The Refinement Calculus, and Literate Development

Carroll Morgan

Programming Research Group, Oxford University, 11 Keble Road, Oxford OX1 3QD, UK

Abstract

The refinement calculus aims to present the development of imperative programs (among others) as a collection of part-abstract, part-concrete program fragments, each related to its predecessor by a mathematical notion of *refinement*. One of the benefits of that approach is the construction of a mathematically coherent history, as a program is developed, of the steps taken; it contains not only the (initial) specification and the (final) code, but also the design decisions in between.

If one views the development history, rather than simply the final code, as the principal product of program construction, then the parallel with Knuth's ideas on "Literate Programming" is clear. This contribution explores those connections, introducing the refinement calculus along the way, and illustrates the notion of a development history and the operations on it by means of a small functional program.

1 Literate Developments

Knuth argued that a computer program is more than simply its source code: it is a narrative explaining the development of the algorithm it embodies [6]. The refinement calculus for imperative programs further supports that view, and in this paper is used as an illustration of it.

1.1 Developments subsume code

High-level computer languages arose to free the programmer from the details of machine coding; as a result, machine- or assembler-code became the concern only of the specialist, or of certain kinds of optimisers. The high-level source code became the 'deliverable' of the project, and was carefully preserved; all subsequent activities necessary for constructing the actual working product — such as making machine code, integrating with subroutine libraries etc. — could be done by the computer itself.

These days we should have as little interest in the source code as we have for some time had in machine code: it is time to move on. We should no longer use *program* listings as the sole (reliable) specification of what a program in the field should be doing (but

perhaps isn't!) They are inadequate for that, and they are similarly limited as a starting point for enhancement.

We need program *developments*. What always results from making a program (but is not always recorded) is:

1. A sequence of steps from specification to program;
2. The reasons those steps were taken; and
3. The justifications for their correctness.

Let us call that a *development*.

We are not suggesting, at this stage, that developments are necessarily formal. Nor do we suggest that the three artefacts above are the whole story. The construction of the specification itself (often carried out as the development proceeds!) is however not our concern here.

But few people these days are under the illusion that developments (in the precise sense above) can or should reflect what actually happened during a project, any more than they believe that delivered programs are written systematically, at the rate of N lines per day, from line 1 to the end. Developments are for understanding what happened; they are not necessarily a record of what actually did happen.

For developments to replace programs as deliverables, they must of course be written down; but more than that it must be possible automatically to generate the program texts from them. The analogue of a compiler is needed.

Further, just as (some) compilers check the source-code for 'correctness' in so far as they are able (looking for type errors, uninitialised variables, unreachable code), so should the compiler's analogue be able to check developments for correctness.

1.2 An Example Development

In Figure 1 is a very small development, in the style of the *refinement calculus* [2, 8, 11]. Point (1) is just its *title*, written say on the outside of the book containing it. Point (2) is the original[1] specification; point (3) indicates a development step (in this example, the only one); point (4) motivates the step; and point (5) is the result.

The *specification* (2) — a multiple assignment — is written in a language well-understood by many people but not, alas, by many computers. That is in the nature of specifications: they strive for clarity at the expense, if necessary, of 'compilability'.

The precise nature of (3), the symbol $\sqsubseteq$, we return to below. The role of the *motivation* (4) is to record for subsequent readers the reason for this step. In this case, it reminds one that the 'obvious' $x := y; y := x$ is incorrect — because x would be overwritten before its value could be transferred to y.

The (compilable) code appears at (5); in this case, it is intended to be *Pascal* code[2]. The code *collector* (let us call it) when given the whole development as input would produce just the final portion (5) as output. That would, in turn, become the input to a Pascal compiler. We might never see that code; nor would we expect any error messages,

[1] So-called, although as mentioned above it might be very different from what was *originally* agreed between client and programmer.

[2] ...or at least close to it.

Swap two variables ←(1)

$x, y := y, x$ ←(2)

(3) → $\sqsubseteq$ "Temporary variable avoids overwrite" ←(4)

begin var t;
  $t := x$;
  $x := y$;
  $y := t$;
end ←(5)

Fig. 1. A small development

any more than compiler-produced assembler code should result in diagnostics from the assembler.

1.3 Refinement

We now return to (3), the symbol $\sqsubseteq$, and the notion of correctness. For programs P and Q, in languages like Pascal, we say that P is refined by Q, written $P \sqsubseteq Q$, if

(for all α: $wp(P, \alpha) \vdash wp(Q, \alpha)$)

holds.

A checking 'by hand' of the above development would then read (where $\alpha[v\backslash w]$ means 'w replacing all (free) v's in α')

$wp(\textbf{begin var } t; t := x; x := y; y := t \textbf{ end}, \alpha)$
iff "Definition **var**; and assuming α contains no t"
$(\forall t: : wp(t := x; x := y; y := t, \alpha))$
iff "Definition of ;"
$(\forall t: : wp(t := x, wp(x := y, wp(y := t, \alpha))))$
iff "Definition of :="
$(\forall t: : \alpha[y\backslash t][x\backslash y][t\backslash x])$
iff "Properties of substitution"
$(\forall t: : \alpha[y, x, t\backslash x, y, x])$
iff "t not free in body"
$\alpha[y, x, t\backslash x, y, x]$
iff "α contains no t"
$\alpha[y, x\backslash x, y]$
iff "Definition :="
$wp(x, y := y, x, \alpha)$.

That may seem rather a lot of work for such a simple development — but part of the reason the development is *simple* is that we've seen it before. (The fact that equivalence 'iff' is shown above is an accident of this example: entailment is all that is required.)

Certainly, remembering standard development steps that we have done before is the key to getting a code collector to do the checking for us. Once a step is checked, we call it a *refinement,* and keep it for later.

While the collector might not be able to calculate weakest preconditions, or establish equivalences in the predicate calculus, it can still check that a given step is an instance of one of a large number of stored refinements, each having been checked independently beforehand and kept for future (re)use.

1.4 A Refinement Rulebase

To support the checking of development steps one takes advantage of refinements previously validated and stored. Three examples are the following, chosen to apply to the development of the previous section:

Choose specific value An expression ? appearing on the right-hand side of an assignment may be replaced by any value whatever. (The assignment $y := ?$ means 'set y to any value'.)

For any expression F,

$$x, y := E, ?$$
$$\sqsubseteq$$
$$x, y := E, F.$$

Introduce local variable Any variable, not already appearing in the program, may be introduced as a new *local* variable and may then be set to any value.

For fresh variable t,

$$x := E$$
$$\sqsubseteq$$
$$\textbf{begin var } t;$$
$$\quad x, t := E, ?$$
$$\textbf{end.}$$

Leading assignment Programs of a certain form (given below) may have a given assignment statement factored out, placed before, and sequentially composed.

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

Using the refinements above, we can repeat the development of Sect. 1.2, where we indicate with $\triangleleft$ the piece of program next to be developed (letting the rest 'carry forward' around it):

$$x, y := y, x$$
$\sqsubseteq$ "Avoid overwrite"
$$\textbf{begin var } t;$$
$$\quad x, y, t := y, x, ? \qquad \triangleleft$$
$$\textbf{end}$$

$\sqsubseteq$ "Substitution"
$x, y, t := y, t[t\backslash x], ?$
$\sqsubseteq$ "Aim for *Leading assignment*"
$x, y, t := y, t[t\backslash x], x$
$\sqsubseteq$ "*Leading assignment*"
$t := x;$
$x, y := y, t$ ◁
$\sqsubseteq$ "t contains no x"
$x, y := y, t[x\backslash y]$
$\sqsubseteq$ "*Leading assignment*"
$x := y;$
$y := t.$

Each of the steps refers either to simple facts about substitution or to a refinement in our small database. All that the collector need check is that the laws have been properly instantiated — in this case, there are no weakest preconditions or logical implications to calculate.

A second point, fortuitously raised by the example, is that developments are not just sequences in which the code appears as the final element: they have a richer structure. In general they are trees, or even directed acyclic graphs, from which the code is collected by a recursive descent of the structure, assembling at each node the code resulting from its subtrees.

In the next section we make that more precise.

1.5 Code Collection

We now give a simple implementation of the above ideas, specialised to the notation of [9] and Pascal. The functional language Orwell [3] is used to describe it. (This 'implementation' is not intended for serious use, of course — its purpose is only to explain the issues arising from code collection in the refinement calculus.)

A development is a multi-way branching tree

```
> dev ::=  Code code
>        | Ref comment prog ref [dev]
```

where we are not specific (here) about the structure of code, comments or programs:

```
> code    == string
> comment == string
> prog    == string
```

They are just character strings.

Note. *These notes give some explanation, for those unfamiliar with Orwell-style functional languages, of the constructions used in the functional programs of this section. The style is to interpolate the relevant symbols (for example ':=' in the sentence following) immediately after their meaning (in this case 'is declared').*

The type `dev` *is declared* `::=` *to have two alternative forms, indicated by the tags* `Code` *and* `Ref`*. The first alternative is of type* `code`*; the second has four components, of types* `comment`, `prog`, `ref`, *and* `[dev]`*. The first three of those components are character strings* `string`*; the last is a list* `[···]` *of (sub)developments.*

Thus a development is either code already (`Code`) or it is a commented refinement step (`Ref`), that takes a program via a refinement rule to a structure containing a number of subsequent developments.

The type `ref` itself contains sufficient information in its values only to identify which instantiation, of which refinement rule, justifies the step:

```
> ref ::=  CV
>        | ILV var
>        | LA var expr
>        | Eq
>
> var  == string
> expr == string
```

Thus a step is either

- Choosing a specific **V**alue for ?;
- Introducing a **L**ocal **V**ariable (whose name is of type `var`, a string thus);
- Factoring out a **L**eading **A**ssignment (for which is noted the name of the variable assigned to, and the expression assigned to it); or
- A trivial rewriting of the program into some other, **Eq**ual to the first.

Note that the type `ref` does not contain enough information itself to apply a rule: `ref` values merely select from a database of refinements, provided separately.

With only the above definitions, we can describe the collection process by the function `collect` below:

```
> collect :: dev -> code
> collect (Code c) = c
>
> collect (Ref c p r ds) = apply r cs, if check refdb r p ps
> where cs = map collect ds
>       ps = map get_prog ds
>       get_prog (Code c) = c
>       get_prog (Ref c p r ds) = p
```

Note. *The function* `collect` *is of type* `::` *function* `->`*, with source* `dev` *and target* `code`*. Its result is given by cases over the two possible forms of its argument: either* `Code c` *for some code* `c`*, or* `Ref c p r ds` *for comment* `c`*, program* `p`*, refinement rule* `r`*, and list of subdevelopments* `ds`*. The standard function* `map` *is used, for example, in the phrase* `map collect ds` *to form a list* `cs` *of codes, by applying* `collect` *to each element of the list* `ds` *of developments.*

Collecting a development produces code. A trivial development, code already, just produces that code. Collecting a development beginning with a refinement step involves more:

- Function **check** is used to check the validity of the step: the current program **p** is compared, in the context of a refinement database **refdb** and the name and parameters of the refinement step **r** selected from it, with the immediately-resulting subprograms **ps**; and
- Function **apply** generates, from the code **cs** resulting from the subdevelopments, the code of the current one.

Note that **check** refers to the subprograms (before collection), whereas **apply** refers to the code resulting from the subdevelopments.

We do not give the details of **check** here; but we can give the definition of **apply**:

```
> apply :: ref -> [code] -> code
>
> apply (ILV v) [c] = "begin var " ++ v ++ "; " ++ c ++ " end"
> apply CV [c] = c
> apply (LA v e) [c] = v ++ ":= " ++ e ++ "; " ++ c
> apply Eq [c] = c
```

Note. *In* `"begin var " ++ v`, *the (constant) string* `"begin var "` *is concatenated* `++` *with the string (variable)* `v`.

The values of type **ref** do contain enough information to insert programming-language constructs, into the code, that are appropriate to the particular refinement step taken.

Then one may verify that `collect d`, where

```
> d = Ref "Avoid overwrite" "x,y:=y,x" (ILV "t") [
>     Ref "Substitution" "x,y,t:= y,x,?" Eq [
>     Ref "Aim for Leading assignment" "x,y,t:=y,t[t\x],?" CV [
>     Ref "Leading assignment" "x,y,t:=y,t[t\x],x" (LA "t" "x") [
>     Ref "t contains no x" "x,y:=y,t" Eq [
>     Ref "Leading assignment" "x,y:=y,t[x\y]" (LA "x" "y") [
>     Code "y:=t"]]]]]]
```

gives the program (with indentation added)

```
begin var t;
  t:=x; x:=y; y:=t
end
```

Note. *The phrase* `[Code "y:=t"]` *is the singleton list of developments, containing the one element* `Code "y:=t"`.

The program contains no comments, and indeed does not need its indentation, because humans need never read it. Neither need humans read the definition of **d** above: it is

the machine-readable form of the development. A recursive procedure **print**, similar to **collect**, given **d** could produce a listing of the development as in Sect. 1.4.

With the simple refinements so far used as illustration, structure of a development is still essentially linear[3], even though the code is distributed through it. Section 1.8 gives rules corresponding to many other refinements of [9], where it is apparent that a development is a multi-way branching tree.

1.6 Making Developments

A convenient way of generating values of type **dev** (like **d** of Sect. 1.5) is again to use a program — making a third, after **collect** and **print**.

That program's main functions would be to construct developments in a way as convenient as possible for the developer: maintaining and displaying the tree structure, suggesting steps to take, and checking as the development is made that the steps are valid. That last would usurp the check within **collect** just as **collect** itself usurps the syntax checking during subsequent compilation. The call to **check** during collection would be unnecessary.

The program for making developments, call it **develop**, is a *refinement editor* [12]. (Compare *syntax-directed* editors for making programs.)

The checks required during the use of **develop** are of course not as simple as may have been suggested by all the above. Rule **WP** in the appendix, for example, corresponds to the rule *weaken precondition* of [9], which reads

Weaken precondition The precondition of a specification may be weakened.

Provided $pre \vdash pre'$,

$$w\colon [pre\ ,\ post] \sqsubseteq w\colon [pre'\ ,\ post]$$

where the proviso is an entailment in the predicate calculus.[4] Naturally, these can be quite complex, tending to be the verification conditions with which we are all very familiar. Their proofs could be delegated to yet another program (*eg.* [1]), though it has proved in practice that some simplification is best performed by **develop** itself.

1.7 Conclusion

A brief sketch has been given of the way in which the ideas of Hoare [5], Wirth [13], Dijkstra [4] and Knuth [6] appear in and inform the use of the refinement calculus, and of how the organisation of program developments is favourably affected.

Developments similar to those we have described above should form one component of the collection of artefacts associated with the making of any program, whether large or small. The program's source code need not.

[3] ... and thus in this example, sub*lists* of developments aren't really necessary.

[4] In [9], the entailment symbol is a triply-barred right arrow.

1.8 Further Refinement Rules

The following declarations are given to cover more completely the rules of [9], and some familiarity with it may be necessary to absorb this section fully. They have been slightly altered so that Pascal-like code results (rather than guarded commands [4]): alternations (**if**), iterations (**do**) are affected.

Note. *In this section we assume rather more familiarity with Orwell also, forgoing further explanations of its constructs.*

Further differences, mainly to do with procedures and their parameters, are beyond us here; background may be found in [7]. But note that call by *value-result* is used, rather than call by *var*.

```
> ref ::=  ...
>        | WP proviso
>        | SP proviso
>        | SC
>        | Con con
>        | Alt1 guard
>        | Alt2 guard
>        | Iter guard
>        | Proc name
>        | Rec name con body
>        | Param [param]
>        | Call
>
> guard   == string
> con     == string
> proviso == string

> param ::=   Value var expr
>           | ValueResult var var
>           | Result var var
```

The `Proc` alternative is used to introduce a procedure with name `n`. The body of the procedure is given as the first subdevelopment, and the calling program is the second: thus either or both may subsequently be developed. The `check` function, or equivalently `develop`, must when processing `Call` determine that the program replaced by just a name (of a suitable procedure) does in fact correspond to a procedure introduction higher up the development tree. This can be accomplished by building up an environment of name-program pairs, augmented by `Proc` steps, where the program in each case is the procedure body *before* subsequent development. (See function `get_prog` in Sect. 1.5.)

The `Rec` alternative introduces recursion. Unlike procedure introduction, the name-program pair is retained in the refinement step, and is not subsequently refined. The 'calling program' is refined, of course, and subsequent `Call` steps in it have access to the recursion name-program pair in exactly the same way as for procedures. (And that is why `p` appears in the development structure even though it is ignored by `collect` below:

it is used by check.) The con component is the logical constant used to force decrease of the variant.

The extra definitions for apply are as follows:

```
> apply (WP 1) [c] = c
> apply (SP 1) [c] = c
> apply SC [c0,c1] = c0 ++ "; " ++ c1
> apply Con [c0] = c0
>
> apply (Alt1 g) [c] =    "if " ++ g
>                       ++ " then begin " ++ c
>                       ++ " end"
>
> apply (Alt2 g) [c0,c1] =    "if " ++ g
>                           ++ " then begin " ++ c0
>                           ++ " end else begin " ++ c1
>                           ++ " end"
>
> apply (Iter g) [c] =    "while " ++ g ++ " do begin "
>                       ++ c
>                       ++ " end"
>
> apply (Proc n) [c0,c1] =    "begin procedure " ++ n ++ "; "
>                           ++ "begin " ++ c0 ++ " end; "
>                           ++ c1
>                           ++ " end"
>
> apply (Rec n p) [c] = "re " ++ n ++ ". " ++ c ++ " er"
>
> apply (Param ps) [c] = "(" ++ c ++ ")[" ++ lefts ++ "\" ++ rights "]"
>
> where lefts  = commas (map left ps)
>       rights = commas (map right ps)
>       left (Value v e) = "value " ++ v
>       left (ValueResult v1 v2) = "value result " ++ v1
>       left (Result v1 v2) = "result " ++ v1
>       right (Value v e) = e
>       right (ValueResult v1 v2) = v2
>       right (Result v1 v2) =  v2
>
> apply Call [c] = c
>
>
> commas = between ","
> between s = foldr f ""
>             where f xs ys = xs ++ s ++ ys
```

2 An Example Development: Square Root

2.1 Abstract Program

We are given a natural number s; 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. Here is our abstract program:

$$\textbf{var } r, s\text{: nat} \bullet$$

$$r := \lfloor \sqrt{s} \rfloor . \tag{i}$$

Our program is abstract, though an assignment, because in this case study we assume that neither $\sqrt{}$ nor $\lfloor \cdots \rfloor$ is code. Our aim in development will be to remove them from the program, replacing them with more basic constructions.

2.2 Routine Steps

These first steps remove the square-root $\sqrt{}$ and floor $\lfloor \cdots \rfloor$ functions from the program by drawing on their mathematical definitions.

$$\begin{array}{ll} \sqsubseteq & \text{“}\textit{simple specification}\text{”} \\ & r\text{: } [r = \lfloor \sqrt{s} \rfloor] \\ \sqsubseteq & \text{“definition } \lfloor\ \rfloor\text{”} \\ & r\text{: } [r \leq \sqrt{s} < r + 1] \\ \sqsubseteq & \text{“definition } \sqrt{}\text{; } r \in \mathsf{nat}\text{”} \\ & r\text{: } \left[r^2 \leq s < (r+1)^2\right] . \end{array} \tag{ii}$$

Now compare (i) and (ii). The first is written for the client: it uses powerful operators, leading to clear and succinct expression. The second, having appealed to the mathematical definitions of those operators, is written for the programmer: it exposes the structure he needs to exploit.

2.3 The Key Step

If we replace $r + 1$ in the postcondition by a new local variable q, we might keep $r^2 \leq s < q^2$ invariant while bringing q and r together. That is a common technique: replace an expression by a local variable, then develop code which makes the local variable equal to the expression it replaced. The development step is the following:

$$\begin{array}{ll} \sqsubseteq & \textbf{var } q\text{: nat} \bullet \\ & q, r\text{: } \left[r^2 \leq s < q^2 \wedge r + 1 = q\right] . \end{array}$$

Having separate bounds on s gives us more scope: initially, q and r could be far apart; finally, we should establish $r + 1 = q$. That suggests an iteration, and the next few steps are routine: we introduce an abbreviation (I for the invariant), establish the invariant (initialisation), and introduce an iteration whose body maintains it.

The abbreviation $I \mathrel{\widehat{=}} \cdots$ is written as a decoration of the refinement: it is available in the development from that point on. The symbol $\triangleleft$ identifies the part of the program to be refined in the very next step.

$$
\begin{array}{ll}
\sqsubseteq & I \mathrel{\widehat{=}} r^2 \le s < q^2 \bullet \\
& q,r\colon [I \wedge r+1 = q] \\
\sqsubseteq & q,r\colon [I]\,; \qquad \text{(iii)} \\
& q,r\colon [I\ ,\ I \wedge r+1 = q] \qquad \triangleleft \\
\sqsubseteq & \text{“invariant } I\text{, variant } q-r\text{”} \\
& \textbf{do } r+1 \neq q \ \rightarrow \\
& \quad q,r\colon [r+1 \neq q\ ,\ I\ ,\ q-r < q_0 - r_0] \qquad \triangleleft \\
& \textbf{od.}
\end{array}
$$

Note that the invariant bounds the variant below (as required). We leave the refinement of (iii) to Ex. 1.

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

$$
\begin{array}{ll}
\sqsubseteq & \textbf{var } p\colon \mathsf{nat} \bullet \\
& p\colon [r+1 < q\ ,\ r < p < q]\,; \qquad \text{(iv)} \\
& q,r\colon [r < p < q\ ,\ I\ ,\ q-r < q_0 - r_0]\,. \qquad \triangleleft
\end{array}
$$

Note that *remove invariant* has removed I from (iv).

We intend to re-establish $r^2 \le s < q^2$ in the postcondition by an assignment $q := p$ or $r := p$. The first requires a precondition $s < p^2$ (at least as strong as that: recall the *assignment* law); the second requires $s \ge p^2$. That suggests a case analysis, leading to an alternation as follows:

$$
\begin{array}{ll}
\sqsubseteq & \textbf{if } \ s < p^2 \ \rightarrow\ q\colon \left[s < p^2 \wedge p < q\ ,\ I\ ,\ q < q_0\right] \qquad \text{(v)} \\
& [\!] \ \ s \ge p^2 \ \rightarrow\ r\colon \left[s \ge p^2 \wedge r < p\ ,\ I\ ,\ r_0 < r\right] \qquad \text{(vi)} \\
& \textbf{fi}
\end{array}
$$

(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.

Now only (iv) is left, and it has many refinements: $p := r+1$ and $p := q-1$ are two. But a faster decrease in the variant — hence a 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. But we need not list the entire program, now or ever; and we need not document it.

2.4 Epilogue

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.

Now we know, though, that 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 English text. 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. 2. Could we choose some other value of p on the indicated line? The development, shown in Fig. 3, 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:

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

No comment could ever have that credibility.

```
|[ var q: nat •
   q, r := s + 1, 0;
   do r + 1 ≠ q →
      |[ var p: nat •
         p := (q + r) ÷ 2;                                    ←
         if s < p² → q := p
         [] s ≥ p² → r := p
         fi
      ]|
   od
]|
```

Fig. 2. Square root code

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 English text 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').

$$
\begin{array}{ll}
 & \textbf{var } r, s: \textbf{nat} \bullet \\
 & r := \lfloor \sqrt{s} \rfloor \\
= & r: \left[r^2 \le s < (r+1)^2\right] \\
\sqsubseteq & \textbf{var } q: \textbf{nat} \bullet \\
 & q, r: \left[r^2 \le s < q^2 \wedge r + 1 = q\right]. \\
\sqsubseteq & I \mathrel{\widehat{=}} r^2 \le s < q^2 \bullet \\
 & q, r: [I \wedge r + 1 = q] \\
\sqsubseteq & q, r: [I]; \qquad \text{(iii)} \\
 & q, r: [I \ , \ I \wedge r + 1 = q] \qquad \triangleleft \\
\sqsubseteq & \textbf{do } r + 1 \neq q \ \rightarrow \\
 & \quad q, r: [r + 1 \neq q \ , \ I \ , \ q - r < q_0 - r_0] \qquad \triangleleft \\
 & \textbf{od} \\
\sqsubseteq & \textbf{var } p: \textbf{nat} \bullet \\
 & p: [r + 1 < q \ , \ r < p < q]; \qquad \text{(iv)} \\
 & q, r: [r < p < q \ , \ I \ , \ q - r < q_0 - r_0] \qquad \triangleleft \\
\sqsubseteq & \textbf{if } \ s < p^2 \ \rightarrow \ q: \left[s < p^2 \wedge p < q \ , \ I \ , \ q < q_0\right] \qquad \text{(v)} \\
 & \ [] \ \ s \ge p^2 \ \rightarrow \ r: \left[s \ge p^2 \wedge r < p \ , \ I \ , \ r_0 < r\right] \qquad \text{(vi)} \\
 & \textbf{fi}
\end{array}
$$

(iii) $\sqsubseteq$ $q, r := s + 1, 0$

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

(v) $\sqsubseteq$ $q := p$

(vi) $\sqsubseteq$ $r := p$

Fig. 3. Square root development

What of testing and debugging? They are still necessary. The code of the *Paragraph* case study [9, Chapter 20] was collected, transliterated by hand,[5] and then tested.

But there was an error in the transliteration: a multiple assignment $x, y := E, F$ was translated in error to $x := E$; $y := F$ (the expression F contained x). Such errors are easily detected, and even avoided, by incorporating the checks in an automated transliterator.

A second error was due to a single mistake in the development, and that was removed by checking the refinement steps in detail without reading the English text. Thus it is the *development* that is debugged: the thought of checking the code itself was shockingly unpleasant.

Those were the only errors, and 'it ran third time': mathematical rigour cannot eliminate mistakes entirely. But it can drastically reduce their likelihood.

2.5 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?

[5] into Modula-2.

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

$$\begin{array}{l} \sqsubseteq\ I \mathrel{\widehat{=}} r^2 \leq s \bullet \\ \quad r\colon [I \wedge s < (r+1)^2]. \end{array}$$

Refine that to code. Compare the efficiency of the result with the code of Figure 3.

3 A Selection of Refinement Laws

A selection of refinement laws is given for the development of imperative programs. The notation is based on Dijkstra's language of guarded commands [4] and the specifications of [9].

The laws appear in alphabetical order by name.

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

$$\begin{array}{ll} & \{pre'\};\ w\colon [pre\ ,\ post] \\ = & w\colon [pre' \wedge pre\ ,\ post]. \end{array}$$

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

$$\begin{array}{ll} & w\colon [pre\ ,\ post];\ [post'] \\ = & w\colon [pre\ ,\ post \wedge post']. \end{array}$$

Law *alternation*

$$\begin{array}{ll} & \{(\bigvee i \bullet G_i)\}\ prog \\ \sqsubseteq & \textbf{if}\ (\llbracket i \bullet G_i \rightarrow \{G_i\}\ prog)\ \textbf{fi}. \end{array}$$

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

$$w, x\colon [pre\ ,\ post] \quad \sqsubseteq \quad w\colon = E.$$

Abbreviation *assumption*

$$\{pre\} \quad \mathrel{\widehat{=}} \quad :[pre\ ,\ \mathsf{true}].$$

Abbreviation *coercion*

$$[post] \quad \widehat{=} \quad : [\mathsf{true} \,,\, post] \,.$$

Law *contract frame*

$$w, x\colon [pre \,,\, post] \quad \sqsubseteq \quad w\colon [pre \,,\, post[x_0 \backslash x]\,] \,.$$

Abbreviation *default precondition*

$$w\colon [post] \quad \widehat{=} \quad w\colon [\mathsf{true} \,,\, post] \,.$$

Law *establish assumption*
An assumption after a specification can be removed after suitable strengthening of the precondition.

$$\begin{array}{ll} & w\colon [pre \,,\, post] \,;\, \{pre'\} \\ = & w\colon [pre \wedge (\forall w \bullet post \Rightarrow pre')\,[w_0 \backslash w] \,,\, post] \,. \end{array}$$

Law *following assignment*
For any term E,

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

Law *initialised iteration*

$$\begin{array}{ll} & w\colon [pre \,,\, inv \wedge \neg G] \\ \sqsubseteq & w\colon [pre \,,\, inv] \,; \\ & \mathbf{do}\ G \rightarrow w\colon [G \wedge inv \,,\, inv \wedge (0 \leq V < V_0)]\ \mathbf{od}. \end{array}$$

Law *introduce assumption*

$$[post] \quad \sqsubseteq \quad [post]\ \{post\}.$$

Law *introduce coercion*
skip is refined by any coercion.

$$\mathbf{skip} \quad \sqsubseteq \quad [post].$$

Law *introduce local block*
If x is fresh, then

$$w\colon [pre\ ,\ post] \quad \sqsubseteq \quad |[\ \mathbf{var}\ x\colon T; \mathbf{and}\ inv \bullet w, x\colon [pre\ ,\ post]\ ]|.$$

Law *introduce logical constant*
If $pre \vdash (\exists c\colon T \bullet pre')$, and c is a fresh name (it does not occur in w, pre, $post$), then

$$\begin{array}{ll} & w\colon [pre\ ,\ post] \\ \sqsubseteq & \mathbf{con}\ c\colon T\bullet \\ & w\colon [pre'\ ,\ post]\,. \end{array}$$

Law *iteration*
Let inv, the *invariant*, be any formula; let V, the *variant*, be any integer-valued expression. Then

$$\begin{array}{ll} & w\colon [inv\ ,\ inv \wedge \neg(\bigvee i \bullet G_i)] \\ \sqsubseteq & \mathbf{do} \\ & \quad ([]i \bullet G_i \rightarrow w\colon [inv \wedge G_i\ ,\ inv \wedge (0 \leq V < V_0)]\ ) \\ & \mathbf{od}. \end{array}$$

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

Law *leading assignment*
For any expression E,

$$\begin{array}{ll} & w, x\colon [pre[x\backslash E]\ ,\ post[x_0\backslash E_0]] \\ \sqsubseteq & x\colon = E; \\ & w, x\colon [pre\ ,\ post]\,. \end{array}$$

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

Law *merge annotations*

$$\begin{array}{l} \{pre'\}\ \{pre\} = \{pre' \wedge pre\} \\ [post]\ [post'] = [post \wedge post']. \end{array}$$

Law *multiple substitution*
Provided neither f nor g occurs in F or G,

$$\begin{array}{ll} & prog[\mathbf{par1}\ f\colon T\backslash F][\mathbf{par2}\ g\colon U\backslash G] \\ \sqsubseteq & prog[\mathbf{par1}\ f\colon T, \mathbf{par2}\ g\colon U\backslash F, G]. \end{array}$$

The substitutions **par1** and **par2** may be any combination of **value**, **result**, and **value result**.

Law *recursion*
For any program *prog*,

$$\begin{array}{ll} & prog \\ \sqsubseteq & \textbf{re}\ N \mathrel{\widehat{=}} \{0 \le E < v\} prog \bullet \\ & \{E = v\} prog. \end{array}$$

The integer-valued expression E is the *variant*. Both N and v are fresh names; and furthermore v is a logical constant, which must be removed to reach code.

Abbreviation *recursive procedure*

$$\begin{array}{ll} & \textbf{procedure}\ P \mathrel{\widehat{=}} prog[N \backslash P] \\ \mathrel{\widehat{=}} & \textbf{procedure}\ P \mathrel{\widehat{=}} \textbf{re}\ N \bullet prog\ \textbf{er}. \end{array}$$

Law *remove alternation*

$$\textbf{if}\ \mathsf{true} \rightarrow branch\ \textbf{fi} \quad = \quad branch.$$

Law *remove assumption*
Any assumption is refined by **skip**.

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

Law *remove coercion*

$$\{pre\}\,[pre] \quad \sqsubseteq \quad \{pre\}.$$

Law *remove false guard*

$$\begin{array}{ll} & \textbf{if}\ ([]i \bullet G_i \rightarrow branch_i) \\ & []\ \mathsf{false} \rightarrow branch \\ & \textbf{fi} \\ = & \textbf{if}\ ([]i \bullet G_i \rightarrow branch_i)\ \textbf{fi}. \end{array}$$

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

$$w\colon [pre\ ,\ inv\ ,\ post] \quad \sqsubseteq \quad w\colon [pre\ ,\ post]\,.$$

Law *remove logical constant*
If X occurs nowhere in program *prog*, then

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

Law *rename formal parameter*
If l does not occur in *prog*,

$$prog[\mathbf{par}\ f{:}T\backslash A] \quad = \quad prog[f\backslash l][\mathbf{par}\ l{:}T\backslash A].$$

Law *result assignment*
Provided f does not occur in E or F,

$$\begin{array}{l} \quad w, a := E, F \\ \sqsubseteq\ [\mathbf{result}\ f{:}T\backslash a]\bullet \\ \quad w, f := E, F. \end{array}$$

Law *result specification*
If f does not occur in *pre*, and neither f nor f_0 occurs in *post*, then

$$\begin{array}{l} \quad w, a{:}\ [pre\ ,\ post] \\ \sqsubseteq\ [\mathbf{result}\ f{:}T\backslash a]\bullet \\ \quad w, f{:}\ [pre\ ,\ post[a\backslash f]]\ . \end{array}$$

Law *right-distribution of assignment over alternation*

$$\begin{array}{l} \quad x := E;\ \mathbf{if}\ ([]i \bullet G_i \rightarrow branch_i)\ \mathbf{fi} \\ = \ \mathbf{if}\ ([]i \bullet G_i[x\backslash E] \rightarrow x := E;\ branch_i)\ \mathbf{fi}. \end{array}$$

Law *select true guard*

$$\begin{array}{l} \quad \mathbf{if}\ ([]i \bullet G_i \rightarrow branch_i) \\ \quad []\ \mathsf{true} \rightarrow branch \\ \quad \mathbf{fi} \\ \sqsubseteq\ branch. \end{array}$$

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

$$a[i := E][j] \quad \widehat{=} \quad \begin{array}{l} E \text{ when } i = j \\ a[j] \text{ when } i \neq j. \end{array}$$

Law *sequential composition*

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

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

Law *sequential composition*
For fresh constants X,

$$\begin{array}{l} \quad w, x\colon [pre\ ,\ post] \\ \sqsubseteq\ \mathbf{con}\ X\bullet \\ \quad x\colon [pre\ ,\ mid]\,; \\ \quad w, x\colon [mid[x_0\backslash X]\ ,\ post[x_0\backslash X]]\,. \end{array}$$

The formula *mid* must not contain initial variables other than x_0.

Law *simple specification*

$$w\colon= E \quad = \quad w\colon [w = E_0]\,,$$

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

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

$$w\colon [pre\ ,\ post] \sqsubseteq \mathbf{skip}.$$

Abbreviation *specification invariant*

$$w\colon [pre\ ,\ inv\ ,\ post] \quad \widehat{=} \quad w\colon [pre \wedge inv\ ,\ inv \wedge post]\,.$$

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

$$w\colon [pre\ ,\ post] \quad \sqsubseteq \quad w\colon [pre\ ,\ post']\,.$$

Law *tail recursion*

$$\begin{array}{l} \quad \mathbf{re}\ L\bullet \\ \qquad \mathbf{if}\ \neg G \rightarrow \mathbf{skip}\ [\!]\ G \rightarrow prog;\ L\ \mathbf{fi} \\ \quad \mathbf{er} \\ =\ \mathbf{do}\ G \rightarrow prog\ \mathbf{od}. \end{array}$$

Law *value assignment*

$$\begin{array}{l} \quad w := E[f\backslash A] \\ \sqsubseteq\ [\textbf{value}\ f{:}T\backslash A]\bullet \\ \quad w := E. \end{array}$$

The actual parameter A may be an expression, and it should have type T.

Law *value specification*
If *post* does not contain f, then

$$\begin{array}{l} \quad w{:}\ [pre[f\backslash A]\ ,\ post[f_0\backslash A_0]] \\ \sqsubseteq\ [\textbf{value}\ f{:}T\backslash A]\bullet \\ \quad w, f{:}\ [pre\ ,\ post], \end{array}$$

where A_0 is $A[w\backslash w_0]$.

Law *value-result assignment*

$$\begin{array}{l} \quad w, a := E[f\backslash a], F[f\backslash a] \\ \sqsubseteq\ (w, f := E, F)[\textbf{value result}\ f{:}T\backslash a]. \end{array}$$

Law *value-result specification*
If *post* does not contain f, then

$$\begin{array}{l} \quad w, a{:}\ [pre[f\backslash a]\ ,\ post[f_0\backslash a_0]] \\ \sqsubseteq\ [\textbf{value result}\ f{:}T\backslash a]\bullet \\ \quad w, f{:}\ [pre\ ,\ post[a\backslash f]]. \end{array}$$

Law *value-result specification*
If *post* does not contain a, then

$$\begin{array}{l} \quad w, a{:}\ [pre[f\backslash a]\ ,\ post[f_0, f\backslash a_0, a]] \\ \sqsubseteq\ [\textbf{value result}\ f{:}T\backslash a]\bullet \\ \quad w, f{:}\ [pre\ ,\ post]. \end{array}$$

Law *weaken precondition*

If $pre \vdash pre'$, then

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

References

1. Abrial, J.-R.: An informal introduction to the B tool. B.P. Project Report, Programming Research Group, Oxford University, 1986.
2. Back, R.-J.R.: A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
3. Bird, R.S., Wadler, P.: *An Introduction to Functional Programming.* Prentice-Hall, 1987.
4. Dijkstra, E.W.: *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.
5. Hoare, C.A.R.: An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.
6. Knuth, D.E.: Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
7. Morgan, C.C.: Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11(1):17–28, 1988. Reprinted in [10].
8. Morgan, C.C.: The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988. Reprinted in [10].
9. Morgan, C.C.: *Programming from Specifications.* Prentice-Hall, 1990.
10. Morgan, C.C., Robinson, K.A., Gardiner, P.H.B.: On the refinement calculus. Technical Report PRG–70, Programming Research Group, 1988.
11. Morris, J.M.: A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
12. Vickers, T.N.: An overview of a theorem proving assistant. *Australian Computer Science Communications*, 12(1):402–411, 1990.
13. Wirth, N.: Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, April 1971.