra introduces for program P and predicate R over the program variables, the *weakest pre-condition* of R with respect to P; he writes it

$$wp(P,\ R)$$

This weakest pre-condition is intended to describe exactly those states from which execution of P is guaranteed to establish R, and Dijkstra goes on to develop a small language by defining for its every construct precise syntactic rules for writing $wp(P,\ R)$ as a predicate itself. For example, the meaning of *assignment* in this language is defined as follows for variable x, expression E, and post-condition R:

$$wp(\text{“}x := E\text{”},\ R) \quad = \quad R[x\backslash E]$$

The notation $[x\backslash E]$ here denotes syntactic replacement in R of x by E in the usual way (avoiding variable capture *etc.*). Thus

$$\begin{array}{ll} & wp(\text{“}x := x - 1\text{”},\ x \geq 0) \\ = & (x \geq 0)[x\backslash x - 1] \\ = & (x - 1) \geq 0 \\ = & x > 0 \end{array} \tag{2}$$

We can *specify* a program P by giving *both* a pre-condition (not necessarily weakest) and a post-condition; our pre-condition and post-condition predicates we will usually call *pre* and *post*:

$$pre \Rightarrow wp(P,\ post) \tag{3}$$

Informally, this is read "if *pre* is true, then execution of P must establish *post*"; formally, we regard the above as admitting only program texts P for which it is valid. Either way, it is a specification in the sense that it directs the implementor to develop a program with the required property.

Our point of divergence from the established style (3) is to write instead

$$[pre\ ,\ post] \sqsubseteq P \tag{4}$$

We take (3) and (4) as identical in meaning, but in (4) the constituents are exposed more clearly: $[pre\ ,\ post]$ is the specification; $\sqsubseteq$ is the relation of refinement; and P is the program to be found. Thus we will read (4) as "the specification $[pre\ ,\ post]$ is refined by P."

The principal advantage of the alternative style is that $[pre\ ,\ post]$ can take on a meaning independent of its particular use in (4) above: we will give it a weakest pre-condition semantics of its own. It is just this which removes the distinction between specification and program — not that they both are executable, but that they both are *predicate transformers*, being suitable first arguments to $wp(\ ,\ )$. Programs are just those specifications which we can execute directly.

The refinement relation $\sqsubseteq$ is likewise generalised, and we do this immediately below.

1.2 Refinement

In (4) we have introduced an explicit symbol "$\sqsubseteq$" for refinement, and we now give its precise definition (as given *e.g.*, in [20]):

Definition 1 *For programs P and Q, we say that P is refined by Q, written $P \sqsubseteq Q$, iff for all post-conditions* post*:*

$$wp(P,\ post) \Rightarrow wp(Q,\ post).$$

♡

We justify the above informally by noting that any occurrence of P in a (proved correct) program is justified by the truth of $wp(P,\ post)$ at

that point, for some predicate *post*. No matter what *post* it is, the relation $P \sqsubseteq Q$ gives us $wp(Q, post)$ as well, so that Q is similarly justified: thus Q can replace P. Operationally, $P \sqsubseteq Q$ whenever Q resolves non-determinism in P, or terminates when P might not.

This refinement relation is independent of the notion of specification, and can be evaluated for *any* two constructs whose weakest pre-condition semantics are known. For example, we have in the guarded command language of [12]

$$\begin{array}{ll} & \textbf{if } a \leq b \rightarrow a := a - b \\ & [] \ b \geq a \rightarrow b := b - a \\ & \textbf{fi} \\ \\ \sqsubseteq & \textbf{if } a \leq b \rightarrow a := a - b \\ & [] \ a \not\leq b \rightarrow b := b - a \\ & \textbf{fi} \end{array}$$

The first program is non-deterministic, executing either branch when $a = b$; the second program is a proper (*i.e.*, non-identical) refinement of it because this non-determinism has been removed. Such refinement relations between *programs* allow us to implement the non-deterministic program above in more conventional (deterministic) languages; we transcribe the deterministic refinement as follows:

```
IF a<=b THEN a:= a-b
        ELSE b:= b-a
END
```

1.3 Specification statements

From section 1.2 above, we can see that in formal terms we should have $[pre\ ,\ post] \sqsubseteq P$ iff for all R

$$wp([pre\ ,\ post]\ ,\ R) \Longrightarrow wp(P,\ R) \tag{5}$$

But for this to have meaning, we must define its antecedent; as in the definition (2) above for assignment statements, we will express $wp([pre\ ,\ post]\ ,\ R)$

as a syntactic transformation of the predicate R. We do this below, moving from simple to more general cases.

1.4 The simple case

In the simplest case we have two predicates *pre* and *post* each over the program variables in a single state. We have

Definition 2 *Let the vector of currently declared program variables be* $\vec{v}$*; for any predicates* pre, post, *and* R, *we define*

$$wp([pre, post], R) \quad = \quad pre \wedge (\forall \vec{v} \,.\, post \Rightarrow R)$$

♡

Note that our quantifiers always extend in scope to the first enclosing parentheses $(\forall . \cdots)$. As indicated, we will use $\vec{v}$ to refer to the vector of all program variables, and will not concern ourselves very much with how they are declared.

Section 2 discusses the consistency of definition 2 and formula (5); here we will justify the definition only informally. We regard $[pre, post]$ as a statement, and its first component *pre* describes the states in which its termination is guaranteed; thus *pre* is a necessary feature of our desired weakest pre-condition, and in fact appears as the first conjunct there. But the weakest pre-condition must guarantee more than termination: it must ensure that on termination, R holds. From the second component of $[pre, post]$, we know that *post* describes the states in which it terminates — and so we require only that in all states described by *post* the desired R holds as well: this is the second conjunct.

We now continue with some notational extensions and abbreviations.

1.4.1 Confining change

We allow a list of variables $\vec{w}$, in which appear all the variables which the statement can change; variables not in $\vec{w}$ must retain their initial values. The precise definition of $\vec{w} : [pre, post]$ is

Definition 3 *Let the vector of currently declared program variables be* $\vec{v}$*, and let* $\vec{w}$ *be a sub-vector of* $\vec{v}$*; for any predicates* pre, post, *and* R, *we define*

$$wp(\vec{w} : [pre, post],\ R) \quad = \quad pre \wedge (\forall\, \vec{w}\ .\ post \Rightarrow R)$$

♡

The only change from definition 2 is that the vector of quantified variables is now $\vec{w}$ rather than $\vec{v}$. Taking for example $\vec{v}$ to be "x, y", we have

$$\begin{array}{ll} & wp(x : [true, x = y],\ R) \\ = & true \wedge (\forall\, x\ .\ x = y \Rightarrow R) \\ = & R[x \backslash y]. \end{array}$$

Since also $wp(x := y,\ R) = R[x \backslash y]$, we have shown "$x : [true, x = y]$" and "$x := y$" to have the same meaning.

1.4.2 Initial values

So far, we can specify only that a certain relationship (*e.g.*, *post*) is to hold between the *final* values of variables. We now adjust our definition so that 0-subscripted variables in the second component of a specification statement can be taken as referring to the *initial* values of variables.

Definition 4 *Let the vector of currently declared program variables be* $\vec{v}$*, and let* $\vec{w}$ *be a sub-vector of* $\vec{v}$*; let* pre *and* R *as before be arbitrary predicates, and let* post *be a predicate referring optionally to 0-subscripted variables* $\vec{v}_0$ *as well. We define*

$$wp(\vec{w} : [pre, post],\ R) \ = \ pre \wedge (\forall\, \vec{w}\ .\ post \Rightarrow R)[\vec{v}_0 \backslash \vec{v}]$$

— *provided* R *contains no 0-subscripted variables* $\vec{v}_0$.
♡

By our definition we have *reserved* the use of 0-subscripts to denote initial values, and so must forego their use for other purposes: this is why R

should contain no $\vec{v}_0$. It is possible, however, to take the view that in R also the variables $\vec{v}_0$ refer to initial values; this leads in fact to the weakest pre-*specification* of Hoare and He [25]. Josephs [27] has investigated this.

We note that if *post* does not refer to initial values, then definition 4 reduces to definition 3.

The substitution $[\vec{v}_0 \backslash \vec{v}]$ may require renaming of the bound variables $\vec{w}$, but this is often unnecessary; for example, taking $\vec{v}$ to be "x, y" as before, we have

$$\begin{array}{ll} & wp(x : [true, x = x_0 + y_0],\ R) \\ = & true \wedge (\forall x \ .\ x = x_0 + y_0 \Rightarrow R)[x_0, y_0 \backslash x, y] \\ = & R[x \backslash x_0 + y_0][x_0, y_0 \backslash x, y] \\ = & R[x \backslash x + y]. \end{array}$$

This is of course $wp(x := x + y,\ R)$, as one would hope.

1.4.3 Implicit pre-conditions

If the pre-condition is omitted, we will supply a default condition for it as follows:

Definition 5 *Let the vector of currently declared program variables be $\vec{v}$, and let $\vec{w}$ be a sub-vector of $\vec{v}$; let* post *be a predicate referring optionally to 0-subscripted variables $\vec{v}_0$. We define*

$$\vec{w}\colon [post] \quad \text{abbreviates} \quad \vec{w}\colon [(\exists\, \vec{w} \bullet post)[\vec{v}_0 \backslash \vec{v}]\ ,\ post]$$

♡

Thus the *implicit* pre-condition is simply "it is possible to establish the post-condition". This is exactly the view taken in Z specifications generally, where only a single predicate is given; in our original square-root example (1) — writing it n: $[n^2 = n_0]$ — the implicit precondition is $(\exists\, n \ .\ n^2 = n_0)[n_0 \backslash n]$ which we can simplify to $(\exists\, k \ .\ k^2 = n)$. That is, termination is guaranteed only if n is a perfect square.

1.4.4 Generalised assignment

The assignment statement $x := E$ establishes the post-condition $x = E$ while changing only x — it has the same meaning, therefore, as the specification statement x: $[x = E[x\backslash x_0]]$ (in which the renaming $[x\backslash x_0]$ is necessary because occurrences of x in E are *initial* values). Exploiting this, we define below a *generalised* assignment statement in which the binary relation $=$ of ordinary assignment can be replaced by any binary relation desired.

Definition 6 *If "$\lhd$" is a binary relation symbol, then for any variable x and expression E,*

$$x :\lhd E \quad \text{abbreviates} \quad x\text{: } [x \lhd E[x\backslash x_0]] .$$

♡

Thus we have that

$x :\prec x$ decreases x; and that
$m :\in s$ chooses a member m from the set s.

Note that in the second case, our implicit pre-condition is "the set s is not empty":

$$\begin{array}{ll} & m :\in s \\ = & m\text{: } [m \in s] \\ = & m\text{: } [(\exists\, m' \bullet m' \in s) \ , \ m \in s] \\ = & m\text{: } [s \neq \{\} \ , \ m \in s] \end{array}$$

This abbreviation was suggested by Jean-Raymond Abrial.

1.4.5 Invariants

Often a formula appears as a conjunct in both the pre- and the post-conditions, thus making it an *invariant* of the statement. The following convention, suggested in [20], allows us to write it only once; we abbreviate $[pre \wedge I \ , \ I \wedge post]$ by

$$[pre \ , \ I \ , \ post]$$

Thus $[pre\ ,\ I\ ,\ post] \sqsubseteq Q$ iff

$$pre \wedge I \Longrightarrow wp(Q,\ I \wedge post).$$

The above convention is useful when developing loops, as we will see in section 3.

2 The refinement theorems

The following theorems justify our choice of semantics for the specification statement. (Their full proofs may be found in [35].) The first theorem shows that for every specification there is a specification statement that satisfies it trivially.

Theorem 1 *If $\vec{u}$ and $\vec{w}$ partition the vector $\vec{v}$ of program variables, then for any predicates* pre *and* post

$$pre \wedge \vec{v} = \vec{v}_0 \quad \Longrightarrow \quad wp(\vec{w} : [pre, post],\ post \wedge \vec{u} = \vec{u}_0)$$

Proof *(outline): The result follows by straightforward application of definition 4 and predicate calculus, except for the possible occurrences of 0-subscripted variables in post $\wedge\ \vec{u} = \vec{u}_0$. Since these are not* program variables *(we never declare* e.g. *x_0 in a program), we can avoid the problem by a systematic renaming, proving instead that*

$$pre \wedge \vec{v} = \vec{v}_1 \quad \Longrightarrow \quad wp(\vec{w} : [pre, post],\ post[\vec{v}_0 \backslash \vec{v}_1] \wedge \vec{u} = \vec{u}_1)$$

This technique is used also in the proof of theorem 8 in section 5.1, given in full.
♡

The consistency mentioned in section 1.4 follows easily from the above, taking $\vec{w} = \vec{v}$ and *post* free of $\vec{v}_0$; clearly other specialisations are profitable as well.

The complementary problem is refining further a given specification statement; the following theorem shows how this can be done.

Theorem 2 *If $\vec{w}$ and $\vec{u}$ partition the program variables $\vec{v}$, and if*

$$pre \wedge \vec{v} = \vec{v}_0 \quad \Longrightarrow \quad wp(P, post \wedge \vec{u} = \vec{u}_0)$$

then

$$\vec{w} : [pre, post] \sqsubseteq P$$

Proof *(outline): The proof again simply applies definitions, this time definitions 1 and 4; the 0-subscripts are avoided as before.*
♡

To summarise: theorem 1 shows that $\vec{w}$: $[pre\ ,\ post]$ is always *a* solution to the specification (of P):

$$pre \wedge \vec{v} = \vec{v}_0 \quad \Longrightarrow \quad wp(P,\ post \wedge \vec{u} = \vec{u}_0)$$

Theorem 2 shows it to be *more general* than any other solution; thus overall we have that is it the most general solution.

3 The refinement calculus

We now move to our main concern. With the definitions of section 1 we can mix specifications and executable constructs freely, and program development becomes a process of transformation within the one framework. But this is only the beginning — the definitions supply the "first principles" from which more specialised techniques spring, and we can use these derived *laws of refinement* directly in our development of programs. Each law is designed to introduce a particular feature into our final program, and the process overall comes to resemble the *natural deduction* style of formal proof, where our goals are not axioms but rather directly executable constructs (the Vienna Development Method [26] has a similar flavour).

We will present the laws in the form

$$\frac{before\text{-}refinement}{after\text{-}refinement} \quad side\text{-}condition$$

and by this we mean: "if *side-condition* is universally valid, then

before-refinement $\sqsubseteq$ *after-refinement*"

Often, there is no side-condition — this indicates that the stated refinement always obtains.

3.1 Strengthening the specification

Generally speaking, refinement *strengthens* a specification, and it is characteristic of our refinement calculus that no check is made against strengthening a specification too much (a notable difference from VDM). The advantage of this is simplicity of the laws (law 11 provides a striking example); a disadvantage is that unproductive refinement steps may go longer unnoticed. But there is no danger of invalidity resulting from over-strengthened specifications, for we will see that they can never provably be refined to executable code.

There is a simple *feasibility test* that can be applied to any specification, and its failure predicts the failure of the refinement process: we simply check that the specification satisfies Dijkstra's *Law of the Excluded Miracle* [12, p. 18] (paraphrased)

"For all executable programs P,

$wp(P, \mathit{false}) = \mathit{false}$"

If the specification failed this law, then so would any refinement of it; and since no *executable* program fails the law, we are forced to conclude that such a specification can never be refined to an executable program. For specifications, direct calculation yields that $\vec{w}$: $[\mathit{pre}\ ,\ \mathit{post}]$ is feasible iff $\mathit{pre} \Rightarrow (\exists\, \vec{w}\ .\ \mathit{post})[\vec{v}\backslash\vec{v}_0]$. This was first pointed out by Robinson [47].

The essence of our advantage is therefore that our laws do *not* force us implicitly to apply a feasibility test at their every application: very often the correctness of a development step is obvious. Further discussion on this topic can be found in [35].

Our first two laws deal with weakening the pre-condition and/or strengthening the postcondition of a specification.

Law 1 *Weakening the pre-condition; the new specification is more robust than the old (*i.e., *it terminates more often):*

$$\frac{\vec{w}\colon [pre\ ,\ post]}{\vec{w}\colon [pre'\ ,\ post]} \quad pre \Rightarrow pre'$$

♡

For example, $n\colon [n > 0\ ,\ n = n_0 - 1] \sqsubseteq n\colon [n \geq 0\ ,\ n = n_0 - 1]$.

Law 2 *Strengthening the post-condition; the new specification allows less choice than the old:*

$$\frac{\vec{w}\colon [pre\ ,\ post]}{\vec{w}\colon [pre\ ,\ post']} \quad pre \Rightarrow (\forall\, \vec{w} \bullet post' \Rightarrow post)\,[\vec{v}_0 \backslash \vec{v}]$$

♡

For example, $n\colon [true\ ,\ n \geq 0] \sqsubseteq n\colon [true\ ,\ n > 0]$.

It is worth noting that a special case of law 2 occurs when $\vec{v}$ and $\vec{w}$ are the same; then we have for the side-condition

$$pre \Longrightarrow (\forall\, \vec{v} \bullet post' \Rightarrow post)\,[\vec{v}_0 \backslash \vec{v}]$$

Renaming $\vec{v}$ to $\vec{v}_0$ throughout, this is equivalent to

$$pre[\vec{v} \backslash \vec{v}_0] \Longrightarrow (\forall\, \vec{v} \bullet post' \Rightarrow post)\,[\vec{v}_0 \backslash \vec{v}][\vec{v} \backslash \vec{v}_0]$$

which we may simplify to

$$pre[\vec{v} \backslash \vec{v}_0] \Longrightarrow (\forall\, \vec{v} \bullet post' \Rightarrow post)$$

The quantifier $\forall\, \vec{v}$ can be discarded since the antecedent contains no $\vec{v}$, and propositional calculus then gives us as our special case the appealing

$$pre[\vec{v} \backslash \vec{v}_0] \wedge post' \Longrightarrow post$$

Law 3 *Restricting change; the new specification can change fewer variables than the old:*

$$\frac{\vec{w}, x\colon [pre\ ,\ post]}{\vec{w}\colon [pre\ ,\ post]}$$

♡

For example, $x, y\colon [x = y_0] \sqsubseteq x\colon [x = y_0]$.

In law 4 below , we use the compact symbols [[and]], instead of the more conventional **begin** and **end**, to delimit the scope of local variable declarations.

Law 4 *Introducing fresh local variables (where "fresh" means not otherwise occurring free):*

$$\frac{\vec{w}\colon [pre\ ,\ post]}{[\![\mathbf{var}\ x;\quad \vec{w}, x\colon [pre\ ,\ post]\ ]\!]} \qquad x \text{ is a fresh variable}$$

♡

For example, $f\colon [f = n!] \quad \sqsubseteq \quad [\![\mathbf{var}\ i;\quad f, i\colon [f = n!]\ ]\!]$.

3.2 Introducing executable constructs

The following laws allow us to introduce constructs from our target programming language.

Law 5 *Introducing* **abort**:

$$\frac{\vec{w}\colon [false\ ,\ post]}{\mathbf{abort}}$$

♡

Since **abort** $\sqsubseteq P$ for any P, we can by transitivity of $\sqsubseteq$ have *any* program as the target of law 5. Thus for any predicate *difficult*(n), we still have the easy refinement $n\colon [n < 0 \wedge n > 0\ ,\ \mathit{difficult}(n)] \quad \sqsubseteq \quad n := 17$.

Law 6 *Introducing* **skip**:

$$\frac{\vec{w}\colon\ [post[\vec{v}_0\backslash\vec{v}]\ ,\ post]}{\textbf{skip}}$$

♡

For example, $x, y\colon\ [x = y\ ,\ x = y_0] \quad \sqsubseteq \quad \textbf{skip}$.

Law 7 *Introducing assignment*:

$$\frac{\vec{w}\colon\ \left[post[\vec{v}_0, \vec{w}\backslash\vec{v}, \vec{E}]\ ,\ post\right]}{\vec{w} := \vec{E}}$$

♡

For example, $x\colon\ [true\ ,\ x = x_0 + y] \quad \sqsubseteq \quad x := x + y$.

The next two laws are the *weakest pre-specification* and *weakest post-specification* constructions of Hoare and He [25], with which one can "divide" one specification A hy another B, leaving a specification Q such that

$$A \sqsubseteq Q;\ B \qquad \text{(law 8: weakest pre-specification)}$$
$$A \sqsubseteq B;\ Q \qquad \text{(law 9: weakest post-specification)}$$

Law 8 *Introducing sequential composition (weakest pre-specification)*:

$$\frac{\vec{w}\colon\ [pre\ ,\ post]}{\vec{w}\colon\ [pre\ ,\ wp(P, post)];\ P} \qquad \vec{w}\colon\ [true] \sqsubseteq P$$

♡

The side condition $w\colon\ [true] \sqsubseteq P$ can be read "P changes only $\vec{w}$". For example, we have

$$x, y\colon\ [true\ ,\ x = y + 1]$$

$$\sqsubseteq \quad x, y\colon\ [true\ ,\ x = 2];$$
$$\qquad y := 1$$

Law 9 *Introducing sequential composition (weakest post-specification):*

$$\frac{\vec{w}\colon [pre\ ,\ post]}{\begin{array}{l}\vec{w}\colon [pre\ ,\ mid]\,;\\ \vec{w}\colon [mid\ ,\ post]\end{array}} \qquad mid,\ post \text{ contain no free } \vec{v}_0$$

♡

For example,

$$x\colon [true\ ,\ x = y + 1]$$

$$\sqsubseteq\ \begin{array}{l} x\colon [true\ ,\ x = y]\,;\\ x\colon [x = y\ ,\ x = y + 1]\end{array}$$

Law 9 can be generalised to the case in which variables $\vec{v}_0$ do appear (as shown in [32]); in that case, one has effectively supplied in *mid* the first component of the sequential composition. For our larger example to follow (section 4), we need only the simpler version.

In laws 10 and 11, we use a quantifier-like notation for generalised disjunction and alternation: if I for example were the set $\{1..n\}$, then $(\bigvee i\colon I\ .G_i)$ would abbreviate $G_1 \vee \cdots \vee G_n$, and **if** $([]\ i\ .G_i \rightarrow S_i)$ **fi** would abbreviate

$$\begin{array}{lll}\textbf{if} & G_1 & \rightarrow\ S_1\\ [] & \cdots & \\ [] & G_n & \rightarrow\ S_n\\ \textbf{fi} & & \end{array}$$

Law 10 *Introducing alternation (***if***):*

$$\frac{\vec{w}\colon [pre \wedge (\bigvee i\colon I\ .G_i)\ ,\ post]}{\textbf{if}\ ([]\ i\colon I\ .G_i \rightarrow \vec{w}\colon [pre \wedge G_i\ ,\ post])\ \textbf{fi}}$$

♡

The predicate *pre* is that part of the pre-condition irrelevant to the case distinction being made by the guards G_i: it is passed on to the branches of

the alternation. For example, taking *pre* to be true, we have

$$
\begin{array}{ll}
 & y\colon [x = 0 \vee x = 1\ ,\ x + y = 1] \\
\sqsubseteq & \textbf{if } x = 0 \rightarrow y\colon [x = 0\ ,\ x + y = 1] \\
 & \ [\!]\ x = 1 \rightarrow y\colon [x = 1\ ,\ x + y = 1] \\
 & \textbf{fi} \\
 & \\
\sqsubseteq & \textbf{if } x = 0 \rightarrow y := 1 \\
 & \ [\!]\ x = 1 \rightarrow y := 0 \\
 & \textbf{fi}
\end{array}
$$

Law 11 *Introducing iteration* (do)*:*

$$
\frac{\vec{w}\colon [\mathit{true}\ ,\ \mathit{inv}\ ,\ \neg(\bigvee i\colon I\ .\ G_i)]}{\begin{array}{l}\textbf{do} \\ \quad ([\!]\ i\colon I\ .\ G_i \rightarrow \vec{w}\colon [G_i\ ,\ \mathit{inv}\ ,\ 0 \leq \mathit{var} < \mathit{var}_0]) \\ \textbf{od}\end{array}}
$$

♡

The predicate *inv* is of course the loop invariant, and the expression *var* is the variant. We use var_0 to abbreviate $var[\vec{v}\backslash\vec{v}_0]$.

An example of law 11 is given in section 4; for now, we note that *inv* can be any predicate and *var* any integer-valued expression. Surprisingly, there are no side-conditions — a bad choice of *inv* or *var* or indeed G_i simply results in a loop body from which no executable program can be developed (see the remarks in section 3.1).

Law 11 is proved in section 5.

4 An example: square root

For an example, we take the square-root development of [12, pp. 61–65]; but our development here will be deliberately terse, because we are illustrating not how to *find* such developments (properly the subject of a whole book), but rather how experienced programmers could *record* such a development.

4.1 Specification

We are given a non-negative integer sq; we must set the integer variable rt to the greatest integer not exceeding $\sqrt{sq}$, where the function $\sqrt{}$ takes the non-negative square root of its argument.

4.2 Specification

$$rt := \lfloor \sqrt{sq} \rfloor$$

$\lfloor x \rfloor$ — the "floor" of x — is the greatest integer not exceeding x.

4.3 Refinement

We assume of course that $\sqrt{}$ is unavailable to us, and proceed as follows to eliminate it from our specification; we eliminate $\lfloor\ \rfloor$ also. "Stacked" predicates denote conjunction.

$$\begin{array}{lll}
 & rt := \lfloor \sqrt{sq} \rfloor & \\
= & rt{:}\ [rt = \lfloor \sqrt{sq} \rfloor\ ] & \text{definition 6} \\
= & rt{:}\ [sq \geq 0\ ,\ rt = \lfloor \sqrt{sq} \rfloor\ ] & \text{definition 5} \\
= & rt{:}\ [sq \geq 0\ ,\ rt \leq \sqrt{sq} < rt + 1] & \text{definition of } \lfloor\ \rfloor \\
\sqsubseteq & rt{:}\ \left[sq \geq 0\ ,\ \begin{array}{c} 0 \leq rt \\ rt^2 \leq sq < (rt+1)^2 \end{array} \right] & \text{law 2}
\end{array}$$

4.4 Refinement

Using laws 4 and 2, we introduce a new variable ru, and strengthen the post-condition; our technique will be to approach the result from above (ru)

and below (rt):

$$\sqsubseteq \quad |[\ \mathbf{var}\ ru\ .\ rt, ru{:}\ \left[\begin{array}{c} sq \geq 0\ , \quad \begin{array}{c} 0 \leq rt \\ rt^2 \leq sq < ru^2 \\ rt + 1 = ru \end{array} \end{array}\right]\ ]|$$

We now work on the inner part.

4.5 Refinement

Anticipating use of $rt + 1 \neq ru$ as a loop guard we concentrate on the remainder of the post-condition, using law 9 with $mid = \left(\begin{array}{c} 0 \leq rt < ru \\ rt^2 \leq sq < ru^2 \end{array}\right)$ to proceed:

$$\sqsubseteq \quad rt, ru{:}\ \left[\begin{array}{c} sq \geq 0\ , \quad \begin{array}{c} 0 \leq rt < ru \\ rt^2 \leq sq < ru^2 \end{array} \end{array}\right]; \tag{6}$$

$$rt, ru{:}\ \left[\begin{array}{c} \begin{array}{c} 0 \leq rt < ru \\ rt^2 \leq sq < ru^2 \end{array} \quad , \quad \begin{array}{c} 0 \leq rt < ru \\ rt^2 \leq sq < ru^2 \\ rt + 1 = ru \end{array} \end{array}\right]$$

Using laws 1 and 7, we can show that for the first component of the sequential composition above — establishing *mid*, to become the loop invariant — we have

$$\sqsubseteq rt, ru := 0, sq + 1$$

We now concentrate on the second component.

4.6 Refinement

We now introduce the loop, rewriting the second component of the sequential composition (6) to bring it into the form required by law 11; writing *inv*

now for our *mid* above, we have

$$= \quad rt, ru\colon [true\ ,\ inv\ ,\ rt + 1 = ru]$$

and then by 11, with variant $ru - rt$, we proceed

$$\begin{array}{ll} \sqsubseteq & \textbf{do } rt + 1 \neq ru \ \rightarrow \\ & \quad rt, ru\colon [rt + 1 \neq ru\ ,\ inv\ ,\ 0 \leq ru - rt < ru_0 - rt_0] \\ & \textbf{od} \end{array}$$

4.7 Refinement

For the loop body, we use law 4 again to introduce a local variable rm to "chop" the interval $rt..ru$ in which the result lies:

$$\begin{array}{ll} \sqsubseteq & [\![\ \textbf{var } rm; \\ & \quad rt, ru, rm\colon [rt + 1 \neq ru\ ,\ inv\ ,\ 0 \leq ru - rt < ru_0 - rt_0] \\ &]\!] \end{array}$$

We first choose rm between rt and ru, using law 9 then law 3 twice to develop:

$$\begin{array}{ll} \sqsubseteq & rm\colon [rt + 1 \neq ru\ ,\ inv\ ,\ rt < rm < ru]; \\ & rt, ru\colon [rt < rm < ru\ ,\ inv\ ,\ 0 \leq ru - rt < ru_0 - rt_0] \end{array}$$

Then with laws 1 and 7, we quickly dispose of the first component, deciding to make our choice of rm divide the interval evenly:

$$\sqsubseteq rm := (rt + ru) \operatorname{div} 2$$

We proceed with the second component.

4.8 Refinement

The natural case analysis is now to consider $rm^2 \leq sq$ versus $rm^2 > sq$; accordingly, with law 10, we so divide our task and immediately apply law

3 to each case; we have

$$\sqsubseteq \ \textbf{if}\ rm^2 \le sq \ \rightarrow\ rt{:}\left[\begin{array}{c} rt < rm < ru \\ rm^2 \le sq \end{array}\ ,\ inv\ ,\ 0 \le ru - rt < ru_0 - rt_0\right]$$

$$[]\ \ rm^2 > sq \ \rightarrow\ rt{:}\left[\begin{array}{c} rt < rm < ru \\ rm^2 > sq \end{array}\ ,\ inv\ ,\ 0 \le ru - rt < ru_0 - rt_0\right]$$

$$\textbf{fi}$$

For the first branch, we have by law 7

$$\sqsubseteq rt := rm$$

For the second branch, we have similarly

$$\sqsubseteq ru := rm$$

This completes our development.

4.9 Consolidation: the implementation

Developments in this style generate a tree structure in which children collectively refine their parents; to obtain the program "neat," we simply flatten the tree. For the square root program, the result is as follows:

```
|[ var ru;
    rt, ru := 0, sq + 1;
    do rt + 1 ≠ ru →
        |[ var rm;
            rm := (rt + ru) div 2;
            if rm² ≤ sq → rt := rm
            [] rm² > sq → ru := rm
            fi
        ]|
    od
]|
```

It is to be stressed that this consolidated presentation is not to be carried off as the only relic of our development. The development itself must remain

as a record of design steps taken and their justifications (and in industrial practice, of who took them!). Mistakes will still be made, and corrections applied; only when a complete record is kept can we make those corrections reliably, without introducing further errors — and learn from the process.

5 Derivation of laws

In this section we will prove the laws 2 and 11 of section 3. We do this for several reasons: to reassure the reader, who may doubt their validity; to demonstrate the use of the weakest pre-condition formula for specifications; and to suggest that the collection of laws can easily be extended by similar proofs.

5.1 Proof of law 2

Law 2 allows us to strengthen the post-condition of a specification; in simplest terms, this means replacing *post* by *post'* as long as we know that $post' \Rightarrow post$. The side-condition is weaker than this, however: it takes both the pre-condition and changing variables into account, making the law more widely applicable.

In the proof below, we will assume that free-standing formulae are *closed* — that is, that their free variables are implicitly quantified (universally). It is this that will allow us to rename variables when necessary.

Theorem 3 *Proof of law 2: if the following side-condition holds*

$$pre \Longrightarrow (\forall\, \vec{w} \bullet post' \Rightarrow post)\,[\vec{v}_0 \backslash \vec{v}]$$

then so does this refinement:

$$\vec{w}\colon [pre\ ,\ post] \sqsubseteq \vec{w}\colon [pre\ ,\ post']$$

Proof *By theorem 2, we need only show*

$$pre \wedge \vec{v} = \vec{v}_0 \Longrightarrow wp(\vec{w}\colon [pre\ ,\ post'],\ post \wedge \vec{u} = \vec{u}_0)$$

Since in definition 4 the predicate R must not contain $\vec{v}_0$, we rename those above to $\vec{v}_1$ (we may do this because the formula is closed); we must show

$$pre \wedge \vec{v} = \vec{v}_1 \Longrightarrow wp(\vec{w}\colon\ [pre\ ,\ post'],\ post \wedge \vec{u} = \vec{u}_1)$$

Definition 4 is now applied; we must show

$$pre \wedge \vec{v} = \vec{v}_1 \Longrightarrow pre \wedge (\forall\, \vec{w}\ \bullet\ post' \Rightarrow post \wedge \vec{u} = \vec{u}_1)\,[\vec{v}_0 \backslash \vec{v}]$$

Clearly we can remove the conjunct pre in the consequent, because it occurs in the antecedent; we can remove $\vec{u} = \vec{u}_1$ because $\vec{u}$ and the quantified $\vec{w}$ are disjoint, and $\vec{v} = \vec{v}_1$ appears in the antecedent. It remains to prove

$$pre \wedge \vec{v} = \vec{v}_1 \Longrightarrow (\forall\, \vec{w}\ \bullet\ post' \Rightarrow post)\,[\vec{v}_0 \backslash \vec{v}]$$

And this follows directly from the side-condition.
♡

5.2 Proof of law 11

We will deal with the following restricted version of law 11, in which we consider a single guard only and take $\vec{v}$ and $\vec{w}$ the same; we must show

$$\frac{[true\ ,\ inv\ ,\ \neg guard]}{\begin{array}{l}\textbf{do}\\ \quad guard\ \rightarrow\ [guard\ ,\ inv\ ,\ 0 \leq var < var_0]\\ \textbf{od}\end{array}}$$

Our proof is based on the loop semantics given in [12]; we will show that for $k \geq 1$

$$inv \wedge (guard \Rightarrow var < k) \Longrightarrow H_k(inv \wedge \neg guard) \qquad (7)$$

From this will follow

$$\begin{array}{ll} & inv\\ = & inv \wedge (guard \Rightarrow (\exists\, k \geq 1\ \bullet\ var < k))\\ = & (\exists\, k \geq 1\ \bullet\ inv \wedge (guard \Rightarrow var < k))\\ \Rightarrow & (\exists\, k\ \bullet\ H_k(inv \wedge \neg guard))\\ = & wp(\textbf{do}\cdots\textbf{od},\ inv \wedge \neg guard)\end{array}$$

Thus by theorem 1 we will have as required that

$$[inv\ ,\ inv \wedge \neg guard] \sqsubseteq \textbf{do}\cdots\textbf{od}$$

It remains therefore to prove (7), and this we will do by induction over k. We note first that $H_0 = inv \wedge \neg guard$, and continue by direct calculation (writing pre' for $pre[\vec{v}\backslash\vec{v}']$ *etc.*, and H_k for $H_k(inv \wedge \neg guard)$):

$$
\begin{array}{lll}
 & H_1 & \\
 & & \\
= & H_0 \vee \left(\begin{array}{c} guard \\ guard \wedge inv \\ (\forall\, \vec{v} \bullet inv \wedge 0 \leq var < var_0 \Rightarrow H_0)\,[\vec{v}_0\backslash\vec{v}] \end{array} \right) & \\
 & & \\
= & H_0 \vee \left(\begin{array}{c} guard \wedge inv \\ (\forall\, \vec{v}' \bullet inv' \wedge 0 \leq var' < var \Rightarrow H_0') \end{array} \right) & \\
 & & \\
\Leftarrow & (\neg guard \wedge inv) \vee (guard \wedge inv \wedge var < 1) & \\
= & inv \wedge (guard \Rightarrow var < 1) &
\end{array}
$$

Our inductive step now concludes the argument:

$$
\begin{array}{lll}
 & H_{k+1} & \\
 & & \\
= & H_0 \vee \left(\begin{array}{c} guard \wedge inv \\ (\forall\, \vec{v}' \bullet inv' \wedge 0 \leq var' < var \Rightarrow H_k') \end{array} \right) & \\
 & & \\
\Leftarrow & H_0 \vee \left(\begin{array}{c} guard \wedge inv \\ \left(\forall\, \vec{v}' \bullet inv' \wedge 0 \leq var' < var \Rightarrow \left(\begin{array}{c} inv' \\ guard' \Rightarrow var' < k \end{array} \right) \right) \end{array} \right) & \\
 & & \\
\Leftarrow & H_0 \vee \left(\begin{array}{c} guard \wedge inv \\ var < (k+1) \end{array} \right) & \\
 & & \\
= & inv \wedge (guard \Rightarrow var < (k+1)) &
\end{array}
$$

♡

The puzzling thing about law 11 is that it has no side-condition, whereas one might expect to find the condition

$$guard \wedge inv \Rightarrow 0 \leq var$$

But closer inspection reveals that whenever the above formula fails, the loop body is infeasible: it must terminate (since $guard \wedge inv$ holds initially) and

must establish $0 \leq var < 0$ (since $0 \not\leq var$ holds initially). By the law of the excluded miracle (see [12]), no executable program can do this — the refinement, though valid, is barren.

For the practising developer, perhaps the side-condition should be explicit; indeed, law 11 can be rewritten this way, with the $0 \leq var$ dropped from the post-condition of the loop body. For the historical record of our development however, we want to prove the very minimum necessary — and feasibility is of no interest. There would be no program, and hence no record, if a feasibility check would have failed.

6 Conclusion

We have claimed that the integration of specifications and executable programs improves the development process. In earlier work [35], the point was made that all the established techniques of refinement are of course still applicable; their being based on weakest pre-condition semantics automatically makes them suitable for *any* construct so given meaning. Indeed an immediate but modest application of this work is our writing for example "choose e from s" directly in our development language as "$e :\in s$".

The refinement calculus is a step further. We are not claiming that it makes algorithms easier to discover, although we hope that this will be so; but it clearly does make it easier to avoid trivial mistakes in development and to keep a record of the steps taken there. A professional approach to software development must record the development *process*, and it must do so with mathematical rigor. We propose the refinement calculus for that at least.

Another immediate possibility is the systematic treatment of Z "case studies" as exercises in development, and we hope to learn from this. (There are a large number of case studies collected in [37].) Such systematic development is already underway for example at the IBM Laboratories at Hursley Park, UK [44].

The techniques of *data* refinement, in which high-level data structures (sets, bags, functions ...) are replaced with structures of the programming

language (arrays, trees ...), fit extremely well into this approach. Also facilitated is the introduction of procedures and functions into a development: the body of the procedure is simply a specification statement "yet to be refined," and the meaning of procedures can once more be given by the elegant *copy rule* of Algol-60. These ideas are explored in [32] and [47], and we hope to publish them more widely.

7 Acknowledgements

It is clear our approach owes its direction to the steady pressure exerted by the work of Abrial, Back, Dijkstra, Hoare, and Jones. More direct inspiration came from the *weakest pre-specification* work of Hoare and He [25], who provide a relational model and a calculus for development; they strongly advocate the *calculation* of refinements as an alternative to refinements proposed then proved. Robinson [47] has done earlier work on the refinement calculus specifically.

We believe the earliest embedding of specifications within Dijkstra's language of weakest pre-conditions to be that reported in Back's thesis [4], and to him we freely give the credit for it. His *descriptions* are single predicates, rather that the predicate pairs we use here, and he gives a very clear and comprehensive presentation of the resulting refinement calculus. Our work extends his in that we consider predicate *pairs*, as does VDM, but — unlike VDM — we do not require those pairs always to describe feasible specifications. Because of this, we obtain a significant simplification in the laws of our refinement calculus.

In [30] L. Meertens explores similar ideas, and we are grateful to him for making us aware of Back's work.

We have benefited from collaboration with the IBM Laboratory at Hursley Park; the joint project [44] aims to transfer research results directly from university to development teams in industry.

Morris [41] independently has taken a similar approach to ours (even to allowing infeasible *prescriptions*); we recommend his more abstract view, which complements our own.

To the referees, and to Stephen Powell of IBM, we are grateful for their helpful suggestions.

Procedures, parameters, and abstraction: separate concerns

Carroll Morgan

Abstract

The notions of *procedures*, *parameters*, and *abstraction* are by convention treated together in methods of imperative program development. Rules for preserving correctness in such developments can be complex.

We show that the three concerns can be separated, and we give simple rules for each. Crucial to this is the ability to embed *specification* — representing abstraction — directly within programs; with this we can use the elegant *copy rule* of ALGOL-60 to treat procedure calls, whether abstract or not.

Our contribution is in simplifying the use of the three features, whether separately or together, and in the proper location of any difficulties that do arise. The *aliasing* problem, for example, is identified as a "loss of monotonicity" with respect to program refinement.

Keywords: Programming methodology; procedure call; parameters; specification; aliasing.

1 Introduction

In developing imperative programs one identifies points of procedural abstraction, where the overall task can be split into subtasks each the subject

[0]Appeared in *Sci. Comp. Prog. 11* (1988)

of its own development subsequently. Integration of the subtasks is accomplished ultimately by parametrized procedure calls in the target programming language. We argue here that these concerns — ***procedures***, ***parametrization***, and ***abstraction*** — can be separated, and that the result is of practical utility.

Abstraction identifies a coherent algorithmic activity that can be split from the main development process; conventionally, a procedure call is left at the point of abstraction, and its necessary properties become the specification of the procedure body. Instead, we leave the specification itself at the point of abstraction, with no ***a priori*** commitment to a procedure call.

Procedure call we treat as simple substitution of text for a name, not caring whether we substitute programming language code (as we would in the final program) or a specification (as we would in a high-level design).

Parametrization we treat as a substitution mechanism that can be applied uniformly to specifications or to program language code, whether or not a procedure call occurs there.

The aim is to give a simple ***orthogonal*** set of rules for treating each concern. Existing practice is in most cases easily realised by appropriate combinations of the rules; but the independence allows greater freedom than before.

2 Procedure call

We return to the simple view, taken in the ALGOL-60 (revised) report [42], that procedure calls are to he understood via a ***copy rule:*** a program that calls a procedure is equivalent to one in which the procedure name is replaced by the text of the procedure body. In the examples to follow, we declare (parameterless) procedures using

procedure *name* $\triangleq$ *body*

With the copy rule, therefore, we have the equality indicated in the following example:

procedure *Inc* $\widehat{=}$ $x := x + 1$ ... $x := 0$; *Inc*; **write** x	=	$x := 0$; $x := x + 1$; **write** x	(1)

The technique has impeccable credentials; it is for example strongly (and deliberately) related to the following one-point rule of predicate calculus:

$$(\forall x \bullet x = T \Rightarrow P) \quad \Rightarrow \quad P[x \backslash T]$$

We write quantifications within parentheses () which delimit their scope, and use a spot • to separate the binding variable from the body. In the formula above, T is some term not containing x free, P a predicate, and $P[x \backslash T]$ the result of replacing x by T in P. We assume that the substitution $[x \backslash T]$ is defined so that it avoids variable capture; similar care is needed with the copy rule.

But the copy rule gives the meaning only of programs written entirely in a programming language. In contrast, the modern "step-wise" approach to program development introduces hybrid programs in which names denote program fragments "yet to be implemented". One understands the effect of these fragments in terms of their specification — abstracting from the detail of implementation — using rules specifically for procedure call such as those given in [22], [14], and [15]. The simple copy rule cannot be applied, for there is not yet program text to copy.

In [22], for example, one finds a *Rule of adaptation*, in the style of the rules of [21], with which procedures specified by pre- and post-conditions can be proved to have been used correctly in a calling program. There is also given in [22] a *Rule of substitution* for dealing separately with the effects of parameter passing. In the more recent [15] and [14], combined rules treat procedures — as specifications — with their parameters, all at once.

Here we reverse the trend, not only retaining the earlier view [22], which separates procedure calling (*adaptation*) from parameter passing (*substitution*), but also splitting procedure call from procedural abstraction. For

procedure calls, therefore, we retain only the copy rule of ALGOL 60 [42, 4.7.3.3]:

> ... the procedure body ... is inserted in place of the procedure statement ... If the procedure is called from a place outside the scope of any non-local quantity of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement ... will be avoided through suitable systematic changes of the latter identifiers.

3 Procedural abstraction

We take the axiomatic view: a procedural abstraction is described by a predicate pair comprising a *pre-condition* and a *post-condition*, both built (mainly) from program variables. We write such *specifications* using the notation $[pre, post]$. In the style of [12] a program P *satisfies* such a specification iff

$$pre \;\Rightarrow\; wp(P, post) \tag{2}$$

Paraphrasing [12, p. 16], we say that

> *pre* characterises a set of initial states such that activation of the mechanism P in any one of them will certainly result in a properly terminating happening leaving the system in a final state satisfying *post*.

But we adopt a different style [35] (similarly [4], [41]), writing more directly but equivalently

$$[pre, post] \;\sqsubseteq\; P \tag{3}$$

This we read "the specification $[pre, post]$ is satisfied by P". And we make specifications "first-class citizens", giving their semantics in the same way as all other programming constructs are defined in [12].

Definition 1 *Let* pre, post, *and* R *be predicates over the program variables* v. *We define the weakest pre-condition of the specification* $[pre, post]$ *with respect to the post-condition* R *as follows:*

$$wp([pre, post], R) \quad = \quad pre \wedge (\forall v. post \Rightarrow R)$$

♡

In that definition and below, single letters v refer to a vector of variables (possibly singleton). Definition 1 is discussed in detail in [41] and [35]; tbe latter allows a more general form in which *post* can refer to the initial state as well.

For the present, we give an informal justification of definition 1: we regard $[pre, post]$ as a statement, and its first component *pre* describes the initial states in which its termination is guaranteed; this is the first conjunct. Its second component *post* describes the final states in which that termination occurs, and so we require also that in all states described by *post* the desired R holds as well; this is the second conjunct.

We now define the relation "is satisfied by" – that is, $\sqsubseteq$ — as in [4], [41], [35], [20]:

Definition 2 *For programs or specifications* P1 *and* P2, *we say that* P1 *is* satisfied *by* P2, *or equivalently that* P2 refines P1, *iff for all post-conditions* R *we have*

$$wp(P1, R) \quad \Rightarrow \quad wp(P2, R)$$

We write this $P1 \sqsubseteq P2$.
♡

With definitions 1 and 2 we can prove that (2) and (3) are equivalent (see [35]). That equivalence allows us to take $[pre, post]$ as the trivial and most general solution for P in (2). Further, definition 1 agrees with the *Rule of adaptation* [22] and with the procedure call rule [14, 12.2.1] in the special case where the abstraction is in fact a procedure.

But we are not necessarily linking procedure call and procedural abstraction: procedure call is useful even when the procedure body is executable

We assume below that a and b are fixed.

$$
\begin{array}{ll}
 & [b^2 - 4ac \geq 0, ax^2 + bx + c = 0] \\
\sqsubseteq & \left[b^2 - 4ac \geq 0,\ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}\right] \qquad \text{(standard mathematics)} \\
\sqsubseteq & [b^2 - 4ac \geq 0,\ x^2 = b^2 - 4ac]; \qquad \text{(sequential composition)} \\
 & x := (x - b)/2a \\
\sqsubseteq & \textbf{procedure } Sqrt \mathrel{\widehat{=}} [b^2 - 4ac \geq 0,\ x^2 = b^2 - 4ac]; \qquad \text{(copy rule)} \\
 & Sqrt; \\
 & x := (x - b)/2a
\end{array}
$$

Figure 1: Development of quadratic-solver

code; and procedural abstraction is useful even if the implementation ultimately is "in-line". Consider the example of figure 1, in which we introduce a parameterless procedure *Sqrt*. There we use specifications $[pre, post]$ as fully-fledged program constructs, as indeed definition 1 allows us to do.

The conclusion of this exercise would be to refine the remaining specification, but the fact that it is the body of a procedure is now irrelevant:

$$
\begin{array}{ll}
 & [b^2 - 4ac \geq 0, x^2 = b^2 - 4ac] \\
\sqsubseteq & x := \sqrt{b^2 - 4ac}
\end{array}
$$

Thus we see that by allowing procedural abstractions — specifications — to mingle with ordinary program constructs, we can with the copy rule accommodate calls to procedures for which we do not yet have the executable code. The specification itself is the text we copy, and definition 1 gives meaning to the result.

4 Parameters

Parameters used to adapt a general program fragment to a particular purpose — whether or not that fragment is a procedure. Historically, procedures and parametrization are closely linked, and parameter *passing* means "parametrizing a procedure call".

Apparently the simplest example of parametrization is ordinary textual substitution. When substituting into *programs*, much the same rules apply as for substitution into formulae: only *global* (compare *free*) occurrences of x are affected; and capture of l must be avoided by systematic renaming of *local* (compare *bound*) variables. And if we are replacing a variable hy a *term*, then that variable cannot appear on the left of :=.

In example (1), we could use substitution to write instead

$y := 0;$
$y := y + 1;$
write y

$= \; y := 0;$ (parametrization)
$(x := x + 1)[x \backslash y];$
write y

$=$ **procedure** $Inc \mathrel{\widehat{=}} x := x + 1;$ (copy rule)
$y := 0;$
$Inc[x \backslash y];$
write y

In the final step above, the substitution suggests — intentionally — supplying an actual parameter y for a formal parameter x in the call of procedure *Inc*. But in the previous step, we see $[x \backslash y]$ as a simple substitution.

That style of parametrization, known as *call by name*, is unfortunately not as simple as it appears. Not only is it difficult to implement (requiring "thunks"), but it can be difficult to reason about, as well. If the actual parameters passed lead to distinct names within the procedure for the *same* variable, then the parametrization may lose the crucial property of *monotonicity:* we won't have that $P1 \sqsubseteq P2$ implies $P1[x \backslash T] \sqsubseteq P2[x \backslash T]$.

That phenomenon is known as *aliasing*, and is traditionally associated with procedure call; writers on program development advise us to avoid it. Because of aliasing, call by name (and similarly call by reference: Pascal's var) must be used with care. But, in fact, aliasing loses monotonicity — and *that* is why we should avoid it. We can separate the problem from procedure call.

Below we show by example that aliasing loses even equality (trivially, monotonicity also): we have

$$(x := 0;\ \ y := 1) \ \ = \ \ (y := 1;\ \ x := 0)$$

but

$$\begin{array}{ll} & (x := 0;\ \ y := 1)[x \backslash y] \\ = & y := 0;\ \ y := 1 \\ = & y := 1 \\ \neq & y := 0 \\ = & y := 1;\ \ y := 0 \\ = & (y := 1;\ \ x := 0)[x \backslash y] \end{array}$$

In the following sections, we define "substitution by value", "by result", and "by value/result"; and we prove that, unlike simple substitution, they are monotonic.

4.1 Substitution by value

For any program P, we write the substitution ***by value*** in P of term T for variable x as follows:

$$P[\textbf{value } x \backslash T]$$

For simplicity in tbe following sections, we use the notation $P < R >$ for $wp(P, R)$ (following [20]).

Definition 3 *Substitution by value: if* x *does not occur free in* R, *then*

$$P[\textbf{value } x \backslash T] < R > \ \ \widehat{=} \ \ P < R > [x \backslash T]$$

♡

Note that the substitution on the right above is ordinary substitution into the predicate $P < R >$: the weakest precondition is calculated first, then the substitution is made. That convention applies everywhere below.

Substitution by value can be implemented with the well-known *call* by value technique of assignment to an anonymous local variable. It is easily shown that for any program P, variable x, term T, and fresh local variable l, we have

$$
\begin{array}{ll}
 & P[\textbf{value } x \backslash T] \\
 & \\
= & \textbf{begin var } l; \\
 & \quad l := T; \\
 & \quad P[x \backslash l] \\
 & \textbf{end}
\end{array}
$$

That implementation, by using ordinary substitution only in a restricted way, avoids the problems we encountered above. First, since the variables l are fresh and distinct, there is no aliasing; second, since the replacing expressions are variables rather than general terms, there is no difficulty when the replaced variables occur on the left of :=.

But our main interest is in monotonicity:

Theorem 1 *Substitution by value is monotonic: if* $P \sqsubseteq Q$ *then*

$$P[\textbf{value } x \backslash T] \quad \sqsubseteq \quad Q[\textbf{value } x \backslash T]$$

Proof: Immediate from definitions 3, 1 and the monotonicity (over $\Rightarrow$) of substitution into predicates: if for predicates X and Y we have $X \Rightarrow Y$, then for any variable v and term T we have also $X[v \backslash T] \Rightarrow Y[v \backslash T]$.
♡

4.2 Substitution by result

For any program P, we write substitution *by result* in P of variable y for variable x as follows:

$$P[\textbf{result } x \backslash y]$$

This is a more restricted form of substitution than substitution by value, because we substitute a variable y rather than a term T. It is defined as follows:

Definition 4 *Substitution by result: if* x *does not occur free in* R, *then*

$$P[\textbf{result } x \backslash y] < R > \quad \widehat{=} \quad (\forall x \bullet P < R[y \backslash x] >)$$

♡

Substitution by result can be implemented by the call by result technique of assignment from an anonymous local variable. It can be shown that for any program P, variable x, term T, and fresh local variable l, we have

$$P[\textbf{result } x \backslash y]$$

$$= \quad \textbf{begin var } l;\ P[x \backslash l];\ y := l \ \textbf{end}$$

For monotonicity, we have

Theorem 2 *Substitution by result is monotonic: if* $P \sqsubseteq Q$ *then*

$$P[\textbf{result } x \backslash y] \ \sqsubseteq \ Q[\textbf{result } x \backslash y]$$

Proof: Immediate from definition 4, as for theorem 1.
♡

4.3 Substitution by value/result

For any program P, we write the substitution *by value/result* in P of term y for variable x as follows:

$$P[\textbf{value result } x \backslash y]$$

Substitution by value/result is a combination of the two substitutions above, and is well-behaved in the same way. We have

Definition 5 *Substitution by value/result: if* x *does not occur free in* R, *then*

$$P[\textbf{value result } x\backslash y] < R > \;\hat{=}\; P < R[y\backslash x] > [x\backslash y]$$

♡

Theorem 3 *Substitution by value/result is monotonic: if* $P \sqsubseteq Q$ *then*

$$P[\textbf{value result } x\backslash y] \;\sqsubseteq\; Q[\textbf{value result } x\backslash y]$$

Proof: Immediate from definition 5, as for theorem 2.
♡

The equivalent program fragment is given by

$$P[\textbf{value result } x\backslash T]$$

$$\begin{array}{ll} = & \textbf{begin var } l; \\ & \quad l := y; \\ & \quad P[x\backslash l]; \\ & \quad y := l \\ & \textbf{end} \end{array}$$

4.4 Apparent limitations

Each of the definitions 3, 4, 5 contains the limitation "if x does not occur free in R". Thus with them we cannot calculate

$$(y := x)[\textbf{value } x\backslash 0] < x = 0 > \qquad (4)$$

It's clear that the weakest precondition in (4) above should be $x = 0$. But calculation (using definition 3 erroneously) reveals instead

$$\begin{array}{ll} & (y := x)[\textbf{value } x\backslash 0] < x = 0 > \\ = & (y := x) < x = 0 > [x\backslash 0] \\ = & (x = 0)[x\backslash 0] \\ = & (0 = 0) \\ = & \textit{true} \end{array}$$

We avoid such problems by extending definitions 3–5 uniformly.

Definition 6 *If the substitution type* **sub** *is* **value**, **result**, *or* **value result**, *we have*

$$P[\textbf{sub } x \backslash T] < R > \;\; \widehat{=} \;\; P[x \backslash l][\textbf{sub } l \backslash T] < R >$$

where l *is a fresh variable, not appearing in* P, T, x, *or* R.
♡

The monotonicity properties persist, and for (4) we now have

$$\begin{array}{ll} & (y := x)[\textbf{value } x \backslash 0] < x = 0 > \\ = & (y := l)[\textbf{value } l \backslash 0] < x = 0 > \\ = & (y := l) < x = 0 > [l \backslash 0] \\ = & (x = 0)[l \backslash 0] \\ = & (x = 0) \end{array}$$

A second limitation is that we have not treated *multiple* parametrization. For example, we cannot calculate

$$(y := x + 1)[\textbf{value } x, \textbf{result } y \backslash x, x] \qquad (5)$$

We use the normal notation for multiple substitutions: in the above, x replaces x by value and y by result.

We proceed as for simple (multiple) substitutions: for formula P, distinct variables x, y, and terms T, U we know that

$$P[x, y \backslash T, U] \;\; = \;\; P[x \backslash l][y \backslash m][l \backslash T][m \backslash U]$$

for fresh variables l and m. Our definition is therefore

Definition 7 *For any substitution types* **sub1** *and* **sub2**, *distinct variables* x *and* y, *and terms* T, U *we have*

$$P[\textbf{sub1 } x, \textbf{sub2 } y \backslash T, U] \;\; \widehat{=} \;\; P[x \backslash l][y \backslash m][\textbf{sub1 } l \backslash T][\textbf{sub2 } m \backslash U]$$

where l, m *are fresh variables.*
♡

The definition is easily generalised to more than two simultaneous substitutions. In (5) above, we now proceed

$$
\begin{array}{ll}
 & (y := z+1)[\textbf{value } x, \textbf{result } y\backslash z, z] < R > \\
= & (m := l+1)[\textbf{value } l\backslash x][\textbf{result } m\backslash z] < R > \\
= & (\forall\, m.\, (m := l+1)[\textbf{value } l\backslash x] < R[z\backslash m] >) \\
= & (\forall\, m.\, (m := l+1) < R[z\backslash m][l\backslash z] >) \\
= & (\forall\, m.\, R[z\backslash m][m\backslash l+1][l\backslash x]) \\
= & R[z\backslash z+1] \\
= & z := z+1 < R >
\end{array}
$$

Hence the program fragment increments z, as expected.

4.5 Real limitations

Unfortunately, we cannot treat the general cases of "substitution by name" or even "substitution by var". As we have seen, simple substitution (*i.e.*, by name) does not respect equality of programs modulo *wp* unless severe restrictions are made on its use. Those very restrictions, whatever they might be[1], are necessary to achieve monotonicity and can be studied as such. With monotonicity, they can be treated as were the substitutions in section 4 above.

Finally, note that in multiple result parametrization an apparent aliasing can occur if two actual parameters are the same, as in $[\textbf{result } x, y\backslash z, z]$. The effect of this must agree with that of multiple assignments $z, z := x, y$ and multiple simple substitutions $[z, z\backslash x, y]$: usually, they are considered syntactically invalid.

5 Conclusion

Rules for parametrized procedural abstraction are complex. We have argued that they are simplified by considering parametrization, procedure call, and specification separately. The result is a more uniform and orthogonal treatment, in which difficulties are properly located: aliasing for example shown to be a non-monotonic construction.

[1] They vary from writer to writer.

Combined rules, such as those of [14] and [22], can be derived from ours. It is the program developer's choice whether to use them, or the more basic rules here, or perhaps some other combination especially relevant to his problem.

The separation we have achieved relies essentially on the embedding of specifications within programs: only this allows ALGOL's copy rule to give the meaning of procedure calls independently of the level of abstraction in the procedure body.

We have not treated the call-by-name and call-by-reference parameter passing techniques because they do not fit easily into the standard axiomatic framework of [21] and [12]. In [46, pp. 160-161], for example, call-by-name is treated in a slightly augmented logic in which one can state as a precondition that aliasing is not to occur. That shortcoming of the standard approach, however, we separate from procedures; as we have shown, the real problem is that in general

$$P =_{wp} Q \quad \not\Rightarrow \quad P[x\backslash T] =_{wp} Q[x\backslash T].$$

That is, equality as predicate transformers "$=_{wp}$" is too coarse for these substitutions.

6 Acknowledgements

The work here depends on the original ideas of Hoare [22] and Gries and Levin [15] for the axiomatic treatment of procedure parameters. I believe Back [4] to have been the first to generalise *wp* in a way similar to ours [35]. He uses single predicates, however, rather than pairs as we do, thus foregoing the advantage of miracles [37].

I am grateful for the very thorough comments of the referees.

Data refinement by miracles

Carroll Morgan

Abstract

Data refinement is the transformation in a computer program of one data type to another. Usually, we call the original data type "abstract", and the final data type "concrete". The concrete data type is said to *represent* the abstract.

In spite of recent advances, there remain obvious data refinements that are difficult to prove. We give such a refinement; and we present a new technique that avoids the difficulty.

Our innovation is the use of program fragments that do not satisfy Dijkstra's *Law of the excluded miracle*. These of course can never be implemented, and so they must be eliminated before the final program is reached. But in the intermediate stages of development, they simplify the calculations.

Keywords: Programming methodology; data refinement; weakest preconditions; laws of programming.

1 Introduction

Data refinement is increasingly becoming a central feature of the modern programming method. Although it is a long-established technique, well-explained for example in [26], it is still developing. Recently it has been extended ([45], [19], [16] and elsewhere) to allow a larger class of refinements than had before been thought desirable. In this paper, we extend it slightly further.

[0]Appeared in *Inf. Proc. Lett. 26(5)* (Jan. 1988).

Given two program fragments A and C, we say that C ***refines*** A iff: C terminates whenever A does; and every result of C is also a possible result of A. In many cases the abstract fragment is a block (or module) using some local (or hidden) variable a, say, and the concrete fragment is to use c instead. The technique of *data* refinement allows the algorithmic structure of the abstract fragment to be carried over into the concrete fragment: that is, if we apply data refinement to the *components* of the abstract fragment, replacing them one-by-one with concrete components, then the concrete fragment refines the abstract fragment *overall*.

We exhibit an "obvious" refinement, in which the abstract algorithmic structure is reproduced in the concrete program, but whose corresponding components cannot be data-refined using existing techniques. The inadequacy is due to the required data refinement's being valid only in certain conditions rather than universally. Furthermore, these conditions cannot be expressed in terms of the abstract variables.

Dijkstra's law of the excluded miracle [12, p. 18] states

For all programs P, $wp(P, false) = false$.

Recent work has suggested that derivation of programs is simplified if *miracles* — statements not satisfying the above — are allowed in the intermediate development steps ([41], [1], [35], [43]). We demonstrate a specific application of miracles by showing that they allow conditional data refinement to proceed even when the condition involves concrete variables. The price paid is that some reasoning is then necessary at the concrete level so that the miracles — which can never be executed — are eliminated.

We use the weakest precondition calculus of Dijkstra ([12], [14]) and its associated programming language.

2 An abstract program

Figure 1 contains a program for summing a bag of integers. Bag comprehensions are indicated by "$\prec \cdots \succ$", and "$+$" is bag (as well as integer) addition. We assume also that variable baghas size N. The statement $x{:\in}\, b$ stores in

```
var  bag : bag of Integer;
     sum : Integer;
     summed : bag of Integer;

sum, summed := 0, ≺≻ ;
do summed ≠ bag →
    || var x: Integer;
         x:∈ (bag − summed);
         sum := sum + x;
         summed := summed+ ≺ x ≻
    ||
od
```

Figure 1: Summing a bag of integers

the variable x an arbitrary element of bag b; it is defined

$$wp(x{:}\in b,\, R) \;\;\widehat{=}\;\; (b \neq \prec\succ) \wedge (\forall x : x \in b : R)$$

3 A difficult data refinement

We now transform the abstract program of figure 1 into a concrete program, replacing the bags *bag* and *summed* by an array a and an integer n. An *abstraction invariant* will provide the link between the two; it is

$$\begin{array}{ll} & 0 \leq n \leq N \\ \wedge & bag = \prec i : i \in 0..N-1 : a[i] \succ \\ \wedge & summed = \prec i : i \in 0..n-1 : a[i] \succ \end{array} \qquad (I)$$

We now use the formulation from [16] for proving a data refinement correct, paraphrased below:

> An abstract fragment A is data-refined by a concrete fragment $\vec{c}$ under abstraction invariant I iff the following holds:

$$I \wedge wp(A, true) \;\Rightarrow\; wp(\vec{c}, \neg wp(A, \neg I)) \qquad (1)$$

With (1) and our chosen abstraction invariant I, we can show the following data refinements:

Abstract	*Concrete*
$summed := \prec\succ$	$n := 0$
$summed \neq bag$	$n \neq N$
$x :\in (bag - summed)$	$x := a[n]$

But we cannot data-refine $summed := summed + \prec x \succ$, for to do so we would need C satisfying

$$\begin{array}{ll} & I \wedge wp(summed := summed + \prec x \succ, true) \\ \Rightarrow & wp(C, \neg wp(summed := summed + \prec x \succ, \neg I)) \end{array} \qquad (2)$$

It can be shown that *no* assignment to n satisfies (2); in particular, C = "$n := n + 1$" does not.

4 Miraculous programs

We introduce the *guarded command* as follows, for condition G and statement S:

$$wp(G \rightarrow S, R) \;\hat{=}\; G \Rightarrow wp(S, R)$$

Guarded commands can never[1] be implemented by real programs, because they violate Dijkstra's law. Like complex numbers, they can appear during calculation, but must be eliminated if a real (compare implemented) solution is to be reached. The worst offender is $false \rightarrow$ **skip**, because we have $wp(false \rightarrow \textbf{skip}, R) = true$, for any R.

Nevertheless, the following statement does satisfy requirement (2):

$$(n \neq N \;\wedge\; a[n] = x) \;\longrightarrow\; n := n + 1 \qquad (3)$$

[1] ... well, hardly ever: only when the guard is *true*.

5 Eliminating miracles

Although guarded commands violate the law of the excluded miracle, they do obey other laws (distributivity of conjunction, for example). In particular, we have the following:

Law 1 Assignment can be distributed through guarding:

$$\begin{array}{ll} & v := exp;\ \ B \to S \\ = & \neg wp(v := exp, \neg B) \ \longrightarrow\ v := exp; S \end{array}$$

Proof: For all postconditions R, we have

$$\begin{array}{ll} & wp(\text{“}v := exp;\ B \to S\text{”},\ R) \\ = & def(exp) \wedge (B \Rightarrow wp(S, R))[v \backslash exp] \\ = & def(exp) \wedge (B[v \backslash exp] \Rightarrow wp(S, R)[v \backslash exp]) \\ = & (def(exp) \Rightarrow B[v \backslash exp]) \Rightarrow (def(exp) \wedge wp(S, R)[v \backslash exp]) \\ = & wp(\neg wp(v := exp, \neg B) \to v := exp; S,\ R) \end{array}$$

(end of law)

With law 1, we can eliminate the miracle; we have in the concrete loop body:

$$\begin{array}{l} x := a[n]; \\ sum := sum + x; \\ (n \neq N \ \wedge\ a[n] = x) \ \longrightarrow\ n := n + 1 \end{array}$$

But this equals

$$\begin{array}{ll} x := a[n]; & \\ (n \neq N \ \wedge\ a[n] = x) \ \longrightarrow & sum := sum + x; \\ & n := n + 1 \end{array}$$

For our final step, we note that

$$\begin{array}{ll} & \neg wp(x := a[n], \neg(n \neq N \wedge a[n] = x)) \\ = & \neg(0 \leq n < N \wedge \neg(n \neq N \wedge a[n] = a[n])) \\ = & true \end{array}$$

```
var  a : array [0..N−1] of Integer;
     sum : Integer;
     n : [0..N];

sum, n := 0, 0;
do n ≠ N →
    || var x: Integer;
        x := a[n];
        sum := sum + x;
        n := n + 1
    ||
od
```

Figure 2: Summing an array of integers

With this, and law 1 again, we reach the promising

$$\begin{array}{lll} true & \rightarrow & x := a[n]; \\ & & sum := sum + x; \\ & & n := n + 1 \end{array}$$

But $(true \rightarrow S) = S$ for all S (another law), and so the guard can be eliminated altogether. We are left with the concrete program of figure 2.

6 Conclusion

In our example, a proof of correctness of the concrete operation $n := n + 1$ requires the precondition $a[n] = x$, which cannot be expressed solely in terms of the abstract variables. Hence the proof method of [16] cannot be used. Allowing concrete variables in the precondition is not the solution, for that would destroy our ability to reason at the abstract level independently of possible representations.

It is possible to rearrange our example program, and then to data-refine as a whole the compound statement

$$\begin{array}{l} x :\in (bag - summed); \\ summed := summed + \prec x \succ \end{array}$$

In this case concrete variables in preconditions are no longer necessary. But then those two statements must *always* appear adjacent: a severe restriction. We would have lost the important technique of distributing data refinement through program structure.

Guarded commands are useful also when stating rules for algorithmic refinement, in many cases making them simpler by widening their applicability. Mistaken refinements — normally prevented by failure of an applicability condition — instead are allowed to proceed, but generate "infeasible" programs from which the guards cannot be eliminated. The disadvantage of this is that such mistakes can go long unnoticed; but the overwhelming advantage is the decreased proof obligation faced by the developer [37].

Guarded commands were first introduced by Dijkstra [12], who used them only within alternation and iteration constructs. As explained in [35], we "discovered" miracles while extending Dijkstra's language to accommodate embedded specifications.

7 Acknowledgements

The connection between infeasible specifications and guarded commands was pointed out by Tony Hoare, who together with He Jifeng and Jeff Sanders demonstrates in [24] that the standard relational model of programs can give a very elegant formulation of data refinement. They show easily that data refinement is a correctness-preserving technique and that it distributes through the ordinary program constructors. Infeasibility occurs naturally within their work as relations whose domains are partial.

The work reported here falls within the larger context of joint research with Jean-Raymond Abrial [1], Paul Gardiner, Mike Spivey, and Trev Vickers; I am grateful to British Petroleum for supporting our collaboration.

I am grateful also to the painstaking and perceptive referees, and to Jean-Raymond Abrial, Paul Gardiner, David Gries, and Jeff Sanders. Their suggestions have improved the paper significantly.

Auxiliary variables in data refinement

Carroll Morgan

15 February 1988

Abstract

A set of local variables in a program is auxiliary if its members occur only in assignments to members of the same set. Data refinement transforms a program, replacing one set of local variables by another set, in order to move towards a more efficient representation of data.

Most techniques of data refinement give a direct transformation. But there is an indirect techniqne, nsing auxiliary variables, that proceeds in several stages. Usually, the two techniques are considered separately.

It is shown that the several stages of the indirect technique are themselves special cases of the direct, thus unifying the separate approaches. Removal of auxiliary variables is formalised incidentally.

Keywords: Programming methodology, auxiliary variables, data refinement, weakest preconditions, program transformation.

1 Introduction

Data refinement transforms a program so that certain local variables — called ***abstract*** — are replaced by other local variables — called ***concrete***. Usually the abstract variables range over mathematically abstract values, such as sets, functions *etc.* The concrete variables take values more efficiently represented in a computer, such as arrays.

[0]To appear in *Inf. Proc. Lett.*

There are many formalisations of data refinement, all more or less equally powerful. In each a rule is given for producing the concrete statements that correspond to given abstract ones. We call such methods *direct*.

An indirect but equally effective approach uses auxiliary variables. First, the concrete variables are introduced *in parallel* with the abstract variables they ultimately replace. The program is then "massaged" (*i.e., algorithmically* refined) to make those abstract variables auxiliary. Then they are removed.

Our contribution is to show that the auxiliary variable technique is a special case of the direct technique: in fact, it is a composition of direct data refinements. That brings the two styles together, and a more uniform view is gained.

2 The direct technique

Data refinement is described in [23, 26, 16]. An *invariant* is chosen that relates the abstract variables to the concrete variables, and it applies to the whole transformation. Using the invariant, each abstract statement is replaced directly by a concrete statement. We use a recent formulation, taken from [40, 13].

Definition 1 *Direct data refinement:* Let A (C) be the abstract (concrete) statement, a (c) the abstract (concrete) variables, and I the invariant. Then we say that

$$A \text{ is data-refined to } C \text{ by } (I, a, c)$$

if for all predicates ϕ not containing concrete variables c

$$(\exists\, a :: I \wedge wp(A, \phi)) \Rightarrow wp(C, (\exists\, a :: I \wedge \phi)).$$

♡

We assume that the concrete variables c do not appear in the abstract program A.

3 The auxiliary variable technique

This use of auxiliary variables is described in [29], [46, Ch.5], and [12, pp.64–65]. An invariant is chosen, as above. Concrete variables are *added* to the program: their declarations are made in parallel with the existing abstract declarations; and abstract statements are extended so that they maintain the invariant. For example, an abstract assignment $a := AE$, where the expression AE involves abstract variables, is extended to $a, c := AE, E$ by an assignment to the concrete variables. The new expression E may contain variables of either kind, as long as the new statement preserves the invariant I:

$$I \Rightarrow wp(\text{`}a, c := AE, E\text{'}, I). \tag{1}$$

In the next step, the program is algorithmically refined to make the abstract variables auxiliary in this sense:

Definition 2 *Auxiliary variables*: A set of local variables is auxiliary if the only executable statements in which its members appear are assignments to members of the same set.
♡

Thus abstract variables must be eliminated from expressions E above and from the guards of alternations and iterations. Finally, the abstract variables are removed from the program entirely; what remains is a data refinement of the original.

4 The correspondence

First, we relate the data refinement (I, a, c) to the composition of two other data refinements.

Lemma 1 *Composition of data refinements:* Let ϵ be the empty list of variables. If A is data-refined to M by (I, ϵ, c) and M to C by $(true, a, \epsilon)$, then A is data-refined to C by (I, a, c).

Proof: Note that empty quantifications $(\exists\, \varepsilon :: \cdots)$ are superfluous. From the assumption and Definition 1, we have for all ϕ not containing c

$$I \wedge wp(A, \phi) \Rightarrow wp(M, I \wedge \phi), \tag{2}$$

and for all ψ (not containing e)

$$(\exists\, a :: wp(M, \psi)) \Rightarrow wp(C, (\exists\, a :: \psi)). \tag{3}$$

Now we have for all ϕ not containing c

	$(\exists\, a :: I \wedge wp(A, \phi))$	
hence	$(\exists\, a :: wp(M, I \wedge \phi))$	from (2)
hence	$wp(C, (\exists\, a :: I \wedge \phi))$	from (3)

That establishes data refinement by (I, a, c).
♡

The correspondence is this: the data refinement (I, ε, c) corresponds to the introduction in parallel of the concrete variables, while preserving I; and the data refinement $(true, a, \varepsilon)$ corresponds to the elimination of the auxiliary variables a. We support that view with the following two lemmas.

Lemma 2 *Introducing concrete variables:* A is data-refined to M by (I, ε, c) if

1. for all ϕ not containing c, $wp(A, \phi) \Rightarrow wp(M, \phi)$; and
2. $I \wedge wp(A, true) \Rightarrow wp(M, I)$.

Proof: For all ϕ not containing c,

	$I \wedge wp(A, \phi)$	
hence	$I \wedge wp(A, \phi) \wedge wp(A, true)$	wp calculus
hence	$wp(M, \phi) \wedge wp(M, I)$	assumptions
hence	$wp(M, I \wedge \phi)$	wp calculus

♡

Assumption 1 of Lemma 2 states that M, over *abstract* variables, is an algorithmic refinement of A. Assumption 2 states that the invariant, linking a and c, is maintained provided A terminates. The example (1) in section 3 is an instance of this.

Lemma 3 *Eliminating auxiliary variables:* M is data-refined to C by $(true, a, \varepsilon)$ if for all ϕ not containing a,

1. $wp(M, \phi) \Rightarrow wp(C, \phi)$; and
2. $wp(M, \phi)$ contains no a.

Proof: For all ψ

	$(\exists\, a :: wp(M, \psi))$	
hence	$(\exists\, a :: wp(M, (\exists\, a :: \psi)))$	wp calculus
hence	$wp(M, (\exists\, a :: \psi))$	assumption 2
hence	$wp(C, (\exists\, a :: \psi))$	assumption 1

♡

Assumption 1 of Lemma 3 states that C, over *concrete* variables, is an algorithmic refinement of M. Assumption 2 states that in M final values of c do not depend on initial values of a — that is, a is auxiliary.

As a final illustration, we apply (3) when a is *not* auxiliary. Taking $c := a$ for M, we must find C such that for all ψ

$$(\exists\, a :: \psi_{a \Rightarrow c}) \Rightarrow wp(C, (\exists\, a :: \psi)). \qquad (4)$$

But there can be no such C, since

	false	
iff	$wp(C, c = 0 \wedge c \neq 0)$	excluded miracle
iff	$wp(C, (\exists\, a :: c = 0)) \wedge wp(C, (\exists\, a :: c \neq 0))$	wp calculus
if	$(\exists\, a :: a = 0) \wedge (\exists\, a :: a \neq 0)$	assumption
iff	*true*	

Therefore that data refinement cannot succeed; such failures underlie the soundness of the method.

Note that a variable that *appears* auxiliary in one place might not he auxiliary in another. For example, in $a := a + 1; \cdots; c := a$ we can transform the first statement to **skip** (indeed, that transformation is always possible). But (3) cannot succeed on the second statement: overall, the transformation still fails.

5 Conclusion

We have shown that the two stages of the auxiliary variable technique are data refinements themselves (Lemmas 2, 3), and we have confirmed that the overall result is a data refinement also (Lemma 1). In passing, we have formalised the removal of auxiliary variables.

Data refinement increasingly seems more than a technique for refining data. The transformation $(true, a, \varepsilon)$ — removing auxiliary variables a — has long been used in programming generally. And the transformation (I, ε, c) introduces new variables c which might remain in the program, affording an alternate representation of the structure a.

The transformation $(I, \varepsilon, \varepsilon)$, applied to the exported operations of a module, allows their preconditions to be strengthened (by assuming I); it is successful (compare the failure of (4)) only if each operation establishes I finally. Thus data refinement can also formalise the strengthening of the invariant within a module, though no variables are added or removed. Finally, algorithmic refinement ("massaging") is a special case of that: $(true, \varepsilon, \varepsilon)$.

Definition 1 is slightly more general than [40]: without restriction, we allow free variables in I that do not necessarily appear in a or c. That allows us our empty lists of variables: $(I, \varepsilon, \varepsilon)$ is an extreme example. It also allows invariants that refer to global variables unaffected by the transformation.

We have not discussed the effect of our transformations on guards nor on initialisations. Details are in [40], in [13] where a more theoretical and general approach is taken, and in [36] where it is shown how the transformations allow data refinements to be calculated in practice.

Acknowledgements

I thank Cliff Jones and Edsger W. Dijkstra for the auxiliary variable technique, the former also for reference [29], and Richard Bird, members of IFIP WG 2.3, and the referee for constructive criticism.

Data Refinement of Predicate Transformers

P.H.B. Gardiner* C.C. Morgan

29 February 1988

Abstract

Data refinement is the systematic substitution of one data type for another in a program. Usually, the new data type is more efficient than the old, but also more complex; the purpose of the data refinement in that case is to make progress in a program design from more abstract to more concrete formulations.

A particularly simple definition of data refinement is possible when programs are taken to be predicate transformers in the sense of Dijkstra. Central to the definition is a function taking abstract predicates to concrete ones, and this function — a generalisation of the abstraction function — therefore is a predicate transformer as well.

Advantages of the approach are: proofs about data refinement are simplified; more general techniques of data refinement are suggested; and a style of program development is encouraged in which data refinements are calculated directly without proof obligation.

1 Introduction

In many situations, it is more simple to describe the desired result of a task than to describe how a task should be performed. This is particularly true in computer science. Computer programs are very complex, both in their

*Supported by British Petroleum Ltd.

[0]Submitted to *Theor. Comp. Sc.*

operation and in their representation of information. Yet the task a program performs is often simple to describe.

More confidence in a program's correctness can be gained by describing its intended task in a formal notation. Such *specifications* can then be used as a basis for a provably correct development of the program. The development can be conducted in small steps, thus allowing the unavoidable complexity of the final program to be introduced in manageable pieces.

The process, called *refinement*, by which specifications are transformed into programs has received much study in the past. In particular [23][12][26] have laid down much of the theory and have recognised two forms of refinement. Firstly, algorithmic refinement: where one makes more explicit the way in which a program operates, usually introducing an algorithm where before there was just a statement of the desired result. And secondly, data refinement: where one changes the structures for storing information, usually replacing some abstract structure that is easily understood, by some more concrete structure that is more efficient.

More recently the emphasis has turned towards providing a uniform theory of program development, in which specifications and programs have equal status. Such a theory is needed to provide the proper setting both for further theoretical work on refinement and for conducting refinement in practice. This goal has been achieved in [4, 41, 35, 37] by extending Dijkstra's language of guarded commands with a *specification statement*. The extended language, by encompassing both programs and specifications, reduces in theory the process of modular program development to program transformation. [41, 35, 37] cover only algorithmic refinement. In this paper we carry on in the same style to include data refinement, and thus give a more complete framework for software development. [40] has made a similar extension.

An important part of our approach is the use of predicate transformers, as in [12], which seem to have several advantages over the relations used in [25]. One is that predicate transformers can represent a form of program conjunction not representable in the relational model. This form of conjunction behaves well under data refinement and can be used to simplify the application of data refinement to specifications. Also, since recursion can be re-expressed in terms of conjunction, this good behaviour allows reasoning about recursion without assuming bounded non-determinism — an

unwanted assumption in a theory of programs which includes specifications. But probably the greatest advantage of using predicate transformers is that the theoretical results are so easily applied in practice. In particular, we use a predicate transformer to represent the relationship between abstract and concrete states of a data refinement, and this predicate transformer can be used to calculate directly the concrete program from the abstract program. The calculation maintains the algorithmic structure of the program and adds very little extra complication. Moreover, these calculations do not have any "applicability conditions". No extra proof of correctness is necessary.

2 Predicate transformers

Following [12], we model programs as functions taking predicates to predicates. We blur intentionally the distinction between predicates and the sets of states satisfying them, and therefore we think also of programs as taking sets of (final) states to sets of (initial) states. In any case, for program P and predicate ψ, called the *postcondition*, the application of P to ψ is written $P\ \psi$ and yields a predicate ϕ, called the *weakest precondition* of ψ with respect to P. We say that P *transforms* ψ into ϕ. This predicate ϕ is the weakest one whose truth *initially* guarantees proper termination of P in a state satisfying ψ (finally). The expression $P\psi$ can also be read simply as "P establishes ψ".

The purpose of predicates in the model is to specify sets of states. For this reason, when giving the meaning of a program as a predicate transformer, we will consider only predicates whose free variables are drawn from the program's set of state variables. We will call this set of state variables the program's alphabet (written αP), and call the predicates whose free variables are drawn from a given set of variables x "the predicates on x". Thus, a predicate transformer P can be defined by giving the value of $P\psi$ for all predicates ψ on αP. Of course, we will have to take care not to apply predicate transformers outside their domains.

For clarity, we will sometimes distinguish between program texts and their corresponding predicate transformers, writing $[\![T]\!]$ for the predicate transformer denoted by the program text T.

We define an order $\leq$ on predicates as follows

$$\phi \leq \psi \quad \textit{iff} \quad \models \phi \Rightarrow \psi$$

The order $\leq$ permits least upper and greater lower bounds of collections of predicates ϕ_i for which we write respectively

$$\bigvee_i \phi_i \text{ and } \bigwedge_i \phi_i.$$

Also the order has a top and bottom

$$\top \text{ and } \bot$$

which correspond to *true* and *false*.

The order on predicates is promoted to predicate transformers in the usual way; for predicate transformers P and Q such that $\alpha P = \alpha Q$:

$$P \sqsubseteq Q \quad \textit{iff} \quad \textit{for all predicates } \phi \textit{ on } \alpha P, \quad P\ \phi \leq Q\ \phi$$

This promoted order has least upper and greatest lower bounds as well as a top and bottom element, and they satisfy the following equations:

$$\begin{aligned}
(\bigsqcup_i P_i)\ \phi &= \bigvee_i (P_i\ \phi) \\
(\bigsqcap_i P_i)\ \phi &= \bigwedge_i (P_i\ \phi) \\
\bot\ \phi &= \bot \\
\top\ \phi &= \top
\end{aligned}$$

All the predicate transformers P we will consider are *monotonic*: for any predicates ϕ and ψ, $\phi \leq \psi$ will imply $P\ \phi \leq P\ \psi$.

3 Algorithmic refinement of predicate transformers

In general, one mechanism is refined by another exactly when every specification satisfied by the first is satisfied also by the second. For predicate transformers we take specifications and satisfaction as follows: a *specification* is a predicate pair $[pre, post]$ comprising the initial assumptions *pre*

and the final requirement *post*; and a program P *satisfies* $[pre, post]$ exactly when

$$pre \Rightarrow P\ post$$

It is now easy to show that P is refined by Q exactly when $P \sqsubseteq Q$.

4 Data refinement of predicate transformers

During algorithmic refinement, *local* variables are usually introduced. And when considering the external behaviour of a program, we ignore the effect it has on its local variables. This gives us a new degree of freedom in refining such programs: we can replace local variables by new ones, so long as the overall effect on the global variables is preserved. This is called *data refinement*.

The following syntax is used to hide (make local) a list of variables x:

$$|[\mathbf{var}\ x \mid I \bullet P]|$$

The predicate I states the initialisation of x, and P is the program within which the variables x may be used. This construct is used only if the alphabet of P contains x. The alphabet of the result is that of P with x removed. The meaning of the construct is as follows: for any predicate ψ on $(\alpha P - x)$

$$[\![\,|[\mathbf{var}\ x \mid I \bullet P]|\,]\!]\psi \ \widehat{=}\ (\forall x \bullet I \Rightarrow [\![P]\!]\psi)$$

We now define data refinement. Let us suppose we wish to replace the list of variables a (the abstract variables) in

$$|[\mathbf{var}\ a \mid I \bullet P]|$$

by some other list of variables c (the concrete variables), and let the variables of αP, other than a, be g (the global variables). We choose any predicate transformer *rep* that takes predicates on the variables a, g to predicates on the variables c, g. Then for programs P and P', we write $P \preceq P'$ to mean that P is data-refined by P'.

Definition 1

$$P \preceq P' \quad \textit{iff} \quad \textit{rep} \circ P \sqsubseteq P' \circ \textit{rep}$$

where the operator $\circ$ *is functional composition (of predicate transformers).*

We will see that for data refinement to be well behaved, we must restrict our choice for *rep*. We choose *rep* satisfying the following two properties:

- *rep* is *monotonic*: $(\forall a \bullet \phi \Rightarrow \psi) \Rightarrow (\forall c \bullet \textit{rep}\ \phi \Rightarrow \textit{rep}\ \psi)$;
- *rep* is $\vee$*-distributive*: $\textit{rep}\ (\bigvee_i \phi_i) = \bigvee_i (\textit{rep}\ \phi_i)$

Note that *strictness* is a special case of $\vee$*-distribution* (i.e. $\textit{rep}\ \bot = \bot$). Note also that this form of monotonicity is stronger than the usual. Further properties that follow from these are proven below. In these proofs and others we will make use of the fact that the two lists of variables a and g are, by definition, disjoint and so the variables a do not occur free in the predicates on g.

Lemma 1 *If* ϕ *is a predicate on* g*, then*

$$\textit{rep}\ \phi \leq \phi$$

proof:

$\neg \phi \Rightarrow (\forall a \bullet \phi \Rightarrow \bot)$	*since* a *is not free in* ϕ
$\neg \phi \Rightarrow (\textit{rep}\ \phi \Rightarrow \textit{rep}\ \bot)$	*monotonicity of rep*
$\neg \phi \Rightarrow (\textit{rep}\ \phi \Rightarrow \bot)$	*strictness of rep*
$\neg \phi \Rightarrow \neg\ \textit{rep}\ \phi$	*predicate calculus*
$\textit{rep}\ \phi \Rightarrow \phi$	*predicate calculus*

$\heartsuit$

Lemma 2 *If* ϕ *is a predicate on* a, g *and* ψ *is a predicate on* g*, then*

$$(\textit{rep}\ \phi) \wedge \psi \leq \textit{rep}(\phi \wedge \psi)$$

proof:

$$\begin{array}{ll}
\psi \Rightarrow (\forall\, a \bullet \phi \Rightarrow \phi \wedge \psi) & \textit{since } a \textit{ is not free in } \psi \\
\psi \Rightarrow (\mathit{rep}\ \phi \Rightarrow \mathit{rep}(\phi \wedge \psi)) & \textit{monotonicity of rep} \\
(\mathit{rep}\ \phi) \wedge \psi \Rightarrow \mathit{rep}(\phi \wedge \psi) & \textit{predicate calculus}
\end{array}$$

♡

In subsequent sections we will discuss a particularly convenient choice for *rep*, and will show how to *calculate* suitable P'. For now, we give the fundamental theorem of data refinement:

Theorem 1 *If for suitable rep (as defined above) we have shown that* $P \preceq P'$, *then*

$$|[\,\mathbf{var}\ a \mid I \bullet P\,]| \sqsubseteq |[\,\mathbf{var}\ c \mid \mathit{rep}\ I \bullet P'\,]|$$

proof: Let ψ be any predicate on g, then

$$\begin{array}{ll}
[\![\, |[\,\mathbf{var}\ a \mid I \bullet P\,]| \,]\!]\psi & \\
\quad = (\forall\, a \bullet I \Rightarrow [\![P]\!]\psi) & \textit{semantics} \\
\quad \leq (\forall\, c \bullet \mathit{rep}\ I \Rightarrow \mathit{rep}\ [\![P]\!]\psi) & \textit{monotonicity of rep} \\
\quad \leq (\forall\, c \bullet \mathit{rep}\ I \Rightarrow [\![P']\!]\mathit{rep}\ \psi) & \textit{hypothesis} \\
\quad \leq (\forall\, c \bullet \mathit{rep}\ I \Rightarrow [\![P']\!]\psi) & \textit{lemma 1 and monotonicity of } P' \\
\quad = [\![\, |[\,\mathbf{var}\ c \mid \mathit{rep}\ I \bullet P'\,]| \,]\!]\psi & \textit{semantics}
\end{array}$$

♡

5 The programming language

The *programming language* is the syntax with which we describe predicate transformers. Here we will use Dijkstra's language [12] with several extensions.

5.1 Extensions

All of the predicate transformers P which can be described by Dijkstra's original language satisfy the following properties:

strictness $P\bot = \bot$;

monotonicity $\phi \leq \psi$ implies $P\ \phi \leq P\ \psi$;

$\wedge$-distributivity $P\ (\bigwedge_i \psi_i) = \bigwedge_i (P\ \psi_i)$, for any non-empty family $\{\psi_i\}_i$;

continuity $P\ (\bigvee_i \psi_i) = \bigvee_i (P\ \psi_i)$, for any chain $\{\psi_i\}_i$.

We will see that some of these properties fail in our extended language.

The first extension was given in section 4 above: the introduction of local variables $[\![\, \mathbf{var}\ x \mid I \bullet P \,]\!]$. It preserves strictness, monotonicity, $\wedge$-distributivity, and continuity.

The second extension is the *specification*. It is written $[pre, post]$ (as in section 2, but here we add it to the *programming* language). The meaning of this construct depends on the alphabet. It is defined for alphabet x as follows:

Definition 2 *For any predicate ψ on x,*

$$[\![\ [pre, post]\ ]\!]\psi \ \widehat{=}\ pre \wedge (\forall x \bullet post \Rightarrow \psi)$$

Specifications are monotonic and $\wedge$-distributive, but $[\top, \bot]$ is not strict, and $[\top, \top]$ is not continuous (take any chain $\{\psi_i\}_i$ with $\bigvee_i \psi_i = \top$ but $\psi_i \neq \top$ for any i).

The third extension is *conjunction* of programs, written $\ddagger_i P_i$ for any family $\{P_i\}_i$ of programs. Its meaning is given as follows:

Definition 3

$$[\![\ddagger_i P_i]\!] \ \widehat{=}\ \bigsqcup_i [\![P_i]\!]$$

Thus the conjunction of a family of programs is the worst program that is better than each member of the family. Conjunction preserves strictness, monotonicity, and continuity, but not $\wedge$-distributivity: consider

$$[\![[\top, \psi] \ddagger [\top, \neg\psi]]\!](\psi \wedge \neg\psi)$$

Conjunction is an important counterpart of the specification, since the specification as it stands in definition 2 does not allow the post-condition to refer to the state before execution. Using conjunction and specification together, we can rectify this problem. For example, the assignment statement $n := n + 1$ can be expressed as $\ddagger_i [n = i, n = i + 1]$.

This completes the extension of Dijkstra's language. We can now see that the only property retained from the original language is monotonicity, since each of the other properties is violated by at least one of the program constructors.

5.2 Generalisations

Having accepted the loss of strictness, continuity and $\wedge$-distributivity, we are able to make other generalisations of the language. Choice and guarding need not be restricted to use within the **do**$\cdots$**od** and **if**$\cdots$**fi** constructs. They can instead be defined as language constructs in their own right.

Definition 4 *Choice: For any family* $\{P_i\}$ *of programs we define their* choice *as follows:*

$$[\![\mathop{[\!]}_i s_i]\!] \;\widehat{=}\; \bigsqcap_i [\![s_i]\!]$$

Definition 5 *Guarding: For predicate* G *and program* P*, we define the* guarded command $G \rightarrow P$ *as follows:*

$$[\![G \rightarrow P]\!]\psi \;\widehat{=}\; G \Rightarrow [\![P]\!]\psi$$

Definition 6 *Recursion: We must consider program* contexts*, which denote functions from predicate transformers to predicate transformers. If* C *is a program context and* P *a program, then* $C(P)$ *is a program also. The context*

C should be thought of as a program structure into which program fragments (for example, P) can be embedded. We have

$$[\![\mu X \bullet C(X)]\!] \;\hat{=}\; \mathbf{fix}[\![C]\!]$$

where **fix** *takes the least fixed point of a function (from predicate transformers to predicate transformers in this case).*

With these definitions, we can if we wish define the conventional **if** $\cdots$ **fi** and **do** $\cdots$ **od** constructors as appropriate combinations. We have

Definition 7 *Alternation:*

$$\begin{array}{ll}\mathbf{if} & G_1 \rightarrow P_1 \\ [\!] & \vdots \\ [\!] & G_n \rightarrow P_n \\ \mathbf{fi} & \end{array}$$

is an abbreviation for

$$(\mathop{[\!]}_{i=1}^{n} G_i \rightarrow P_i) \,[\!]\, \neg\, (\bigvee_i G_i) \rightarrow \mathbf{abort}$$

Definition 8 *Iteration:*

$$\begin{array}{ll}\mathbf{do} & G_1 \rightarrow P_1 \\ [\!] & \vdots \\ [\!] & G_n \rightarrow P_n \\ \mathbf{od} & \end{array}$$

is an abbreviation for

$$(\mu X \bullet (\mathop{[\!]}_{i=1}^{n} G_i \rightarrow P_i; X) \,[\!]\, \neg\, (\bigvee_i G_i) \rightarrow \mathbf{skip})$$

6 Distribution of data refinement

After theorem 1, the most important property of data refinement is that it distributes through the algorithmic constructors of our programming language. Only then can one carry over the algorithmic structure of the abstract program onto the concrete program. We prove this distribution for each constructor below:

Lemma 3 *Sequential composition: If* $P \preceq P'$ *and* $Q \preceq Q'$ *then* $P; Q \preceq P'; Q'$

proof:

$$
\begin{array}{lr}
rep \circ [\![P; Q]\!] & \\
\quad = rep \circ [\![P]\!] \circ [\![Q]\!] & \textit{semantics} \\
\quad \sqsubseteq [\![P']\!] \circ rep \circ [\![Q]\!] & \textit{hypothesis} \\
\quad \sqsubseteq [\![P']\!] \circ [\![Q']\!] \circ rep & \textit{hypothesis and monotonicity of } P' \\
\quad = [\![P'; Q']\!] \circ rep & \textit{semantics}
\end{array}
$$

♡

Lemma 4 *Skip:* $\mathbf{skip} \preceq \mathbf{skip}$

proof:

$$
\begin{array}{lr}
rep \circ [\![\mathbf{skip}]\!] & \\
\quad = rep \circ Id & \textit{semantics} \\
\quad = rep & \textit{property of } Id \\
\quad = Id \circ rep & \textit{property of } Id \\
\quad = [\![\mathbf{skip}]\!] \circ rep & \textit{semantics}
\end{array}
$$

♡

Lemma 5 *Abort:* $\mathbf{abort} \preceq \mathbf{abort}$

proof:

$$
\begin{array}{lr}
rep \circ [\![\mathbf{abort}]\!] & \\
\quad = rep \circ \bot & \textit{semantics} \\
\quad = \bot & \textit{strictness of } rep \\
\quad = \bot \circ rep & \textit{property of } \bot \\
\quad = [\![\mathbf{abort}]\!] \circ rep & \textit{semantics}
\end{array}
$$

♡

Lemma 6 *Guarding: To deal with guarded commands we will need another function from abstract predicates to concrete predicates.*

$$\overline{rep}\ \psi \ \widehat{=}\ \neg\,(rep\ \neg\ \psi)$$

We then have the following result for guarded commands. If $P \preceq P'$ *then* $(G \rightarrow P) \preceq ((\overline{rep}\ G) \rightarrow P')$

proof:

$$\begin{array}{lll}
rep\ [\![G \rightarrow P]\!]\ \psi & & \\
\quad = rep\ (G \Rightarrow [\![P]\!]\ \psi) & & \textit{semantics} \\
\quad = rep\ (\neg\ G \vee [\![P]\!]\ \psi) & & \textit{predicate calculus} \\
\quad = (rep\ \neg\ G) \vee (rep\ [\![P]\!]\ \psi) & & \vee\textit{-distributivity of rep} \\
\quad = \neg\ (\overline{rep}\ G) \vee (rep\ [\![P]\!]\ \psi) & & \textit{definition of } \overline{rep} \\
\quad = (\overline{rep}\ G) \Rightarrow (rep\ [\![P]\!]\ \psi) & & \textit{predicate calculus} \\
\quad \leq (\overline{rep}\ G) \Rightarrow ([\![P']\!]\ rep\ \psi) & & \textit{hypothesis} \\
\quad = [\![(\overline{rep}\ G) \rightarrow P']\!]\ rep\ \psi & & \textit{semantics}
\end{array}$$

♡

Lemma 7 *Choice: If for each* i $P_i \preceq P_i'$ *then* $\|_i P_i \preceq \|_i P_i'$

proof:

$$\begin{array}{lll}
rep \circ [\![\ \|_i P_i\ ]\!] & & \\
\quad = rep \circ (\sqcap_i [\![P_i]\!]) & & \textit{semantics} \\
\quad \sqsubseteq \sqcap_i (rep \circ [\![P_i]\!]) & & \textit{monotonicity of rep} \\
\quad \sqsubseteq \sqcap_i ([\![P_i']\!] \circ rep) & & \textit{hypothesis} \\
\quad = (\sqcap_i [\![P_i']\!]) \circ rep & & \textit{property of } \sqcap \\
\quad = [\![\ \|_i P_i'\ ]\!] \circ rep & & \textit{semantics}
\end{array}$$

♡

Lemma 8 *Conjunction: If for each* i $P_i \preceq P_i'$ *then* $\ddagger_i P_i \preceq \ddagger_i P_i'$

proof:

$$
\begin{aligned}
& rep \circ [\![\ddagger_i P_i]\!] \\
&\quad = rep \circ (\sqcup [\![P_i]\!]) && \textit{semantics} \\
&\quad = \sqcup (rep \circ [\![P_i]\!]) && \vee\textit{-distributivity of } rep \\
&\quad \sqsubseteq \sqcup ([\![P_i']\!] \circ rep) && \textit{hypothesis} \\
&\quad = (\sqcup [\![P_i']\!]) \circ rep && \textit{property of } \sqcup \\
&\quad = [\![\ddagger_i P_i]\!] \circ rep && \textit{semantics}
\end{aligned}
$$

♡

Lemma 9 *Recursion: We first promote data refinement to program contexts: we say that $C \preceq C'$ exactly when for all pairs of programs P and P' such that $P \preceq P'$, we have $C(P) \preceq C'(P')$ as well. The result for recursion is then as follows:*

If $C \preceq C'$, then

$$(\mu X \bullet C(X)) \preceq (\mu X \bullet C'(X))$$

proof: The Knaster-Tarski theorem asserts the existence of an ordinal γ such that $\mathbf{fix}\, F = F^\gamma \perp$, where

$$
\begin{aligned}
F^0\, X &= X \\
F^{\alpha+1}\, X &= F(F^\alpha\, X) \\
F^\gamma\, X &= \bigsqcup_{\beta<\gamma} (F^\beta\, X)
\end{aligned}
$$

Hence, it is sufficient to prove $[\![C]\!]^\gamma \perp \preceq [\![C']\!]^\gamma \perp$ for all γ. This can be proven by induction. The base case follows from Lemma 5, the step case follows from $C \preceq C'$ and the limit case follows from Lemma 8. ♡

7 Data refinement of specifications

In the preceding section, we showed that data refinement can be performed piecewise (a term we borrow from [40]), thus maintaining the algorithmic

structure of a program. We now consider the pieces lying within the structure.

There are two constructs to consider, the specification and the assignment. In fact, we can ignore the assignment statement since it is readily transformed into a simple specification.

The following theorems provide a method for calculating the data refinement of specifications, and show that this method produces the most general data refinement.

In both theorems we will, again, write a for the list of abstract variables, c for the list of concrete variables, g for the remaining (global) variables and rep for the representation predicate transformer.

Theorem 2

$$[pre, post] \preceq [rep\ pre, rep\ post]$$

proof: Let ψ be any predicate on g, a, then

$$\begin{array}{lr}
rep\ [\![\ [pre, post]\ ]\!]\ \psi & \\
\quad = rep\ (pre \wedge (\forall\, g, a \bullet post \Rightarrow \psi)) & \textit{semantics} \\
\quad \leq (rep\ pre) \wedge rep\ (\forall\, g, a \bullet post \Rightarrow \psi) & \textit{monotonicity of rep} \\
\quad \leq (rep\ pre) \wedge (\forall\, g, a \bullet post \Rightarrow \psi) & \textit{lemma 1} \\
\quad \leq (rep\ pre) \wedge (\forall\, g, c \bullet rep\ post \Rightarrow rep\ \psi) & \textit{monotonicity of rep} \\
\quad = [\![\ [rep\ pre, rep\ post]\ ]\!]\ rep\ \psi & \textit{semantics}
\end{array}$$

♡

Theorem 3 *If* $[pre, post] \preceq P$ *then* $[rep\ pre, rep\ post] \sqsubseteq P$.

proof: Let ψ be any predicate on g, c, then

$$\begin{array}{lr}
[\![\ [rep\ pre, rep\ post]\ ]\!]\psi & \\
\quad = (rep\ pre) \wedge (\forall\, g, c \bullet rep\ post \Rightarrow \psi) & \textit{semantics} \\
\quad \leq (rep\ pre) \wedge (\forall\, g, a \bullet post \Rightarrow \bigvee_{\forall g, c \bullet rep\ x \Rightarrow \psi} x) & \textit{properties of} \vee \\
\quad \leq rep\ (pre \wedge (\forall\, g, a \bullet post \Rightarrow \bigvee_{\forall g, c \bullet rep\ x \Rightarrow \psi} x)) & \textit{lemma 2}
\end{array}$$

$$= rep\ (\llbracket\ [pre, post]\ \rrbracket_{\bigvee_{\forall g, c \circ rep\ x \Rightarrow \psi}} x) \qquad \textit{semantics}$$

$$\leq P\ (rep \bigvee_{\forall g, c \circ rep\ x \Rightarrow \psi} x) \qquad \textit{hypothesis}$$

$$= P\ (\bigvee_{\forall g, c \circ rep\ x \Rightarrow \psi} rep\ x) \qquad \vee\textit{-distributivity of rep}$$

$$\leq P\ \psi \qquad \textit{properties of} \vee \textit{and monotonicity of } P$$

♡

8 Data refinement in practice

So far we have given no indication as to how one chooses a suitable representation transformer *rep*. In fact, there may be many classes of program transformation that can be supported by the theory of the proceeding sections. We can, though, cite one example that proves very useful in practice. This definition of *rep*, which is described below, gives the same form of data refinement as that in [16], [40] and also, under the name of downward simulation, in [24].

When performing data refinement, one always intends that the abstract and concrete states should correspond in some way. This correspondence can be expressed as a predicate over the two sets of state variables. From such a predicate (I say) the representation transformer can be defined as follows.

$$rep\ \phi \ \widehat{=}\ (\exists\, a \bullet I \wedge \phi)$$

It is easy to verify that this choice of *rep* has the required properties (i.e. *monotonicity* and $\vee$ *−distributivity*).

$\overline{rep}$ also has a simple form.

$$\overline{rep}\ \phi = (\forall\, a \bullet I \Rightarrow \phi)$$

Now that we have a definition for *rep* with such a simple form, we can see how easily data refinements can be calculated. The two simple transformers, *rep* and $\overline{rep}$ can be applied directly to the predicates of any abstract program so as to calculate a concrete refinement. In the abstract program all the pre and post conditions of specifications are replaced by their image under

rep and the guards by their image under $\overline{rep}$ thus leaving the algorithmic structure unchanged. The result is guaranteed correct by the theorems of the previous sections.

9 Conclusions

We have presented the familiar technique of data refinement in the novel context of predicate transformers. In doing so we have drawn on other recent work in program development: the factoring of Dijkstra's language into smaller pieces (Definitions 7 and 8); the use of recursion in practice rather than iteration as the basis for unbounded computations in Dijkstra's language [11]; and the mixing of specification and program [4, 41, 35, 37]. And in Definition 3 we give a further factorisation: with program conjunction, the technique of "logical" variables is formalised. All of this comes together to promote a style of program design in which steps are made by calculation rather than via proof obligations.

It is clear, though, that proof cannot be avoided altogether! In practice, the necessary truths of predicate calculus are drawn on when strengthening postconditions and weakening preconditions — and virtually nowhere else. In this respect perhaps the proofs have moved rather than disappeared. Their confinement though makes the *other* rules easier to apply in practice. Certainly they are easier to remember: the formulation of data refinement in theorem 2 is simpler than any other of which we are aware.

Using predicate transformers as a model, rather than the relations of [24], affords several advantages. One is that the dependence on continuity is more easily broken. That allowed us to extend the work of [24] so that it applies to a language that includes constructs for specification as well as programming. Another advantage is the ease with which the theoretical results are applied in practice. Both these advantages are related to our conjunction, which can not be represented in the relational model. Conjunction simplifies the expression of specifications in our language; and this, in turn, permits the very simple method of refinement calculation. In contrast, the method of calculation in [24], although theoretically simple, gives rise to very large and unwieldy expressions in practice.

Our relaxing of Dijkstra's "healthiness" conditions has left us only with

monotonicity: continuity, strictness, and $\wedge$-distributivity are gone. That is similar to [41], where continuity and strictness are dropped so that the guard and choice symbols can be given meaning as operators in their own right. We too proposed this in [35], but have taken the process further, dropping also $\wedge$-distributivity, so that we can define the conjunction operator which is the key to simplifying the calculation of data refinements.

Our results are potentially more general than those of [40], since we recognise how the abstraction condition is, itself, applied as a predicate transformer, and base all are proofs on two properties of it. By doing this we make the structure of the proofs more explicit, and also leave open the possibility of finding other predicate transformers with these properties which can, therefore, also be used for data refinement.

We also prove several results that [40] does not. Theorem 1 forms the important link between data refinement of local variables and operational refinement. Theorem 3 shows that our method of calculation yields the weakest program that is a data refinement of the original and thus that no loss of choice is incurred by calculation.

10 Acknowledgements

An early description of data refinement appeared in [23], and it later was made part of the Vienna Development Method [26]. Specifications were embedded within programs by [4], who also treated data refinement. The first connection between data refinement and weakest preconditions was made by [4], though it had been earlier presented by [12] as a technique based on auxiliary variables. ([31] explains the connection between these.) Most recently, [40] has given the same formulation as section 8 above, and his work has improved ours in several ways.

Our own work owes much to collaboration with Jean-Raymond Abrial and Mike Spivey who have been a constant source of new and exciting ideas. Much of our contact with other researchers has been made possible by the generosity of British Petroleum Ltd.

Data refinement by calculation

Carroll Morgan P.H.B. Gardiner*

9 July 1988

Abstract

Data refinement is the systematic substitution of one data type for another in a program. Usually, the new data type is more efficient than the old, but possibly more complex; the purpose of the data refinement in that case is to make progress in program construction from more abstract to more concrete formulations.

A recent trend in program construction is to *calculate* programs from their specifications; that contrasts with proving that a *given* program satisfies some specification. We investigate to what extent the trend can be applied to data refinement.

1 Introduction

In [3], Back proposed an extension of Dijkstra's calculus [12] where specifications and programs are given equal status during program construction. Later interest in specifications generally has led quite recently to further work on such constructions [41, 35, 37, 5, 34, 6, 39]. The style is now known as the *refinement calculus*.

Characteristic of any calculus is that it is used for *calculation*, not just description. The refinement calculus, therefore, should allow programs to

*Supported by British Petroleum Ltd.
[0]Submitted to *Acta Informatica*.

be calculated from their specifications. It does indeed allow presentations in which each intermediate design follows from a previous design according to some *law of refinement*. That contrasts with the more well-known style in which intermediate designs are first proposed and then proved to follow from their antecedents. Our hope is that constructions in the refinement calculus will proceed more smoothly, and that proof obligations will be reduced. That is the point of a calculus, and it can be observed elsewhere: for example, in the differential calculus one uses laws of differentiation, not proofs from first principles. For differentiation, the process is now mechanical. In the integral calculus, we have laws too — but there, as in the refinement calculus, success is not guaranteed.

Data refinement is a special case of refinement: one replaces an abstract type by a more concrete type in a program while preserving its algorithmic structure. Abstract operations are similarly replaced by corresponding concrete operations. It is a well-established technique, with its own specialised proof rules [23, 26].

Our principal contribution is to draw data refinement into the calculational style: we show how to *calculate* data refinements rather than prove them. Our emphasis here is on practice, in contrast to our earlier [13]: this paper gives applications of that theory, though for convenience we have presented afresh some proofs which are corollaries by specialisation of [13]. Recent work by Morris [40] addresses the same concerns that we do.

In passing we formalise *logical constants*, long used in program derivation, but not until now treated rigorously. Their use in programs loses the property of conjunctivity, another of Dijkstra's healthiness laws [12]. (The law of the excluded miracle, and continuity, have already been abandoned [35, 41, 43, 11].)

This work relies on the ideas of the refinement calculus, reviewed in Sections 2 and 3 below. More detail can be found in [35, 37, 41, 4].

2 Refinement

We consider Dijkstra's programming language [12], whose meaning is given by *predicate transformers*. For any program P, we write $[\![P]\!]$ for its *meaning*; and that meaning is a function from (desired) final assertions to (necessary) initial ones:

> For any formula ψ over state variables, and program P, $[\![P]\!]\psi$ is the *weakest* formula whose truth in an initial state ensures that activation of P will lead to a final state in which ψ is true.

Thus we write $[\![P]\!]\psi$ for Dijkstra's $wp(P, \psi)$.

2.1 Algorithmic refinement

A program P is *algorithmically* refined by another P' whenever every specification satisfied by P is satisfied by P' also. We restrict our specifications, however, to formulae $\phi \Rightarrow [\![P]\!]\psi$ which state "the program P must be such that its activation in a state in wbich ϕ is true will lead to a state in which ψ is true." We do not, for example, specify time or space constraints.

Definition 1 *Algorithmic refinement:* Program P is algorithmically refined by program P' precisely when, for all formulae ϕ and ψ over the program variables,

$$\phi \Rightarrow [\![P]\!]\psi \quad \text{implies} \quad \phi \Rightarrow [\![P']\!]\psi.$$

We write $P \sqsubseteq P'$ for tbat relationship.
♡

The following is an easy consequence of Definition 1, and is what we will use in practice:

Lemma 1 *Algorithmic refinement:* For programs P and P', we have $P \sqsubseteq P'$ precisely when

$$[\![P]\!]\psi \Rightarrow [\![P']\!]\psi \quad \text{for all formulae } \psi \text{ over the program variables.}$$

Proof: For *if*, note that $\phi \Rightarrow \langle\!|P|\!\rangle\psi$ and $[\![P]\!]\psi \Rightarrow [\![P']\!]\psi$ imply $\phi \Rightarrow \langle\!|P'|\!\rangle\psi$ as required; for *only if*, take ϕ to be $[\![P]\!]\psi$ itself.
♡

We assume that in Lemma 1 we may limit our choice of formulae ψ to those containing only variables free either in P or P' or both.

2.2 Data refinement

Data refinement arises as a special case of algorithmic refinement. A program P is data-refined to another program P' by a transformation in which some so-called *abstract* data-type in P is replaced by a *concrete* data-type in P'. The overall effect is an algorithmic refinement of the block in which the abstract data type is declared.

For that, we add *local variables* to Dijkstra's language in the following (standard) way:

Definition 2 *Local variables:* For (list of) variables l, formula I (the initialisation), and program P, the construction

$$|[\ \mathbf{var}\ l \mid I \bullet P\ ]|$$

is a *local block* in which the local variables l are introduced for the use of program P; they are first assigned initial values such that I holds. We define, for ψ not containing l,

$$[\![\ |[\ \mathbf{var}\ l \mid I \bullet P\ ]|\ ]\!]\psi \ \ \hat{=}\ \ (\forall l \bullet I \Rightarrow [\![P]\!]\psi)$$

♡

Note that the scope of quantifiers is indicated explicitly by parentheses $(\forall \cdots)$; the spot $\bullet$ reads "such that".

Where a postcondition ψ does contain the local variable l, Definition 2 can be applied after systematic change of the local l to some fresh l'. We assume therefore that such clashes do not occur.

Where appropriate, we consider *types* to be simply sets of values, and will write $[\![\textbf{var } l\colon T \mid I \bullet P]\!]$ for $[\![\textbf{var } l \mid (l \in T \wedge I) \bullet P]\!]$; thus a variable is initialised to some value in its type. And if I is just *true* we may omit it, writing $[\![\textbf{var } l \bullet P]\!]$ or $[\![\textbf{var } l\colon T \bullet P]\!]$ as appropriate.

Now data-refinement transforms an abstract block $[\![\textbf{var } a \mid I \bullet P]\!]$ to a concrete block $[\![\textbf{var } c \mid I' \bullet P']\!]$. We assume that the concrete variables c do not occur in the abstract program I and P, and *vice versa*. The transformation has these characteristics:

1. The concrete block algorithmically refines the abstract block:

$$[\![\textbf{var } a \mid I \bullet P]\!] \sqsubseteq [\![\textbf{var } c \mid I' \bullet P']\!].$$

2. The abstract variable declarations **var** a are replaced by concrete variable declarations **var** c.

3. The abstract initialisation I is replaced by a concrete initialisation I'.

4. The abstract program P, referring to variables a but not c, is replaced by a concrete program P' referring to variables c but not a; moreover, the algorithmic structure of P is reproduced in P' (see below).

The four characteristics are realised as follows. An *abstraction invariant* AI is chosen which links the abstract variables a and the concrete variables c. It may be *any* formula, but usually will refer to a and c at least. (See Section 4.1 below for a discussion of the impracticality of choosing *false* as the abstraction invariant.) The concrete initialisation I' must be such that $I' \Rightarrow (\exists a \bullet AI \wedge I)$. For the concrete program we define a relation $\preceq$ of data-refinement:

Definition 3 *Data refinement:* A program P is said to be data-refined by another program P', using abstraction invariant AI, abstract variables a and concrete variables c, whenever for all formulae ψ not containing c free we have

$$(\exists a \bullet AI \wedge [\![P]\!]\psi) \Rightarrow [\![P']\!](\exists a \bullet AI \wedge \psi)$$

We write this relation $P \preceq_{AI,a,c} P'$, and omit the subscript $_{AI,a,c}$ when it is understood from context.
♡

Definition 3 is appropriate for two reasons. The first is that it guarantees characteristic 1, as we now show.

Theorem 1 *Soundness of data-refinement:* If $I' \Rightarrow (\exists\, a \bullet AI \wedge I)$ and $P \preceq P'$, then

$$|[\ \mathbf{var}\ a \mid I \bullet P\ ]| \sqsubseteq |[\ \mathbf{var}\ c \mid I' \bullet P'\ ]|$$

Proof: Consider any ψ not containing a or c free. We have

$$\begin{array}{lll}
& [\![\ |[\ \mathbf{var}\ a \mid I \bullet P\ ]|\ ]\!]\psi & \\
= & (\forall a \bullet I \Rightarrow [\![P]\!]\psi) & \text{Definition 2} \\
= & (\forall c, a \bullet I \Rightarrow [\![P]\!]\psi) & c \text{ not free in above} \\
\Rightarrow & (\forall c, a \bullet AI \wedge I \Rightarrow AI \wedge [\![P]\!]\psi) & \\
\Rightarrow & (\forall c \bullet (\exists a \bullet AI \wedge I) \Rightarrow (\exists a \bullet AI \wedge [\![P]\!]\psi)) & \\
\Rightarrow & (\forall c \bullet I' \Rightarrow (\exists a \bullet AI \wedge [\![P]\!]\psi)) & \text{assumption} \\
\Rightarrow & (\forall c \bullet I' \Rightarrow [\![P']\!](\exists a \bullet AI \wedge \psi)) & \text{assumption; Definition 3} \\
\Rightarrow & (\forall c \bullet I' \Rightarrow [\![P']\!]\psi) & \text{monotonicity; } a \text{ not free in } \psi \\
= & [\![\ |[\ \mathbf{var}\ c \mid I' \bullet P'\ ]|\ ]\!]\psi & \text{Definition 2}
\end{array}$$

♡

The second reason our Definition 3 is appropriate is that it distributes through program composition. This is shown in [40, 13], and we refer the reader there for details. Here, for illustration, we treat sequential composition; alternation and iteration are dealt with in Sections 4 and 6 below.

Lemma 2 *Data-refinement distributes through sequential composition:* If $P \preceq P'$ and $Q \preceq Q'$ then $(P; Q) \preceq (P'; Q')$.

Proof: Let ψ be any formula not containing c. Then

$$\begin{array}{lll}
& (\exists a \bullet AI \wedge [\![P; Q]\!]\psi) & \\
= & (\exists a \bullet AI \wedge [\![P]\!]([\![Q]\!]\psi)) & \text{semantics of ";"} \\
\Rightarrow & [\![P']\!](\exists a \bullet AI \wedge [\![Q]\!]\psi) & P \preceq P' \\
\Rightarrow & [\![P']\!]([\![Q']\!](\exists a \bullet AI \wedge \psi)) & Q \preceq Q'\text{; monotonicity} \\
= & [\![P'; Q']\!](\exists a \bullet AI \wedge \psi) & \text{semantics of ";"}
\end{array}$$

♡

It is the distributive property illustrated by Lemma 2 that accounts for characteristic 4 above: if for example P is $P_1;\ P_2;\ \cdots P_n$ then we can construct P' with $P \preceq P'$ simply by taking $P' = P_1';\ P_2';\ \cdots P_n'$ with $P_i \preceq P_i'$ for each i. It is in this sense that the structure of P is preserved in P'. We will see in Section 4 below that this carries through for alternations and iterations also.

3 Language extensions

We extend Dijkstra's language in two ways. With the *specification statement* we allow specifications and executable program fragments to be mixed, thus promoting a more uniform development style. With *program conjunction* we make more rigorous the use of so-called *logical constants*, which appear in specifications but not in executable programs.

3.1 Specification statements

A specification statement is a list of changing variables called the *frame* (say w), a formula called the *precondition* (say *pre*), and a formula called the *postcondition* (say *post*). Together they are written

$$w\colon\ [pre\ ,\ post]\,.$$

Informally this construct denotes a program which,

> if *pre* is true in the initial state, will establish *post* in the final state by changing only variables mentioned in the list w.

For the precise meaning, we have

Definition 4 *Specification statement:* For formulae *pre*, *post* over the program variables, and list of variables w,

$$[\![\, w\colon [pre\ ,\ post]\ ]\!]\psi \mathrel{\widehat{=}} pre \wedge (\forall w \bullet post \Rightarrow \psi)$$

♡

The symbol $\mathrel{\widehat{=}}$ is read "is defined to be equal to".

Specification statements allow program development to proceed at the level of refinement steps $\sqsubseteq$ rather than directly in terms of weakest preconditions, and are discussed in detail in [35, 37]. They are similar to the *descriptions* of [4] and the *prescriptions* of [41]. For now we extract from the above works a collection of *refinement laws*, given in the appendix to this paper. We illustrate their use with the following small program development:

"assign to y the absolute value of x"

$= y\colon [true\ ,\ y = |x|\ ]$

$= y\colon [(x \leq 0) \vee (x \geq 0)\ ,\ y = |x|\ ]$

$\sqsubseteq$ Law 13

```
if x ≤ 0 → y: [x ≤ 0 , y = |x| ]
[] x ≥ 0 → y: [x ≥ 0 , y = |x| ]
fi
```

$=$

```
if x ≤ 0 → y: [−x = |x| , y = |x| ]
[] x ≥ 0 → y: [x = |x| , y = |x| ]
fi
```

$\sqsubseteq$ Law 12 twice

```
if x ≤ 0 → y := −x
[] x ≥ 0 → y := x
fi
```

3.2 Program conjunction

Given a program P we write the generalised *program conjunction* of P over some variable i as $[\![\text{ con } i \bullet P]\!]$. We call it conjunction because that new program is a refinement $\sqsubseteq$ of the original program P for *all* values of the *logical constant* i. For example, consider the statement x: $[x = i \ , \ x = i + 1]$, and suppose our variables range over the natural numbers. Its generalised conjunction over i refines all of the following:

$$
\begin{array}{l}
x\text{: } [x = 0 \ , \ x = 1] \\
x\text{: } [x = 1 \ , \ x = 2] \\
x\text{: } [x = 2 \ , \ x = 3] \\
\vdots
\end{array}
$$

Each of those programs deals with a specific value of x, and can abort for all others. Yet, as Definition 5 will show, that generalised conjunction equals the statement $x := x + 1$, which is guaranteed to terminate.

Definition 5 *Program conjunction:* For program P and variable i not free in ψ,

$$[\![\, [\![\text{ con } i \bullet P]\!] \,]\!]\psi \triangleq (\exists i \bullet [\![P]\!]\psi)$$

♡

As in Definition 2, systematic renaming can deal with occurrences of i in ψ.

Thus for the example above we can calculate

$$
\begin{array}{lll}
& [\![\, [\![\text{ con } i \bullet i\text{: } [x = i \ , \ x = i + 1] \,]\!] \,]\!]\psi & \\
= & (\exists i \bullet [\![\, i\text{: } [x = i \ , \ x = i + 1] \,]\!]\psi) & \text{Definition 5} \\
= & (\exists i \bullet x = i \wedge (\forall x \bullet x = i + 1 \Rightarrow \psi)) & \text{Definition 4} \\
= & (\exists i \bullet x = i \wedge \psi[x \backslash i + 1]) & \\
= & \psi[x \backslash i + 1][i \backslash x] & \\
= & \psi[x \backslash x + 1] & i \text{ not free in } \psi \\
= & [\![x := x + 1]\!]\psi &
\end{array}
$$

The notation $[x\backslash i+1]$ indicates syntactic replacement of x by $i+1$ with any changes of bound variable necessary to avoid capture.

Variables declared by con we call *logical constants*. They usually appear in program developments where some initial value must be fixed, in order to allow later reference to it. For example in the Hoare style [21], we might write "find a program P, changing only x, such that $\{x = X\}P\{x = X+1\}$". Here the upper case X makes use of a convention that such variables are not to appear in the final program: it is not $x := X + 1$ that is sought, but $x := x + 1$. We would just write

$$|[\ \mathbf{con}\ X \bullet x{:}\ [x = X\ ,\ x = X + 1]\ ]|,$$

it being understood that we are looking for a refinement of that. Since our final programming language does not allow declarations con, we are forced to use refinements whose effect is to eliminate X. We do not need an upper-case convention.

It is interesting that program conjunction is the dual of local variable declaration (compare Definitions 2 and 5); thus logical constants are in that sense dual to local variables. It is shown in [13] that data refinement distributes through program conjunction.

4 Data refinement calculators

In Section 2 we defined the relation $\preceq$ of data-refinement between two statements S and S'. We gave there also a sufficient relation between the abstract initialisation I and the concrete initialisation I'.

In this section we show how the extensions of Section 3 allow us to *calculate* data-refinements S' and I' which satisfy the sufficient relations automatically. Following [28], we call these techniques *calculators*.

For the rest of this section, we will assume that the data-refinement is given by

abstract variables:	a
concrete variables:	c
abstraction invariant:	AI

Moreover, we assume that the concrete variables c do not appear free in the abstract program.

4.1 The *initialisation* calculator

For concrete initialisation I' to data-refine the abstract I we know from Theorem 1 that $I' \Rightarrow (\exists\, a \bullet AI \wedge I)$ is sufficient; therefore we define I' to be $(\exists\, a \bullet AI \wedge I)$ itself. Law 5 (appendix) shows that we lose no generality, since any concrete initialisation I', where $I' \Rightarrow (\exists\, a \bullet AI \wedge I)$, can be reached in two stages: first replace I by the calculated $(\exists\, a \bullet AI \wedge I)$; then strengthen that, by Law 5, to I'.

If AI is *false*, then the calculated I' will be *false* also; indeed, Law 5 allows a refinement step to *false* initialisation directly. That is valid, though impractical, for the following reason: Definition 2 shows that the resulting program is *miraculous*:

$$[\![\, |[\ \mathbf{var}\ l \mid \mathit{false} \bullet P\ ]|\,]\!]\mathit{false} = \mathit{true}.$$

It can never be implemented in a programming language. (And that is why programming languages do not have empty types.)

4.2 The *specification* calculator

Lemma 3 to follow gives us a calculator for the data-refinement of any abstract statement of the form $a, x{:}\ [\mathit{pre}\ ,\ \mathit{post}]$, where a and x are disjoint (and either may be empty). Lemma 4 shows that taking that data-refinement loses no generality. The two results are combined in Theorem 2. Finally, we give as a corollary a calculator for statements $b, x{:}\ [\mathit{pre}\ ,\ \mathit{post}]$ where b is a subset of a; that is an abstract statement which may require some abstract variables not to change.

Lemma 3 *Validity:* The following data-refinement is always valid:

$$a, x{:}\ [\mathit{pre}\ ,\ \mathit{post}]$$

$$\preceq\ c,x\colon [(\exists\, a \bullet AI \wedge pre)\ ,\ (\exists\, a \bullet AI \wedge post)]$$

Proof: We take any formula ψ containing no free c, and proceed as follows:

$$\begin{array}{lll}
 & (\exists\, a \bullet AI \wedge [\![\ a,x\colon\ [pre\ ,\ post]\ ]\!]\psi) & \\
= & (\exists\, a \bullet AI \wedge pre \wedge (\forall\, a,x \bullet post \Rightarrow \psi)) & \text{Definition 4} \\
= & (\exists\, a \bullet AI \wedge pre) \wedge (\forall\, c,a,x \bullet post \Rightarrow \psi) & c \text{ not free in } post,\ \psi \\
\Rightarrow & (\exists\, a \bullet AI \wedge pre) \wedge (\forall\, c,x,a \bullet AI \wedge post \Rightarrow AI \wedge \psi) & \\
\Rightarrow & (\exists\, a \bullet AI \wedge pre) \wedge (\forall\, c,x \bullet (\exists\, a \bullet AI \wedge pre) \Rightarrow (\exists\, a \bullet AI \wedge \psi)) & \\
= & [\![\ c,x\colon\ [(\exists\, a \bullet AI \wedge pre)\ ,\ (\exists\, a \bullet AI \wedge post)]\ ]\!](\exists\, a \bullet AI \wedge \psi) &
\end{array}$$

$\heartsuit$

Lemma 4 *Generality:* For all programs CP, if $a,x\colon [pre\ ,\ post] \preceq CP$ then

$$c,x\colon [(\exists\, a \bullet AI \wedge pre)\ ,\ (\exists\, a \bullet AI \wedge post)] \sqsubseteq CP$$

Proof: We take any ψ containing no free a, and proceed as follows:

$$\begin{array}{lll}
 & [\![\ c,x\colon\ [(\exists\, a \bullet AI \wedge pre)\ ,\ (\exists\, a \bullet AI \wedge post)]\ ]\!]\psi & \\
= & (\exists\, a \bullet AI \wedge pre) & \text{Definition 4} \\
 & \quad \wedge (\forall\, c,x \bullet (\exists\, a \bullet AI \wedge post) \Rightarrow \psi) & \\
= & (\exists\, a \bullet AI \wedge pre) & c \text{ not free in } post, \\
 & \quad \wedge (\forall\, a,x \bullet post \Rightarrow (\forall\, c \bullet AI \Rightarrow \psi)) & a \text{ not free in } \psi \\
= & (\exists\, a \bullet AI \wedge pre \wedge (\forall\, a,x \bullet post \Rightarrow (\forall\, c \bullet AI \Rightarrow \psi))) & \\
= & (\exists\, a \bullet AI \wedge [\![a,x\colon\ [pre\ ,\ post]]\!](\forall\, c \bullet AI \Rightarrow \psi)) & \text{Definition 4} \\
\Rightarrow & [\![CP]\!](\exists\, a \bullet AI \wedge (\forall\, c \bullet AI \Rightarrow \psi)) & \text{assumption, Definition 3} \\
\Rightarrow & [\![CP]\!]\psi & a \text{ not free in } \psi, \text{ monotonicity}
\end{array}$$

$\heartsuit$

We now have the specification calculator we require: Lemma 3 states that it *is* a data refinement; Lemma 4 states that any other data-refinement

of the abstract specification is an *algorithmic* refinement of the calculated one. We summarise that in Theorem 2:

Theorem 2 *The specification calculator:* For all programs CP,

$$a, x\colon [pre\ ,\ post] \preceq CP$$

if and only if

$$c, x\colon [(\exists\ a\ \bullet\ AI \wedge pre)\ ,\ (\exists\ a\ \bullet\ AI \wedge post)] \sqsubseteq CP$$

Proof: From Lemmas 3 and 4.
♡

Note that the quantifications $(\exists\ a \cdots)$ ensure that the abstract variables a do not appear in the concrete program.

We conclude this section with a corollary of Lemma 3; it calculates the data-refinement of an abstract specification in which not all variables are changing. In its proof we are able to reason at the higher level of the relations $\sqsubseteq$ and $\preceq$; weakest preconditions are not required.

This corollary is the first occasion we have to use logical constants in data refinement. Like local variables, logical constants are *bound* in a program; and it is the **con** declaration which binds the abstract variables a in Corollary 1, since the quantification $(\exists\ b \cdots)$ alone may leave some abstract variables free.

Corollary 1 For any subset (not necessarily proper) b of the abstract variables a, the abstract specification $b, x\colon [pre\ ,\ post]$ is data-refined by

$$\begin{array}{l} [\![\ \textbf{con}\ a\ \bullet \\ \quad c, x\colon\ [AI \wedge pre\ ,\ (\exists\ b\ \bullet\ AI \wedge post)] \\ ]\!] \end{array}$$

Proof: Let b and y partition a, and let B and Y partition A correspondingly. Then

$$b, x\colon [pre\ ,\ post]$$

$=$ Law 9

$$|[\ \mathbf{con}\ Y \bullet\ b, y, x\colon [pre \wedge y = Y\ ,\ post \wedge y = Y]\ ]|$$

$\preceq$ Lemma 3

$$|[\ \mathbf{con}\ Y \bullet\ c, x\colon [(\exists\, b, y \bullet AI \wedge pre \wedge y = Y)\ ,\ (\exists\, b, y \bullet AI \wedge post \wedge y = Y)]\ ]|$$

$=$

$$|[\ \mathbf{con}\ Y \bullet\ c, x\colon [(\exists\, b \bullet AI \wedge pre)[y \backslash Y]\ ,\ (\exists\, b \bullet AI \wedge post)[y \backslash Y]]\ ]|$$

$=$ Law 8

$$|[\ \mathbf{con}\ y \bullet\ c, x\colon [(\exists\, b \bullet AI \wedge pre)\ ,\ (\exists\, b \bullet AI \wedge post)]\ ]|$$

$=$ Law 6

$$|[\ \mathbf{con}\ a \bullet\ c, x\colon [AI \wedge pre\ ,\ (\exists\, b \bullet AI \wedge post)]\ ]|$$

♡

4.3 The *guard* calculator

We saw in Corollary 1 that the specification calculator introduces **con** a and existentially quantifies over changing abstract variables only. For guards, changing nothing, we would expect that quantification to be empty. We have

Theorem 3 *The guard calculator:* If $S_i \preceq S_i'$ for each i, then the following refinement is valid:

$$\mathbf{if}\ ([]i \bullet G_i \rightarrow S_i)\ \mathbf{fi}$$

$$\preceq \; [\![\; \mathbf{con}\ a \bullet \\ \quad \mathbf{if}\ ([]i \bullet AI \wedge G_i \rightarrow S_i')\ \mathbf{fi} \\ ]\!]$$

Proof: For any ψ not containing c, we have

$$\begin{array}{ll} & (\exists\, a \bullet AI \wedge [\![\ \mathbf{if}\ ([]i \bullet G_i \rightarrow S_i)\ \mathbf{fi}\ ]\!]\psi) \\ = & \qquad\qquad \text{definition } [\![\mathbf{if} \cdots \mathbf{fi}]\!] \\ & (\exists\, a \bullet AI \wedge (\bigvee i \bullet G_i) \wedge (\bigwedge i \bullet G_i \Rightarrow [\![S_i]\!]\psi)) \\ = & (\exists\, a \bullet (\bigvee i \bullet AI \wedge G_i) \wedge (\bigwedge i \bullet AI \wedge G_i \Rightarrow AI \wedge [\![S_i]\!]\psi)) \\ \Rightarrow & (\exists\, a \bullet (\bigvee i \bullet AI \wedge G_i) \wedge (\bigwedge i \bullet AI \wedge G_i \Rightarrow (\exists\, a \bullet AI \wedge [\![S_i]\!]\psi))) \\ \Rightarrow & \qquad\qquad \text{since } S_i \preceq S_i' \\ & (\exists\, a \bullet (\bigvee i \bullet AI \wedge G_i) \wedge (\bigwedge i \bullet AI \wedge G_i \Rightarrow [\![S_i']\!](\exists\, a \bullet AI \wedge \psi))) \\ = & [\![\ [\![\ \mathbf{con}\ a \bullet \cdots]\!]\ ]\!](\exists\, a \bullet AI \wedge \psi) \end{array}$$

♡

A similar construction is possible for **do** ⋯ **od**, but in this general setting it is better to use **if** ⋯ **fi** and recursion. There are special cases for **do**, however, and they are discussed in Section 6.

5 Example of refinement: the "mean" module

We can present a data refinement independently of its surrounding program text by collecting together all the statements that refer to the abstract variables or to variables in the abstraction invariant. Such a collection is called a *module*, and we can confine our attention to it for this reason: statements which do *not* refer to abstract variables, or to the ahstraction invariant, are refined by themselves and we need not change them.

Consider the module of Figure 1 for calculating the mean of a sample of numbers. We write hag comprehensions hetween brackets $\prec\succ$, and use $\sum b$ and $\#b$ for the sum and size respectively of bag b. The operator $+$ is used for bag addition. The statement **error** is some definite error indication, and we assume that **error** $\preceq$ **error**. The initialisation is $b \in$ **bag of** *Real*.

```
module Calculator ≙
  var b: bag of Real;

  procedure Clear ≙ b := ≺≻ ;
  procedure Enter (value r) ≙ b := b + ≺ r ≻ ;
  procedure Mean (result m) ≙
    if b ≠ ≺≻ → m := ∑ b / #b
    [] b = ≺≻ → error
    fi
end
```

Figure 1: The "mean" module

The module is operated by: first *clearing*; then *entering* the sample values, one at a time; then finally taking the *mean* of all those values.

For the data refinement, we represent the bag by its sum s and size n at any time.

abstract variables:	b
concrete variables:	s, n
abstraction invariant:	$s = \sum b \wedge n = \#b$

We data-refine the module by replacing the abstract variables b by the concrete variables s, n and applying the calculations of Section 4 to the initialisation and the three procedures. Stacked formulae below denote their conjunction.

- For the initialisation, we have from Section 4.1 for the concrete initialisation

$$\left(\exists\, b \bullet \begin{array}{c} b \in \textbf{bag of } \mathit{Real} \\ s = \sum b \\ n = \#b \end{array} \right)$$

$$= \begin{pmatrix} s \in \mathit{Real} \\ n \in \mathit{Natural} \\ n = 0 \Rightarrow s = 0 \end{pmatrix}$$

- For the procedure *Clear*, we have from Section 4.2

$$\begin{aligned}
& b := \prec\succ \\
&= b{:}\ [\mathit{true}\ ,\ b = \prec\succ] \\
&\preceq \text{Lemma 3} \\
&\quad s, n{:}\ \left[\begin{pmatrix} \exists\, b \bullet \begin{matrix} s = \sum b \\ n = \#b \end{matrix} \end{pmatrix}\ ,\ \begin{pmatrix} \exists\, b \bullet \begin{matrix} s = \sum b \\ n = \#b \\ b = \prec\succ \end{matrix} \end{pmatrix} \right] \\
&\sqsubseteq \text{Law 1} \\
&\quad s, n{:}\ [\mathit{true}\ ,\ s = 0 \wedge n = 0] \\
&\sqsubseteq \text{Law 12} \\
&\quad s, n := 0, 0
\end{aligned}$$

- For the procedure *Enter*, we have from Section 4.2

$$\begin{aligned}
& b := b{+} \prec r \succ \\
&= [\![\mathbf{con}\ B \bullet b{:}\ [b = B\ ,\ b = B{+} \prec r \succ]]\!] \\
&\preceq \text{Lemma 3} \\
&\quad [\![\mathbf{con}\ B \bullet s, n{:}\ \begin{bmatrix} s = \sum B \\ n = \#B \end{bmatrix}\ ,\ \begin{matrix} s = \sum (B{+} \prec r \succ) \\ n = \#(B{+} \prec r \succ) \end{matrix} \Big]]\!] \\
&\sqsubseteq \text{Laws 12, 7} \\
&\quad s, n := s + r, n + 1
\end{aligned}$$

- For *Mean* we have first that from Section 4.2

$$\begin{aligned}
& m := \textstyle\sum b / \#b \\
&= m{:}\ [\#b \neq 0\ ,\ m = \textstyle\sum b / \#b] \\
&\preceq \text{Corollary 1 (noting the quantification is empty)} \\
&\quad [\![\mathbf{con}\ b \bullet \\
&\qquad m, s, n{:}\ \begin{bmatrix} \#b \neq 0 \\ s = \sum b \\ n = \#b \end{bmatrix}\ ,\ \begin{matrix} m = \sum b / \#b \\ s = \sum b \\ n = \#b \end{matrix} \\
&\quad]\!]
\end{aligned}$$

$\sqsubseteq$ **Laws 10, 2, 3, 1**

$\quad [\![\textbf{con } b \bullet m\colon\ [n \neq 0\ ,\ m = s/n]]\!]$

$\sqsubseteq$ **Laws 12, 7**

$\quad m := s/n$

Then we conclude from Theorem 3 that

$$\begin{array}{l}\textbf{if } b \neq \prec\succ \rightarrow m := \sum b / \# b \\ [\!] \ b = \prec\succ \rightarrow \textbf{error} \\ \textbf{fi}\end{array}$$

$$\preceq\ [\![\textbf{con } b \bullet \textbf{if} \left(\begin{array}{l} b \neq \prec\succ \\ s = \sum b \\ n = \# b \end{array}\right) \rightarrow m := s/n \ [\!] \left(\begin{array}{l} b = \prec\succ \\ s = \sum b \\ n = \# b \end{array}\right) \rightarrow \textbf{error} \ \textbf{fi} \ ]\!]$$

To make further progress with *Mean*, we need to eliminate the abstract variable b from the guards; then Law 7 applies. That is assisted by the following lemma (which is generally applicable to the refinement of alternations, whether or not they occur within data refinements):

Lemma 5 *Refining guards:* Given the conditions

1. $(\bigvee i \bullet G_i) \Rightarrow (\bigvee i \bullet G'_i)$
2. $(\bigvee i \bullet G_i) \Rightarrow (G'_i \Rightarrow G_i)$ *for each* i

the following refinement is valid:

$$\textbf{if } ([\!]i \bullet G_i \rightarrow S_i) \textbf{ fi} \ \sqsubseteq\ \textbf{if } ([\!]i \bullet G'_i \rightarrow S_i) \textbf{ fi}$$

Proof: By Lemma 1 and $[\![\textbf{if} \cdots \textbf{fi}]\!]$ we must show for all formulae ψ that

$$(\bigvee i \bullet G_i) \wedge (\bigwedge i \bullet G_i \Rightarrow [\![S_i]\!]\psi)$$
$$\Rightarrow (\bigvee i \bullet G'_i) \wedge (\bigwedge i \bullet G'_i \Rightarrow [\![S_i]\!]\psi)$$

That follows by propositional calculus from assumptions 1 and 2 above. ♡

We have immediately the following corollary:

Corollary 2 *Weakening guards.* The following refinement is valid for *any* formula X:

$$\textbf{if } ([]i \bullet G_i \wedge X \rightarrow S_i) \textbf{ fi} \sqsubseteq \textbf{if } ([]i \bullet G_i \rightarrow S_i) \textbf{ fi}$$

♡

Now we can continue the refinement of *Mean*:

$\sqsubseteq$ Lemma 5, Law 7
if $n \neq 0 \rightarrow m := s/n$
[] $n = 0 \rightarrow$ **error**
fi

In Figure 2 we give the resulting data refinement for the whole module.

To see the need for the initialisation, consider this alternative definition of *Clear*:

procedure *Clear* $\widehat{=}$
if $b \neq \prec\succ \rightarrow b := \prec\succ$
[] $b = \prec\succ \rightarrow$ **skip**
fi

That is semantically identical to the original, in Figure 1, but might be cheaper overall if the operation $b := \prec\succ$ were expensive. Its calculated data refinement is

```
module Calculator ≙
  var s: Real; n: Natural;

  procedure Clear ≙ s, n := 0, 0;
  procedure Enter(value r) ≙ s, n := s + n, n + 1;
  procedure Mean(result m) ≙
    if n ≠ 0 → m := s/n
    [] n = 0 → error
    fi

  initially n = 0 ⇒ s = 0
end
```

Figure 2: The "mean" module, after data refinement

```
procedure Clear ≙
  if n ≠ 0 → s, n := 0, 0
  [] n = 0 → skip
  fi
```

That would *not* work correctly if used immediately after an initialisation, say, of $s = 1 \wedge n = 0$! So our stated initialisation is necessary, after all; note however that since initialisations can always be strengthened (Law 5), we could use the simpler $s = 0$ if desired.

6 Specialised techniques

Now we specialise the techniques of Section 4: we consider *guards*, *functional* data-refinement, and the use of *auxiliary variables*.

6.1 Data-refining guards

We have seen that data refinement takes an abstract guard G to a concrete guard $G \wedge AI$, where AI is the abstraction invariant. The occurrences of abstract variables in this concrete guard must then be eliminated. We use Lemma 5 for that: we replace each of the calculated guards $G_i \wedge AI$ by the guard $(\forall a \bullet AI \Rightarrow G_i)$, which does not contain a free. By that lemma, we must show

1. $(\bigvee i \bullet G_i \wedge AI) \Rightarrow (\bigvee i \bullet (\forall a \bullet AI \Rightarrow G_i))$
2. $(\bigvee i \bullet G_i \wedge AI) \Rightarrow ((\forall a \bullet AI \Rightarrow G_i) \Rightarrow G_i \wedge AI)$ *for each* i

The validity of 2 is evident; and by rewriting 1 we can see that it requires only that the data-refined *disjunction* of the abstract guards implies the disjunction of the concrete guards. Thus we have the following

Lemma 6 *Data refinement of alternations:* Given abstraction invariant AI, abstract guards G_i, and abstract statements S_i, let the concrete guards G_i' and concrete statements S_i' be such that

1. $G_i' = (\forall a \bullet AI \Rightarrow G_i)$
2. $S_i \preceq S_i'$

Then provided $(\exists a \bullet AI \wedge (\bigvee i \bullet G_i)) \Rightarrow (\bigvee i \bullet G_i')$, the following data refinement is valid:

$$\textbf{if } ([]i \bullet G_i \rightarrow S_i) \textbf{ fi} \;\preceq\; \textbf{if } ([]i \bullet G_i' \rightarrow S_i') \textbf{ fi}$$

♡

For iterations the result is the same: we use the recursive formulation

$$\textbf{do } ([]i \bullet G_i \rightarrow S_i) \textbf{ od} \triangleq (\mu P \bullet \textbf{if } ([]i \bullet G_i \rightarrow S_i;\ P) \;[]\; \neg(\bigvee i \bullet G_i) \rightarrow \textbf{skip } \textbf{fi})$$

and hence must determine the conditions under which

$$
\begin{array}{ll}
 & \textbf{if } (\llbracket i \bullet AI \wedge G_i \rightarrow S_i;\ P) \\
 & \llbracket\ AI \wedge \neg(\bigvee i \bullet G_i) \rightarrow \textbf{skip} \\
 & \textbf{fi} \\
 & \\
\sqsubseteq & \textbf{if } (\llbracket i \bullet G'_i \rightarrow S'_i;\ P) \\
 & \llbracket\ \neg(\bigvee i \bullet G'_i) \rightarrow \textbf{skip} \\
 & \textbf{fi}
\end{array}
$$

As before we have defined G'_i to be $(\forall a \bullet AI \Rightarrow G_i)$. Straightforward application of Lemma 5 gives us

Lemma 7 *Data refinement of iterations:* Under the same conditions as Lemma 6, the following refinement is valid:

$$\textbf{do } (\llbracket i \bullet G_i \rightarrow S_i) \textbf{ od } \preceq \textbf{ do } (\llbracket i \bullet G'_i \rightarrow S'_i) \textbf{ od}$$

♡

Our choice of G'_i is used also in [40], where those two rules are proved from first principles (that is, from Definition 3). We have shown therefore how that technique is an instance of our Theorem 3.

6.2 Functional refinement

In many cases, the abstraction invariant is *functional* in the sense that for any concrete value there is at most one corresponding abstract value. In [26], for example, this is the primary form of data-refinement considered.

Functional abstraction invariants can always be written as a conjunction

$$
\begin{array}{l}
a = AF(c) \\
CI(c)
\end{array}
$$

where AF we call the *abstraction function* and CI the *concrete invariant*; the formula CI of course contains no occurrences of abstract variables a. We assume that $CI(c)$ implies well-definedness of AF at c.

Functional data-refinements usually lead to simpler calculations. First, the concrete formula $(\exists\ a \bullet AI \wedge \phi)$ — where ϕ is *pre* or *post* in the abstract specification — is simplified:

$$\begin{array}{ll} & (\exists\ a \bullet AI \wedge \phi) \\ = & (\exists\ a \bullet (a = AF(c)) \wedge CI(c) \wedge \phi) \\ = & CI(c) \wedge \phi[a \backslash AF(c)] \end{array}$$

Thus in this case data-refinement calculations are no more than simple substitutions. Note also that the resulting concrete formula contains no free abstract variables, and this allows any $[\![\ \mathbf{con}\ a \bullet \cdots]\!]$ to be eliminated immediately. We have this corollary of Theorem 2:

Corollary 3 *Functional data-refinement*: Given an abstraction invariant $a = AF(c) \wedge CI(c)$, the following data-refinement is always valid:

$$a, x{:}\ [pre\ ,\ post] \preceq c, x{:}\ \left[\begin{array}{c} pre[a \backslash AF(c)] \\ CI(c) \end{array}\ ,\ \begin{array}{c} post[a \backslash AF(c)] \\ CI(c) \end{array}\right]$$

Moreover, it is the most general.
♡

A second advantage is in the treatment of guards, as is shown also in [40]. We replace as before G_i by $G_i \wedge AI$, which becomes

$$\begin{array}{ll} & G_i \wedge (a = AF(c)) \wedge CI(c) \\ = & G_i[a \backslash AF(c)] \wedge (a = AF(c)) \wedge CI(c) \end{array}$$

Now by Corollary 2, we can eliminate the conjunct $a = AF(c)$ immediately, and hence the enclosing $[\![\ \mathbf{con}\ a \bullet \cdots]\!]$ as well. (And we can eliminate the $CI(c)$, but that is optional: it contains no a.) So we have the following result for the functional data-refinement of alternations:

Lemma 8 *Functional data-refinement of alternations:* Given abstraction invariant $(a = AF(c)) \wedge CI(c)$, abstract guards G_i, and abstract statements S_i, let concrete guards G_i' and concrete statements S_i' be such that

1. $G_i' = G_i[a \backslash AF(c)] \wedge CI(c)$
2. $S_i \preceq S_i'$

Then the following data refinement is always valid

$$\textbf{if } (\llbracket i \bullet G_i \rightarrow S_i) \textbf{ fi} \;\preceq\; \textbf{if } (\llbracket i \bullet G_i' \rightarrow S_i') \textbf{ fi}$$

♡

The same remarks apply to iteration (and again, the conjunct $CI(c)$ is optional in the concrete guards):

Lemma 9 *Functional data-refinement of iterations:* Under the same conditions as Lemma 8, the following data refinement is valid

$$\textbf{do } (\llbracket i \bullet G_i \rightarrow S_i) \textbf{ od} \;\preceq\; \textbf{do } (\llbracket i \bullet G_i' \rightarrow S_i') \textbf{ od}$$

♡

6.3 Auxiliary variables

A set of local variables is *auxiliary* if its members occur only in statements which assign to members of that set. They can be used for data refinement as follows.

There are three stages. In the first, an abstraction invariant is chosen, relating abstract variables to concrete. Declarations of those concrete variables are added to the program, but the declarations of the abstract variables are *not* removed. The initialisation is strengthened so that it implies the abstraction invariant; every guard is strengthened by conjoining the abstraction invariant; and every assignment statement is extended, if necessary, by assignments to concrete variables which *maintain* the the abstraction invariant.

In the second stage, the program is *algorithmically* refined so that the abstract variables become auxiliary. In the third stage, the (now) auxiliary abstract variables are removed (their declarations too), leaving only the concrete — and the data-refinement is complete.

That technique was proposed by [29], and a simple example is given in [12, p.64]. It is a special case of our present technique, as we now show. Suppose our overall aim is the following data-refinement:

abstract variables:	a
concrete variables:	c
abstraction invariant:	AI

We decompose this into two data-refinements, applied in succession. In the first, there are *no* abstract variables:

abstract variables:	(none)
concrete variables:	c
abstraction invariant:	AI

Clearly this removes no declarations, and from Definition 3 requires for $S \preceq S'$ (remembering that the quantification $(\exists\, a \bullet \cdots)$ is empty) only that for all ψ not containing c free, we have

$$AI \wedge [\![S]\!]\psi \Rightarrow [\![S]\!](AI \wedge \psi)$$

That is precisely the first stage explained informally above.

The second stage remains: it is only algorithmic refinement. For the third stage, we use the following data refinement in which there are no *concrete* variables:

abstract variables:	a
concrete variables:	(none)
abstraction invariant:	$true$

From Definition 3, here for $S \preceq S'$ we must show that for all formulae ψ

$$(\exists\, a \bullet [\![S]\!]\psi) \Rightarrow [\![S']\!](\exists\, a \bullet \psi)$$

And this holds only when the abstract variables a are auxiliary.

We illustrate the auxiliary technique with two lemmas, derived from our general rules for data refinement:

Lemma 10 *Introducing concrete variables while maintaining the invariant:* Let the abstract variables be *none*, the concrete variables be c, and the abstraction invariant AI. Then for abstract expression AE and concrete expression CE, we have

$$a := AE \preceq a, c := AE, CE$$

provided $AI \Rightarrow [\![a, c := AE, CE]\!] AI$.

Proof:

$$\begin{array}{lr}
AI \wedge [\![a := AE]\!]\psi & \\
\quad = AI \wedge \psi[a \backslash AE] & \text{by semantics of } := \\
\quad \Rightarrow [\![a, c := AE, CE]\!] AI \wedge \psi[a \backslash AE] & \text{by assumption} \\
\quad \Rightarrow AI[a, c \backslash AE, CE] \wedge \psi[a \backslash AE] & \text{by semantics of } := \\
\quad = AI[a, c \backslash AE, CE] \wedge \psi[a, c \backslash AE, CE] & \text{since } \psi \text{ contains no } c \\
\quad = [\![a, c := AE, CE]\!](AI \wedge \psi) & \text{by semantics of } :=
\end{array}$$

♡

Lemma 11 *Eliminating auxiliary variables:* Let the abstract variables he a, the concrete variables he *none*,and the abstraction invariant *true*. Then

1. $a := AE \preceq \mathbf{skip}$
2. $c := CE \preceq c := CE$

provided CE contains no occurrence of a.

Proof: For 1 we have

$$(\exists\, a \bullet [\![a := AE]\!]\psi)$$

$$
\begin{array}{ll}
= (\exists\, a \bullet \psi[a \backslash AE]) & \text{by semantics of } := \\
\Rightarrow (\exists\, a \bullet \psi) & \text{predicate calculus} \\
= [\![\mathbf{skip}]\!] (\exists\, a \bullet \psi) &
\end{array}
$$

For 2 we have

$$
\begin{array}{ll}
(\exists\, a \bullet [\![c := CE]\!] \psi) & \\
\quad = (\exists\, a \bullet \psi[c \backslash CE]) & \text{by semantics of } := \\
\quad = (\exists\, a \bullet \psi)\,[c \backslash CE] & \text{since } CE \text{ contains no } a \\
\quad = [\![c := CE]\!] (\exists\, a \bullet \psi) &
\end{array}
$$

(Note that in case 2 we did not assume that ψ contained no c.)
♡

If the abstract statement is a specification a: $[pre\ ,\ post]$, then in the first stage we replace it by a, c: $[pre \wedge AI\ ,\ post \wedge AI]$. If by the third stage (after algorithmic refinement) we still have a specification — say a, c: $[pre'\ ,\ post']$, then the removal of a as an auxiliary variable leaves us with c: $[(\exists\, a \bullet pre')\ ,\ (\exists\, a \bullet post')]$.

Let us as a final illustration try to remove a variable which is *not* auxiliary: we take the data-refinement as for the third stage, and suppose that $c := a \preceq CP$ for some concrete program CP. We expect this to fail, since a is clearly not auxiliary in $c := a$. Now we have for all constants n that

$$
\begin{array}{ll}
\mathit{true} & \\
\quad = (\exists\, c \bullet c = n) & \text{predicate calculus} \\
\quad = (\exists\, a \bullet (c = n)[c \backslash a]) & \text{renaming bound variable } c \text{ to } a \\
\quad = (\exists\, a \bullet [\![c := a]\!](c = n)) & \text{by semantics of } := \\
\quad \Rightarrow [\![CP]\!] (\exists\, a \bullet c = n) & \text{by assumption} \\
\quad = [\![CP]\!](c = n) &
\end{array}
$$

Since the above holds for any n, we have that CP always establishes both $c = 0$ and $c = 1$. Because no executable program can do this, we have shown that there is no such CP — as hoped, a cannot be eliminated from $c := a$. But what if we write $c := a$ as a specification? In that case, we have

$c := a$

$= c\colon [true\ ,\ c = a]$

$\preceq$ Corollary 1 (noting the quantification is empty)
$[\![\ \mathbf{con}\ a \bullet c\colon [true\ ,\ c = a]\ ]\!]$

So here we *have* a data-refinement, after all. But that is consistent with the above in the following way: there is no executable program CP (whether containing a or not) such that $c\colon [true\ ,\ c = a] \sqsubseteq CP$. Thus the $[\![\ \mathbf{con}\ a \bullet \cdots]\!]$ still cannot be eliminated.

In [31] the auxiliary variable technique is presented independently of the refinement calculus.

7 Conclusions

Our calculators for data refinement make it possible in principle to see that activity as the routine application of laws. The example of Section 5 is a demonstration for a simple case. It is important in practice, however, to take advantage of the specialised techniques of Section 6; otherwise, the subsequent algorithmic refinement will simply repeat the derivation of the techniques themselves, again and again.

That subsequent algorithmic refinement is in fact a lingering problem. In many cases, particularly with larger and more sophisticated refinements, the refined operations present fearsome collections of formulae concerning data structures for which we do not have an adequate body of theory. Their subsequent manipulations in the predicate calculus resemble programming in machine code. Fortunately, there is work on such theories (and *their* calculi, for example [10]), and we see little difficulty in taking advantage of them.

Our work on data refinement has been aided and improved by collaboration with Morris and Back, who present their work in [40] and [4] respectively. We extend Morris's approach by our use of logical constants (which, however, he has discovered in another context [39]). A second extension is

our "if and only if" result in Theorem 2. That is necessary, we feel, for a data refinement to be called a calculator: $P \leq Q$ is a *calculator* only if taking Q loses no generality. And Morris retains some restrictions on abstraction invariants which we believe are unnecessary. Conversely, Morris's specialised alternation calculator [40, Theorem 4] improves ours (Lemma 6) by introducing a miracle as the refined program [33]; his rule needs no proof obligation. Our work extends Back's by our emphasis on calculation, and our use of logical constants.

8 Acknowledgements

We are grateful to have had the opportunity to discuss our work with Ralph Back and Joe Morris, and for the comments made by members of IFIP WG 2.3. Much of our contact with other researchers has been made possible by the generosity of British Petroleum Ltd.

9 Appendix: refinement laws

Below is a collection of laws which can in principle take most specification statements through a series of refinements into executable code. We have not tried to make them complete. "Executable code" means program text which does not include either specification statements or logical constants.

"In principle" means that these basic rules, used alone, will in many cases give refinement sequences which are very long indeed — rather like calculating derivatives from first principles. But with experience, one collects a repertoire of more powerful and specific laws which make those calculations routine.

Some of the laws below are equalities $=$; some are proper refinements $\sqsubseteq$. In all cases they have been proved using the *weakest precondition* semantics of the constructs concerned.

Section 9.2 contains notes relating to the laws of Section 9.1.

9.1 Laws of program refinement

Most of these laws are extracted from [37], retaining only those used in this paper. Logical constant laws have been added.

1. *Weakening the precondition:* If $pre \Rightarrow pre'$ then

$$w\colon [pre\ ,\ post] \sqsubseteq w\colon [pre'\ ,\ post]$$

2. *Strengthening the postcondition:* If $post' \Rightarrow post$ then

$$w\colon [pre\ ,\ post] \sqsubseteq w\colon [pre\ ,\ post']$$

See Note 1. 3. *Assuming the precondition in the postcondition:*

$$w\colon [pre\ ,\ (\exists\, w \bullet pre) \wedge post] = w\colon [pre\ ,\ post]$$

4. *Introducing local variables:* If x does not appear free in *pre* or *post*, then

$$w\colon [pre\ ,\ post] \sqsubseteq |[\,\mathbf{var}\ x \mid I \bullet w,x\colon [pre\ ,\ post]\,]|$$

5. *Strengthening the initialisation:* If $I' \Rightarrow I$, then

$$|[\,\mathbf{var}\ x \mid I \bullet S\,]| \sqsubseteq |[\,\mathbf{var}\ x \mid I' \bullet S\,]|$$

See Note 2. 6. *Introducing logical constants:* If x does not appear free in *post*, then

$$w\colon [(\exists\, x \bullet pre)\ ,\ post] = |[\,\mathbf{con}\ x \bullet w\colon [pre\ ,\ post]\,]|$$

See Note 3. 7. *Eliminating logical constants:* If x does not appear free in P, then

$$|[\,\mathbf{con}\ x \bullet P\,]| = P$$

8. *Renaming logical constants:* If y is disjoint from w, and does not occur free in *pre* or *post*, then

$$\begin{aligned} &|[\,\mathbf{con}\ x \bullet w\colon [pre\ ,\ post]\,]| \\ &= |[\,\mathbf{con}\ y \bullet w[x\backslash y]\colon [pre[x\backslash y]\ ,\ post[x\backslash y]\,]\,]| \end{aligned}$$

9. *Expanding the frame:* If x and y are fresh variables, disjoint from each other, then

$$w\colon [pre\ ,\ post] \;=\; |[\,\mathbf{con}\ y \bullet w, x\colon [pre \wedge x = y\ ,\ post \wedge x = y]\,]|$$

10. *Contracting the frame:* If w and x are disjoint, then

$$w, x\colon [pre\ ,\ post] \sqsubseteq w\colon [pre\ ,\ post]$$

11. *Introducing* **skip**:

$$w\colon [post\ ,\ post] \sqsubseteq \mathbf{skip}$$

12. *Introducing assignment:* If E is an expression, then See Note 4.

$$w\colon [post[w \backslash E]\ ,\ post] \sqsubseteq w := E$$

13. *Introducing alternation:*

$$\begin{array}{l} w\colon [pre \wedge (\bigvee i \bullet G_i)\ ,\ post] \\ = \mathbf{if}\ ([]\, i \bullet G_i \rightarrow w\colon [pre \wedge G_i\ ,\ post])\ \mathbf{fi} \end{array}$$

9.2 Notes

1. Law 3 applies when information from the precondition is needed in the postcondition. We use it below to derive a stronger version of Law 2:

If $((\exists w \bullet pre) \wedge post') \Rightarrow post$, then

$w\colon [pre\ ,\ post]$

$\sqsubseteq$ by Law 2 and the assumption

$w\colon [pre\ ,\ (\exists w \bullet pre) \wedge post']$

$\sqsubseteq$ by Law 3

$w\colon [pre\ ,\ post']$

2. Usually Law 6 is used to introduce an equality into the precondition which "saves an initial value for later." That is summarised in the following derived law:

If y is disjoint from w, and does not occur free in *pre* or *post*, then

w: $[pre\ ,\ post]$
$\sqsubseteq$ by Law 1
w: $[(\exists y \bullet x = y \wedge pre)\ ,\ post]$
$\sqsubseteq$ by Law 6
$|[\mathbf{con}\ y \bullet w$: $[x = y \wedge pre\ ,\ post]\ ]|$

3. *Logical constants*, introduced by **con**, are variables which we can use during program development but not in final programs. Usually they are used to fix initial values, as in

$|[\mathbf{con}\ X \bullet x$: $[x = X\ ,\ x = X + 1]\ ]|$
$\sqsubseteq$ by Law 12
$|[\mathbf{con}\ X \bullet x := x + 1\ ]|$
$\sqsubseteq$ by Law 7
$x := x + 1$

Since the keyword **con** is does not occur in our executable programming language — just as specification statements do not — it must be eliminated (using Law 7 as above) during the development process. Thus logical constants never appear in the final program, since they cannot be declared there.

4. Law 12 is usually applied together with Laws 10 and 1, as in the following derived rule:

If the variables w and x are disjoint, E is an expression, and $pre \Rightarrow post[w \backslash E]$, then

w, x: $[pre\ ,\ post]$
$\sqsubseteq$ by Law 10
w: $[pre\ ,\ post]$
$\sqsubseteq$ by Law 1 and the assumption
w: $[post[w \backslash E]\ ,\ post]$
$\sqsubseteq$ by Law 12
$w := E$

Laws of program refinement: a summary

Carroll Morgan

1 December 1987

1 Introduction

In Section 2 below is a collection of laws which can in principle take most specification statements through a series of refinements into "executable code." *Executable code* means program text which does not include either specification statements or logical constants (see below).

"In principle" means that these basic rules, used alone, will in many cases give refinement sequences which are very long indeed — rather like calculating derivatives from first principles. But with experience, one collects a repertoire of more powerful and specific laws which make these calculations routine.

Some of the laws below are equalities $=$; some are proper refinements $\sqsubset$. In all cases they have been proved using the *weakest precondition* calculus of Dijkstra, as explained in [12], [14], and [7]. This means in particular that specification statements too have a weakest precondition semantics.

Section 3 contains notes relating to the laws of Section 2. Section 4 presents a small example. Section 5 gives some useful abbreviations, and Section 6 some examples of derived laws. Section 7 gives weakest precondition semantics for the language extensions.

[0]Taken from [32].

2 Laws

1. *Weakening the precondition:* If $pre \Rightarrow pre'$ then

$$w\colon [pre\ ,\ post] \sqsubseteq w\colon [pre'\ ,\ post]$$

2. *Strengthening the postcondition:* If $post' \Rightarrow post$ then

$$w\colon [pre\ ,\ post] \sqsubseteq w\colon [pre\ ,\ post']$$

See Note 1. 3. *Assuming the precondition in the postcondition:*

$$w\colon [pre\ ,\ (\exists\, w \bullet pre) \wedge post] = w\colon [pre\ ,\ post]$$

See Note 2. 4. *Introducing local variables:* If x does not appear free in pre or $post$, then

$$w\colon [pre\ ,\ post] = |[\ \mathbf{var}\ x \mid I \bullet w, x\colon [pre\ ,\ post]\ ]|$$

5. *Strengthening the initialisation:* If $I' \Rightarrow I$, then

$$|[\ \mathbf{var}\ x \mid I \bullet S\ ]| \sqsubseteq |[\ \mathbf{var}\ x \mid I' \bullet S\ ]|$$

6. *Exporting the initialisation:*

$$|[\ \mathbf{var}\ x \mid (\exists\, x \bullet I') \wedge I \bullet S\ ]| = |[\ \mathbf{var}\ x \mid I' \bullet l\colon [I'\ ,\ I]\ ;\ S\ ]|$$

See Note 3. 7. *Renaming local variables:* If y is disjoint from w, and does not occur free in pre or $post$, then

$$
\begin{aligned}
& |[\,\mathbf{var}\ x \mid I \bullet w\colon [pre\ ,\ post]\,]| \\
= \ & |[\,\mathbf{var}\ y \mid I[x\backslash y] \bullet w[x\backslash y]\colon [pre[x\backslash y]\ ,\ post[x\backslash y]]\,]|
\end{aligned}
$$

8. *Introducing logical constants:* If x does not appear free in *post*, then See Notes 4, 5.

$$w\colon [(\exists\, x \bullet pre)\ ,\ post] = |[\,\mathbf{con}\ x \bullet w\colon [pre\ ,\ post]\,]|$$

9. *Eliminating logical constants:* If x does not appear free in P, then See Note 6.

$$|[\,\mathbf{con}\ x \bullet P\,]| = P$$

10. *Renaming logical constants:* If y is disjoint from w, and does not occur free in *pre* or *post*, then

$$
\begin{aligned}
& |[\,\mathbf{con}\ x \bullet w\colon [pre\ ,\ post]\,]| \\
= \ & |[\,\mathbf{con}\ y \bullet w[x\backslash y]\colon [pre[x\backslash y]\ ,\ post[x\backslash y]]\,]|
\end{aligned}
$$

11. *Expanding the frame:* If x and y are fresh variables, disjoint from each other, then See Notes 7, 8.

$$w\colon [pre\ ,\ post] = |[\,\mathbf{con}\ y \bullet w, x\colon [pre \wedge x = y\ ,\ post \wedge x = y]\,]|$$

12. *Contracting the frame:* If w and x are disjoint, then

$$w, x\colon [pre\ ,\ post] \sqsubseteq w\colon [pre\ ,\ post]$$

13. *Introducing* **skip**:

$$w\colon [post\ ,\ post] \sqsubseteq \mathbf{skip}$$

14. *Introducing* **abort**:

$$w\colon [\mathit{false}\ ,\ \mathit{post}] = \mathbf{abort}$$

15. *Introducing assignment:* If E is an expression, then See Note 9.

$$w\colon [\mathit{post}[w \backslash E]\ ,\ \mathit{post}] \sqsubseteq w := E$$

See Note 10. 16. *Introducing sequential composition:*

$$w\colon [\mathit{pre}\ ,\ \mathit{post}] \sqsubseteq w\colon [\mathit{pre}\ ,\ \mathit{mid}]\ ;\ w\colon [\mathit{mid}\ ,\ \mathit{post}]$$

17. *Introducing alternation:*

$$\begin{array}{l} w\colon [\mathit{pre} \wedge (\bigvee i \bullet G_i)\ ,\ \mathit{post}] \\ = \mathbf{if}\ ([]\, i \bullet G_i \rightarrow w\colon [\mathit{pre} \wedge G_i\ ,\ \mathit{post}])\ \mathbf{fi} \end{array}$$

See Notes 11, 12. 18. *Introducing iteration:*

$$\begin{array}{l} w\colon [\mathit{inv}\ ,\ \mathit{inv} \wedge \neg(\bigvee i \bullet G_i)] \\ \sqsubseteq\ \mathbf{do} \\ \quad ([]\, i \bullet G_i \rightarrow |[\,\mathbf{con}\ v \bullet w\colon [\mathit{inv} \wedge G_i \wedge v = E\ ,\ \mathit{inv} \wedge 0 \leq E < v]\,|]) \\ \mathbf{od} \end{array}$$

The predicate *inv* is the invariant; the expression E is the integer-valued variant; and the logical constant v is fresh.

3 Notes

1. Law 3 applies when information from the precondition is needed in the postcondition. We use it below to derive a stronger version of Law 2:

   If $((\exists w \bullet \mathit{pre}) \wedge \mathit{post}') \Rightarrow \mathit{post}$, then

w: $[pre\ ,\ post]$

$\sqsubseteq$ by Law 2 and the assumption

w: $[pre\ ,\ (\exists\, w \bullet pre) \wedge post']$

$\sqsubseteq$ by Law 3

w: $[pre\ ,\ post']$

2. In setting out derivations we allow the following abbreviation to Law 4:

w: $[pre\ ,\ post]$

$\sqsubseteq$ **var** x

w, x: $[pre\ ,\ post]$

Note 4 is similar.

3. We say "disjoint" rather than "distinct" because we allow variable names to stand for *vectors* of variables as well.

4. In setting out derivations we allow the following abbreviation to Law 8:

w: $[(\exists\, x \bullet pre)\ ,\ post]$

$\sqsubseteq$ **con** x

w: $[pre\ ,\ post]$

Note 2 is similar; Note 5 gives an example.

5. Usually Law 8 is used to introduce an equality into the precondition which "saves an initial value for later." This is summarised in the following derived law:

If y is disjoint from w, and does not occur free in *pre* or *post*, then

w: $[pre\ ,\ post]$

$\sqsubseteq$ by Law 1

w: $[(\exists\, y \bullet x = y \wedge pre)\ ,\ post]$

$\sqsubseteq$ con y

w: $[x = y \wedge pre\ ,\ post]$

6. *Logical constants*, introduced by **con**, are variables which we can use during program development but not in final programs. Usually they are used to fix initial values (but see Note 7), as in

$$
\begin{array}{l}
\quad [\![\mathbf{con}\ X \bullet x\colon\ [x = X\ ,\ x = X + 1]\ ]\!] \\
\sqsubseteq \text{ by Law 15} \\
\quad [\![\mathbf{con}\ X \bullet x := x + 1\ ]\!] \\
\sqsubseteq \text{ by Law 9} \\
\quad x := x + 1
\end{array}
$$

Since the keyword **con** does not occur in our executable programming language — just as specification statements do not — it must be eliminated (using Law 9 as above) during the development process. Thus logical constants never appear in the final program, since they cannot be declared there.

7. The idiom $[\![\mathbf{con}\ y \bullet w, x\colon\ [pre \wedge x = y\ ,\ post]\ ]\!]$, where *post* refers to y (that is, to the *initial* value of x), is so common that we adopt the following convention:

> *If a specification statement contains* 0*-subscripted variables — say* x_0 *— in its postcondition, then equalities* $x = x_0$ *are assumed in its precondition together with an enclosing declaration* $[\![\mathbf{con}\ x_0 \bullet \cdots]\!]$ *of* x_0 *as a logical constant.*

We will reserve 0-subscripts for variables used in this way.

8. With the convention of Note 7, the right-hand side of Law 11 can be simplified to

$$w, x\colon\ [pre\ ,\ post \wedge x = x_0]$$

9. Law 15 is usually applied together with Laws 12 and 1, as in the following derived rule:

> If the variables w and x are disjoint, E is an expression, and $pre \Rightarrow post[w \backslash E]$, then
>
> $$w, x\colon\ [pre\ ,\ post]$$

$\sqsubseteq$ by Law 12

w: $[pre\ ,\ post]$

$\sqsubseteq$ by Law 1 and the assumption

w: $[post[w\backslash E]\ ,\ post]$

$\sqsubseteq$ by Law 15

$w := E$

10. Often Law 16 will introduce logical constants which are not the simple equalities of Note 7. Consider for example the following:

$|[\ \mathbf{con}\ X \bullet x\colon\ [x = X\ ,\ x = \log \sin X]\ ]|$

$\sqsubseteq$ by Law 16

$|[\ \mathbf{con}\ X \bullet$
$\quad x\colon\ [x = X\ ,\ x = \sin X];$
$\quad x\colon\ [x = \sin X\ ,\ x = \log \sin X]$
$]|$

$\sqsubseteq$ by Note 9 twice

$|[\ \mathbf{con}\ X \bullet x := \sin x;\ x := \log x\ ]|$

$\sqsubseteq$ by Law 9

$x := \sin x;\ x := \log x$

11. With the convention of Note 7, the right-hand side of Law 18 can be simplified to

do
$\quad ([]i \bullet\ G_i \rightarrow w\colon\ [inv \wedge G_i\ ,\ inv \wedge 0 \leq E < E_0])$
od

The 0-subscripted *expression* E_0 is obtained by 0-subscripting all its free variables.

12. The inequality in this rule can be seen by taking inv and G — a single guard — both *false*.

4 An example

Below we present a derivation of a *greatest common divisor* algorithm. The steps taken are large, as an experienced developer would make them.

To save space, we will on occasion indicate with a dagger † the part of a program that is to be refined in the next step. In that case the text surrounding the refined part will not be repeated in subsequent steps.

$$x, y{:}\ [x \geq 1 \wedge y \geq 1\ ,\ x = \gcd(x_0, y_0)]$$

$\sqsubseteq$ by Note 7, Law 10

$$|[\mathbf{con}\ X, Y \bullet \quad x, y{:}\ \begin{bmatrix} x \geq 1 \\ y \geq 1 \\ x = X \\ y = Y \end{bmatrix}\ ,\ x = \gcd(X, Y) \Big] \quad \dagger \quad]|$$

$\sqsubseteq$ by Law 1, Law 2

$$x, y{:}\ \begin{bmatrix} x \geq 1 \\ y \geq 1 \\ \gcd(x, y) = \gcd(X, Y) \end{bmatrix}\ ,\ \begin{bmatrix} x \geq 1 \\ y \geq 1 \\ \gcd(x, y) = \gcd(X, Y) \\ x = y \end{bmatrix}$$

$\sqsubseteq$ by Law 18, Law 1, Note 7

$$\mathbf{do}\ x > y \rightarrow x, y{:}\ \begin{bmatrix} x > y \geq 1 \\ \gcd(x, y) = \gcd(X, Y) \end{bmatrix}\ ,\ \begin{bmatrix} x \geq 1 \\ y \geq 1 \\ \gcd(x, y) = \gcd(X, Y) \\ 0 \leq x + y < x_0 + y_0 \end{bmatrix}$$

$$y > x \rightarrow x, y{:}\ \begin{bmatrix} y > x \geq 1 \\ \gcd(x, y) = \gcd(X, Y) \end{bmatrix}\ ,\ \begin{bmatrix} x \geq 1 \\ y \geq 1 \\ \gcd(x, y) = \gcd(X, Y) \\ 0 \leq x + y < x_0 + y_0 \end{bmatrix}$$

$\mathbf{od}$

$\sqsubseteq$ by Note 7, Note 9

$$\begin{array}{l}\textbf{do } x > y \to x := x - y \\ \quad\; y > x \to y := y - x \\ \textbf{od}\end{array}$$

5 Abbreviations

The following notations abbreviate commonly occuring idioms.

5.1 Generalised assignment

If "$\lhd$" is a binary relation symbol, then $w : \lhd E$ for some expression E, abbreviates

$$w\colon\ [w \lhd\ E[w \backslash w_0]]$$

The variable w' is fresh, not appearing in E.

For example, $e\colon \in s$ abbreviates $e\colon\ [s \neq \{\}\ ,\ e \in s]$.

5.2 Invariants

For predicates *pre*, *inv* and *post* we have

$$w\colon\ [pre\ ,\ inv\ ,\ post] \text{ abbreviates } w\colon \begin{bmatrix} pre & , & post \\ inv & & inv \end{bmatrix}$$

This allows for example the rule for iteration introduction (Law 18, Note 8) to be written

$$\begin{array}{l} w\colon\ [true\ ,\ inv\ ,\ \neg(\bigvee i \bullet\ G_i)] \\ \sqsubseteq\ \textbf{do} \\ \quad (\llbracket i \bullet\ G_i \to w\colon\ [G_i\ ,\ inv\ ,\ 0 \leq E < E_0]) \\ \textbf{od}\end{array}$$

6 Derived laws

In this section we give some examples of derived laws.

6.1 Introducing assignment

This law applies directly to a specification whose postcondition contains 0-subscripted variables (that is, refers to initial values):

> If $pre \Rightarrow post[w\backslash e][v_0\backslash v]$, where the variable(s) v are those appearing 0-subscripted (free) in *post*, then
>
> $$w, x{:}\ [pre\ ,\ post] \sqsubseteq w := e$$

6.2 Introducing skip

This law also applies directly to a specification whose postcondition contains 0-subscripted variables:

> If $pre \Rightarrow post[v_0\backslash v]$, where the variable(s) v are those appearing 0-subscripted (free) in *post*, then
>
> $$w{:}\ [pre\ ,\ post] \sqsubseteq \textbf{skip}$$

6.3 Introducing initial assignments

This law introduces an assignment statement *before* a given specification.

> If w is disjoint from x, then for any expression E
>
> $$\begin{array}{l} w, x{:}\ [pre[w\backslash E]\ ,\ post[w_0\backslash E_0]] \\ \sqsubseteq\ w := E; \\ \quad w, x{:}\ [pre\ ,\ post] \end{array}$$

6.4 Introducing final assignments

This law introduces an assignment statement *after* a given specification.

If w is disjoint from x, then for any expression E

$$\begin{array}{ll} & w, x\colon [pre\ ,\ post] \\ \sqsubseteq & w, x\colon [pre\ ,\ post[w \backslash E]]\,; \\ & w := E \end{array}$$

6.5 Removing invariants

This law allows an invariant to be *removed* from a specification whenever it contains no changing variables.

If w does not occur free in inv, then

$$w\colon [pre\ ,\ inv\ ,\ post] \sqsubseteq w\colon [pre\ ,\ post]$$

7 Semantics of the extensions

7.1 Local variables

For variable x, program P, and postcondition R not containing x free,

$$[\![[\,\mathbf{var}\, x \mid I \bullet P\,]]\!] R \mathrel{\widehat{=}} (\forall x \bullet I \Rightarrow [\![P]\!] R)$$

7.2 Logical constants

For variable x, program P, and postcondition R not containing x free,

$$[\![[\,\mathbf{con}\, x \bullet P\,]]\!] R \mathrel{\widehat{=}} (\exists x \bullet [\![P]\!] R)$$

7.3 Specifications

For variable w, predicates pre and $post$, and postcondition R,

$$[\![w\colon [pre\ ,\ post]]\!]R \mathrel{\widehat{=}} pre \wedge (\forall\, w \bullet post \Rightarrow R)$$

References

[1] J.-R. Abrial. Generalised substitutions. 26 Rue des Plantes, Paris 75014, France.

[2] J.-R. Abrial, P.H.B. Gardiner, C.C. Morgan, and J.M. Spivey. A formal approach to large software construction. 1988.

[3] R.-J. Back. On the correctness of refinement steps in program development. Report A-1978-4, Department of Computer Science, University of Helsinki, 1978.

[4] R.-J. Back. Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[5] R.-J. Back. A calculus of refinemnet for program derivations. Report Ser.A 54, Departments of Information Processing and Mathematics, Swedish University of Åbo, Åbo, Finland, 1987.

[6] R.-J. Back. Procedural abstraction in the refinement calculus. Report Ser.A 55, Departments of Information Processing and Mathematics, Swedish University of Åbo, Åbo, Finland, 1987.

[7] R. Backhouse. *Program construction and verification.* Prentice-Hall, 1986.

[8] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[9] F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, and B. Krieg-Brückner. A wide spectrum language for program development. In *3rd Int. Symp. Programming, Paris*, pages 1–15, 1978.

[10] R.S. Bird. An introduction to the theory of lists. Technical monograph PRG–56, Programming Research Group, 8–11 Keble Road, Oxford OX1 3QD, U.K., October 1986.

[11] H. Boom. A weaker precondition for loops. *Trans. Prog. Lang. Sys.*, 4:668–677, 1982.

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

[13] P.H.B. Gardiner and C.C. Morgan. Data refinement of predicate transformers. Submitted to *Theoretical Computer Science*. Reprinted in this collection.

[14] D. Gries. *The Science of Programming*. Springer, 1981.

[15] D. Gries and D. Levin. Assignment and procedure call proof rules. *Trans. Prog. Lang. Sys.*, 2(4), October 1980.

[16] D. Gries and J. Prins. A new notion of encapsulation. In *Symp. Language Issues in Programming Environments*. SIGPLAN, June 1985.

[17] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. Technical Report 5, Digital Systems Research Center, July 1985.

[18] I.J. Hayes. *Specification Case Studies*. Prentice-Hall, 1987.

[19] J.F. He, C.A.R. Hoare, and Sanders J.W. Data refinement refined. Programming Research Group, Oxford.

[20] E.C.R. Hehner. *The Logic of Programming*. Prentice-Hall, London, 1984.

[21] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 583, October 1969.

[22] C.A.R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Lecture Notes in Mathematics 188*. Springer, 1971.

[23] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[24] C.A.R. Hoare, J.F. He, , and J.W. Sanders. Prespecification in data refinement. *Inf. Proc. Lett.*, 25(2), May 1987.

[25] C.A.R. Hoare and J.F. He. The weakest prespecification. *Fundamenta Informaticae*, IX:51–84, 1986.

[26] C.B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1986.

[27] M.B. Josephs. Formal methods for stepwise refinement in the Z specification language. Programming Research Group, Oxford.

[28] M.B. Josephs. The data refinement calculator for Z specifications. *Inf. Proc. Lett.*, 27:29–33, 1988.

[29] P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory Vienna, 1968.

[30] L. Meertens. Abstracto 84: The next generation. In *Annual Conf.* ACM, 1979.

[31] C.C. Morgan. Auxiliary variables in data refinement. Accepted by *Inf. Proc. Lett.* Reprinted in this collection.

[32] C.C Morgan. Software engineering course notes. In draft.

[33] C.C. Morgan. Data refinement using miracles. *Inf. Proc. Lett.*, 26(5):243–246, January 1988. Reprinted in this collection.

[34] C.C Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11, 1988. Reprinted in this collection.

[35] C.C. Morgan. The specification statement. *Trans. Prog. Lang. Sys.*, 10(3), July 1988. Reprinted in this collection.

[36] C.C. Morgan and P.H.B. Gardiner. Data refinement by calculation. Submitted to *Acta Informatica*. Reprinted in this collection.

[37] C.C Morgan and K.A. Robinson. Specification statements and refinement. *IBM Jnl. Res. Dev.*, 31(5), September 1987. Reprinted in this collection.

[38] C.C Morgan and B.A Sufrin. Specification of the UNIX filing system. *IEEE Trans. Soft. Eng.*, SE–10(2), March 1984.

[39] J.M. Morris. Invariance theorems for recursive procedures. In draft.

[40] J.M. Morris. Laws of data refinement. Submitted to *Acta Informatica*.

[41] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):298–306, December 1987.

[42] P. Naur (Ed.). Revised report on the algorithmic language Algol 60. *Comm. ACM*, 6(1):1–17, January 1963.

[43] G. Nelson. A generalization of dijkstra's calculus. Technical Report 16, Digital Systems Research Center, April 1987.

[44] J.E. Nicholls and Sørensen I.H. Collaborative project in software development. IBM Hursley Park and Programming Research Group Oxford.

[45] T. Nipkow. Non-deterministic data types. *Acta Informatica*, 22:629–661, 1986.

[46] J.C. Reynolds. *The Craft of Programming*. Prentice-Hall, London, 1981.

[47] K.A. Robinson. From specifications to programs. Department of Computer Science, University of New South Wales, Australia.

[48] J.M. Spivey. *Understanding Z: a Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[49] N. Wirth. *Programming in Modula-2*. Springer, 1982.

Authors' addresses

Paul Gardiner and Carroll Morgan Programming Research Group, 8–11 Keble Road, Oxford OX1 3QD, U.K.

Ken Robinson Dept. of Computer Science, University of New South Wales, P.O. Box 1, Kensington 2033, Australia.

and

Ralph Back Dept. of Computer Science, Åbo Akademi, Lemminkäinengatan 14, SF–20500 Åbo, Finland.

Joe Morris Dept. of Computing, Glasgow University, Glasgow G12 8QQ, U.K.

OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP
8-11 Keble Road, Oxford OX1 3QD, England

Technical Monographs to September 23, 1988

PRG-2 *Outline of a Mathematical Theory of Computation*
by Dana Scott. November 1970, 24 p., £0.50

PRG-3 *The Lattice of Flow Diagrams*
by Dana Scott. November 1970, 57 p., £1.00

PRG-5 *Data Types as Lattices*
by Dana Scott. September 1976, 65 p., £2.00

PRG-6 *Toward a Mathematical Semantics for Computer Languages*
by Dana Scott and Christopher Strachey. August 1971, 43 p., £0.60

PRG-9 *The Text of OSPub*
by Christopher Strachey and Joseph Stoy. July 1972, 2v. 126, 151 p., £3.50

PRG-10 *The Varieties of Programming Language*
by Christopher Strachey. March 1973, 20 p., £0.50

PRG-17 *Report on the Programming Notation 3R*
by Andrew P. Black. August 1980, 58 p., £2.30

PRG-18 *The Specification of Abstract Mappings and their Implementation as B+ Trees*
by Elizabeth Fielding. September 1980, 74 p. + Appendix, £1.30

PRG-20 *Partial Correctness of Communicating Processes and Protocols*
by Zhou Chao Chen and C.A.R. Hoare. May 1981, 23 p., £1.75

PRG-22 *A Model for Communicating Sequential Processes*
by C.A.R. Hoare. June 1981, 26 p., £1.30

PRG-23 *A Calculus of Total Correctness for Communicating Processes*
by C.A.R. Hoare. April 1981, 31 p., £1.75

PRG-26 *The Consistency of the Calculus of Total Correctness for Communicating Sequential Processes*
by Zhou Chao Chen. February 1982, 38 p., £1.80

PRG-29 *Specifications, Programs and Implementations*
by C.A.R. Hoare. June 1982, 29 p., £1.75

PRG-32 *The Lispkit Manual*
by Peter Henderson, Geraint A. Jones and Simon B. Jones. 1983. 2v., 127, 136 p., £4.00 for both volumes

PRG-34 *Abstract Machine Support for Purely Functional Operating Systems*

by Simon B. Jones. August 1983, 33 p. + Appendix, £1.75

PRG–36 *The Formal Specification of a Conference Organising System*
by Tim Clement. August 1983, 52 p. + Appendix, £1.75

PRG–37 *Specification-Oriented Semantics for Communicating Processes*
by E.R. Olderog and C.A.R. Hoare. February 1984, 81 p., £1.50

PRG–38 *Making Nets Abstract and Structured* and *Nets and their Relation to CSP*
by Ludwik Czaja. January/June 1984, 23, 26 p., £1.30

PRG–39 *μFP - An Algebraic VLSI Design Language*
by Mary Sheeran. Ph.D. thesis November 1983, 139 p., £2.50

PRG–40 *The Design and Implementation of Programming Languages*
by John Hughes. Ph.D. thesis July 1983, 130 p. + Appendix, £2.50

PRG–42 *A Range of Operating Systems Written in a Purely Functional Style*
by Simon B. Jones. February 1985, 44 p., £1.30

PRG–44 *The Weakest Prespecification*
by C.A.R. Hoare and He Jifeng. June 1985, 60 p., £0.85

PRG–45 *Laws of Programming - A Tutorial Paper*
by C.A.R. Hoare, He Jifeng, I.J. Hayes, C.C. Morgan, J.W. Sanders, I.H. Sørensen, J.M. Spivey, B.A. Sufrin, A.W. Roscoe.
May 1985, 43 p., £2.35

PRG–46 *Specification Case Studies*
by Ian Hayes. July 1985, 68 p., £2.50

PRG–47 *Specifying the CICS Application Programmer's Interface*
by Ian Hayes. July 1985, 82 p., £3.10

PRG–48 *CAVIAR: A Case Study in Specification*
by Bill Flinn and Ib Holm Sørensen. July 1985, 46 p., £2.00

PRG–49 *Specification Directed Module Testing*
by Ian Hayes. July 1985, 30 p., £0.90

PRG–50 *The Distributed Computing Software Project*
by Roger Gimson and Carroll Morgan. July 1985, 85 p., £4.00

PRG–51 *JSD Expressed in CSP*
by K.T. Sridhar and C.A.R. Hoare. July 1985, 40 p., £1.45

PRG–52 *Algebraic Specification and Proof of Properties of Communicating Sequential Processes*
by C.A.R. Hoare and He Jifeng. November 1985, 72 p., £0.90

PRG–53 *The Laws of Occam Programming*
by A.W. Roscoe and C.A.R. Hoare. February 1986. 86 p., £2.50

PRG–54 *Exploiting Parallelism in the Graphics Pipeline*
by **Theoharis A. Theoharis.** June 1986, 101 p., £2.50

PRG–55 *Functional Programming with Side–Effects*
by **Mark B. Josephs.** Ph.D. thesis, June 1986, 101 p., £3.00

PRG–56 *An Introduction to the Theory of Lists*
by **Richard S. Bird.** October 1986, 28 p., £1.50

PRG–57 *The Pursuit of Deadlock Freedom*
by **A.W. Roscoe and Naiem Dathi.** November 1986, 38 p., £1.50

PRG–58 *Formal Methods Applied to a Floating Point Number System*
by **Geoff Barrett.** January 1987, 47 p., £1.60

PRG–59 Not yet allocated

PRG–60 *The Formal Specification of a Microprocessor Instruction Set*
by **Jonathan Bowen.** January 1987, 72 p., £2.00

PRG–61 *The Specification of Network Services*
by **Jonathan Bowen, Roger Gimson, Stig Topp-Jørgensen.** August 1987, 100 p., £2.60

PRG–62 *The Formal Documentation of a Block Storage System*
by **Roger Gimson.** August 1987, 112 p., £2.90

PRG–63 *Specifying System Implementations in Z*
by **Jonathan Bowen, Roger Gimson, Stig Topp-Jørgensen.** February 1988, 88 p., £2.60

PRG–64 *A Calculus of Functions for Program Derivation*
by **Richard S. Bird.** December 1987, 21 p., £1.00

PRG–65 *An Introduction to CSP*
by **J.W. Sanders.** 1988, 29 p., £1.20

PRG–66 *The Sliding-Window Protocol in CSP*
by **K. Paliwoda, J.W. Sanders.** 1988, 26 p., £1.20

PRG–67 *Two Papers on CSP*
by **A.W. Roscoe.** 1988, 80 p., £2.20

PRG–68 *Z: Grammar and Concrete and Abstract Syntaxes*
by **Steve King, Ib Holm Sørensen, Jim Woodcock.** 1988, 48 p., £1.60