South African
Computer
Journal
Number 13
April 1995

Suid-Afrikaanse
Rekenaar-
tydskrif
Nommer 13
April 1995

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the Computer Society of South Africa and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Rekenaarverenging van Suid-Afrika en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083
Email: dkourie@dos-lan.cs.up.ac.za

Subeditor: Information Systems
Prof Lucas Introna
Department of Informatics
University of Pretoria
Hatfield 0083
Email: lintrona@econ.up.ac.za

Production Editor
Dr Riël Smit
Mosaic Software (Pty) Ltd
P.O.Box 23906
Claremont 7735
Email: gds@mosaic.co.za

World-Wide Web: http://www.mosaic.co.za/sacj/

Subscriptions

	Annual	Single copy
Southern Africa:	R50,00	R25,00
Elsewhere:	$30,00	$15,00

An additional $15 per year is charged for airmail outside Southern Africa

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Phone: +27 (11) 315-1319 Fax: +27 (11) 315-2276

Introduction

WOFACS '94: The Second Workshop on Formal Aspects of Computer Science

Chris Brink
Laboratory for Formal Aspects and Complexity in Computer Science, Department of Mathematics, University of Cape Town,
cbrink@maths.uct.ac.za

Following the success of WOFACS '92 (for the *Proceedings* of which see SACJ 9 1993) another such event was held at the University of Cape Town, from 27 June to 8 July 1994.

These Workshops on Formal Aspects of Computer Science serve several purposes. First, they help to strengthen a culture of studying formal aspects and developing formal methods in Computer Science. Second, in doing so they provide an impetus towards collaboration and interdisciplinarity – in this case bringing together Logic, Mathematics and Computer Science. Third, they provide a vehicle for the inter-institutional training of postgraduate students. And fourth, they contribute to international collaboration by bringing a number of eminent scientists to South Africa.

WOFACS '94 was co-hosted by the Departments of Mathematics and Computer Science at the University of Cape Town, and was organised by the Laboratory for Formal Aspects and Complexity in Computer Science. It offered 2-week lecture courses as follows:

- Dr Falko Bause (Dortmund University, Germany), *Petri Nets*
- Prof Ed Brinksma (Technological University of Twente, Netherlands), *Formal Design of Distributed Systems*
- Prof Robert Goldblatt (Victoria University of Wellington), *Modal Logics of Programs*
- Prof Austin Melton (Michigan Technological University), *Denotational Semantics*
- Dr Carroll Morgan (Oxford University), *The Refinement Calculus*

About 80 participants attended WOFACS '94; of these roughly half were academic staff and half were graduate students from a number of Southern African Universities. Participation from neighbouring countries was very encouraging. The lecture courses were again made available to all attending students as credit-bearing courses at their home Universities (by prior arrangement with the Departments concerned). About 30 students were eventually evaluated and had their WOFACS results incorporated into their respective Honours programmes. Apart from travel and accommodation costs and a small administrative charge for registration WOFACS '94 was a free service to the community – no lecture fees of any kind were involved.

This volume of SACJ contains the *Proceedings* of WOFACS '94: papers relating to their lecture courses by Brinksma, Goldblatt, Melton and Morgan (and co-authors). Unfortunately the Bause contribution to WOFACS '94 was committed to a publisher before the event, and hence could not be included here.

For the success of WOFACS '94 I would like to express my grateful thanks to:

- our invited speakers, for their hard work and the quality of their presentations;
- the Foundation for Research Development, the participants of the Programme for Research Manpower in Computer Science, the UCT Research Committee and the Department of National Education, for direct and indirect sponsorship;
- my co-host, Professor Pieter Kritzinger;
- the indefatigable FACCS-Lab staff, particularly Diana Dixon, and
- all participants.

SACJ is produced with kind support from
Mosaic Software (Pty) Ltd.

The Refinement Calculus

Carroll Morgan
Programming Research Group , Wolfson Building, Parks Road, Oxford, OX1, 3QD, UK

Abstract

The refinement calculus, based on weakest preconditions, elaborates imperative program developments as a series of steps linked by a mathematical refinement relation. Specifications and executable code are regarded equally as programs, allowing a gradual transformation from one to the other.

The extra generality afforded by admitting specifications as code allows a uniform treatment of procedures, parameters, recursion, data refinement, typing and miracles.

Keywords: *refinement, weakest preconditions, program development, specifications, data refinement, miracles.*
Computing Review Categories: *F.3.1*

1 Summary of following sections

2. What is the refinement calculus? This section explains *specifications*, how they are inserted into an existing programming language, and the effect of doing so: the new prominence of *refinement* between programs.

3. The programming language. Adding specifications to Dijkstra's language of guarded commands, in particular, suggests futher extensions to it. This section summarises the entire programming language placed at our disposal: the 'core' language of guarded commands; the 'conventional' extensions such as variable declarations, types, procedures and recursion; and finally the more 'exotic' feaures directly related to specifications.

4. A small example. *Taking an integer square root by binary chop* is a familiar example of rigorous program derivation. In this section it is recast in the refinement calculus, aiming to illustrate the typical style of such developments.

6. Functions and relations This contains a brief summary of the functional and relational notation used in Section 7.

7. Specification and design State-based specification of *systems* (as opposed to merely programs) is a tradition inherited from *Z* and *VDM*. The example of this section specifies a *mail system* at a very high level, then developing it by a series of data refinements towards a more practical implementation including asynchronous transmission.

8. Summary of refinements This is an abridged collection of some of the more useful refinement steps.

2 What is the refinement calculus?

What has come to be known as *the refinement calculus* arose out of a simple extension to Dijkstra's language of guarded commands; the *specification*, here written

$$v\colon [\alpha\ ,\ \beta] \tag{1}$$

(and explained below) is a command, in the programming language like all others, that describes the intended effect of a computation. Unlike conventional programming commands, however, it does not necessarily suggest a mechanism for that computation: it gives the *what*, but not the *how*.

We explain (1) as follows:

v is a list (including empty) of variables that the specification can alter;

α is a precondition — a formula over the program variables — whose truth initially is sufficient to guarantee termination of the specification; and

β is the postcondition, also a formula, expressing what must hold when the specification terminates (if it does).

The above is written as if specifications are *executable*, and indeed many of them are. But we are not assuming that any specification can be given to a computer which will then execute it directly, although that is possible in certain limited cases. Rather we consider "executing a specification" to include the execution of any program code that satisfies it: thus one way of 'executing' a specification is to hire a programmer to develop the program for you, and then run the program. No matter how it is executed, the result must then have all the properties referred to above — which leads us to the concept of program *refinement*.

Thus the second essential ingredient of the refinement calculus is not a program construct itself; rather it is *about* programs. To write

$$aaa \sqsubseteq bbb \ ,$$

for two programs aaa and bbb, is to say that aaa is *refined* by bbb; and that, in turn, means informally that any client who has asked for the program aaa will be happy if he is given bbb.

Program development in the refinement calculus is usually carried out via a series of such refinement steps, starting from a specification aaa, say, and ending with a program ggg. In between might occur a number of 'hybrid' programs, containing both specifications and executable fragments:

$$aaa \sqsubseteq \cdots \sqsubseteq bbb \sqsubseteq ccc \sqsubseteq ddd \sqsubseteq eee \sqsubseteq fff \sqsubseteq \cdots \sqsubseteq ggg \ .$$

The overall desired $aaa \sqsubseteq ggg$ follows from the transitivity of the refinement relation $\sqsubseteq$.

As an example, some of the flavour of refinement may be gained by considering the following program, in which "$d \mid p$" means "d divides p exactly":

if $2 \mid x \rightarrow x := x \operatorname{div} 2$
[] $3 \mid x \rightarrow x := x \operatorname{div} 3$
fi .

If x initially is 2, then x finally will be 1; if initially 3, again finally 1; if initially 6, then finally (but unpredictably) either 2 or 3; and if initially 7, then "finally" is not meaningful, since the program may abort.

Now consider instead this 'C-like' program:

```
IF (x%2 == 0) x = x DIV 2; ELSE x = x DIV 3 .
```

In all the cases we've considered, the behaviour of the second program is better, from the client's point of view, than the behaviour of the first. Cases 'initially-2' and 'initially-3' are unchanged; in case 'initially-6' the nondeterminism is resolved to 'finally-3' (about which the client can have no legitimate complaint, since he was prepared to accept *either* 2 or 3); and in case 'initially-7', the result finally is 7 `DIV` 3, whatever that is (and again there is no ground for complaint, since the client was prepared for the program to abort).

Usually refinements begin with a program that is a single specification and end finally with a program that is directly executable by computer.

The connection between refinement-style and wp-style development is given by this theorem, in which v for the moment is taken to be *all* program variables:

$$v\colon [\alpha \ , \ \beta] \sqsubseteq aaa$$

has exactly the same meaning as $\Rrightarrow$

$$\alpha \Rrightarrow wp(aaa, \beta) \ .$$

In the following sections we find the mathematical definitions of specifications and the relation $\sqsubseteq$, an indication of the notations used generally, and in separate sections two extended examples. The first — square root — is simple and well-known (and much-published), and is included to allow the reader to consider the refinement calculus from a position of strength (in familiarity with the example). The second example concentrates on the larger-scale issue of specification and development of a system (rather than just a program).

Good starting points in the literature more generally are

- *Generalised substitutions* [1];
- *A Calculus of Refinements for Program Derivations* [2];
- *A Theoretical Basis for Stepwise Refinement and the Programming Calculus* [7];
- *The Specification Statement* [4];
- *Programming from Specifications* [5]; and
- *On the Refinement Calculus* [6].

3 The programming language

Our tour of the programming language is in three parts: first a reminder of what we inherit from Dijkstra's language of guarded commands; then an exposition of the syntax of every-day features, such as procedures, that were not treated explicitly by Dijkstra; and finally a look at the more exotic extensions that seem natural once we have accepted specifications as programs themselves.

Legacy

These constructions come directly from the guarded command language [3].

- **skip** is the command that terminates, but otherwise does nothing.
- **abort** is the command whose behaviour is wholly unpredictable: it might terminate; it might not; if it terminates it might give the right answer — but it might not.
- *Assignment* commands are written $v := E$, for some variable v and expression E. We take the (convenient) view that *all* assignments terminate.
- *Sequential composition* is written with a semicolon ";", so that the command *aaa*; *bbb* 'first does *aaa*, then does *bbb*'.
- *Alternation* is indicated by the brackets **if** and **fi**, enclosing a []-separated list of guarded commands. A *guarded* command is written $G \rightarrow aaa$, where G is a guard and *aaa* is a command. A *guard* is a (logical) formula.

  The effect of the alternation is to execute exactly one of the enclosed guarded commands whose guard is true, behaving as **abort** if there is none.
- *Iteration* is indicated by the brackets **do** and **od**, enclosing a []-separated list of guarded commands. The effect of the iteration is to execute exactly one of the enclosed guarded commands whose guard is true, and then to repeat that process. When no guard is true, then the iteration terminates.

Practicality

Conventional programming languages aften contain constructions such as those given below, extending the 'core' language described in Section 3. We look at them in no particular order.

Local variables

The construction

| [**var** $v \cdot aaa$] |

introduces a (new, local) variable v for use within the program *aaa*; the brackets | [and] | indicate its scope. Within scope it can be accessed as any other variable; outside its scope it is inaccessible.

The · separates such declarations from following commands.

Procedures

The construction

| [**procedure** *Proc* (**value** *a*; **result** *b*) $\widehat{=}$ *aaa* ·
bbb
] |

introduces a (new, local) procedure *Proc*, possibly with parameters (of which only an example is given above), for use within the program *bbb*; again the brackets indicate its scope. Within scope, using the 'command' *Proc* (x, y) is exactly equivalent to using instead the procedure body *aaa*, providing substitution of actual parameters (x and y) for formal parameters (a and b) is carried out; outside its scope it is inaccessible.

Modules

The construction

| [**module**
export list;
variable, procedure, module declarations;
initially *I*
end ·
aaa
] |

introduces a related set of declarations for use within the program *aaa*. Variables exported from the module may be accessed but not modified by *aaa*; procedures exported from the module may be called by *aaa*.

Variables and procedures declared within the module, but not listed in the export list, are accessible only by the procedures of the module itself.

The *initialisation* I is a formula constraining the values that the variables of the module are given initially.

All components of the module — procedures, internal modules and initialisation — may refer to variables outside the module in the (normal) way that a command at that point could (thus respecting the usual scope rules).

Heresy

Here finally we find the constructions characteristic of the refinement calculus specifically. Each of them is given a weakest-precondition semantics so that a smooth integration is achieved with the existing language.

Specification (statements)

The main and motivating extension is the *specification statement* (just "specification" for short). It is a triple comprising a list of variables called the frame (say v), a formula called the precondition (α) and a formula called the postcondition (β); all together they are written

$$v\colon [\alpha\ ,\ \beta]\ .$$

The *frame* lists the variables which the specification can change. Other variables, not in the frame, are accessible but not modifiable. Thus the specification above can modify v.

The *precondition* gives the condition under which the specification is guaranteed to terminate. If that condition is not met, then the specification is equivalent to **abort**. Thus the specification above is guaranteed to terminate only if executed in a state satisfying α.

The *postcondition* gives the effect that the specification will establish if it terminates. The specification above will establish β, if executed in an initial state satisfying α. Overall, therefore, the specification above

"if executed in a state satisfying α will, by changing only variables in v, terminate in a state satisfying β."

The precise meaning of specifications is given by the following definitions.

Definition 1 *Specification* For any postcondition ω,

$$wp(v\colon [\alpha\ ,\ \beta]\ ,\omega) \mathrel{\widehat{=}} \alpha \wedge (\forall v \cdot \beta \Rightarrow \omega)\ .$$

□

In some cases an extended form of specification is used in which β contains so-called *initial* variables: indicated by a subscripted 0, they refer to values held in the initial state. For those specifications we have

Definition 2 *Specification with initial variables* For any postcondition ω,

$$wp(v\colon [\alpha\ ,\ \beta]\ ,\omega) \mathrel{\widehat{=}} \alpha \wedge (\forall v \cdot \beta \Rightarrow \omega)\,[v_0\backslash v]\ ,$$

where 0-subscripted variables — *initial variables* — in the postcondition refer to their values held initially. The notation *something*$[old\backslash new]$ denotes syntactic replacement of all free occurrences of *old* in *something* by *new*, with appropriate renaming if necessary to avoid variable capture.

□

Thus for example the specification

$$i\colon [i > 0\ ,\ 0 \le i < i_0] \tag{2}$$

strictly decreases i, but not below 0, aborting if i is not positive to start with. We can easily verify that by a weakest-precondition calculation:

$$\begin{array}{ll} & wp(i\colon [i > 0\ ,\ 0 \le i < i_0]\ , i < C) \\ \equiv & i > 0 \wedge (\forall i \cdot 0 \le i < i_0 \Rightarrow i < C)\,[i_0\backslash i] \\ \equiv & \text{"see below}^1\text{"} \\ & i > 0 \wedge (i_0 \le 0 \vee i_0 \le C)[i_0\backslash i] \\ \equiv & i > 0 \wedge (i \le 0 \vee i \le C) \\ \equiv & 0 < i \le C\ . \end{array}$$

Thus for Specification (2) to establish $i < C$ finally, i must be positive but no more than C initially.

There are various abbreviations available for special cases of specifications:

- A precondition of true may be omitted, so that

  $v\colon [\beta]$ means $v\colon [\text{true}\ ,\ \beta]$.

  Thus the specification $v\colon [\beta]$ always terminates.
- If the frame is empty (no variables can change) and the postcondition is true, then both may be omitted, so that

  $\{\alpha\}$ means : $[\alpha\ ,\ \text{true}]$.

  Such commands are called *assumptions*, and they are directly executable (irrespective of the formula α used).

  The precondition α, in assumptions, is written between braces { and } for historical reasons (and to distinguish it from the following).
- If the frame is empty (no variables can change) and the precondition is true, then both may be omitted, so that

  $[\beta]$ means : $[\text{true}\ ,\ \beta]$.

[1] Really? Consider

$$\begin{array}{ll} & (\forall i \cdot 0 \le i < i_0 \Rightarrow i < C) \\ \equiv & (\forall i \cdot 0 > i \vee i \ge i_0 \vee i < C) \\ \equiv & (\forall i \cdot i < (0 \sqcup C) \vee i_0 \le i) \\ \equiv & (0 \sqcup C) \ge i_0 \\ \equiv & i_0 \le 0 \vee i_0 \le C\ . \end{array}$$

These are called *coercions*, and may alternatively be written $(\beta \rightarrow)$. One interpretation of coercions is that they force *backtracking* if their formula does not hold.

- Conjuncts common to both precondition and postcondition may be written once, in between. Thus
  v: $[\alpha\ ,\ \beta\ ,\ \gamma]$ means v: $[\alpha \wedge \beta\ ,\ \beta \wedge \gamma]$.

Logical constants

The dual of local variables (Section 3 above) is logical constants. They formalise the conventional use of upper-case names to denote 'extra-program' quantities: such names are declared as *logical constants*, using the keyword **con**; thus

$|\,[\,\mathbf{con}\ x, y \cdot$
$\quad a, b\colon [a = x \wedge b = y\ ,\ a = b = \gcd(x, y)]$
$\,]\,|$

is a program that sets (both) a and b to the gcd of their original values.

Parametrization via substitution

The definition of procedure call (above) depends partly on the notion of parametrization of procedures. Any program — procedure call or otherwise — can be the target of a substitution of which there are three kinds:

1. Substitution *by value* is written $aaa[\mathbf{value}\ x \backslash E]$, and causes x to be given the value E before aaa is executed.
2. Substitution *by result* is written $aaa[\mathbf{result}\ x \backslash y]$, and causes y to be given the value that x has after the execution of aaa.
3. Substitution *by value-result* is written $aaa[\mathbf{value\ result}\ x \backslash y]$, and is a combination of the above two.

Recursion

A *recursion block* is surrounded by the brackets **re** and **er**, and introduces a local name by which a recursive call may be indicated. The following, for example, illustrates the usual equivalence between recursion and iteration:

$\mathbf{do}\ G \rightarrow aaa\ \mathbf{od}$

is equivalent to

$\mathbf{re}\ X \cdot$
$\quad \mathbf{if}\ G\ \mathbf{then}\ aaa;\ X\ \mathbf{fi}$
$\mathbf{er}$,

where in general $\mathbf{if}\ G\ \mathbf{then}\ aaa\ \mathbf{fi}$ abbreviates

$\mathbf{if}\ G \rightarrow aaa\ [\!]\ \neg G \rightarrow \mathbf{skip}\ \mathbf{fi}$.

Types

Variables can be declared to be of any type (even empty) that can be built with the ordinary operators of set theory; in addition, types may be built by recursive definitions incorporating disjoint unions. Extended alternation and iteration constructions allow selection of the components of disjoint unions.

For example, taking

$\mathbf{type}\ tree \mathrel{\widehat{=}} \mathsf{nil} \mid \mathsf{node}\ A\ tree\ tree$

to define the type of finite binary trees with values of type A at the nodes, the following recursive procedure assigns to the sequence as the frontier of its argument tree at:

$\mathbf{var}\ as : \mathbf{seq}\ tree;$

$\mathbf{procedure}\ Frontier\ (\mathbf{value}\ at : tree)$
$\mathrel{\widehat{=}}\ \mathbf{if}\ at\ \mathbf{is}\ \mathsf{nil} \rightarrow as := \langle\rangle$
$\quad [\!]\ at\ \mathbf{is}\ \mathsf{node}\ a\ at'\ at'' \rightarrow$
$\qquad Frontier\ (at');$
$\qquad as := as +\!\!+ \langle a \rangle;$
$\qquad Frontier\ (at'')$
$\quad \mathbf{fi}$.

Local invariants

A *local invariant* is a formula that, applied to a program fragment, imposes that formula as a pre- and postcondition onto every command of that fragment; it is a generalisation of typing. Thus, for example, the following two programs are equivalent:

$|\,[\,\mathbf{var}\ x : \mathbb{N} \cdot aaa\,]\,|$

is equivalent to

$|\,[\,\mathbf{var}\ x : \int;\ \mathbf{and}\ x \geq 0 \cdot aaa\,]\,|$.

Exits and exceptions

An exception block $[\![\cdots]\!]$ acts to catch 'exits' caused by the execution of an **exit** command within the block. Thus

if $\neg G$ **then** *aaa* **fi**

is equivalent to

$[\![$ **if** G **then exit fi;**
aaa
$]\!]$.

Treatment of exits requires a specially-extended form of specification

v: $[\alpha\ ,\ \beta > \gamma]$

that, assuming α initially, either terminates normally establishing β or terminates 'exceptionally' establishing γ. In either case, only variables in v are affected.

The 'normal' specification v: $[\alpha\ ,\ \beta]$ is a special case of the extension:

v: $[\alpha\ ,\ \beta] = v$: $[\alpha\ ,\ \beta >$ **false**$]$.

The exit command itself is defined

exit $=:$ [**true** , **false** $>$ **true**] ,

showing that it changes no variables (empty frame), always terminates (precondition **true**), but cannot terminate normally (first postcondition **false**).

Refinement

The definition of refinement is deceptively simple:

Definition 3 *Refinement* For programs *aaa* and *bbb*, the refinement relation

$aaa \sqsubseteq bbb$,

pronounced "*aaa* is refined by *bbb*", holds iff for all formulae ω

$wp(aaa, \omega) \Rrightarrow wp(bbb, \omega)$.

□

In the informal terms used earlier, that definition may be justified as follows:

A client expecting aaa, and intending to use it to establish ω, will have to ensure initially that $wp(aaa, \omega)$ holds. The precondition $wp(bbb, \omega)$ actually required is however a weaker one, and thus the desired ω will be established in any case.

4 Square Root: a small example

In this section we follow a small but complete development from beginning to end.

Abstract program: the starting point

We are given a natural number s, and we must set the natural number r to the greatest integer not exceeding $\sqrt{s}$, where $\sqrt{}$ takes the non-negative square root of its argument. Thus starting from $s = 29$, for example, we would expect to finish with $s = 29 \wedge r = 5$.

Here is our abstract program:

var $r, s : \mathbb{N} \cdot$
$r := \lfloor \sqrt{s} \rfloor$. (i)

Although an assignment, the command (i) is not code, because in this case study we assume that neither $\sqrt{}$ nor $\lfloor\ \rfloor$ is code. Our aim in the development to follow will be to remove them from the program.

Remove 'exotic' operators

These first refinement steps remove the square-root and floor functions ('exotic' only because they are not code) from the program by drawing on their mathematical definitions. The steps are routine, and leave us with a specification from which $\sqrt{}$ and $\lfloor\ \rfloor$ have disappeared:

$=$ "*simple specification* 36"
r: $[r = \lfloor\sqrt{s}\rfloor]$
$=$ "definition $\lfloor\ \rfloor$"
r: $[r \leq \sqrt{s} < r+1]$
$=$ "definition $\surd$"
r: $[r^2 \leq s < (r+1)^2]$. (ii)

We have now moved from assignment to specification, and for two reasons: we need the freedom of a formula, rather than just an expression, to exploit the definitions of $\surd$ and $\lfloor\ \rfloor$; and a specification is easier to develop *from* than an assignment.

Look for an invariant

The postcondition in *iteration* 31 is of the form $inv \wedge \neg GG$, and so we should investigate rewriting our postcondition in (ii) that way. There are two immediate possibilities:

$$r^2 \leq s \wedge \neg(s \geq (r+1)^2)$$
$$\text{and}\quad s < (r+1)^2 \wedge \neg(r^2 > s)\ .$$

The first would lead to an iteration

$$\mathbf{do}\ s \geq (r+1)^2 \rightarrow \cdots\ \mathbf{od}\ ,$$

with invariant $r^2 \leq s$. (The assignment $r := 0$ could establish the invariant initially.) The second would lead to

$$\mathbf{do}\ r^2 > s \rightarrow \cdots\ \mathbf{od}\ ,$$

with invariant $s < (r+1)^2$ (whose initialisation is not quite so straightforward — but perhaps $r := s$ would do).

Either of those two approaches would succeed (and in the exercises you are invited to try them). But the resulting programs are not as efficient as the one we are about to develop. We rewrite the postondition as

$$r^2 \leq s < q^2 \wedge r+1 = q\ ,$$

taking advantage of a new variable q that will be introduced for that purpose. (We use "rewrite" here a bit loosely, since the two postconditions are definitely *not* equivalent. But the new one *implies* the original, as it should — remember *strengthen postcondition* 40.) That surprising step is nevertheless a fairly common one in practice: one replaces an expression by a variable, adding a conjunct to make them equal.

The refinement is the following:

(ii) $\sqsubseteq$ **var** $q : \mathbb{N}$ ·
q, r: $[r^2 \leq s < q^2 \wedge r+1 = q]$.

Now having separate bounds on s gives us more scope: initially, r and q could be far apart. Finally, we must establish $r+1 = q$, and that will be the source of our increased efficiency: we can move them in big steps.

The next few refinements are routine when introducing an iteration: declare an abbreviation (I for the invariant, just to avoid writing it out again and again), establish the invariant with an assignment (initialisation), and introduce an iteration whose body maintains it.

The abbreviation $I \mathrel{\widehat{=}} \cdots$ is written as a decoration of the refinement. Like other decorations there (**var**, **con**), it is available in the development from that point on.

$\sqsubseteq$ $I \mathrel{\widehat{=}} r^2 \leq s < q^2$ ·
q, r: $[I \wedge r+1 = q]$
$\sqsubseteq$ q, r: $[I]$; (iii)
q, r: $[I\ ,\ I \wedge r+1 = q]$ ◁
$\sqsubseteq$ "invariant I, variant $q - r$"
do $r+1 \neq q \rightarrow$
q, r: $[r+1 \neq q\ ,\ I\ ,\ q - r < q_0 - r_0]$ ◁
od .

Note that the invariant bounds the variant below — that is, we have $I \Rrightarrow 0 \leq q - r$ — and so we need not write the "$0 \leq \cdots$" explicitly in the postcondition. We leave the refinement of (iii) to Exercise 1.

Our next step is motivated by the variant: to decrease it, we must move r and q closer together. If we move one at a time, whichever it is will take a value strictly between r and q. So we introduce a local variable for that new value, and make this step:

$\sqsubseteq$ **var** $p : \mathbb{N}$ ·

p: $[r+1 < q\ ,\ r < p < q]$; (iv)
q, r: $[r < p < q\ ,\ I\ ,\ q - r < q_0 - r_0]$. ◁

Strictly speaking, there should be an I in the postcondition of (iv), since in our use of *sequential composition* 35 the formula *mid* is clearly $r < p < q \wedge I$. (It is necessarily the same as the precondition of the ◁-marked command, which includes I: recall *specification invariant* 39.) But in fact I is established by (iv) whether we write it there or not, since it was in the precondition of the iteration body and does not contain p (the only variable that (iv) can change). Thus informally we can see that (iv) cannot falsify I — but in fact we have appealed to this law:

Law 4 *remove invariant* Provided w does not occur in *inv*,

$$w\text{: } [pre\ ,\ inv\ ,\ post] \sqsubseteq w\text{: } [pre\ ,\ post]\ .$$

□

We now intend to re-establish $r^2 \leq s < q^2$ in the postcondition with an assignment: either $q := p$ or $r := p$. By calculating $(r^2 \leq s < q^2)[q \backslash p]$, we can see that the first requires a precondition $s < p^2$ (or at least as strong as that: recall *assignment* 20); similarly, the second requires $s \geq p^2$. That case analysis supplies the guards for our alternation:

$\sqsubseteq$ **if** $s < p^2 \rightarrow q$: $[s < p^2 \wedge p < q\ ,\ I\ ,\ q < q_0]$ (v)
[] $s \geq p^2 \rightarrow r$: $[s \geq p^2 \wedge r < p\ ,\ I\ ,\ r_0 < r]$ (vi)
fi

(v) $\sqsubseteq\ q := p$
(vi) $\sqsubseteq\ r := p$.

Note that the refinement markers (v) and (vi) refer to the bodies of the alternation branches, and do not include the guards.

The simplifications of the variant inequalities are possible because we have used *contract frame* 23 in each case. In (v) for example, removing r from the frame allows us to rewrite $q - r < q_0 - r_0$ as $q - r < q_0 - r$, thence just $q < q_0$.

Now only (iv) is left, and it has many refinements: the assignment $p := r + 1$ and $p := q - 1$ are two. But a faster decrease in the variant — hence our more efficient program — will result if we choose p midway between q and r:

(iv) $\sqsubseteq\ p := (q + r)$ div 2 .

There we have reached code.

Epilogue

We need not list the entire program, now or ever; and we need not document it. Proper commenting and laying out of the final code is important only when there is no history of the development of the program: then, the code is all we have. An analogy with present practice (where machine-processable developments are not retained) is that commenting of assembler code is necessary only when the high-level source code has been thrown away.

Code is not meant to be read: it is meant to be executed by computer. And we have developments, such as the one above. It is a sequence of steps, every one justified by a refinement law, whose validity is independent of the surrounding commentary. The initial, abstract, program is at the beginning, and the final executable code is easily (mechanically) recoverable, at the end. The structure of the program is revealed as well: logically related sections of code are identified simply by finding a common ancestor. Furthermore, the development allows the program to be modified safely.

The code of our example is collected in Fig. 1. Could we choose some other value of p on the commented line? The development, shown in Fig. 2, gives the answer: the commented command in the code can be replaced by $p := r + 1$ without affecting the program's correctness. The validity of the following refinement step is all that is needed, and the rest of the program can be completely ignored:

$$p\text{: } [r+1 < q\ ,\ r < p < q] \sqsubseteq p := r + 1\ .$$

No comment could ever have that credibility.

There are still good reasons for collecting code. One is that certain optimisations are not possible until logically separate fragments are found to be executed close together. That is like a peephole optimiser's removing redundant loads to registers from compiler-generated machine code: the opportunity is noticed only when the machine code is assembled together. And those activities have more in common, for both are carried out without any knowledge of the program's purpose. It is genuine post-processing.

For us, the documentation is the commentary accompanying the development history (including the quoted decorations on individual refinement steps). Because it plays no role in the correctness of the refinements, we are free to tailor it to specific needs. For teaching, it reveals the strategies used; for production programs, it might contain hints for later modification ("binary chop").

Exercises

Exercise 1 Refine (iii) to code.

Exercise 2 Why can we assume $r + 1 < q$ in the precondition of (iv)? Would $r < q$ have been good enough? Why?

```
|[ var q : ℕ ·
   q, r := s + 1, 0;
   do r + 1 ≠ q →
      |[ var p : ℕ ·
         p := (q + r) div 2;        ← binary chop
         if s < p² → q := p
         [] s ≥ p² → r := p
         fi
      ]|
   od
]|
```

Figure 1. Square root code

```
   var r, s : ℕ ·
   r := ⌊√s⌋
⊑ "simple specification 36"
   r: [r = ⌊√s⌋]
⊑ "definition ⌊ ⌋"
   r: [r ≤ √s < r + 1]
⊑ "definition √ ; r ∈ ℕ"
   r: [r² ≤ s < (r + 1)²]
⊑ var q : ℕ ·
   q, r: [r² ≤ s < q² ∧ r + 1 = q]
⊑ I ≙ r² ≤ s < q² ·
   q, r: [I ∧ r + 1 = q]
⊑ "sequential composition 35"
   q, r: [I] ;                                          (iii)
   q, r: [I , I ∧ r + 1 = q]                            ◁
⊑ "invariant I, variant q − r"
   do r + 1 ≠ q →
      q, r: [r + 1 ≠ q , α , q − r < q0 − r0]           ◁
   od
⊑ var p : ℕ ·
   p: [r + 1 < q , r < p < q]                           (iv)
   q, r: [r < p < q , I , q − r < q0 − r0]              ◁
⊑ if s < p² →
      q: [s < p² ∧ p < q , α·, q < q0]                  (v)
   [] s ≥ p² →
      r: [s ≥ p² ∧ r < p , I , r0 < r]                  ([])vi]
   fi
(v) ⊑ q := p
(vi) ⊑ r := p
(vi) ⊑ p := (q + r) div 2 .
```

Figure 2. Square root development

```
module Tag
  var u : set ℕ;

  procedure Acquire (result t : ℕ)
  ≙ t, u: [u ≠ ℕ , t ∉ u₀ ∧ u = u₀ ∪ {t}];

  procedure Return (value t : ℕ)
  ≙ u := u − {t};

  initially u = {}
end
```

Figure 3. Module declaration

```
module Tag'
  var n : ℕ;

  procedure Acquire (result t : ℕ)
  ≙ t, n := n, n + 1;

  procedure Return (value t : ℕ)
  ≙ skip
end
```

Figure 4. A refinement of Figure 3

Exercise 3 Justify the branches (v) and (vi) of the alternation: where does $p < q$ come from in the precondition of (v)? Why does the postcondition of (vi) contain an *increasing* variant?

Exercise 4 Return to (ii) and make instead the refinement

$$\sqsubseteq I \mathrel{\widehat{=}} r^2 \leq s \cdot$$
$$r\colon [I \wedge s < (r+1)^2] \ .$$

Refine that to code. Compare the efficiency of the result with the code of Figure 2.

5 Data refinement

Why bother?

Consider the modules *Tag* and *Tag'* of Figures 3 and 4. We can show *informally* that *Tag'* refines *Tag* as a whole module. For suppose the contrary, that *Tag* $\not\sqsubseteq$ *Tag'*: then a client expecting *Tag* would have to be disappointed by *Tag'*. That means in turn that there is some program whose behaviour would be detectably different if *Tag* were replaced by *Tag'* — otherwise, the client would have no grounds for complaint! But there is no such program: using *Tag'*, any series of *Acquire* and *Return* will produce successively higher values of t, starting from some randomly chosen value. And using *Tag*, exactly the same could have happened. (One might argue that it is unlikely: but still it is *possible*.) Thus *Tag* $\not\sqsubseteq$ *Tag'* is *not* true.

On the other hand, we can see easily that *Tag'* $\not\sqsubseteq$ *Tag* is indeed the case. For *Tag* can *Acquire* 1 then 0, and that is something *Tag'* could never do. Thus *Tag* $\sqsubseteq$ *Tag'* but *Tag'* $\not\sqsubseteq$ *Tag*, and so *Tag* $\sqsubset$ *Tag'*: a strict refinement. The former could be a definition module, and the latter one of its implementation modules.

In this section we investigate techniques for proving rigorously that one module is refined by another.

State transformation

State transformation, carried out on the interior of a module, results in refinement of its external behaviour. We consider two specific transformations: one adds variables to a module; the other removes variables from a module.

To add variables, a *coupling invariant* is chosen, relating the existing variables to the new ones; it can be any formula over the local variables of the module. Declarations of the new variables are added; the initialisation is strengthened by conjoining the coupling invariant; every guard may assume the coupling invariant; and every command is extended by modifications to the new variables that *maintain* the coupling invariant. The resulting module then refines the original.

To remove variables, they must be first made auxiliary by refining the procedures of the module individually. A set of variables is *auxiliary* if its elements occur only in assignments or specifications whose changing variables are in the same set, so that other variables cannot depend on them. Then the declarations and all occurrences of those variables are removed. Again, the resulting module refines the original.

Often the two steps are carried out in succession — augmentation to add variables, then diminution to remove them — though in special cases we can bundle them together in one step.

Adding variables: augmentation

Each of the following laws deals with an aspect of adding new variables. We assume throughout that the new variables are c, and that the coupling invariant is CI. Note that they are not *refinement* laws, and so do not contain the symbol $\sqsubseteq$. Rather they are transformations, for which we use the word "*becomes*".

In our examples below, we suppose the module already contains a variable p, and that the new variables are q and r. The coupling invariant is $p = q + r$, and all three variables have type $\mathbb{N}$.

Declarations

Declarations of the new variables are added to the module. For the example, we would add **var** $q, r : \mathbb{N}$.

Initialisation

The coupling invariant is conjoined to the initialisation.

Law 5 *augment initialisation* The initialisation I *becomes* $I \wedge CI$.

□

If the initialisation were $p = 1$, it would become $p = 1 \wedge p = q + r$.

Specifications

The coupling invariant is conjoined to both the pre- and postcondition of specifications, and the frame is extended to allow the new variables to change.

Law 6 *augment specification* The specification w: $[pre\ ,\ post]$ *becomes*

$$w, c: [pre\ ,\ CI\ ,\ post]\ .$$

□

For example, the command p: $[p > 0\ ,\ p < p_0]$ *becomes*

$$p, q, r: [p > 0\ ,\ p = q + r\ ,\ p < p_0]\ .$$

Guards

Each guard can be replaced by another whose equivalence to the first follows from the coupling invariant.

Law 7 *augment guard* The guard G may be replaced by G' provided that

$$CI \Rrightarrow (G \Leftrightarrow G')\ .$$

□

Note that $CI \wedge G$ is always a suitable G' above, as is $CI \Rightarrow G$. The guards $p > 0$ and $p < 0$ could become $p > 0 \wedge p = q + r$ and $p < 0 \wedge p = q + r$.

Removing auxiliary variables: diminution

Each of the following laws deals with an aspect of removing variables. In each one we assume that the auxiliary variables are a and that their type is A, and we continue with the example of the previous section. There is no coupling invariant for diminutions.

Declarations

The declarations of auxiliary variables are simply deleted. In our example, the declaration **var** $p : \mathbb{N}$ is removed.

Initialisation

Existential quantification removes auxiliary variables from the initialisation.

Law 8 *diminish initialisation* The initialisation I *becomes*

$$(\exists a : A \cdot I)\ .$$

```
module Calculator
  var b : bag ℝ;

  procedure Clear ≙ b := ≺≻;

  procedure Enter (value r : ℝ)
  ≙ b := b + ≺r≻;

  procedure Mean (result m : ℝ)
  ≙ {b ≠ ≺≻} m := ∑ b/#b
end
```

Figure 5. The mean module

□

The example initialisation, augmented in Section 5, *becomes* $q + r = 1$ when p is removed.

Specifications

The following law removes variables from a specification. In many practical cases, however, the law is not needed: often the variables can be removed by ordinary refinement. (See our more substantial example, in Section 5.)

Law 9 *diminish specification* The specification w, a: $[pre\ ,\ post]$ *becomes*

$$w\colon\ [(\exists a : A \cdot pre)\ ,\ (\forall a_0 : A \cdot pre_0 \Rightarrow (\exists a : A \cdot post))]\ ,$$

where pre_0 is $pre[w, a \backslash w_0, a_0]$. The frame beforehand *must* include a.

□

The example from Section 5 yields

$$q, r\colon\ [q + r > 0\ ,\ q_0 + r_0 > 0 \Rightarrow q + r < q_0 + r_0]\ .$$

And by strengthening the postcondition that refines to

$$q, r\colon\ [q + r > 0\ ,\ q + r < q_0 + r_0]\ .$$

Guards

Guards must be rewritten so that they contain no auxiliary variables. The law *alternation guards* 19 can be used for that, since it is applicable to the refinement of alternations generally. In the example, we get $q + r > 0$ and $q + r < 0$.

An example of data refinement

As our first serious[1] example, consider the module of Figure 5 for calculating the mean of a sample of real numbers. We write $\sum b$ and $\#b$ for the sum and size respectively of bag b.

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.

We transform the module, replacing the abstract bag b by a more concrete representation s, n, a pair of numbers. Throughout, we refer to b as the abstract variable, and to s, n as the concrete variables. First s and n are added, then b is removed.

Adding concrete variables

We shall represent the bag by its sum s and size n:

abstract variable:	$b : \mathbf{bag}\,\mathbb{R}$
concrete variables:	$s : \mathbb{R};\ n : \mathbb{N}$
coupling invariant:	$s = \sum b \wedge n = \#b$.

The first step is to add the declarations of new variables s, n and apply the augmentation techniques of Section 5 to the initialisation and the three procedures.

- For the initialisation, we have from *augment initialisation* 5

$$s = \sum b \wedge n = \#b\ .$$

[1] It is serious: the refinement we calculate here is exactly the one used in pocket calculators.

module *Calculator*
  var b : **bag** $\mathbb{R}$;
    $s : \mathbb{R}$; $n : \mathbb{N}$;

  procedure *Clear* $\widehat{=}$ $b, s, n := \prec\succ, 0, 0$;

  procedure *Enter* (**value** $r : \mathbb{R}$)
  $\widehat{=}$ $b, s, n := b + \prec r\succ, s + r, n + 1$;

  procedure *Mean* (**result** $m : \mathbb{R}$)
  $\widehat{=}$ m: $[n \neq 0\ ,\ m = s/n]$;

  initially $s = \sum b \wedge n = \#b$
end

Figure 6. After addition of concrete variables

- For *Clear*, we have from *augment assignment* 22

  $$b, s, n := \prec\succ, 0, 0\ .$$

- For *Enter*, we have from *augment assignment* 22

  $$b, s, n := b + \prec r\succ, s + r, n + 1\ .$$

- For *Mean* we have from *augment specification* 6 (after rewriting)

  $$m, s, n\text{: } [b \neq \prec\succ\ ,\ s = \textstyle\sum b \wedge n = \#b\ ,\ m = \sum b/\#b]\ ,$$

  and we can carry on, making these refinements immediately:

  $$\sqsubseteq\ m, s, n\text{: } [n \neq 0\ ,\ s = \textstyle\sum b \wedge n = \#b\ ,\ m = s/n]$$
  $$\sqsubseteq\ m\text{: } [n \neq 0\ ,\ s = \textstyle\sum b \wedge n = \#b\ ,\ m = s/n]$$
  $\sqsubseteq$ "*remove invariant* 4"
  $$m\text{: } [n \neq 0\ ,\ m = s/n]\ .$$

The result is shown in Figure 6.

Remember that augmentation (or diminution) is not in itself a refinement: the assignment $\{b \neq \prec\succ\}\ m := \sum b/\#b$ is *not* refined by $\{n \neq 0\}\ m := s/n$. The relation between them is augmentation (or diminution), relative to the abstract and conctrete variables and the coupling invariant. That is why we write *becomes* rather than $\sqsubseteq$.[2]

Removing abstract variables

The abstract variable is b, and its removal from Figure 6 is straightforward for the assignment commands, because it is auxiliary (*diminish assignment* 25). Its removal from *Mean* is unnecessary — it has been removed already! That leaves only the initialisation. We use *diminish initialisation* 8, giving

$$n = 0 \Rightarrow s = 0\ .$$

Now the abstract b has been removed altogether; the result is given in Figure 7.

Abstraction functions

The laws of Section 5 dealt with a very general case of data refinement, in which the coupling invariant linking the abstract and concrete states could be anything whatever. In particular, several abstract variables could be collapsed onto a single concrete representation, as shown in the example of Section 5:.

both $b = \prec 1, 2, 3\succ$
and $b = \prec 2, 2, 2\succ$ are represented by $s = 6 \wedge n = 3$.

That is actually a fairly rare occurrence in every-day program development however: it is much more common for the coupling invariant to be functional from concrete to abstract. (The above is not, but our earlier example $p = q + r$ is.)

[2]Compare change of variable in an integral: faced with $\int_a^b dx/\sqrt{1-x^2}$ we might consider the substitution $x = \sin\theta$ (which is the analogue of the coupling invariant). But it would be wrong to claim that $dx/\sqrt{1-x^2}$ and $(\cos\theta\ d\theta)/\cos\theta$ were *equal*, even though the two definite integrals as a whole are equal.

```
module Calculator
  var s : ℝ;  n : ℕ;

  procedure Clear ≙ s, n := 0, 0;

  procedure Enter (value r : ℝ)
  ≙ s, n := s + r, n + 1;

  procedure Mean (result m : ℝ)
  ≙ m: [n ≠ 0 , m = s/n] ;

  initially n = 0 ⇒ s = 0
end
```

Figure 7. The mean module, after transformation

An example is the representation of sets by sequences, in which many distinct sequences may represent a given set: the elements may appear in different orders, may be duplicated, or even both. But to each sequence there corresponds at most one set; that is the functional nature of the abstraction, and what distinguishes is from the calculator example of Section 5.

The general form for such coupling invariants, called *funcional abstractions* is

$$a = \mathsf{af}\ c \wedge \mathsf{dti}\ c\ , \tag{3}$$

where af is a function, called the *abstraction function* and dti is a predicate, in which a does not appear, called the *data-type invariant*. In the case of sets and sequences, for example, the abstraction function is set, the function that makes a set from a sequence. Various data-type invariants may be included as well, for example that the sequences are kept in order (in which case we would write $a = \mathsf{set}\ c \wedge \mathsf{up}\ c$), or that the sequences contain no duplicated elements.

The reason for our interest in the special cases of data refinement is that when the augmentation and diminution laws are specialised to coupling invariants of the form (3) they become very much simpler, and the augmentation and diminution may be done together in one step.

Data-refining initialisations

Suppose that here (and in the following subsections) the coupling invariant is in the form (3), whence we may speak of abstraction function af and a data-type invariant dti.

One merely replaces all occurrences of abstract variables a by their concrete counterparts af c, conjoining the data-type invariant dti c to the result. That law is

Law 10 *data-refine initialisation* Under abstraction function af and data-type invariant dti, the initialisation I *becomes*

$$I[a \backslash \mathsf{af}\ c] \wedge \mathsf{dti}\ c\ .$$

□

Data-refining specifications

Here the law is substantially simpler: it is again substition (abstraction function) and conjunction (data-type invariant):

Law 11 *data-refine specification* Under abstraction function af and data-type invariant dti, the specification w, a: $[pre\ ,\ post]$ *becomes*

$$w, c\colon\ [pre[a \backslash \mathsf{af}\ c]\ ,\ \mathsf{dti}\ c\ ,\ post[a_0, a \backslash \mathsf{af}\ c_0, \mathsf{af}\ c]]\ .$$

□

Data-refinement of guards

Law *augment guard* 7 allows us to replace G by $G[a \backslash \mathsf{af}\ c] \wedge \mathsf{dti}\ c$, where as in *augment guard* 7 there is some flexibility: the conjunct dti c is optional. Subsequent adjustments may be made by *alternation guards* 19 as before. We have

Law 12 *data-refine guard* Under abstraction function af and data-type invariant dti, the guard G may be replaced by $G[a \backslash \mathsf{af}\ c] \wedge \mathsf{dti}\ c$, or if desired simply by $G[a \backslash \mathsf{af}\ c]$ on its own.

□

Exercises

Exercise 5 *Log-time multiplication* The following program terminates in time proportional to $\log N$:

$l, m, n := 0, 1, N$;
do $n \neq 0 \rightarrow$
  if even $n \rightarrow m, n := 2 \times m, n \operatorname{div} 2$
  $[]$ odd $n \rightarrow l, n := l + m, n - 1$
  fi
od .

Propose an iteration invariant that could be used to show that the program refines

l, m, n: $[l = N]$,

given the declarations $l, m, n, N : \mathbb{N}$.

Augment the program by variables l' and m', coupled as follows:

$$\begin{aligned} l' &= M \times l \\ m' &= M \times m \,. \end{aligned}$$

What is the resulting program, and what value is then found in l' on its termination?

Now go on to diminish the program by removing all variables not needed for the calculation of l'; then rename variables to remove primes. What is the resulting program?

Exercise 6 *Log-time exponentiation* Augment the program of Exercise 5 by variables l' and m', coupled as follows:

$$\begin{aligned} l' &= M^l \\ m' &= M^m \,. \end{aligned}$$

What is the resulting program, and what value is then found in l' on its termination?

Diminish the program by removing all variables not needed for the calculation of l'; then rename variables to remove primes. What is the resulting program?

Exercise 7 *Log-time transitive closure* Let A be given, and define

$$\mathsf{tc}\, n \mathrel{\widehat{=}} \textstyle\sum_{0 \leq i < n} A^i \,.$$

Augment the program of Exercise 5 by variables l' and m', coupled as follows:

$$\begin{aligned} l' &= \mathsf{tc}\, l \\ m' &= \mathsf{tc}\, m \,. \end{aligned}$$

What is the resulting program, and what value is then found in l' on its termination?

You will need an identity that gives $\mathsf{tc}(a + b)$ in terms of $\mathsf{tc}\, a$, $\mathsf{tc}\, b$ and A^b; what is it? How does that identity help you to decide what the 'definition' of $\mathsf{tc}\, 0$ should be?

Your augmented program should not contain any occurrences of tc, but may contain expressions A^m.

Further augment the program — add another variable, suitably coupled — so that the exponentiation can be removed. Note that the coupling invariant may be assumed when simplifying expressions.

Now diminish the program so that, after suitable renaming, a program remains that calculates $\mathsf{tc}\, N$ in logarithmic time.

Exercise 8 Apply *diminish specification* 26 directly to

$$m, s, n\text{: } \left[b \neq \prec\succ \,,\; s = \textstyle\sum b \wedge n = \#b \,,\; m = \sum b / \#b\right] \,,$$

without first doing the refinements on p.76. Then simplify the result. Which is easier: this exercise, or p.76?

Exercise 9 Suppose the mean procedure were instead

procedure *Mean* (**result** $m : \mathbb{R}$)
$\widehat{=}$ **if** $b \neq \prec\succ \rightarrow m := \sum b / \#b$
  $[]$ $b = \prec\succ \rightarrow$ **error**
  fi ,

where **error** is some definite error indication unaffected by data refinement. What would the concrete procedure be?

Exercise 10 Show that in the *Tag* module of Figure 3, the body of *Return* can be replaced by **skip**. *Hint*: Remember that you cannot transform just part of a module. Use new variable v and coupling invariant $u \subseteq v \wedge v \in$ **finset** $\mathbb{N}$ to transform all of it. The appearance of changing only *Return* is then gained by renaming v back to u.

Exercise 11 Use the functional abstraction laws to do the data refinement of Section 5 in reverse: that is, show that the module of Figure 5 is a refinement of that in Figure 7. Does that mean that the modules are equal? How does equality differ from refinement?

Hint: Convert the assignments to specifications first.

6 Summary of functions and relations

Some of these notations are used in the example of Section 7 to follow; they are mainly simple set-theoretic constructions.

Functions

Partial functions, domain and range

The square root function $\surd$, taking real numbers to real numbers, can be said to have type

$$\mathbb{R} \pfun \mathbb{R} .$$

The left-hand $\mathbb{R}$, the *source*, is the set from which the arguments are drawn; the right-hand $\mathbb{R}$, the *target*, is the set within which the results lie. The direction of the arrow indicates which is which (*from* source *to* target), and the stroke on the arrow indicates that the function is *partial*: there are some elements of its domain for which it is undefined.

Our mathematical view of functions is that they are sets of pairs, with each pair containing one element from the domain and the corresponding element from the range. Thus these pairs are some of the elements of $\surd$:

$(0, 0)$
$(1, 1)$
$(1.21, 1.1)$
$(\pi^2, \pi) \cdots$

All of the pairs are elements of the Cartesian product of $\mathbb{R}$ with $\mathbb{R}$, written $\mathbb{R} \times \mathbb{R}$: in general the elements of the set $S \times T$ are pairs (s, t) with one element drawn from S and the other from T. Thus any function in $S \pfun T$ is a subset of $S \times T$.

Associated with functions, as a type, are certain operations. The *domain* of a function is that subset of its source on which it is defined: for $f : S \pfun T$,

$$\mathsf{dom}\, f \mathrel{\widehat=} \{s : S;\ t : T \mid (s, t) \in f \cdot s\} .$$

Since f is itself a set, we can write that more succinctly as

$$\{(s, t) : f \cdot s\}$$

if we allow tuples as bound variables (which therefore we do).

The *range* of a function is that subset of the target which it might actually produce (given the right arguments):

$$\mathsf{ran}\, f \mathrel{\widehat=} \{(s, t) : f \cdot t\} .$$

Thus $\mathsf{dom}(\surd) = \mathsf{ran}(\surd)$ = "the non-negative reals".

Note that $\mathsf{ran}(\surd)$ = "the non-negative reals" means in particular that every non-negative real number is the square root of something.

Total functions

For any function $f : S \pfun T$, we have $\mathsf{dom}\, f \subseteq S$ and $\mathsf{ran}\, f \subseteq T$. When equality holds in either case, we can be more specific: function f above is total when it is defined on all its source. In other words,

f is *total* means that $\mathsf{dom}\, f = S$.

If f can produce every element of its range, we say that it is onto (or surjective):

f is *onto*, or *surjective*, means that $\mathsf{ran}\, f = T$.

For total functions we have the special notation of 'uncrossed' arrow, so that declaring $f : S \fun T$ is the same as declaring $f : S \pfun T$ and stating additionally that f is total.

Totality of a function is relative to its declared souce: although partial over $\mathbb{R}$, the square root function is total over the non-negative reals. The same applies to whether the function is onto: thus square root is total and onto if declared to be from non-negative reals to non-negative reals.

Function application and overriding

Given $f : S \pfun T$ and some $s : \mathsf{dom}\, f$ (which implies that $s \in S$ as well), we apply the function f to its argument s by writing $f\ s$. The result is an element of $\mathsf{ran}\, f$, and of T (since $\mathsf{ran}\, f \subseteq T$). An alternative way of writing the application is $f[s]$.

('Ordinarily', such function application is written $f(s)$. We have chosen instead to reserve parentheses for grouping, and indicate application by simple juxtaposition. The $f[s]$ variant is suggested by analogy with sequences, since they are functions from their indices to their values.)

As an 'abuse' of notation (actually a convenience), we allow $f[ss]$, given ss : set S as a *set* of values, and by it we mean the set of results obtained by applying f to elementsof ss separately (and ignoring those that are undefined). Thus

$$f[ss] \mathrel{\widehat{=}} \{(s,t) : f \mid s \in ss \cdot t\} \ .$$

We can modify a function at one or more of its source values, so that

$$f[s := t]$$

is the function f *overridden* by $s := t$. Letting g be $f[s := t]$, we have

$$\begin{array}{rrcl} & g[s] & = & t \quad \text{(no matter what } f[s] \text{ is)}, \\ \text{and} & g[s'] & = & f[s'] \quad \text{for any } s' \neq s. \end{array}$$

If $s \neq s'$ and f is not defined at s', then neither is g.

Similarly,

$$\begin{array}{rcll} (f[ss := t])[s] & = & t & \text{if } s \in ss \\ & = & f[s] & \text{if } s \notin ss \ . \end{array}$$

More generally still, we can override f by another function g; the resulting function $f \oplus g$ behaves like g if it can (if g is defined at that argument), otherwise like f. Thus

$$\begin{array}{rcll} (f \oplus g)[s] & = & g[s] & \text{if } s \in \mathsf{dom}\, g \\ & = & f[s] & \text{otherwise.} \end{array}$$

If neither f nor g is defined at s, then $f \oplus g$ is undefined there also.

In terms of sets,

$$f \oplus g = \{(s,t) : f \cup g \mid s \in \mathsf{dom}\, g \Rightarrow (s,t) \in g\} \ .$$

That last formulation takes undefinedness automatically into account.

Our earlier notations for overriding can now be seen as special cases of the above, because $f[s := t]$ is just f overridden by the (singleton) function $\{(s,t)\}$, which takes s to t but is undefined everywhere else. In the $f[ss := t]$ case the overriding function is $\{s : ss \cdot (s,t)\}$, defined only on ss.

The overriding notations "$[s := \cdots$" apply to sequences also, as they are a special case of functions.

Restriction and corestriction

Finally we have operators for restricting functions to smaller domains and ranges. Given $f : S \nrightarrow T$, ss : set S and tt : set T, we define

$$\begin{array}{rcl} ss \lhd f & \mathrel{\widehat{=}} & \{(s,t) : f \mid s \in ss\} \\ ss \lhd\!\!\!\cdot f & \mathrel{\widehat{=}} & \{(s,t) : f \mid s \notin ss\} \\ f \rhd tt & \mathrel{\widehat{=}} & \{(s,t) : f \mid t \in tt\} \\ f \rhd\!\!\!\cdot tt & \mathrel{\widehat{=}} & \{(s,t) : f \mid t \notin tt\} \ . \end{array}$$

An immediate use for $\lhd$ is an even more compact definition of overriding:

$$f \oplus g = ((\mathsf{dom}\ g) \lhd f) \cup g \ .$$

Relations

Generalised functions

Relations are a generalisation of functions: for source S and target T the corresponding relational type is written

$$S \leftrightarrow T \ ,$$

and, like functions, relations are sets of pairs. In fact,

$$S \leftrightarrow T = \mathbf{set}(S \times T) \ ,$$

which means that *any* subset of $S \times T$ is a relation. In contrast, only some subsets of $S \times T$ are functions; just which subsets they are we shall see shortly.

The generalisation of relations beyond functions is that relations are 'multi-valued': whereas for function f and source value s there is at most one $f[s]$, for relation r there may be many related target values.

Compare for example the function pred of type $\mathbb{N} \nrightarrow \mathbb{N}$ (it subtracts 1 from positive natural numbers) with the relation "less than" $<$, of type $\mathbb{N} \leftrightarrow \mathbb{N}$. The two agree on source element 0 (because pred is undefined there, and no natural number is less than 0), and on source element 1 (because pred $1 = 0$ and the only natural number less than 1 is 0). But beyond 1 we find that $<$ is more generous:

source value s	pred s	less than s
2	1	0,1
3	2	0,1,2
4	3	0,1,2,3

The function returns just one value, whereas the relation relates s to many values: the predecessor of s is *one of* the values less than it.

As sets, we have

$$\begin{aligned} \mathsf{pred} &= \{(1,0),(2,1),(3,2),\cdots\} \\ (<) &= \{(1,0),(2,1),(2,0),(3,2),(3,1),(3,0),\cdots\}\ , \end{aligned}$$

and thus we see clearly the difference between a relation and a function: in this case it's just that pred $\subseteq (<)$.

Most of the operators and notations we have defined for functions work for relations as well, and we summarise them here: for $r : S \leftrightarrow T$,

$$\begin{aligned} \mathsf{dom}\ r &= \{(s,t) : r \cdot s\} \\ \mathsf{ran}\ r &= \{(s,t) : r \cdot t\} \\ r \text{ is total} &\ \text{iff}\ \ \mathsf{dom}\ r = S \\ r \text{ is onto} &\ \text{iff}\ \ \mathsf{ran}\ r = T \\ \text{if } ss : \mathsf{set}\, S, \text{ then } r[ss] &= \{(s,t) : r \mid s \in ss \cdot t\}\ . \end{aligned}$$

For overriding we have

$$r \oplus r' = ((\mathsf{dom}\ r') \lhd r) \cup r'\ ,$$

and for the more specific cases then

$$\begin{aligned} r[s := t] &= r \oplus \{(s,t)\} \\ r[ss := t] &= r \oplus \{s : ss \cdot (s,t)\}\ . \end{aligned}$$

Thus for example $r[s := t]$ replaces *all* associations from s with a *single* new association to t.

7 A mail system

Electronic mail systems have their main features in common but vary a lot in the detail. Thus our initial specification, given later in Figure 8, is more or less just the bare minimum that could be said to comprise electronic mail. But the immediately following two subsections nevertheless discuss deficiencies apparent even at that level of abstraction, and propose design changes to avoid them.

A first specification

The system will provide just three procedures for passing messages:

- *Send*: A message and set of intended recipients are supplied; a 'unique' identifier is returned for that transmission. The identifier is used to refer to the message while it remains in the system.
- *Receive*: The set of identifiers is returned for mail that has been received by a given user, but not yet read.
- *Read*: The actual message corresponding to a given identifier is presented to its recipient, and the message is removed from the system.

Generally the system works as follows. A user *me* sends a message *msg* to a group *tos* of other users using Procedure *Send*; its result parameter *id* provides a unique reference to that transmission, which could be used by the sender, for example, to enquire after the status of a message or even to cancel it. (See Exercises 12 and 13.)

Procedure *Receive* is called by potential recipients who wish to know whether mail has been sent to them. Its result is a set *ids* of transmission identifiers that refer to messages sent to them that they have not yet read.

Supplying an identifier to Procedure *Read* will return the message text associated with the identifier, and delete that message from the system.

To construct an abstract module consistent with the informal description above, we begin by choosing types. Let users come from a type *Usr*, messages from *Msg* and identifiers from *Id*. The state of the system could then be given by the two variables

$msgs : Id \nrightarrow Msg$ and $sent : Usr \leftrightarrow Id$,

in which the partial function *msgs* associates message identifiers with the corresponding message text, and the relation *sent* records the (identifiers of) messages sent but not yet read. By making *sent* a proper relation we allow many users (many possible recipients) to be associated with a single identifier: that is how we shall deal with 'broadcasts', in which the same message is sent to many different users.

For the procedure *Send* we have the following parameters: which user is sending the message; the message itself; to which users it is being sent; and (as a result parameter) the identifier that subsequently can be used to refer to it. Here is the procedure heading:

procedure *Send* (**value** $me : Usr$; $msg : Msg$; tos : **set** Usr;
result $id : Id$) .

Within the procedure we first have a new identifier selected, which then will be associated both with the message and with the set of recipients. We use simply i: $[i \notin \mathsf{dom}\ msgs]$ for the selection, on the understanding that *Id* must be an infinite set so that the supply of unused identifiers can never 'run out'. That does raise several problems — but its simplicity is so appealing that we shall do it nevertheless.

Once the identifier is selected it is a straightforward matter to construct the neccesary links to the recipients and the message text, and the body of the procedure is as a whole

id: $[id \notin \mathsf{dom}\ msgs]$;
$msgs[id] := msg$;
$sent := sent \cup (tos \times \{id\})$.

Note that in the assignment $msgs[id] := msg$ the id need not be in the domain of *msgs*; one effect of the assignment is to put it there.

For the procedure *Receive* we need only return for user me the set of identifiers of messages waiting in *sent* to be read, and for that we use $ids := sent[me]$. (The application $sent[me]$ of a relation to an element we take as an abbreviation for $\{i : Id \mid (me, i) \in sent\}$.)

Finally, procedure *Read* must retrieve a message, given its identifier, and here we deal with some tricky questions. Suppose user me supplies identifier id legitimately — that is, that (me, id) is an element of *sent*, meaning that me is one of its intended recipients: then the message to be returned is found in *msgs*, and $msg := msgs[id]$ will retrieve it. But if the identifier id is not legitimate for me, what then? Making 'legitimacy' a precondition of the procedure (we need only include the assumption $\{(me, id) \in sent\}$ as its first command) would relieve the implementor of the obligation to deal with such matters: the procedure would simply abort if the request were not legitimate.

A more forgiving design would insist on termination in any case (not aborting, therefore); but it would allow any message whatsoever to be returned for illegitimate requests. That ranges from the helpful "Identifier does not refer to a message that has been received." through the cryptic "`MSGERR BAD ID`" finally to the mischievous option of returning likely-looking but wholly-invented messages that were never sent. (That last could be useful if one user is suspected of trying to read messages intended for others.) Thus we use the specification

msg: $\left[(me, id) \in sent \Rightarrow msg = msgs[id]\right]$,

leaving as the last detail the removal of the message from the system. The command $sent := sent - \{(me, id)\}$ does that, with the set subtraction as usual having no effect on *sent* if the pair $\{(me, id)\}$ is not there.

Straightforward initialisation to "the system is empty" gives us finally the module of Figure 8.

Reuse of identifiers

We now consider the possible implementation problems caused by our use of id: $[id \notin \mathsf{dom}\ msgs]$ in *Send*: that an unending supply of identifiers is required. Looking at *Read* in Figure 8, however, we can see that once a message has been read by all of its intended recipients, the antecedent $(me, id) \in sent \Rightarrow \cdots$ of the first command will never again be true, and so subsequent calls for the same identifier need not refer to *msgs* — the message texts can simply be invented. Thus, provided that at the end of *Read* we have $id \notin \mathsf{ran}\ sent$ (because the last (me, id) pair has just been removed), we can remove (id, msg) from *msgs*. That is done with the $\lhd$-marked command in the revised *Read* shown in Figure 9, which removes all such pairs at once.

But on what basis have we been saying "should" and "can"? Are we changing the specification, or are we merely refining the original module of Figure 8? How do we find out?

The effect of the change is to make the state component *msgs* a smaller function than before, taking care however never to delete identifiers still in $\mathsf{ran}\ sent$. We are therefore led to consider a data refinement in which the abstract variable is *msgs*, the concrete is *msgs'*, say, and the coupling invariant is

$$msgs' = (\mathsf{ran}\ sent) \lhd msgs \ . \qquad (4)$$

module *MailSys*
  var $msgs : Id \pfun Msg$;
    $sent : Usr \rel Id$ ·

  procedure *Send* (**value** $me : Usr$; $msg : Msg$; $tos :$ **set** Usr;
                result $id : Id$)
  $\widehat{=}$ id: $[id \notin \mathsf{dom}\ msgs]$;
    $msgs[id] := msg$;
    $sent := sent \cup (tos \times \{id\})$;

  procedure *Receive* (**value** $me : Usr$; **result** $ids :$ **set** Id)
  $\widehat{=}$ $ids := sent[me]$;

  procedure *Read* (**value** $me : Usr$; $id : Id$; **result** $msg : Msg$)
  $\widehat{=}$ msg: $[(me, id) \in sent \Rightarrow msg = msgs[id]]$;
    $sent := sent - \{(me, id)\}$;

  initially $msgs = sent = \{\}$
end

Figure 8. Initial specification of mail system

  procedure *Read* (**value** $me : Usr$; $id : Id$; **result** $msg : Msg$)
  $\widehat{=}$ msg: $[(me, id) \in sent \Rightarrow msg = msgs[id]]$;
    $sent := sent - \{(me, id)\}$;
    $msgs := (\mathsf{ran}\ sent) \lhd msgs$ ◁

Figure 9. Attempted recovery of Id's

(In all of the data refinements of this chapter we shall use 'primed' names for concrete variables; when the result of the data refinement is presented (and thus the abstract variables are gone), we simply remove the primes. That will help prevent a proliferation of names.)

Things are straightforward until finally we reach the first command of *Send*. There, we have

id: $[id \notin \mathsf{dom}\ msgs]$

becomes "*augment specification* 6"

$$\left[\frac{id, msgs'}{\dfrac{msgs' = (\mathsf{ran}\ sent) \lhd msgs}{\dfrac{msgs' = (\mathsf{ran}\ sent) \lhd msgs}{id \notin \mathsf{dom}\ msgs}}}\right]$$

= "$msgs$, $sent$ not in frame, thus $msgs'$ cannot change"

id: $[msgs' = (\mathsf{ran}\ sent) \lhd msgs\ ,\ id \notin \mathsf{dom}\ msgs]$

becomes "*diminish specification* 26"

$$\left[\frac{id}{\dfrac{(\exists\, msgs \cdot msgs' = (\mathsf{ran}\ sent) \lhd msgs)}{(\forall\, msgs_0 \cdot msgs' = (\mathsf{ran}\ sent) \lhd msgs_0 \Rightarrow id \notin \mathsf{dom}\ msgs_0)}}\right]$$

= "simplify precondition"

$$\left[\frac{id}{\dfrac{\mathsf{dom}\ msgs' \subseteq \mathsf{ran}\ sent}{(\forall\, msgs_0 \cdot msgs' = (\mathsf{ran}\ sent) \lhd msgs_0 \Rightarrow id \notin \mathsf{dom}\ msgs_0)}}\right]$$

= "simplify postcondition[3]"

id: $[\mathsf{dom}\ msgs' \subseteq \mathsf{ran}\ sent\ ,\ id \in \mathsf{ran}\ sent - \mathsf{dom}\ msgs']$.

We have been reasoning with equality rather than refinement $\sqsubseteq$, because we want to be sure of finding a concrete command if there is one. (Using $\sqsubseteq$ we might accidentally introduce infeasible behaviour and thus miss a data refinement that would actually have been acceptable.)

[3] Some of these comments conceal quite a lot of non-trivial predicate calculation, in this case discussed below. Similarly, 'routine' steps in engineering design sometimes generate quite tough integrals to be calculated. But the *principles* remain simple.

```
module MailSys
  var msgs : Id ⇸ Msg;
      sent : Usr ↔ Id ·

  procedure Send (value me : Usr;  msg : Msg;  tos : set Usr;
                  result id : Id)
  ≙ id: [id ∉ ran sent];                                    ◁
    msgs[id] := msg;
    sent := sent ∪ (tos × {id});

  procedure Receive (value me : Usr;  result ids : set Id)
  ≙ ids := sent[me];

  procedure Read (value me : Usr;  id : Id;  result msg : Msg)
  ≙ msg: [(me, id) ∈ sent ⇒ msg = msgs[id]];
    sent := sent − {(me, id)};

  initially msgs = sent = {}
end
```

Figure 10. Reuse of identifiers

```
  procedure Receive (value me : Usr;  result ids : set Id)
  ≙ ids: [ids ⊆ sent[me]]                                   ◁
```

Figure 11. Attempt at specifying delayed receipt

In the last step, refinement $\sqsubseteq$ is not difficult to show. (The antecedent gives dom $msgs' =$ ran $sent \cap$ dom $msgs_0$, and thus that ran $sent -$ dom $msgs'$ and dom $msgs_0$ are disjoint.) But for the equality we need also the reverse refinement $\sqsupseteq$; for that we choose an arbitrary $m : Msg$ and define $large \mathrel{\widehat{=}} Id \times \{m\}$, whose domain is all of Id. Taking $msgs_0$ to be $msgs' \cup (\text{ran } sent) \lhd large$ satisfies the antecedent, and the consequent is then equivalent to $id \in \text{ran } sent - \text{dom } msgs'$.

But alas it has all in any case been for nothing, since our conclusion is *not* feasible: the precondition allows dom $msgs' =$ ran $sent$, making the postcondition identically false. Not having shown refinement with this particular coupling invariant does not of course mean that no other would work; but it does encourage us to look more closely at whether we *are* after all performing refinement.

In fact we are not proposing a refinement: the concrete module can return the same id from $Send$ on separate occasions, which is something the abstract module cannot do. (See Exercise 14.)

A second specification: reuse

We are forced to admit that reusing identifiers requires a *change* in the specification that is *not* a refinement of it. Having to accept therefore that we are still in the 'design stage', we consider a simpler change with the same effect: we leave $Read$ in its original state, changing $Send$ instead so that 'new' identifiers are chosen in fact simply outside the range of $sent$ (since it is precisely the identifiers in $sent$ that refer to messages not yet read by all recipients). The result is shown in Figure 10, with the altered command marked.

A slightly unhelpful aspect of this new specification, and of the earlier attempt, now comes to light: it is that the reuse of identifiers is enabled by $Read$ as soon as (me, id) is removed from $sent$. In an eventual implementation that would probably require communication, in some form, from the receiver back to the sender. Our first specification did not require that, since the generation of identifiers was managed locally in $Send$, an essentially self-contained activity.

A second unrealistic aspect of this specification is that messages arrive instantly at the destination: a $Receive$ no matter how quickly after a $Send$ will return the identifier of the newly-sent message, and this too is unlikely to be implementable in practice.

Thus we are led to consider a third version of our specification.

A third specification: delay

In order to allow delay between sending a message and receiving it, one might think of altering $Receive$ as shown in Figure 11: only some, not necessarily all, of the identifiers of sent messages are returned by $Receive$. The rest are 'in transit'.

But if the subset returned is chosen afresh on each occasion, then messages could be received only later to be 'unreceived'

```
module MailSys
  var msgs : Id ⇸ Msg;
      sent, recd : Usr ↔ Id ·

  procedure Send (value me : Usr; msg : Msg; tos : set Usr;
                  result id : Id)
  ≙ id: [id ∉ ran sent];
    msgs[id] := msg;
    sent := sent ∪ (tos × {id});

  procedure Receive (value me : Usr; result ids : set Id)
  ≙ recd: [recd0 ⊆ recd ⊆ sent];                    ◁
    ids := recd[me];

  procedure Read (value me : Usr; id : Id; result msg : Msg)
  ≙ msg: [(me, id) ∈ recd ⇒ msg = msgs[id]];
    sent, recd := sent − {(me, id)}, recd − {(me, id)};

  initially msgs = sent = recd = {}
end
```

Figure 12. Delayed receipt of messages

again. In order to specify that once a message is received it stays received, we must introduce an extra variable *recd* that records which messages have been received already. That would be necessary in any case to make *Read* sensitive to whether a message has been received or not.

Thus while *Send* will use *sent* as before, in *Read* we find the new variable *recd* instead. The transfer of messages between *sent* and *recd* occurs in *Receive*, as shown in Figure 12. Note that in *Read* both *sent* and *recd* must have (me, id) removed: if left in *recd* the message could be read again; if removed from *recd* but left in *sent* it could be received again; and if left in both its identifier would never be recovered.

We will not attempt to show that Figure 12 refines our earlier Figure 10: indeed it cannot, because with our new module the program fragment

$$Send\ (me, msg, \{you\}, id);\ \ Receive\ (you, ids)$$

can terminate with $id \notin ids$ (because id is still in transit), and in our earlier module that is not possible. Nevertheless we should investigate carefully what we have done: is delay the *only* change we have made?

For our investigation, we go back and alter (but do not necessarily *refine*) our 'prompt' module of Figure 10 to express our minimum expectations of introducing delay. First, we must accept that *Receive* will not return *all* identifiers of messages sent, and so we use in this 'mock-up' the alternative procedure in Figure 11. Second, we split *Read* into two procedures: one for reading received messages, and the other for reading (or attempting to read) not-yet-received ones. The former should behave as *Read* does in Figure 10; the latter should return a randomly-chosen message, but change nothing else. The result is shown in Figure 13.

We should be quite clear about the role of Figure 13: it is not a refinement of Figure 10 (a customer having specified a prompt mail system will not accept an implementation containing delay); nor is it even a satisfactory specification of a system with delay (it is too weak). It is only the most we can say about delay while retaining the state of Figure 10.

Because we constructed our system with delay (Figure 12) essentially by guesswork, we are now double-checking against Figure 13 to see whether it has those 'reasonable' properties *at least*.

To compare Figure 12 with Figure 13, we must make the same distinction in Figure 12 between reading received messages and attempting to read not-yet-received ones. We can do that with a pair of coercions.

Recall that a coercion $[post]$ behaves like **skip** if *post* holds, and like **magic** otherwise: if *post* does not hold then $[post]$ is essentially 'unexecutable'. We make our procedures *ReadReceived* and *ReadNotReceived* from Figure 12 by exploiting that unexecutability. The body of *ReadReceived* will be as for *Read* but with an initial coercion expressing "the message has been received":

$$[(me, id) \in recd];\ msg: [(me, id) \in recd \Rightarrow msg = msgs[id]];\ sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\}\ .$$

Naturally, we can use the coercion to simplify the rest of the procedure; it becomes

```
module MailSys
  var msgs : Id ⇸ Msg;
      sent : Usr ↔ Id ·

  procedure Send (value me : Usr;  msg : Msg;  tos : set Usr;
                  result id : Id)
  ≙ id: [id ∉ ran sent];                                    ◁
    msgs[id] := msg;
    sent := sent ∪ (tos × {id});

  procedure Receive (value me : Usr;  result ids : set Id)
  ≙ ids: [ids ⊆ sent[me]];                                  ◁

  procedure ReadReceived (value me : Usr;  id : Id;
                          result msg : Msg)
  ≙ msg: [(me, id) ∈ sent ⇒ msg = msgs[id]];
    sent := sent − {(me, id)};

  procedure ReadNotReceived (value me : Usr;  id : Id;
                             result msg : Msg)
  ≙ choose msg;

  initially msgs = sent = {}
end
```

Figure 13. Delay 'mock-up' — compare Figure 10

$[(me, id) \in recd]$;
$msg := msgs[id]$;
$sent, recd := sent - \{(me, id)\}, recd - \{(me, id)\}$.

We simply note that the coercion $(me, id) \in recd$ simplifies the following postcondition to $msg = msgs[id]$.
The body of *ReadNotReceived* will have the opposite coercion added, and we are able to simplify it as well: it becomes

$[(me, id) \notin recd]$;
choose msg;
$sent := sent - \{(me, id)\}$.

The result of all of these changes is shown in Figure 14, and we now — finally — investigate whether Figure 13 is refined by Figure 14. We choose as coupling invariant $recd \subseteq sent$, with $recd$ being our concrete variable; we have no abstract variable.

Only *ReadNotReceived* causes difficulties:

choose msg

becomes

$\{recd \subseteq sent\}$;
$msg, recd := ?, ?$;
$[recd \subseteq sent]$.

And here we have a problem. Our target code, in *ReadNotReceived* of Figure 14, appears to alter $sent$; the above code doesn't. The best we can do with the above is to refine it to

$[(me, id) \notin sent]$;
choose msg;
$sent := sent - \{(me, id)\}$.

The coercion $[(me, id) \notin sent]$, however, is not the one we want. It is too strong, and we can do nothing about it: stronger coercions cannot be refined into weaker ones. Thus our actual behaviour differs from our desired behaviour precisely when those two coercions differ: when $(me, id) \notin recd$ (from Figure 14) but $(me, id) \in sent$ (negating the above).

Thus we have not been able to show that Figure 14 refines 13, and must conclude therefore that our introduction of delay, in Figure 12, may have brought with it some unexpected consequences. (See Exercise 15.)

The problem was essentially a coding trick that came back to haunt us: in the original specification of *Read* we allowed the command $sent := sent - \{(me, id)\}$ to be executed even when (me, id) is not an element of $sent$. Later that became, without our noticing it, "executing $sent := sent - \{(me, id)\}$ even when (me, id) is not an element of $recd$" — altogether different, quite dangerous, and hard to detect without some kind of formal analysis.

```
module MailSys
  var msgs : Id ↛ Msg;
      sent, recd : Usr ↔ Id ·

  procedure Send (value me : Usr;  msg : Msg;  tos : set Usr;
                  result id : Id)
  ≙ id: [id ∉ ran sent];
    msgs[id] := msg;
    sent := sent ∪ (tos × {id});

  procedure Receive (value me : Usr;  result ids : set Id)
  ≙ recd: [recd0 ⊆ recd ⊆ sent];                                ◁
    ids := recd[me];

  procedure ReadReceived (value me : Usr;  id : Id;
                          result msg : Msg)
  ≙ [(me, id) ∈ recd];
    msg := msgs[id];
    sent, recd := sent − {(me, id)}, recd − {(me, id)};

  procedure ReadNotReceived (value me : Usr;  id : Id;
                             result msg : Msg)
  ≙ [(me, id) ∉ recd];
    choose msg;
    sent := sent − {(me, id)};

  initially msgs = sent = recd = {}
end
```

Figure 14. Delayed receipt of messages— instrumented

```
module MailSys
  var msgs : Id ⇸ Msg;
      sent, recd : Usr ↔ Id ·

  procedure Send (value me : Usr;  msg : Msg;  tos : set Usr;
                  result id : Id)
  ≙ id: [id ∉ ran sent];
    msgs[id] := msg;
    sent := sent ∪ (tos × {id});

  procedure Receive (value me : Usr;  result ids : set Id)
  ≙ recd: [recd0 ⊆ recd ⊆ sent];
    ids := recd[me];

  procedure Read (value me : Usr;  id : Id;  result msg : Msg)
  ≙ if (me, id) ∈ recd →
        msg := msgs[id];
        sent, recd := sent − {(me, id)}, recd − {(me, id)}
    [] (me, id) ∉ recd → choose msg
    fi;

  initially msgs = sent = recd = {}
end
```

Figure 15. The 'final' specification

```
  procedure Deliver ≙ recd: [recd0 ⊆ recd ⊆ sent]

  procedure Receive (value me : Usr;  result ids : set Id)
  ≙ ids := recd[me]
```

Figure 16. Asynchronous delivery, modifying Figure 15

We remedy matters by using more-straightforward coding in *Read*, as shown in Figure 15. If we now performed the above analysis, the concrete *ReadNotReceived* procedure would simply be

```
[(me, id) ∉ recd];
choose msg ,
```

which we could reach without difficulty by direct refinement from Figure 13.

A first development: asynchronous delivery

With Figure 15 we have — finally — a specification that describes a reasonably realistic system in which messages may take some time to be delivered. We take it as our 'final' specification. (Why the quotes? Because very few specification are never changed, final or not.)

Our first move towards implementation will be concerned with the 'delay' built in to Procedure *Receive* of Figure 15. That describes the user's-eye view of it, but of course the delay doesn't happen necessarily in *Receive* itself; that is only where it is noticed.

Through the implementor's eyes instead, we would see messages moving towards their destination even while no user-accessible procedures are called. The subset relation in $recd \subseteq sent$ merely reflects the effect of calling *Receive* before they have arrived.

Our first development step is to introduce asynchronous message delivery, 'in the background'. Not having refinement rules for concurrency, however, we proceed informally: the actual delivery $recd$: $[recd_0 \subseteq recd \subseteq sent]$ is relocated from Procedure *Receive* into a new procedure *Deliver* which, it is understood, is called 'by the operating system' to move messages about. (Calls of *Deliver* are not even seen by the users.) The result is shown in Figure 16; note that *Deliver* needs no parameters.

With such a modest excursion into concurrency as this new module represents, we need only require that in an actual implementation there never be destructive interference between apparently concurrent calls on the procedures: a simple way of doing that is to introduce mutual exclusion so that at any time at most one procedure is active within it. In fact we would need to do that in any case — even without asynchronous delivery — if we were to share the mail system module

between concurrently executing users.

Although we admit we are not strictly-speaking implementing a refinement, we still should strive for as much confidence as possible in justifying the new behaviour, having learned from our earlier mistakes above. The key change, the new procedure, is not visible to users at all; it is called, 'in between' users' access to the module, by the operating system. Can we isolate that change, bringing the rest within the reach of our rigorous techniques?

If we were to add an 'asynchronous' procedure *Deliver* to our specification, Figure 15, we would only have to make its body **skip** to be sure that the change would not affect the users' perception of the module's behaviour. (We would have to ignore however the possibility that *Deliver* could be called 'so often' that users' access to *MailSys* is forever delayed, just as we have already ignored such starvation of one user by another.) Thus to justify the step we have just taken, we go back and add

$$\textbf{procedure}\ \mathit{Deliver} \mathrel{\widehat{=}} \textbf{skip}$$

to Figure 15, and attempt to show that Figure 16 is a refinement of that. To find a coupling invariant, imagine executions of the abstract and concrete *Deliver* together: we can see that even if *recd* and *recd'* were equal beforehand, afterwards the concrete *recd'* could have grown. Thus we choose *recd* as abstract variable, introduce concrete variable *recd'*, and couple the two with

$$\mathit{recd} \subseteq \mathit{recd}' \subseteq \mathit{sent}\ .$$

We look at the procedures in turn, taking a slightly informal view where matters seem clear enough.

As earlier, we can look at *Send* informally: since *sent* is only increased, the coupling invariant cannot be broken. For *Deliver* we must introduce a statement allowing *recd'* to grow, and we reason as follows:

$$\textbf{skip}$$

becomes "augment specification 6"

$$\mathit{recd}'\colon \left[\mathit{recd} \subseteq \mathit{recd}' \subseteq \mathit{sent}\ ,\ \mathit{recd} \subseteq \mathit{recd}' \subseteq \mathit{sent}\right]$$

$$\sqsubseteq\ \mathit{recd}'\colon \left[\mathit{recd}'_0 \subseteq \mathit{recd}' \subseteq \mathit{sent}\right]\ .$$

In *Receive* we must on the other hand *remove* the statement affecting *recd*. We have first

$$\mathit{recd}\colon [\mathit{recd}_0 \subseteq \mathit{recd} \subseteq \mathit{sent}]$$

becomes "augment specification 6"

$$\left[\begin{array}{c} \mathit{recd}, \mathit{recd}' \\ \hline \mathit{recd} \subseteq \mathit{recd}' \subseteq \mathit{sent} \\ \hline \mathit{recd} \subseteq \mathit{recd}' \subseteq \mathit{sent} \\ \mathit{recd}_0 \subseteq \mathit{recd} \subseteq \mathit{sent} \end{array}\right]$$

$$\sqsubseteq\ \textbf{skip}\ .$$

Note that it is in that last step that we need $\mathit{recd}' \subseteq \mathit{sent}$ in the coupling invariant.

In *Read*, however, we have a problem. The guards of the alternation make essential use of *recd*: we cannot replace them by guards involving *recd'* only, as *recd'* is essentially an arbitrary superset of *recd*. We are stuck.

The difference between that and our earlier problem, with *Receive*, is that the abstract *Receive* allowed a nondeterministic alteration of *recd*: deliveries could occur there. But deliveries cannot occur in our abstract *Read*. (See Exercise 16.)

Thus our Figure 16 is not a refinement of Figure 15 with its extra

$$\textbf{procedure}\ \mathit{Deliver} \mathrel{\widehat{=}} \textbf{skip}\ .$$

We are forced instead to add a third variable *deld* for "delivered", independent of *sent* and *recd*. Our proposed concrete module is shown in Figure 17.

To show refinement between the extended Figure 15 and Figure 17, our coupling invariant is $\mathit{read} \subseteq \mathit{deld}' \subseteq \mathit{sent}$; there are no abstract variables. Arguing informally, we can see that *Send* is successfully data-refined, since *sent* is only increased. Similarly in *Read*, variables *sent*, *deld'* and *recd* are decreased 'in step'. For Procedure *Deliver* we have

$$\textbf{skip}$$

becomes "augment specification 6"

$$\mathit{deld}'\colon \left[\mathit{recd} \subseteq \mathit{deld}' \subseteq \mathit{sent}\ ,\ \mathit{recd} \subseteq \mathit{deld}' \subseteq \mathit{sent}\right]$$

$$\sqsubseteq\ \mathit{deld}'\colon \left[\mathit{deld}'_0 \subseteq \mathit{deld}' \subseteq \mathit{sent}\right]\ .$$

Note that even though the last command appears miraculous, we can as in Section 7 introduce a module invariant, in this case $\mathit{deld}' \subseteq \mathit{sent}$ (direct from the coupling invariant in fact), that would allow us to write

$$\mathit{deld}'\colon \left[\mathit{deld}' \subseteq \mathit{sent}\ ,\ \mathit{deld}'_0 \subseteq \mathit{deld}' \subseteq \mathit{sent}\right]$$

if we wished to.

Finally, for *Receive* we have

```
module MailSys
  var msgs : Id ⇸ Msg;
      sent, deld, recd : Usr ↔ Id ·

  procedure Send (value me : Usr;  msg : Msg;  tos : set Usr;
                  result id : Id)
  ≙ id: [id ∉ ran sent];
    msgs[id] := msg;
    sent := sent ∪ (tos × {id});

  procedure Deliver
  ≙ deld: [deld₀ ⊆ deld ⊆ sent]

  procedure Receive (value me : Usr;  result ids : set Id)
  ≙ recd := deld;
    ids := recd[me];

  procedure Read (value me : Usr;  id : Id;  result msg : Msg)
  ≙ if (me, id) ∈ recd →
        msg := msgs[id];
        sent, deld, recd := sent − {(me, id)},
                            deld − {(me, id)},
                            recd − {(me, id)}
    [] (me, id) ∉ recd → choose msg
    fi;

  initially msgs = sent = deld = recd = {}
end
```

Figure 17. Asynchronous delivery — corrected

Persistent problems with electronic funds transfer led to chaos recently in the financial markets. The cause was traced to code in which the ordinary electronic mail system had been used to generate the unique identifiers needed for funds transfers.

Noticing that the specification of the mail system guaranteed never to repeat an identifier, a programmer had obtained them as needed by broadcasting null messages to no-one. (Such 'empty broadcasts' were, not surprisingly, particularly efficient and generated no network traffic.)

The mail system originally was implemented with such a large set of possible identifiers it was thought they would never run out. Use grew so rapidly, however, that recently the system was upgraded to recover old identifiers — yet it was not verified that the new system was a refinement of the original, and in fact it was not. Had the absence of refinement been noticed, the well-established principles of the Institute of Systems and Software Engineering would then have required a routine check to be made for dependencies on the original behaviour.

The institutions affected are suing for damages; meanwhile the financial community waits anxiously for other effects to come to light.

Figure 18.

$$recd\colon [recd_0 \subseteq recd \subseteq sent]$$

becomes augment specification 6

$$\left[\begin{array}{c} recd,\ deld' \\ \hline recd \subseteq deld' \subseteq sent \\ \hline recd \subseteq deld' \subseteq sent \\ recd_0 \subseteq recd \subseteq sent \end{array}\right]$$

$$\sqsubseteq\ recd := deld' .$$

and our data-refinement is proved. We therefore accept Figure 17 as our first development step, introducing asynchronous delivery of messages.

Exercises

Exercise 12 Modify the original specification of Figure 8 to include a procedure *Cancel* that can be used to remove all unreceived copies of a message from the system. Does the specification contain enough detail to prevent 'unauthorised' removal?

Exercise 13 Modify the original specification of Figure 8 to include a procedure *Unread* that can be used to determine which users have not yet read a given message.

Exercise 14 Explain precisely what is wrong with the specification in Figure 8, as amended in Figure 9. *Hint*: See Figure 18.

Exercise 15 Explain precisely what is wrong with the specification of Figure 12. *Hint*: See Figure 19.

Exercise 16 Explain precisely what is wrong with the specification of Figure 15. *Hint*: See Figure 20.

Exercise 17 Imagine a building with one lift serving several floors. Outside the lift door, on each floor, is a panel of buttons and lights with one button/light pair for each floor. Inside the lift are no buttons or lights at all.

To use the lift one presses the button, next to the doors, for the desired destination; the corresponding light should light if it's not lit already. When the doors open, one enters the lift in the hope that it will eventually visit that destination (whose light should be lit).

Design a module based on the type

$$Floor \mathrel{\widehat{=}} 0 \rightarrow F$$

The appellants withdrew today in the *CMSK* (Common Mail-System Kernel) case, after it was shown that the specification of the system indeed allowed accidental deletion of messages before they had been read. Unexplained message loss had been widely reported and documented in the user community, and in a joint action by users of *CMSK* it was claimed that since the specification guaranteed no loss, the manufacturer was liable for damages.

In a rare move in such cases, the manufacturer showed that its own specification did after all allow such undesirable behaviour, in particular when message identifiers were used for reading before they had been registered as received: a randomly-chosen message was in that case returned to the user, and the legitimate message was deleted from the system.

Users generally are now looking more closely at the published specification of *CMSK*, the future of which has been thrown into in doubt.

Figure 19.

A large-scale computer fraud was discovered today, involving an accounting loophole in the national electronic mail system.

It had been noticed that messages could be read before their delivery was reported. Believing their own specification, however, the mail authority had installed accounting software only at the actual point of reporting delivery. Thus 'unreported' messages could be read free of charge.

The loophole was exploited by a company that offered greatly-reduced rates on bulk electronic mail. Its customers' messages would be collected and sent all at once as a single very long message. The identifier returned would then be sent as the body of an immediately following very short message. Since the mail system tended to allow short messages to overtake long ones, the second message was often delivered before the first: the identifier it contained would then be used to read the first, bypassing report of delivery — and bypassing charging as well. The occasional failure of the second message to overtake the first was easily covered by the enormous profit made overall.

Figure 20.

module *TelephoneExchange*
var *xns* : ???
rqs : ??? ·

procedure *Request* (**value** *tt* : ???)
≙ "Request a conversation *tt*";

procedure *Connect* ≙ "described in the text"

procedure *Converse* (**value** *t* : *T*;
result *tt* : ???)
≙ "Identify *all* participants in any conversation involving *t*"

procedure *HangUp* (**value** *t* : *T*)
≙ "Withdraw *t* from any conversation in which it is involved"

initially "no conversations"
end

Note that

- A single telephone may be part of many requests (but of at most one conversation).
- *Connect* may be thought of as being executed at suitable moments by the exchange itself.
- *HangUp* should allow other participants in a conversation to continue.

Figure 21. Telephone module

that contains these procedures with the meanings informally indicated:

- *Press* (**value** *f*, *b* : *Floor*) — Press button *b* outside the lift doors on floor *f*. (Called by lift user.)
- *Check* (**value** *f*, *l* : *Floor*; **result** *b* : *Boolean*) — Check whether the light *l* on floor *f* is lit. (Called by lift user.)
- *Visit* (**result** *f* : *Floor*) — Close the doors, select a floor *f* 'randomly' which it would be useful to visit, go there, and open the doors. (Called by lift operator.)

Hint: There are probably unanswered questions about the informal specification above; answer them yourself. Consider using set-valued variables inside the module.

Exercise 18 Let *T* be a set of *telephones* connected to an exchange that supports conference calls, so that collections of (people using) telephones can hold group conversations.

Declare a variable *xns* of appropriate type that could represent the set of conversations in progress at any moment; write then, in English *and* in mathematics, an invariant that ensures there is no telephone in more than one conversation.

Now suppose *rqs* is to represent the set of conversations requested but not in progress (thus 'pending'). Specify and justify an operation (with the default precondition, true)

xns, *rqs*: [???]

that connects as many new conversations as is possible without disturbing existing conversations. Note that the invariant over *xns* must be respected. *Hint*: The set *xns* should be made locally maximal in some sense.

Then use the structures above to supply (abstract) program text for the informally-described module in Figure 21. (You need not fill in *Connect*, already specified in the text above.)

Finally, give a sensible definition of a new procedure *Chat* (**value** *t* : *T*) that causes *t* immediately to join a single 'chat line', able then to converse with all others that have not executed *HangUp* since they last executed *Chat*. Modify your other definitions if necessary (but the less, the better).

8 A collection of refinement laws

This list contains some of the more useful refinement laws: but it is *not* meant to suggest that program development is just a 'routine' matter of applying these (or any other) packaged developments.

A large dose of ingenuity continues to be required.

Law 13 *absorb assumption* An assumption before a specification can be absorbed directly into its precondition.

$\{pre'\}\ w{:}\ [pre\ ,\ post] = w{:}\ [pre' \wedge pre\ ,\ post]$.

□

Law 14 *absorb coercion* A coercion following a specification can be absorbed into its postcondition.

$$w{:}\ [pre\ ,\ post];\ [post'] = w{:}\ [pre\ ,\ post \wedge post']\ .$$

□

Law 15 *advance assumption*

$$w := E\ \{pre\} = \{pre[w \backslash E]\}\ w := E\ .$$

□

Law 16 *advance coercion*

$$w := E\ [post] = [post[w \backslash E]]\ w := E\ .$$

□

Law 17 *alternation* If $pre \Rrightarrow GG$, then

$$w{:}\ [pre\ ,\ post] \sqsubseteq \textbf{if}\ ([]\ i \cdot G_i \rightarrow w{:}\ [G_i \wedge pre\ ,\ post])\ \textbf{fi}\ .$$

□

Law 18 *alternation*

$$\{(\bigvee i \cdot G_i)\}\ prog = \textbf{if}\ ([]\ i \cdot G_i \rightarrow \{G_i\}\ prog)\ \textbf{fi}\ .$$

□

Law 19 *alternation guards* Let GG mean $G_0 \vee \cdots \vee G_n$, and HH similarly. Then provided

1. $GG \Rrightarrow HH$, and
2. $GG \Rrightarrow (H_i \Rightarrow G_i)$ for each i separately,

this refinement is valid:

$$\textbf{if}\ ([]\ i \cdot G_i \rightarrow prog_i)\ \textbf{fi} \sqsubseteq \textbf{if}\ ([]\ i \cdot H_i \rightarrow prog_i)\ \textbf{fi}\ .$$

□

Law 20 *assignment* If $(w = w_0) \wedge pre \Rrightarrow post[w \backslash E]$, then

$$w,x{:}\ [pre\ ,\ post] \sqsubseteq w := E\ .$$

□

Law 21 *augment assignment* The assignment $w := E$ can be replaced by the fragment

$$\{CI\}\ w,c := E,?\ [CI]\ .$$

□

Law 22 *augment assignment* The assignment $w := E$ can be replaced by the assignment $w,c := E,F$ provided that

$$CI \Rrightarrow CI[w,c \backslash E,F]\ .$$

□

Law 23 *contract frame*

$$w,x{:}\ [pre\ ,\ post] \sqsubseteq w{:}\ [pre\ ,\ post[x_0 \backslash x]]\ .$$

□

Law 24 *data-refine assignment* Under abstraction function af and data-type invariant dti, the assignment $w,a := E,F$ can be replaced by the assignment $w,c := E,G$ provided that E and G contain no a, and that

$$\begin{aligned} & \mathsf{dti}\, c \Rrightarrow F[a \backslash \mathsf{af}\, c] = \mathsf{af}\, G \\ \text{and}\quad & \mathsf{dti}\, c \Rrightarrow \mathsf{dti}\, G\ . \end{aligned}$$

□

Law 25 *diminish assignment* If E contains no variables a, then the assignment $w, a := E, F$ can be replaced by the assignment $w := E$.

□

Law 26 *diminish specification* The specification w: $[pre\ ,\ post]$ *becomes*

$$w{:}\ \left[(\exists a : A \cdot pre)\ ,\ (\forall a : A \cdot pre_0 \Rightarrow post)\right]\ ,$$

where pre_0 is $pre[w \backslash w_0]$. The frame beforehand must not include a, and *post* must not contain a_0.

□

Law 27 *expand frame*

$$w{:}\ [pre\ ,\ post] = w, x{:}\ [pre\ ,\ post \wedge x = x_0]\ .$$

□

Definition 28 *feasibility* The specification w: $[pre\ ,\ post]$ is *feasible* in context *inv* iff

$$(w = w_0) \wedge pre \wedge inv \Rrightarrow (\exists w : T \cdot inv \wedge post)\ ,$$

where T is the type of w.

□

Law 29 *fix initial value* For any term E such that $pre \Rrightarrow E \in T$, and fresh name c,

$$\begin{array}{l} \quad w{:}\ [pre\ ,\ post] \\ \sqsubseteq\ \textbf{con}\ c : T \cdot \\ \quad w{:}\ [pre \wedge c = E\ ,\ post]\ . \end{array}$$

□

Law 30 *following assignment* For any term E,

$$\begin{array}{l} \quad w, x{:}\ [pre\ ,\ post] \\ \sqsubseteq\ w, x{:}\ [pre\ ,\ post[x \backslash E]]\ ; \\ \quad x := E\ . \end{array}$$

□

Law 31 *iteration* Let *inv*, the *invariant*, be any formula; let V, the *variant*, be any integer-valued expression. Then if GG is the disjunction of the guards,

$$\begin{array}{l} \quad w{:}\ [inv\ ,\ inv \wedge \neg GG] \\ \sqsubseteq\ \textbf{do}\ ([]i \cdot G_i \rightarrow w{:}\ [inv \wedge G_i\ ,\ inv \wedge (0 \leq V < V_0)])\ \textbf{od}\ . \end{array}$$

Neither *inv* nor G_i may contain initial variables. The expression V_0 is $V[w \backslash w_0]$.

□

Law 32 *leading assignment* For any expression E,

$$\begin{array}{l} \quad w, x{:}\ [pre[x \backslash E]\ ,\ post[x_0 \backslash E_0]] \\ \sqsubseteq\ x := E; \\ \quad w, x{:}\ [pre\ ,\ post]\ . \end{array}$$

The expression E_0 abbreviates $E[w, x \backslash w_0, x_0]$.

□

Law 33 *leading assignment* For disjoint w and x,

$$w, x := E, F[w \backslash E] = w := E;\ \ x := F\ .$$

□

Abbreviation 34 *sequence assignment* For any sequence as, if $0 \leq i, j \leq \#as$ then

$$as[i := E][j] \mathrel{\widehat{=}} \begin{array}{ll} E & \text{when}\ \ i = j \\ as[j] & \text{when}\ \ i \neq j\ . \end{array}$$

□

Law 35 *sequential composition*

$$\begin{array}{l} \quad w, x{:}\ [pre\ ,\ post] \\ \sqsubseteq\ x{:}\ [pre\ ,\ mid]; \\ \quad w, x{:}\ [mid\ ,\ post]\ . \end{array}$$

The formula *mid* must not contain initial variables; and *post* must not contain x_0.

□

Law 36 *simple specification* Provided E contains no w,

$$w := E = w: [w = E] .$$

If w and E are lists, then the formula $w = E$ means the equating of corresponding elements of the lists.

□

Abbreviation 37 *simple specification* For any relation $\odot$,

$$w: \odot E = w: [w \odot E_0] ,$$

where E_0 is $E[w\backslash w_0]$.

□

Law 38 *skip command* If $(w = w_0) \wedge pre \Rrightarrow post$, then

$$w: [pre , post] \sqsubseteq \textbf{skip} .$$

□

Abbreviation 39 *specification invariant* Provided *inv* contains no initial variables,

$$w: [pre , inv , post] \mathrel{\widehat{=}} w: [pre \wedge inv , inv \wedge post] .$$

□

Law 40 *strengthen postcondition* If $post' \Rrightarrow post$, then

$$w: [pre , post] \sqsubseteq w: [pre , post'] .$$

□

Law 41 *strengthen postcondition* If $pre[w\backslash w_0] \wedge post' \Rrightarrow post$, then

$$w: [pre , post] \sqsubseteq w: [pre , post'] .$$

□

Law 42 *weaken precondition* If $pre \Rrightarrow pre'$, then

$$w: [pre , post] \sqsubseteq w: [pre' , post] .$$

□

References

1. J R Abrial. 'Generalised substitutions'. 26 Rue des Plantes, Paris 75014, France, 1987.
2. R J Back. 'A calculus of refinements for program derivations'. *Acta Informatica*, **25**:593–624, (1988).
3. E Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.
4. C Morgan. 'The specification statement'. *ACM Transactions on Programming Languages and Systems*, **10**(3), (July 1988). Reprinted in [6].
5. C Morgan. *Programming from Specifications*. Prentice-Hall, 1990.
6. C Morgan and T Vickers, eds. *On the Refinement Calculus*. FACIT Series in Computer Science. Springer, 1994.
7. J Morris. 'A theoretical basis for stepwise refinement and the programming calculus'. *Science of Computer Programming*, **9**(3):287–306, (December 1987).

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for *review* should be prepared according to the following guidelines.

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets [1–3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) LaTeX file(s), either on a diskette, or via e-mail/ftp – a LaTeX style file is available from the production editor;
2. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

   Further instructions on how to reduce page charges can be obtained from the production editor.
3. In camera-ready format – a detailed page specification is available from the production editor;
4. In a typed form, suitable for scanning.

Authors will be expected to sign a copyright release form.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R30-00 per final page for LaTeX or camera-ready contributions that require no further attention. The maximum is R120-00 per page for contributions in typed format (charges include VAT).

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 2 and 4 above may be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Camera-ready submissions will only be accepted if they are in strict accordance with the detailed guidelines. It is the responsibility of the authors to ensure that their submissions are error-free.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

1. E Ashcroft and Z Manna. 'The translation of 'goto' programs to 'while' programs'. In *Proceedings of IFIP Congress 71*, pp. 250–255, Amsterdam, (1972).
2. C Bohm and G Jacopini. 'Flow diagrams, turing machines and languages with only two formation rules'. *Communications of the ACM*, 9:366–371, (1966).
3. S Ginsburg. *Mathematical theory of context free languages*. McGraw Hill, New York, 1966.

South African Computer Journal

Number 13, April 1995
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 13, April 1995
ISSN 1015-7999

Contents

INTRODUCTION

WOFACS '94: The Second Workshop on Formal Aspects of Computer Science
C Brink 1

PROCEEDINGS

Functionality Decomposition by Compositional Correctness Preserving Transformation
E Brinksma and R Langerak 2

Modal Logics for Programs
R Goldblatt 14

A Gentle Introduction to Domain Theory: Domains and Powerdomains
J Goslett and A Melton 45

The Refinement Calculus
C Morgan 64

UD Drukkers Printers