The Rudiments of Algorithm Refinement

J. C. P. WOODCOCK
Oxford University Computing Laboratory, Programming Research Group, 11 Keble Road, Oxford OX1 3QD

We describe the rudiments of algorithm refinement: the business of taking a specification and producing code that correctly implements it. The paper starts with a general discussion of the concepts, and then turns to a particular calculus for algorithm refinement.

Received June 1992

1. INTRODUCTION

rĕfī'nement (-nm-) *n.* Refining or being refined; fineness of feeling or taste, polished manners etc.; subtle or ingenious manifestation *of,* piece of elaborate arrangement, (*all the refinements of reasoning, torture; a countermine was a refinement beyond their skill*); instance of improvement (*up*)*on*; piece of subtle reasoning, fine distinction.

The Concise Oxford Dictionary, 7th Edition

In this paper, we give an account of algorithm refinement: the process of producing code that correctly implements a specification. We describe the laws that allow us to introduce programming constructs progressively, and that may be used as part of a programming method based upon calculation.

In Section 2 we come to a definition of refinement by considering the circumstances under which one program may be substituted for another without a user being able to notice the difference. From this definition, we derive the usual proof rules for refinement: that of weakening the precondition and strengthening the postcondition.

In Section 3 we show how specifications and programs may be set in the same semantic framework, so that proof rules may be derived for refinement steps using the various program combinators. However, by writing our specifications in a particular way, the burden of proof may be dramatically reduced. For this purpose, at the end of this section, we arrive at the specification statement.

In Section 4 we give the refinement calculus: some basic laws for manipulating specification statements and developing code from them; and in Section 5 we give some small program developments.

2. DEFINING REFINEMENT

In this section we arrive at a definition of refinement, and explore a few of its properties.

Suppose that you are a supplier of computer programs, and that you have a customer. You enter into a binding contract to supply your customer with the program A. However, you already have another, different program B, which you intend to substitute for A, thereby saving development costs. Are you going to get away with this subterfuge? If the customer can detect the fact that you have delivered B, then you will have to face the full force of the law; if there is no way of detecting your misdoings, then you must succeed.

Let us try to formalise this. The customer must not be able to tell if you substitute B for A. Let us write $A \sqcup B$ to denote the program which behaves either like A or like B, the choice being arbitrary in the sense that it cannot be influenced by the user of the program. In fact, the choice is yours: to deliver A, or to deliver B. Now we can describe the circumstances under which the deception can succeed:

$$A \sqcup B = A,$$

that is, the program $A \sqcup B$ cannot be distinguished from the program A.

In order to formalise the $\sqcup$ operator, we should fix out notation for programs: we shall use the Z notation to describe them.[5, 17] In Z, there is already a choice operator that combines programs: schema disjunction. Suppose that we have a program state that consists simply of the variable x that ranges over the set s. A program may be specified by describing the relation between x's value before the program is executed and its value after the program has been executed. We distinguish the latter from the former by decorating x with a prime, thus: x'. Suppose that p and q are both predicates involving x and x', then the programs A and B might be specified as

$$A \mathrel{\hat{=}} [x, x' : s \mid p]$$
$$B \mathrel{\hat{=}} [x, x' : s \mid q].$$

The choice of programs $A \vee B$ behaves either like A, or like B, and is given by

$$A \vee B = [x, x' : s \mid p \vee q],$$

that is, either the relation specified by p holds between x and x', or the relation specified by q holds between them.

The precondition of a program describes all the states in which the program is guaranteed to terminate with the correct result. In our example, the precondition of A consists of all those values of x which are related to an x' by p: the domain of the relation. If the precondition of A is denoted by $\operatorname{pre} A$, then we have

$$\operatorname{pre} A = [x : s \mid \exists x' : s \bullet p].$$

That is, all those values of x in s, such that there exists an x' in s satisfying p. It is easy to show that the precondition operator distributes through the schema disjunction operator, so

$$\operatorname{pre}(A \vee B) = (\operatorname{pre} A) \vee (\operatorname{pre} B).$$

Suppose that we have a particular value of x, say x_0, which lies in the precondition of A, but outside the precondition of B. Clearly, it is in the precondition of $A \vee B$. Should it also be in the precondition of $A \sqcup B$? The answer must be no: if the user activates the program $A \sqcup B$ in this state, and you the supplier have chosen to deliver the program B, then the result will not be what is expected: namely, termination in a correct state. Thus, the user has a test that can distinguish whether or not you have substituted B for A. This leads us to suppose that x_0 is in the precondition of $A \sqcup B$ if it is both in the precondition of A and in the precondition of B. At last, we can define our $\sqcup$ operator:

$$A \sqcup B \mathrel{\hat=} \forall S, S' \bullet (\operatorname{pre} A) \wedge (\operatorname{pre} B) \wedge (A \vee B),$$

where S denotes the before-state, and S' denotes the after-state.*

Thus we have the difference between the two choice operators: $A \vee B$ is very obliging, since if termination is possible, it will happen; $A \sqcup B$ is more awkward, since if non-termination is possible, it will happen. We describe the former as *angelic*, and the latter as *demonic*.

If we have the equation

$$A \sqcup B = A,$$

we say that B is a refinement of A, denoted

$$A \sqsubseteq B.$$

So, if B is a refinement of A, we can use B wherever using A is acceptable, safe in the knowledge that no test can detect the difference.

Proposition 1. *Demonic choice is idempotent, commutative, and associative*:

$$A \sqcup A = A$$
$$A \sqcup B = B \sqcup A$$
$$A \sqcup (B \sqcup C) = (A \sqcup B) \sqcup C.$$

Proposition 2. *Refinement is reflexive, antisymmetric, and transitive*:

$$A \sqsubseteq A$$
$$A \sqsubseteq B \wedge B \sqsubseteq A \Rightarrow A = B$$
$$A \sqsubseteq B \wedge B \sqsubseteq C \Rightarrow A \sqsubseteq C.$$

Transitivity forms the basis for stepwise development. Thus, if a user cannot tell the difference between A and B, nor between B and C, then of course the user cannot tell the difference between A and C. This transitivity result is a consequence of the associativity of the demonic choice operator.

Refinement consists in two ideas: weakening the precondition and strengthening the effect. This is demonstrated by the following proposition.

Proposition 3. *If A is refined by B, B has a stronger effect (within the precondition of A), and a weaker precondition.*

$$\begin{aligned} &A \sqsubseteq B \Leftrightarrow \\ &\quad (\forall S, S' \bullet (\operatorname{pre} A) \wedge B \Rightarrow A) \wedge \\ &\quad (\forall S \bullet (\operatorname{pre} A) \Rightarrow (\operatorname{pre} B)). \end{aligned}$$

What do these two ideas mean? Suppose again that B is a refinement of A. That the precondition of B may be weaker than the precondition of A means that B must be defined everywhere that A is. It may well be defined elsewhere, but that is irrelevant. The other idea is that wherever A is defined, B must be as strong as A: any 'extra' behaviour is disregarded. Now what do we mean by saying that B must be at least as strong as A? We mean that whatever B does is correct with respect to A. Everywhere that A is defined, B is required to produce a result that A could produce.

Our approach of defining refinement using demonic choice is similar, for example, to that of Sekerinski.[16]

* This definition may be read as 'The non-deterministic choice between A and B is defined to be, for all before-states S and all after-states S', the precondition of A and the precondition of B and the disjunction of A and B'.

3. LINKING SPECIFICATIONS AND CODE

The purpose of this paper is to discuss algorithm refinement, that is, the process of developing code that implements a specification. In this section, we discuss the formal link between specifications and code. We start off by giving an introduction to the problem of algorithm refinement by means of a simple example. In order to establish our formal link we need a way of giving meaning to specifications and to code, and we do this in terms of the method of weakest preconditions; this also gives us a way of addressing the concerns of refinement. We describe how a step of development may be carried out using the existing weakest precondition apparatus. This leads to a method that is somewhat cumbersome, and we then set out to improve the method of eliminating some of this burden of proof. We introduce the specification statement, the basis of our streamlined method.

Let us take a simple example. Suppose that we have the little specification†

$$Update \mathrel{\hat=} [x, x' : \mathbb{N} \mid x' = x + 2];$$

thus, it is *Update*'s task to increment x's value by 2. In the unlikely event that we have a fast way of incrementing by 1, but not by 2, we might choose to implement *Update* with the code

```
x := x + 1;
x := x + 1.
```

How can we be certain that this is correct? Well, if the fragment of code is a refinement of *Update*, all will be well. We need a way of understanding the program combinators, such as the ';', and program commands, such as the assignment operator, ':=', in the same framework as our specification; then we shall we be able to say whether or not this development step is correct.

One simple way of giving a meaning to a programming language is by using predicate transformers. In this approach, each combinator and command of the language is defined by explaining, for any postcondition, the precondition that will guarantee termination in a final state satisfying the postcondition. As we have taken schemas as our means of specifying programs, we shall use a variation on this theme: a schema transformer. Our language fragment is an operation on the state, and we shall take as our postcondition a schema which relates before-and-after values of the state. The weakest precondition for an operation to establish a postcondition will be a set of states, described as a schema.‡

Definition 1. *Suppose that Op is an operation on the state S; that R is a schema relating the before-state S and the after-state S'; then the* weakest precondition *for Op to establish R is*:

$$\mathrm{wp}_{S'}(Op, R) \mathrel{\hat=} (\exists S' \bullet Op) \wedge (\forall S' \bullet Op \Rightarrow R).$$

† Where $\mathbb{N}$ denotes the *natural numbers*; that is, $0, 1, 2, 3, \ldots$

‡ This follows (Joseph, 1988); weakest preconditions are explained in Dijkstra.[6]

By comparing Proposition 3 and our definition, notice that a consequence is that refinement can be explained in a third way.

Proposition 4. *Refinement can be defined using the weakest precondition construction*:

$$A \sqsubseteq B \Leftrightarrow \forall S \bullet \mathrm{pre}\, A \Rightarrow \mathrm{wp}_{S'}(B, A).$$

Now that we have a way of giving meaning to our programming language, let us try to formalize the development of *Update*. First, we need a meaning for sequential composition. Before we can start, we need a little bit of notation. If *Op* is a schema that relates a before-state (undecorated) with an after-state (decorated with a prime), Op° is a schema that relates a before-state (undecorated) with an after-state (decorated with a $^{\circ}$); similarly, $^{\circ}Op$ is a schema that relates a before-state (decorated with a $^{\circ}$) with an after-state (decorated with a prime).

Definition 2

$$\mathrm{wp}_{S'}(First; Second, R) \mathrel{\hat{=}} \mathrm{wp}_{S^{\circ}}(First^{\circ}, \mathrm{wp}_{S'}({}^{\circ}Second, R)).$$

Given

$$IncBy1 \mathrel{\hat{=}} [x, x' : \mathbb{N} \mid x' = x + 1],$$

we have

$$\begin{aligned}
&\mathrm{wp}_{S'}(IncBy1; IncBy1, Update)\\
&\Leftrightarrow \mathrm{wp}_{S^{\circ}}(IncBy1^{\circ}, \mathrm{wp}_{S'}({}^{\circ}IncBy1, Update))\\
&\Leftrightarrow \mathrm{wp}_{S^{\circ}}(IncBy1^{\circ}, (\exists S' \bullet {}^{\circ}IncBy1) \wedge (\forall S' \bullet {}^{\circ}IncBy1\\
&\qquad\qquad \Rightarrow Update))\\
&\Leftrightarrow \mathrm{wp}_{S^{\circ}}(IncBy1^{\circ}, \forall S' \bullet {}^{\circ}IncBy1 \Rightarrow Update)\\
&\Leftrightarrow (\exists S^{\circ} \bullet IncBy1^{\circ}) \wedge (\forall S^{\circ} \bullet IncBy1^{\circ}\\
&\qquad\qquad \Rightarrow (\forall S' \bullet {}^{\circ}IncBy1 \Rightarrow Update))\\
&\Leftrightarrow (\forall S^{\circ} \bullet IncBy1^{\circ} \Rightarrow (\forall S' \bullet {}^{\circ}IncBy1 \Rightarrow Update))\\
&\Leftrightarrow \forall S^{\circ}, S' \bullet IncBy1^{\circ} \wedge {}^{\circ}IncBy1 \Rightarrow Update\\
&\Leftrightarrow (\forall S^{\circ}, S' \bullet x^{\circ} = x + 1 \wedge x' = x^{\circ} + 1 \Rightarrow x' = x + 2\\
&\Leftrightarrow true.
\end{aligned}$$

Using the weakest precondition calculus directly thus is rather tiresome: what is needed is a set of simple but sufficient conditions, proved in advance, to tell us whether the refinement step is correct. We can derive such a set from the definitions that we have.

Proposition 5. *In order to show that*

$$A \sqsubseteq B; C$$

it is sufficient to prove the three theorems

$$\begin{aligned}
&\mathrm{pre}\, A \vdash \mathrm{pre}\, B\\
&\mathrm{pre}\, A \wedge B^{\circ} \vdash \mathrm{pre}\, {}^{\circ}C\\
&\mathrm{pre}\, A \wedge B^{\circ} \wedge {}^{\circ}C \vdash A.
\end{aligned}$$

In our example, since *Update* and *IncBy*1 are both total, the only thing to show is that

$$IncBy1^{\circ} \wedge {}^{\circ}IncBy1 \vdash Update$$

that is, that

$$[x, x^{\circ} : \mathbb{N} \mid x^{\circ} = x + 1] \wedge [x^{\circ}, x' : \mathbb{N} \mid x' = x^{\circ} + 1] \vdash [x, x' : \mathbb{N} \mid x' = x + 2],$$

which is rather trivial (as it should be).

In general, having to prove three theorems in order to introduce the sequential composition operator is excessive; can anything be done to smooth the process? There are two notational changes that may be made to our specifications that will help somewhat. The first is to reuse past results, and to record the precondition for our specifications separately from their postconditions. Thus, specifications become precondition/postcondition pairs. Suppose that each condition is given as a predicate on a single state (we shall revisit this presently). Then we can eliminate the need for proof completely in a development step that introduces sequential composition. Let us try to give a meaning to this new idea. Suppose that *pre* is a predicate on the state S; that *post* is a predicate on the state S'; and that R is a relation between the before-state S and the after-state S'. Then the meaning of the precondition/postcondition specification is given by the definition

$$\mathrm{wp}_{S'}([pre, post], R) \mathrel{\hat{=}} pre \wedge (\forall S' \bullet post \Rightarrow R).$$

Note that this is very similar to the meaning of a schema specification, say *Op*, but with the asserted precondition in place of the term $\exists S' \bullet Op$: the actual precondition.

Now, we can hazard a guess at a refinement rule. If we need to implement the specification $[pre, post]$, then why not choose a mid-point, *mid* to achieve in a first step, and to proceed from in a second step? Our rule would be

$$[pre, post] \sqsubseteq [pre, mid']; [mid, post].$$

Is this correct? Before we answer that, we can make a notational change which simplifies things even further: let us write both the precondition and the postcondition without any decoration.

Definition 3. *Suppose that pre, post, and R are predicates on the state S. Then, the meaning of the precondition/postcondition specification is given by the definition*

$$\mathrm{wp}([pre, post], R) \mathrel{\hat{=}} pre \wedge (\forall S \bullet post \Rightarrow R).$$

Consider the following:

$$\begin{aligned}
&[pre, post] \sqsubseteq [pre, mid]; [mid, post]\\
&\Leftrightarrow \forall S \bullet pre \Rightarrow \mathrm{wp}([pre, mid]; [mid, post], post)\\
&\Leftrightarrow \forall S \bullet pre \Rightarrow \mathrm{wp}([pre, mid], \mathrm{wp}([mid, post], post))\\
&\Leftrightarrow \forall S \bullet pre \Rightarrow \mathrm{wp}([pre, mid], mid \wedge \forall S \bullet post \Rightarrow post)\\
&\Leftrightarrow \forall S \bullet pre \Rightarrow \mathrm{wp}([pre, mid], mid)\\
&\Leftrightarrow \forall S \bullet pre \Rightarrow pre \ \wedge \forall S \bullet mid \Rightarrow mid\\
&\Leftrightarrow \forall S \bullet pre \Rightarrow pre\\
&\Leftrightarrow true.
\end{aligned}$$

Thus our refinement rule is correct.

Proposition 6. *For any predicate mid,*

$$[pre, post] \sqsubseteq [pre, mid]; [mid, post].$$

On the fact of it, this is a very nice refinement rule: choose your mid-point, and then snap the specification in half about that point. This works for any *mid*, although if *mid* is very weak, the first subtask will be easy to achieve, but the second will be difficult; if *mid* is very strong, the first subtask will be difficult to achieve, but the second will be easier. However, if we return to our very simple problem of refining *Update*, we have a more immediate difficulty: how do we write *Update* in terms of a pair of predicates, each of which is of a single state?

The program

$$|[\mathbf{con}\ X : T \bullet prog]|$$

introduces a logical constant named X, which ranges over T. Its scope is delimited by the local block brackets. We can think of **con** X as choosing a value for X which makes subsequent preconditions *true*, is possible. This is a kind of *angelic non-determinism*, and may be contrasted with **var** x, which is *demonic*. Logical constants should not be confused with the sort of constants that one might find in a real programming language.

Logical constants may be used to give names to things that must exist. A simple example of this is the way in which they may be used to fix the before-value of a variable. That is what is being done in the block

$$|[\mathbf{con}\, X \bullet x\!:\![x = X, x = X+2]]|.$$

The logical constant X takes a value that makes subsequent preconditions true, if possible. Within the scope of X there is only one precondition $x = X$, and so X takes on this value: namely, the value of x before the specification statement. This kind of thing happens so frequently that we introduce an abbreviation.

Abbreviation 1 (initial variables). *Zero-subscripted variables in the postcondition of a specification statement refer to the before-values of those variables*:

$$\begin{array}{l} w\!:\![pre, post] \mathrel{\hat{=}} \\ \quad |[\mathbf{con}\, X\!:\!T \bullet \\ \qquad w\!:\![pre \wedge x = X, post[X/x_0]] \\ \quad]|. \end{array}$$

The fact that we can use zero-subscripted variables does not change our view that a postcondition is a predicate on just a single state, since these variables are explained using logical constants.

Example 1. *Consider the specification statement*

$$x, y\!:\![y = x+1, x = x_0 + y_0].$$

It is abbreviation for

$$|[\mathbf{con}\, X, Y\!:\!\mathbb{N} \bullet x, y\!:\![y = x+1 \wedge x = X \wedge y = Y, x = X+Y]]|.$$

Now, we would like to be able to refine *Update* to the sequential composition

$$[true, x = x_0+2] \sqsubseteq [true, x = x_0+1]; [true, x = x_0+1].$$

How do we justify this development step?

If we choose as our mid-point $x = x_0+1$, then we have a problem, since it is no good for the precondition of the second part, since in our notation that *must* be a predicate of a single state. Furthermore, the second part ought to have the same postcondition as the specification: $x = x_0+2$, and the x_0 should refer to the value of the variable x before the execution of the first part. These two problems have a common solution: rename the troublesome x_0s. If we chose $x = X+1$ as the precondition of the second part, and $x = X+2$ as its postcondition, that would conform to our notation. Thus we are proposing the development step

$$\begin{array}{l} [true, x = x_0+2] \sqsubseteq \\ \quad |[\mathbf{con}\, X \bullet \\ \qquad [true, x = x_0+1]; \\ \qquad [x = X+1, x = X+2] \\ \quad]|. \end{array}$$

The first part of the composition increments x by 1. The second part assumes that $x = X+1$. As X is a logical constant, it is chosen so as to make this true, and thus we have that $X = x_0$; that is, X fixes the value of x before the execution of the composition. The second part now increases x value again, so that it is equal to $X+2$, two more than the value of x before the execution of the composition. Now our notation allows the introduction of a second logical constant, whose scope encloses the second part of the composition, and we obtain the program

$$\begin{array}{l} |[\mathbf{con}\, X \bullet \\ \quad [true, x = x_0+1]; \\ \quad |[\mathbf{con}\, Y \bullet \\ \qquad [x = X+1 \wedge x = Y, x = X+2] \\ \quad]| \\ ]|. \end{array}$$

Now, if we use the equality $Y = X+1$ in the postcondition, and then weaken the precondition, we obtain

$$\begin{array}{l} |[\mathbf{con}\, X \bullet \\ \quad [true, x = x_0+1]; \\ \quad |[\mathbf{con}\, Y \bullet \\ \qquad [x = Y, x = Y+1] \\ \quad]| \\ ]| \end{array}$$

and using the abbreviation,

$$\begin{array}{l} |[\mathbf{con}\, X \bullet \\ \quad [true, x = x_0+1]; \\ \quad [true, x = x_0+1] \\ ]|. \end{array}$$

If we drop the redundant logical constant, we have finished.

Finally, we introduce one last piece of notation: the program frame. Suppose that the program state contains the variables x, y and z, but that we want to change only the variable x. Normally, we must say explicitly that y and z do not change: the meaning of silence is that anything might happen. At the specification level, it is important to remember that if we do not mention the fate of a variable in an operation, anything might happen to its value. An operation is a relation, and we constrain its behaviour further by adding a stronger predicate. In a sequential programming language such as Pascal, if a program statement does not mention a variable (in an assignment for example), it does not change. In the refinement calculus, we see a half-way house: if a variable is not in the frame, then, as in a programming language, it cannot be changed, no matter what we say in the postcondition; if it is mentioned in the frame, it may be changed, and we can constraint its after-value by adding a stronger predicate. We shall write the names of variables which may be changed by the code implementing a specification; if a variable does not appear in this list it must be kept constant. We can now write down the anatomy of a specification statement:

$$\underbrace{w}_{\text{frame}} \!:\![\overbrace{pre}^{\text{precondition}}, \underbrace{post}_{\text{postcondition}}].$$

The frame is a – possibly empty – list of variables, and the precondition and postcondition are each a predicate on a single state: the before-state and the after-state, respectively. The specification statement describes a task for an implementor: a program is required that terminates whenever *pre* is true, and when it does so it produces a correct result, satisfying *post*. The specification statement is pronounced in something like the following manner:

By changing only the variables w, and by assuming that the state satisfies *pre*, change it so that it satisfies *post*.

4. THE REFINEMENT CALCULUS

The so-called refinement calculus comprises several laws for manipulating specification statements. In this section we give some examples of specifications, and some of the laws that we use to turn them into implementations. The work on refinement calculi has been reported by Back,[1-4] Gardiner and Morgan,[7] Morgan,[10-12] Morgan and Robinson,[13] and Morris.[14, 15]

Example 2. *The operation that finds the root in the interval* $[a, b]$ *of a continuous function* f, *providing that it has one, is*

$$Bisect \mathrel{\hat{=}} m{:}[f(a)\times f(b) \leqslant 0 \wedge a \leqslant b, |f(m)| < 0.1 \wedge a \leqslant m \leqslant b].$$

4.1 Extremities

Certain specification statements have particular names. The specification statement '$w{:}[false, true]$' is called **abort**; it is never guaranteed to terminate (it has precondition *false*); if it does terminate, it might produce any result (postcondition *true*). The specification statement '$w{:}[true, true]$' is called **choose** w; it is always guaranteed to terminate, but it might produce any result. The specification statement '$:[true, true]$' is called **skip**; it is always guaranteed to terminate, without changing anything (its frame is empty). The specification statement '$w{:}[true, false]$' is called **magic**; it is always guaranteed to terminate, and when it does so, it establishes the impossible (a state satisfying *false*).

4.2 Some simple laws

A program in the refinement calculus may consist of a single specification statement, or it may consist entirely of code in the guarded command language (an idealised programming language), or it may consist of a mixture of specification statements and code. An example of the latter is

$$x, y{:}[x = X \wedge y = Y, x = X - Y \wedge y = X]; x := y - x.$$

The task of developing code in the refinement calculus usually starts with a single specification statement, goes through several intermediate stages of mixed programs, and ends up with code in the guarded command language, free from specification statements. If a program contains no specification statements, then we call it *code*. It might be that a particular development leads to a dead-end, so sometimes no code can be produced from a specification statement. In this section we give some laws for refining specification statements.

One way of improving a specification is to do more than was required. Thus, if *post'* is a stronger predicate than *post*, then any client who was satisfied with $w{:}[pre, post]$ must also be satisfied with $w{:}[pre, post']$.* This corresponds to the removal of non-determinism.

* Remember that in this example, and in what follows, *post'* is just another mathematical variable.

Law 1 (strengthen postcondition 'sp'). *If* $post' \Rightarrow post$ *then*

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

Example 3. *Since* $x < 0.01 \Rightarrow x < 0.1$, *we have that*

$$m{:}[f(a)\times f(b) \leqslant 0 \wedge a \leqslant b, |f(m)| < 0.1 \wedge a \leqslant m \leqslant b]$$
$$\sqsubseteq \textbf{`sp'}$$
$$m{:}[f(a)\times f(b) \leqslant 0 \wedge a \leqslant b, |f(m)| < 0.01 \wedge a \leqslant m \leqslant b].$$

This refinement step has improved the accuracy of the result computed by this specification statement. The annotation **'sp'** *denotes the particular law that justifies the refinement.*

Another simple way of improving a specification is to make it apply to more situations than was asked for. Thus, if *pre'* is a weaker predicate than *pre*, any client who was satisfied with $w{:}[pre, post]$ must also be satisfied with $w{:}[pre', post']$, since it works at least as often. This corresponds to the widening of preconditions.

Law 2 (weaken precondition 'wp'). *If* $pre \Rightarrow pre'$ *then*

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

Example 4. *Since*

$$f(a)\times f(b) \leqslant 0 \wedge a \leqslant b \Rightarrow f(a)\times f(b) \leqslant 0,$$

we have that

$$m{:}[f(a)\times f(b) \leqslant 0 \wedge a \leqslant b, |f(m)| < 0.01 \wedge a \leqslant m \leqslant b]$$
$$\sqsubseteq \textbf{`wp'}$$
$$m{:}[f(a)\times f(b) \leqslant 0, |f(m)| < 0.01 \wedge a \leqslant m \leqslant b].$$

This refinement step means that we now require an implementation to produce a correct result even when a and b do not describe an interval. It seems that we have given up too much: if $b < a$, *then the postcondition is not going to be satisfiable.*

Definition 4 (feasibility). $w{:}[pre, post]$ *is feasible iff*

$$pre \Rightarrow \exists w{:}T \bullet post.$$

Recalling our definition of the precondition of an operation defined using a schema, we can see that feasibility is a check that the predicate that we claim is the precondition, *pre*, is at least as strong as the real precondition. If a specification is infeasible, then we cannot refine it to code; all code is feasible. Thus, an infeasible specification cannot lead to incorrect code, since it cannot lead to any code at all. For this reason, we are not obliged to perform feasibility checks during our development; however, there is always the opportunity of doing so.

Example 5. *Since*

$$\neg (f(a)\times f(b) \leqslant 0 \Rightarrow \exists m \bullet |f(m)| < 0.01 \wedge a \leqslant m \leqslant b),$$

the specification

$$m{:}[f(a)\times f(b) \leqslant 0, |f(m)| < 0.01 \wedge a \leqslant m \leqslant b]$$

is infeasible.

4.3 Assignment

Law 3 (assignment 'assI'). *If* $pre \Rightarrow post[E/w]$ *then*

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

If we use initial variables, we must be careful with the application of certain laws. The assignment introduction law must be changed in order to cope with the fact that the precondition and postcondition use different names for the same values.

Law 4 (assignment introduction§ 'assI§'). *If* $pre \wedge w = w_0 \Rightarrow post[E/w]$ *then*

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

Example 6. *Since*

$$\begin{aligned}&x = x_0\\ &\Leftrightarrow x+1 = x_0+1\\ &\Leftrightarrow (x = x_0+1)[x+1/x],\end{aligned}$$

we have

$x{:}[true, x = x_0+1] \sqsubseteq x := x+1.$

Example 7. *Since*

$$\begin{aligned}&x = X\\ &\Rightarrow 5 = 5 \wedge x = X \wedge 6 = 6\\ &\Leftrightarrow (x = 5 \wedge y = X \wedge z = 6)[5, x, 6/x, y, z],\end{aligned}$$

$$\begin{aligned}&x, y, z{:}[x = X, x = 5 \wedge y = X \wedge z = 6]\\ &\sqsubseteq \text{'assI'}\\ &x, y, z := 5, x, 6.\end{aligned}$$

In the refinement calculus, we introduce program variables using a declaration that is rather like an axiomatic definition: we name the variables, say what sets they range over, and add an invariant.

Law 5 (introduce local block 'varI'). *If w and x are disjoint, then*

```
w:[pre, post] ⊑
   |[var x: T | inv •
      w, x:[pre, post]
   ]|.
```

Example 8

```
Bisect
= 'by definition'
   m:[f(a) × f(b) ⩽ 0 ∧ a ⩽ b, |f(m)| < 0.1 ∧ a
                                           ⩽ m ⩽ b]
⊑ 'varI'
   |[var x, y •
      x, y, m:[f(a) × f(b) ⩽ 0 ∧ a ⩽ b, |f(m)|
                                  < 0.1 ∧ a ⩽ m ⩽ b]
   ]|.
```

We usually abbreviate this as

```
Bisect
⊑ var x, y •
   x, y, m:[f(a) × f(b) ⩽ 0 ∧ a ⩽ b, |f(m)|
                               < 0.1 ∧ a ⩽ m ⩽ b].
```

The presence of invariants means that we must tighten up our notion of feasibility.

Definition 5 (feasibility). $w{:}[pre, post]$ *is feasible iff*

$pre \wedge inv \Rightarrow \exists w{:}T \bullet inv \wedge post.$

Law 6 (skip command 'skipI'). *If* $pre \Rightarrow post$ *then*

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

Example 9. *Since* $x = 5 \wedge y = x^3 \Rightarrow x = 5$, *we have that*

$x, y{:}[x = 5 \wedge y = x^3, x = 5]$
$\sqsubseteq$ **skip**.

Law 7 (sequential composition 'semI'). *For any predicate mid*

$w{:}[pre, post]$
$\sqsubseteq w{:}[pre, mid]; w{:}[mid, post].$

If a zero-subscripted variable is used in a postcondition P, it refers to the value of the variable before that specification statement. If we break the statement in two using **emI**, the zero-subscripted variable in P now refers to the value of the variable *at the mid-point*. The law is updated to avoid this mistake.

Law 8 (sequential composition introduction§ 'semI§'). *For fresh constants X,*

```
w, x:[pre, post]
⊑
   |[con X •
      x:[pre, mid];
      w, x:[mid[X/x₀], post[X/x₀]]
   ]|.
```

The predicate mid must not contain initial variables other than x_0.

Example 10

```
x:[true, x = x₀ + 2]
⊑ 'semI§' con X •
   x:[true, x = x₀ + 1];
   x:[x = X + 1, x = X + 2].
```

Example 11. *In Example 7 we used a multiple assignment to implement a specification; not all programming languages have this facility, so here is a development that does the necessary assignments in sequence:*

```
x, y, z:[x = X, x = 5 ∧ y = X ∧ z = 6]
⊑ 'semI'
x, y, z:[x = X, y = X];                        [◁]
x, y, z:[y = X, x = 5 ∧ y = X ∧ z = 6]         [†]
⊑ 'assI'
y := x
† ⊑ 'semI'
x, y, z:[y = X, x = 5 ∧ y = X]                 [◁]
x, y, z:[x = 5 ∧ y = X, x = 5 ∧ y = X ∧ z = 6] [‡]
⊑ 'assI'
x := 5
‡ ⊑ 'assI'
z := 6.
```

Note the use of marginal markers. We shall be using a convention that the left-hand (◁) always points to the next part of the program to be refined. Other marginal markers, such as the obelisk and double obelisk used here, refer to parts of the program whose development proceeds at a later point. Thus we end up with a flattened tree as the record of the development. It is not difficult to see how the tree may be walked in order to extract the code from the development. It is easier to see the final code once it has been retrieved from the development; however, it becomes more difficult to see how it was obtained once the development record has been thrown away.

Example 12. *Notice that we were fortunate in our last development to find a correct order for our assignments. What would have happened if we had not been so fortunate? Consider this:*

$x, y, z:[x = X, x = 5 \wedge y = X \wedge z = 6]$
$\sqsubseteq$ **'semI'**
$x, y, z:[x = X, x = 5];$ $[\lhd]$
$x, y, z:[x = 5, x = 5 \wedge y = X \wedge z = 6]$ $[\dagger]$
$\sqsubseteq$ **'assI'**
$x := 5$
$\dagger \sqsubseteq$ **'semI'**
$x, y, z:[x = 5, x = 5 \wedge y = X];$ $[\ddagger]$
$x, y, z:[x = 5 \wedge y = X, x = 5 \wedge y = X \wedge z = 6]$ $[\lhd]$
$\sqsubseteq$ **'assI'**
$z := 6.$

The specification marked with ‡ has not been reduced to code. We cannot assign the value X to y, since X is a logical constant, and so '$y := X$' would not be code. In fact, this specification is infeasible: no code can be produced to satisfy ‡. Having destroyed the value of x, we cannot re-create it.

Very often, we require combinations of some of the previous basic laws. The following is a good example of this.

Law 9 (following assignment 'fassI§'). *For any expression E,*

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

This is an easy law to apply. First we decide upon the assignment that we would like to perform, then we calculate the new specification statement.

Example 13

$x, y, z:[x = X, x = 5 \wedge y = X \wedge z = 6]$
$\sqsubseteq$ **'fassI'**
$x, y, z:[x = X, x = 5 \wedge y = X];$ $[\lhd]$
$z := 6$
$\sqsubseteq$ **'fassI'**
$x, y, z:[x = X, y = X];$ $[\lhd]$
$x := 5$
$\sqsubseteq$ **'assI'**
$y := x.$

Notice that we could have used **assI** *a third time, and obtained the code*

skip;
$y := x;$
$x := 5;$
$z := 6.$

Law 10 (leading assignment 'lassI§'). *For any expression E,*

$w, x:[pre[E/x], post[E_0/x_0]]$
$\sqsubseteq$
$x := E;$
$w, x:[pre, post].$

4.4 Conditional statements

In the guarded command language, the conditional statement has the form

if $G_1 \rightarrow com_1$
$\Box\, G_2 \rightarrow com_2$
$\vdots$
$\Box\, G_n \rightarrow com_n$
fi.

Each of the branches $G_i \rightarrow com_i$ is called a guarded command, with G_i the guard, and com_i the command. When the conditional is activated the guards $G_1, G_2, \ldots, G_n$ are evaluated, and one of the commands whose guard is true is executed. If no guard is true, the program aborts. A more compact notation for the conditional uses a generalised notation

if $\Box\, i \bullet G_i \rightarrow com_i$ **fi**.

Law 11 (conditional 'ifI'). *If* $pre \Rightarrow \bigvee i \bullet G_i$,

$w:[pre, post]$
$\sqsubseteq$ **if** $\Box\, i \bullet G_i \rightarrow w:[G_i \wedge pre, post]$ **fi**.

Whenever the specification is required to terminate, the conditional must not abort; thus the precondition must establish that at least one guard is true. Whichever branch is taken must implement the specification, but we can strengthen the precondition with the guard in the knowledge that it must be true for that branch to have been taken.

Example 14. *Given two variables x and y, we require a program that will ensure that $x \leqslant y$, by preserving their values, or swapping them if necessary:*

$x, y:[x = X \wedge y = Y, (X \leqslant Y \Rightarrow x = X \wedge y = Y)$
$\wedge (Y \leqslant X \Rightarrow x = Y \wedge y = X)]$
$\sqsubseteq$ **'ifI'**
if $x \leqslant y \rightarrow x, y:[x \leqslant y \wedge x = X \wedge y = Y,$
$(X \leqslant Y \Rightarrow x = X \wedge y = Y) \wedge (Y \leqslant X$
$\Rightarrow x = Y \wedge y = X)]$ $[\lhd]$
$\Box\, y \leqslant x \rightarrow x, y:[y \leqslant x \wedge x = X \wedge y = Y,$
$(X \leqslant Y \Rightarrow x = X \wedge y = Y) \wedge (Y \leqslant X$
$\Rightarrow x = Y \wedge y = X)]$ $[\dagger]$
fi
$\sqsubseteq$ **skip**
$\dagger \sqsubseteq$ **'assI'** $x, y := y, x.$

Notice that the disjunction of the guards is true, thus validating the introduction of the conditional. The program is

if $x \leqslant y \rightarrow$ **skip**
$\Box\, y \leqslant x \rightarrow x, y := y, x$
fi.

4.5 Logical constants

We can introduce a logical constant in much the same way as we would introduce an existential quantifier.

Law 12 (introduce logical constant 'conI'). *If* $pre \Rightarrow (\exists C: T \bullet pre')$, *and C is a fresh name, then*

$w:[pre, post]$
$\sqsubseteq |[$**con** $C: T \bullet w:[pre', post]]|.$

Getting rid of logical constants is important, since they are not code. If a logical constant is no longer mentioned in a program, we can eliminate it (again, in much the same way as removing an existential quantifier).

Law 13 (eliminate logical constant 'conE'). *If C occurs nowhere in prog,*

$|[$**con** $C: T \bullet prog]| \sqsubseteq prog.$

Finally in this section we give a law for contracting the frame in a specification statement: if we drop the name of a variable from the frame, it cannot change; thus, we can drop the zero-subscript on any of its occurrences in the postcondition.

Law 14 (contract frame 'conF')

$$w, x : [pre, post] \sqsubseteq w : [pre, post\,[x/x_0]].$$

4.6 Iteration

In the guarded command language, the loop construct is rather similar in form to the conditional:

$$\begin{array}{l}\mathbf{do}\ G_1 \to com_1 \\ \square\ G_2 \to com_2 \\ \quad\vdots \\ \square\ G_n \to com_n \\ \mathbf{od},\end{array}$$

with the generalised notation

$$\mathbf{do}\ \square\ i \bullet G_i \to com_i\ \mathbf{od}.$$

When the loop is activated the guards $G_1, G_2, \ldots, G_n$ are evaluated, and one of the commands whose guard is true is executed. This is done repeatedly until no guard is true, when the loop terminates. We give a simplified form of the introduction rule, which is applicable to developing a loop with a single branch.

Law 15 (loop introduction 1 'doI1')

$$\begin{array}{l} w : [inv, inv \wedge \neg G] \\ \sqsubseteq \mathbf{do}\ G \to w : [inv \wedge G, inv \wedge 0 \leqslant V < V_0]\ \mathbf{od}. \end{array}$$

The task of the developer is to discover an invariant inv, a guard G, and a variant V.

The more general form of the law involves many branches.

Law 16 (loop introduction 'doI')

$$\begin{array}{l} w : [inv, inv \wedge \neg (\vee\, i \bullet G_i)] \\ \sqsubseteq \mathbf{do}\ \square\ i \bullet G_i \to w : [inv \wedge G_i, inv \wedge 0 \leqslant V < V_0]\ \mathbf{od}. \end{array}$$

5. CASE STUDIES

5.1 Initialising an array

Suppose that we wish to initialise an array so that every entry is zero. An array may be modelled mathematically as a total function from a set of indices to values (in this case numbers):

$$ar : (1\,..\,n) \to \mathbb{N}.$$

The initialisation operation has the task of assigning the value 0 to every element in the array. Its specification is given by

$$Init \mathrel{\hat{=}} ar : [true, \operatorname{ran} ar = \{0\}].$$

Our first step in developing the code for this rather simple operation is to use an obvious transformation of the postcondition:

$$Init = ar : [true, \forall j : 1\,..\,n \bullet ar\,j = 0].$$

The motivation for this is that we intend to implement the operation using a loop, and the universal quantifier points to the way that the loop might be developed. One strategy for loop development is to take such a quantified expression, and replace a constant by a variable. The following shorthand helps us in doing this:

$$zeroed(i, ar) \mathrel{\hat{=}} \forall j : 1\,..\,i \bullet ar\,j = 0.$$

The development of the code follows in rather a detailed manner. The refinement calculus should be used with a light touch, rather than in this heavy-handed manner; however, we go into greater detail so that the reader may follow this, the first case study.

$$\begin{array}{lr}
ar : [true, zeroed(n, ar)] & \\
\sqsubseteq \mathbf{var}\ j \mid 1 \leqslant j \leqslant n+1 \bullet\text{*} & \\
\quad j, ar : [true, zeroed(n, ar)] & \\
\sqsubseteq \text{'}\mathbf{semI}\text{'}\dagger & \\
\quad j, ar : [true, zeroed(j-1, ar)]; & [\lhd] \\
\quad j, ar : [zeroed(j-1, ar), zeroed(n, ar)] & [1] \\
\sqsubseteq \text{'}\mathbf{assI}\text{'}\ddagger & \\
\quad j := 1 & \\
[1] \sqsubseteq \text{'}\mathbf{sp}\text{'}\S & \\
\quad j, ar : [zeroed(j-1, ar), zeroed(j-1, ar) \wedge j = n+1] & \\
\sqsubseteq \text{'}\mathbf{doI1}\text{'}\| & \\
\quad \mathbf{do}\ j \neq n+1 \to & \\
\qquad j, ar : [j \neq n+1 \wedge zeroed(j-1, ar), & \\
\qquad\quad 0 \leqslant n-j+1 < n-j_0+1 \wedge zeroed(j-1, ar)] & \\
\quad \mathbf{od} & \\
\sqsubseteq \text{'}\mathbf{fassI}\text{'}\P & \\
\quad j, ar : [j \neq n+1 & \\
\qquad \wedge\ zeroed(j-1, ar), j \neq n+1 \wedge zeroed(j, ar)]; & [\lhd] \\
\quad j := j+1 & \\
\sqsubseteq \text{'}\mathbf{assI}\text{'**} & \\
\quad ar := ar \oplus \{j \mapsto 0\}. &
\end{array}$$

We included the proof obligations as part of the annotations of the refinement steps; here is a summary of those obligations (the trivial arithmetic ones have been discarded):

$$\begin{array}{l}
zeroed(0, ar) \\
zeroed(j-1, ar) \wedge j = n+1 \Rightarrow zeroed(n, ar) \\
j \neq n+1 \wedge zeroed(j-1, ar) \Rightarrow j \neq n+1 \\
\qquad\qquad\qquad\qquad \wedge\ zeroed(j, ar \oplus \{j \mapsto 0\}).
\end{array}$$

Obviously, the second predicate follows from Leibniz's law; the first and third are simple properties of *zeroed*.

Summarizing our development, we have that

$$\begin{array}{l}
Init \\
\sqsubseteq \\
\quad |[\mathbf{var}\ j \mid 1 \leqslant j \leqslant n+1 \bullet \\
\qquad j := 1; \\
\qquad \text{do}\ j \neq n+1 \to \\
\qquad\quad ar := update(ar, j, 0);
\end{array}$$

* The variable j will be used as a loop counter; thus it will range from the smallest element in the domain of *ar* to just after the highest.

† We introduce the semicolon in order to choose the loop invariant. At the beginning of the loop, and after each iteration, we will have zeroed all the blocks up, but not including j. The specification statement before the semicolon must establish the invariant, and the one after must be developed into the loop.

‡ Provided
$(zeroed(j-1, ar) \wedge 1 \leqslant j \leqslant n+1)[1/j]$.
If we set j to 1, we have zeroed no blocks.

§ Provided
$zeroed(j-1, ar) \wedge j = n+1 \Rightarrow zeroed(n, ar)$.
We need to put the invariant into the postcondition before we can apply the loop introduction rule.

‖ We must choose a variant $n-j+1$ will do: when we enter the loop with $j = 1$, we have n more iterations to perform.

¶ In the implementation of the body of the loop, we shall need to increment j. Since we started j with the value 1, the assignment to j must be done at the end of the loop.

** Provided
$j \neq n+1 \wedge zeroed(j-1, ar)$
$\Rightarrow (j \neq n+1 \wedge zeroed(j, ar))[(ar \oplus \{j \mapsto 0\})/ar]$.
The only thing left to do is to free the next element of *ar*, that is, the jth element.

```
        $j := j+1$
      od
    ]|.
```

This might be translated into Pascal as

```
procedure Init;
  for j := 1 to n do ar[j] := 0.
```

5.2 Translating numbers

We would like to develop an algorithm that converts numbers from a base β to the base 10. For an $n+1$ digit number, a solution that requires more than n multiplications is not acceptable.

The key to this development is to recall Horner's rule about evaluating polynomials:

$$\sum_{i=1}^{n} a_i \times x^{i-1} = H_{1,n}$$

where

$$H_{n,n} = a_n$$
$$H_{i,n} = a_i + x \times H_{i+1,n} \quad \text{for} \quad i < n.$$

Now, suppose that we have a number in base β with digits $a_n a_{n-1} \ldots a_2 a_1$; then our algorithm must satisfy the specification

$$d\!:\!\left[true, d = \sum_{i=1}^{n} a_i \times \beta^{i-1}\right].$$

Now, if we substitute β for x in the definitions of H, we obtain

$$G_{n,n} = a_n$$
$$G_{i,n} = a_i + \beta \times G_{i+1,n} \quad \text{for} \quad i < n$$

and our specification can be rewritten as

$$d\!:\![true, d = G_{1,n}].$$

The strategy for calculating the code for this algorithm is quite clear: we can develop a loop which varies the first index of G. It is easy enough to establish $G_{n,n}$, and we want to end up with $G_{1,n}$, so the loop counter is decreasing, and the invariant will involve $d = G_{j,n}$, for loop counter j.

```
$d\!:\![true, d = G_{1,n}]$
$\sqsubseteq$ var $j\!:1..n$ ●
  $d,j\!:\![true, d = G_{1,n}]$
$\sqsubseteq$ 'semI'
  $d,j\!:\![true, d = G_{j,n}]$;                                  [◁]
  $d,j\!:\![d = G_{j,n}, d = G_{1,n}]$                             [†]
$\sqsubseteq$ 'assI'
  $d,j := a_n, n$
† $\sqsubseteq$ 'sp'
  $d,j\!:\![d = G_{j,n}, d = G_{1,n} \wedge j = 1]$
$\sqsubseteq$ 'sp'
  $d,j\!:\![d = G_{j,n}, d = G_{j,n} \wedge j = 1]$
$\sqsubseteq$ 'doI'
  do $j \neq 1 \rightarrow$
    $d,j\!:\![j \neq 1 \wedge d = G_{j,n}, 0 \leqslant j < j_0 \wedge d = G_{j,n}]$
  od
$= d,j\!:\![(j \geqslant 0 \wedge d = G_{j+1,n})[j-1/j], (0 \leqslant j$
                                  $\leqslant j_0 \wedge d = G_{j,n})[j_0 - 1/j_0]]$
$\sqsubseteq$ 'lassI'
  $j := j-1$;
  $d,j\!:\![j \geqslant 0 \wedge d = G_{j+1,n}, 0 \leqslant j \leqslant j_0 \wedge d = G_{j,n}]$
$\sqsubseteq$ 'doI, con F'
  $d := a_j + x \times d$.
```

Thus, we have derived the following program:

```
|[var $j\!:1..n$ ●
   $d,j := a_n, n$;
   do $j \neq 1 \rightarrow$
     $j := j-1$;
     $d := a_j + x \times d$
   od
]|.
```

Acknowledgments

I am indebted to Carroll Morgan for first showing me the benefits of a refinement calculus; I am grateful for the many discussions on the subject that I have had with Paul Gardiner, Steve King and Jock McDoowi. I wish to thank the anonymous referees for their helpful comments. This paper is based on lectures given in Oxford, various industrial courses, and tutorials given at Noordwijkerhout in October 1991.

REFERENCES

1. R.-J. Back, *On the Correctness of Refinement Steps in Program Development*. Report A-1978-4, Department of Computer Science, University of Helsinki (1978).
2. R.-J. Back, *Correctness-preserving Program Refinements: Proof Theory and Applications*. Tract 131, Mathematisch Centrum, Amsterdam (1980).
3. R.-J. Back, *A Calculus of Refinement for Program Derivations*. Report Ser. A 54, Departments of Information Processing and Mathematics, Swedish University of Åbo, Åbo, Finland (1987).
4. R.-J. Back, *Procedural Abstraction in the Refinement Calculus*. Report Ser. A 55, Departments of Information Processing and Mathematics, Swedish University of Åbo, Åbo, Finland (1987).
5. S. M. Brien, P. H. B. Gardiner, P. J. Lupton and J. C. P. Woodcock, *A Semantics for Z*. Programming Research Group, Oxford University Computing Laboratory (1992).
6. E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, Hemel Hempstead (1976).
7. P. H. B. Gardiner and C. C. Morgan, Data refinement of predicate transformers. *Theoretical Computer Science*.
8. C. B. Jones, *Systematic Software Development using VDM*. Prentice-Hall, Hemel Hempstead (1986).
9. M. B. Josephs, *Formal Methods for Stepwise Refinement in the Z Specification Language*. Programming Research Group, Oxford University Computing Laboratory, Technical Report TR-1-86 (1988).
10. C. C. Morgan, The specification statement. *Transaction in Programming Language Systems*, **10** (3) (1988).
11. C. C. Morgan, Data refinement using miracles. *Information Processing Letters*, **26** (5), 243–246 (1988).
12. C. C. Morgan, Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, **11** (1988c).
13. C. C. Morgan and K. A. Robinson, Specification statements and refinement. *IBM Journal of Research and Development*, **31** (5) (1987).

14. J. M. Morris, Laws of data refinement. *Acta Informatica*, **26**, 287–308 (1989).
15. J. M. Morris, A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, **9** (3), 298–306 (1987).
16. E. Sekerinski, A calculus for predicative programming. *Proceedings of the 2nd International Conference on the Mathematics of Program Construction, Oxford, June 1992.* To be published in Lecture Notes in Computer Science, Springer Verlag, Heidelberg (1992).
17. J. M. Spivey, *The Z Notation: a Reference Manual.* Prentice-Hall International, London (1989).
18. J. C. P. Woodcock, A tutorial on the refinement calculus. In *VDM'91: Formal Software Development Methods.* Lecture Notes in Computer Science 552, pp. 79–140. Springer-Verlag, Heidelberg (1991).

Book Reviews

Y. Takefuji. *Neural Network Parallel Computing.* Kluwer Academic Publishers Group, Dordrecht, 1992, £44.25 ISBN 0-7923-9190-X

In his ackowledgements Takefuji admits to being inspired by the work of Hopfield and Tank on using neural network architectures to solve problems in optimisation. The book sets out many extensions to the approach where mathematical models of constraint satisfaction problems are implemented and tested in neural nets. The variety of topics that are addressed and the variety of methods used to convey the ideas are distinctive characteristics.

The first ten chapters deal with many hard optimisation problems such as N-queen problems and k-colorability problems. The remaining chapters deal with hardware implementations and mathematical derivations. Some Turbo PASCAL code is provided and each chapter has its own list of references. Most of the chapters have student exercises attached. The exercises are clearly an attempt to make this a text book, but I feel that students generally benefit most when model answers are supplied. However, after a cursory glance, I did wonder just how the average undergraduate might cope with Exercise 11.5 No. 4 – 'Survey research on silicon neural network implementations and optical implementations'.

Takefuji's general approach is to provide various gradient descent methods for solving constraint satisfaction problems. The aim is to construct something known as a motion equation which species a 'fabricated computational energy function' (p. 4). An artificial neural net is then developed to implement parallel gradient descent as a method to minimise the fabricated energy function. Various types of artificial neurons are considered, and each is defined relative to a different input/output function. Takefuji discusses the basic McCulloch and Pitts neuron defined as a binary threshold logic unit (TLU). An alternative to this is a unit using a sigmoid input/output function as studied by Hopfield. Nets comprising either sort of unit are then examined. Detailed comparisons revealed that the TLU net tended to converge faster than the sigmoid net, although the TLU net did exhibit unfortunate oscillatory behaviour. To overcome this, Takefuji discusses using units that employ an hysteresis McCulloch–Pitts input/output function. With this, the idea is to have a unit with essentially two thresholds. The upper threshold sets a value above which inputs must be for the unit to turn on, the lower threshold sets a value below which the unit turns off. Input values between the upper and lower bounds have no effect on the unit. Nets with these units no longer exhibited oscillation, but problems remained over how best to set the threshold values.

Although Takefuji discusses Boltzmann machines in passing, his favoured model is one known as the 'maximum neuron model'. This is a variant on the theme of a winner-take-all net. One of the advantages of these nets is that they are 'guaranteed to generate satisfactory solutions' (p. 181). Another is that 'turing coefficients parameters in the motion equation is not required'. From this last statement I am happy to conclude either that my rather slender grasp of mathematics lets me down, or that something has gone wrong in the typesetting.

Nevertheless, my general impression was that the book has been cobbled together. Some of it is rather too obviously the product of 'cut-and-paste', there are inconsistencies in style (compare the formatting of the references for Chapters 1 and 13), and some of the writing is dreadful. Moreover, the formula typesetting is untidy. This can be off-putting to those who find unpacking mathematical expressions into natural language daunting. Equation 1.8 (p. 9) is a case in point (see also the proof on p. 191).

This is truly a book for computer scientists with a strong background in mathematics. Although a wide range of topics is covered – from natural brains to VLSI chips – the book is perhaps going to fit most comfortably in the hands of applied mathematicians.

P. Quinlan
York

L. C. Paulsons. *ML for the Working Programmer*, Cambridge University Press. £27.50. ISBN 0 521 39022 2

Standard ML (henceforth referred to as SML) is a major influence in the design of programming languages. It is widely used in the research community and is increasingly used for teaching computer science. It is even beginning to find its way into the commercial world, both as a prototyping tool and as a delivery language. Therefore there is a significant demand for a good introductory text. Paulson's book, while not perfect, meets that demand well.

As the title suggests, Paulson aims his book at people who already know how to program and who want to use SML on real programs. His book is also suitable for advanced undergraduate teaching.

Paulson introduces the features of SML by example. His explanations are generally clear and form a good introduction to both functional and imperative programming with SML. Most of his examples are based on code that he has used himself, rather than purely illustrative code from the classroom. So the reader is shown how to write basic tools such as binary trees, priority queues, tree searches, parsers and pretty-printers in a functional style. Some 'lazy' data structures are also examined in detail. Paulson also gives syntax diagrams at the back of the book, which newcomers to SML will find useful.

Other examples introduce the basic ideas of first-order classical logic and the lambda-calculus, which are used in two major case studies once the presentation of the language is complete. Obviously these examples aren't directly useful to programmers from other fields, but Paulson is on home ground here, and presents his examples well. The code for many of the examples in the book is available by anonymous FTP.

There is a particularly strong need for a textbook that explains how to use the SML modules system. Paulson does a good job of presenting the basics. There is more that he could have said, but he shows enough for people to use the language to build real programs. Similarly, although I disagree with his discussion of abstract types in SML, his presentation is good enough for people to get things done.

Paulson also offers a chapter on formal reasoning about functional programs. He cover the ground well, and discusses both the limitations and the virtues of the techniques. However, I felt that he would have done better to integrate this discussion with the main text. Putting it in a separate chapter may make it rather indigestible to the 'working programmer' of the title.

Working programmers need efficient programs, and Paulson does discuss the efficiency of his examples. However, he doesn't present any techniques for analysing the efficiency of recursive programs. Perhaps surprisingly for a book aimed at imperative programmers, he also doesn't deal directly with questions such as 'How do I write a loop in functional language?'. Although the techniques are described, readers are left to find them for themselves.

Overall, I like the book. The treatment of the core language is very good – I could quibble with details, but these are mainly matters of personal taste and style. I have more disagreements with his treatment of modules, and he leaves room for a more comprehensive coverage of the modules system, but his presentation is both adequate in itself and better than the competition. Apart from this, my main criticism is that there is not enough discussion of when to use which features of the language. However, the examples provide a useful guide. In my opinion, this is the best general SML textbook currently available.

Dave Berry
Edinburgh