JAN 187 -0

Mechanized Support for Stepwise Refinement

Jan L.A. van de Snepscheut
California Institute of Technology

This note describes a notation for formula manipulation and an editor that provides support for the production of programs through the process of stepwise refinement.

1 Introduction

Stepwise refinement is the method of gradually developing programs from their specification through a number of steps. This method was first proposed by E.W. Dijkstra [2], [3], [4] and N. Wirth [8], [9]. As [8] puts it

> In each step, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language, and must therefore be guided by the facilities available on that computer or language. ...
>
> Every refinement step implies some design decisions. It is important that these decisions be made explicit, and that the programmer be aware of the underlying criteria and of the existence of alternative solutions.

Both authors give elegant and convincing examples of the application of this method. In both cases, however, the process is an informal one. In [1], R.J. Back lays a mathematical foundation under this process by viewing refinement as a partial order on state transformers.

Although stepwise refinement is a simple method, it is not widely used in practice because it is often tedious and, as a result, error-prone. In this note, we describe an editor that is geared to the production of programs via stepwise refinement by automating the tedious parts and by making explicit the transformations carried out in each step as well as the conditions under which they apply. Numerous systems support program transformation or theorem proving but (almost) none of them reduce the amount of labor required by the practicing programmer who uses the system. There is (almost) always some aspect of the mechanization that forces the programmer to pay attention to details that are only tangential to the program development itself. The driving force behind our design is to compete with paper and pencil, so to speak, by actually reducing the amount of work done by the programmer. The editor is called *proxac* for program and proof transformation and calculation.

2 Overview

The editor presents a number of windows, including a window that contains the text being edited and a window that contains the transformation rules that can be applied. For example, if the edit window contains the text

$$s = mss.(n+1) \;\; \wedge \;\; 0 \le n < N$$

then application of rule

$$mss.n = \mathbf{MAX}(j \mid 0 \le j \le n \triangleright mes.j)$$

transforms the text into

$$\begin{array}{ll} & s = mss.(n+1) \;\;\wedge\;\; 0 \le n < N \\ = & \{ \;\; mss[n := n+1] \;\; \} \\ & s = \mathbf{MAX}(j \mid 0 \le j \le n+1 \triangleright mes.j) \;\;\wedge\;\; 0 \le n < N \end{array}$$

(We will turn to the interpretation of these formulae later on.) In the current version of the proxac system, a rule is selected by clicking with the mouse on the rule (see [7] for details). The editor supports the tedious part of this rewriting in the sense that it matches the given text to the selected rule; it determines the "longest" subformula that matches one side of the rule (namely, $mss.(n+1)$ if variable n in the rule is replaced by $n+1$); it then carries out this substitution in the right-other side of the rule to produce the rewrite. The old and new lines are connected by the hint $mss[n := n+1]$ to indicate which rule was applied and which substitution was carried out. Including this information in the text helps in making the transformations explicit. The author of the text is the one who selects the rule that is being applied, the edit program carries out the other actions. Notice that the text being produced is in the format suggested by W.H.J. Feijen.

We have cheated a little bit in the example since we did not indicate in the rule that n is a variable and all the other quantities are constants. Also, transformation rules are applicable only under certain conditions; in this case the condition is $0 \leq n \leq N$. The full version of the transformation rule is, therefore, as follows.

$$\textbf{rule}\ mss : (n \mid 0 \leq n \leq N \triangleright mss.n = \mathbf{MAX}(j \mid 0 \leq j \leq n \triangleright mes.j))$$

In addition to the actions described earlier, the editor checks that the applicability condition is met. Since the transformation is applied in a conjunction where $0 \leq n < N$ is one of the terms, the condition holds and the rule applies.

We continue the example with one more rule.

$$\textbf{rule}\ split : (x, y, z \triangleright (x \leq y \leq z + 1) = (x \leq y \leq z \quad \vee \quad x \leq y = z + 1))$$

Rules can be viewed in different ways. The *split* rule is an algebraic identity, not a definition. But a rule like *mss* can be viewed as an explicit definition of function *mss*. The second view is a special case of the first view. We prefer the first view since it provides a great economy in formal labor, even though it has the danger of leading to inconsistencies (since the algebraic properties are postulated instead of proved).

Application of these rules leads to the following text.

$$\begin{array}{ll}
 & s = mss.(n+1) \quad \wedge \quad 0 \leq n < N \\
= & \{ \quad mss[n := n+1] \quad \} \\
 & s = \mathbf{MAX}(j \mid 0 \leq j \leq n+1 \triangleright mes.j) \quad \wedge \quad 0 \leq n < N \\
= & \{ \quad split[x := 0, y := j, z := n] \quad \} \\
 & s = \mathbf{MAX}(j \mid 0 \leq j \leq n \triangleright mes.j) \uparrow mes.(n+1) \quad \wedge \quad 0 \leq n < N \\
= & \{ \quad mss \quad \} \\
 & s = mss.n \uparrow mes.(n+1) \quad \wedge \quad 0 \leq n < N
\end{array}$$

Notice that the last step is the *mss* rule applied in the opposite direction. Also notice that the second transformation step produces term

$$s = \mathbf{MAX}(j \mid 0 \leq j \leq n \quad \vee \quad j = n+1 \triangleright mes.j)$$

but the editor reduces this further to

$$s = \mathbf{MAX}(j \mid 0 \leq j \leq n \triangleright mes.j) \uparrow mes.(n+1)$$

through an application of the range disjunction and one-point rules for quantification. It shows that $\uparrow$ is the infix operator that corresponds to quantifier **MAX** just like $\vee$ corresponds to $\exists$ and $+$ corresponds to $\sum$. These correspondences are not built into the editor; they are specified through the following statements.

$$\begin{array}{l}
\textbf{declare}\ \mathit{INFIX}\ 20\ \uparrow \\
\textbf{property}\ \mathit{ASSOCIATIVE}(\uparrow) \quad \wedge \quad \mathit{DUAL}(\uparrow) = \uparrow \\
\textbf{declare}\ \mathit{QUANTIFIER}\ \mathbf{MAX} \\
\textbf{property}\ \mathit{INFIXOPERATOR}(\mathbf{MAX}) \ = \ \uparrow
\end{array}$$

The first line declares $\uparrow$ to be an infix operator with precedence level 20. The second line states that it is associative and commutative (that is, it is its own dual). The third line declares quantifier **MAX** and the fourth line gives the correspondence between the two new operators. The associativity and commutativity of $\uparrow$ are necessary to make **MAX** a well-defined quantifier. They also enable a lot of simplifications that are automatically applied by the system. By writing the rules and properties in a small but rather general language instead of a richer language with more built-in facts, we gain the ability to extend the application domain of our editor to algebraic manipulations that were not necessarily foreseen. In particular, we show how it can be used to set up a calculus of stepwise refinement.

3 Refinement calculus

In this section, we develop a formalization of the refinement calculus within the framework of our transformation method. The refinement calculus introduced by R.J. Back in [1] is based on the weakest preconditions introduced in [4]. It is based on an ordering relation on programs, written as $s0 \sqsubseteq s1$ for programs $s0$ and $s1$ to denote that $s0$ can be refined by $s1$. Two properties are essential for stepwise refinement. The first is that $\sqsubseteq$ be reflexive and transitive because this justifies the fact that a sequence of steps can be used to refine a specification into an executable program. The second is monotonicity of the program constructs because this justifies that refining one subprogram by another refines the whole program. Notice that this view of refinement requires that programs and specifications be treated on equal footing. Hence, specifications are treated as programs, but we continue refining a program until it contains no specifications. (See the quote in section 1.)

As a first attempt, we may introduce some program constructs. For example, sequential composition will be denoted by semicolon and the empty statement by *skip*.

declare *INFIX* 0 $\sqsubseteq$

property $TRANSITIVE(\sqsubseteq) \quad \wedge \quad REFLEXIVE(\sqsubseteq)$

declare *INFIX* 1 ;

declare *skip*

property $UNIT(;) = skip$

property $ASSOCIATIVE(;)$

property $\forall(s0, s1, t0, t1 \mid s0 \sqsubseteq s1 \wedge t0 \sqsubseteq t1 \triangleright s0; t0 \sqsubseteq s1; t1)$

Notice that this does not provide a definition of ; even though it is claimed to be an associative operator. A definition-based style would have to prove this result from the definition, which would depend on the associativity of function composition. The last line states the monotonicity of sequential composition.

Though mathematically elegant, formalization of weakest preconditions leads to a complication in their practical use. The complication is due to the difference between program variables and mathematical variables. J.J. Lukkien provided the following example to illustrate the confusion that may arise. Suppose we want to prove the correctness of program

$$i := 100;\ DO\ i \neq 0 \rightarrow i := i - 1\ OD$$

with respect to precondition *true* and postcondition *true*. All we need to do is to prove termination. Using invariant $i \geq 0$ and bound function i, our proof obligation is to show that the conjunction of the invariant and the guard implies a decrease of the bound function, that is,

$$i \geq 0 \wedge i \neq 0 \wedge i = C \;\Rightarrow\; wp.(i := i - 1).(i < C)$$

for all constants C. Using a naive formalization, we may proceed as follows

$$\begin{array}{ll} & i \geq 0 \wedge i \neq 0 \wedge i = C \;\Rightarrow\; wp.(i := i - 1).(i < C) \\ = & \{\ i = C\ \} \\ & i \geq 0 \wedge i \neq 0 \wedge i = C \;\Rightarrow\; wp.(i := i - 1).(C < C) \\ = & \{\ \text{algebra}\ \} \\ & i \geq 0 \wedge i \neq 0 \wedge i = C \;\Rightarrow\; wp.(i := i - 1).\mathit{false} \\ = & \{\ \text{law of excluded miracle}\ \} \\ & i \geq 0 \wedge i \neq 0 \wedge i = C \;\Rightarrow\; \mathit{false} \\ = & \{\ \text{algebra}\ \} \\ & \neg(i = C > 0) \end{array}$$

and we are stuck. The problem, of course, is that one should not allow the substitution of C for i in the argument of wp. The solution is to distinguish between i on the left-hand side and i on the right-hand side by making both sides boolean functions instead of boolean scalars. In particular, the second argument of wp becomes a boolean function that maps argument i to the boolean value $i < C$. We write this function as $(i \triangleright i < C)$. In this way the problem disappears. Unfortunately, so does the practicality of the wp calculus. For example, the weakest precondition of statement $i := i - 1$ with respect to postcondition $i < C$ is written as $wp.(i := i - 1).(i \triangleright i < C)$. It becomes even worse when the statement is to be understood in a state where i is not the only program variable. If the program has variables i, j, and k, then the aforementioned precondition becomes $wp.(i := i - 1).(i, j, k \triangleright i < C)$. The size of the formula grows with the number of program variables and this greatly impacts its practical use.

In [6], C. Morgan provides an alternative formalization of the refinement calculus. It is based on the specification statement, written as $v : [pre, post]$, in which v is called the frame, and *pre* and *post* are the precondition and postcondition. Its effect is given as (see [6])

> If the initial state satisfies the precondition *then* change only the variables listed in the frame so that the resulting final state satisfies the postcondition.

The rules for calculating with specification statements do not involve wp's and thereby avoid the problem mentioned above.

The notation used for a specification statement is not that of an infix operator. It is a notation involving three arguments; the first is a list of variables, and the other two are single expressions. In our formalism, we write

notation $!LIST\ :\ [\ !\ ,\ !\ ]$

and, presently, we cannot express the restriction that the elements if the first list are variables. We have no need for expressing the semantics of the specification statement other than how it can be refined by other programs, as discussed below.

Using the specification statement, we postulate the following property of sequential composition.

$$v : [P, R] \ \sqsubseteq\ v : [P, Q];\ v : [Q, R]$$

for all predicates P, Q, and R. Next, we introduce the assignment statement. Assignment statement $x := E$ is a refinement of any specification statement that contains x in its frame (in addition to a possibly empty list of variables v) and such that the postcondition in which x is replaced by E is implied by the precondition. In our formalism, we write

$$v, x : [P, Q] \ \sqsubseteq\ x := E$$

for all v, x, E, P, and Q provided $P \ \Rightarrow\ Q[x := E]$. Finally, we introduce the loop construct. We write

notation $DO\ !\ \rightarrow !\ OD$

for a loop with one guarded command and we postulate

$$\begin{array}{l} \quad v : [P, P \ \wedge\ \neg b] \\ \sqsubseteq \\ \quad DO\ b \rightarrow v : [P \ \wedge\ b \ \wedge\ bf = BF, P \ \wedge\ bf < BF]\ OD \end{array}$$

provided $P \wedge b \ \Rightarrow\ bf > 0$. For our example, we can get away with a simpler form of the loop in which there is an integer variable that is increased in steps of one from one given value to another given value. Using a more specific refinement rule implies less work upon application since part of the proof obligations can be taken care of when constructing (or postulating) this rule. The rule we will use is given in the text below. For the sake of completeness we also list a rule for strengthening the postcondition and a rule for introducing local variables. The latter notation is a bit more complicated because it restricts the scope of local variables, an issue that we are not concerned with here. This text is the entire refinement calculus as far as we need it for the example. In other cases we need more rules and the full version is about four times the size of the short version listed here. Whenever we develop a new program, we want to use these refinement rules and definitions in the same way we want to use a module of procedures and definitions in a program. We use the same mechanism: rules and definitions can be collected in a module, and the module can be imported by another text.

```
module $refine$
notation $!LIST\ :\ [\ !\ ,\ !\ ]$
notation $DO\ !\ \rightarrow\ !\ OD$
notation $VAR\ ?LIST(v)\ BEGIN\ !(v)\ END$
declare $INFIX\ 2\ :=$
declare $INFIX\ 1\ ;$
property $ASSOCIATIVE(;)$
rule $StrengthenPost : (v, P, Q, R \triangleright$
          $v : [P, Q] \sqsubseteq v : [P, Q \wedge R])$
rule $Block : (v, w, P, Q \triangleright$
          $v : [P, Q] \sqsubseteq VAR\ w\ BEGIN\ w, v : [P, Q]\ END\ )$
rule $Assignment : (v, x, E, P, Q \mid P \Rightarrow Q[x := E] \triangleright$
          $v, x : [P, Q] \sqsubseteq x := E)$
rule $Semicolon : (v, P, Q, R \triangleright$
          $v : [P, R] \sqsubseteq v : [P, Q]; v : [Q, R])$
rule $SemicolonAssignment : (v, x, E, P, Q \triangleright$
          $v, x : [P, Q] \sqsubseteq v, x : [P, Q[x := E]]; x := E)$
rule $UpLoop : (v, i, pre, P, from, to \mid P \Rightarrow from \leq to \triangleright$
          $v, i : [pre,\ P \wedge i = to]$
      $\sqsubseteq$
          $v : [pre,\ P[i := from]]; i := from;$
          $DO\ i \neq to \rightarrow v : [P \wedge from \leq i < to,\ P[i := i + 1] \wedge from \leq i < to];$
                            $i := i + 1$
          $OD)$
property $\forall(s0, s1, t0, t1 \mid s0 \sqsubseteq s1 \wedge t0 \sqsubseteq t1 \triangleright s0; t0 \sqsubseteq s1; t1)$
property $\forall(b, s0, s1 \mid s0 \sqsubseteq s1 \triangleright DO\ b \rightarrow s0\ OD \sqsubseteq DO\ b \rightarrow s1\ OD\ )$
property $\forall(w, s0, s1 \mid s0 \sqsubseteq s1 \triangleright$
          $VAR\ w\ BEGIN\ s0\ END \sqsubseteq VAR\ w\ BEGIN\ s1\ END\ )$
```

Notice that we have included a rule, viz. *SemicolonAssignment*, that is strictly superfluous because it follows from the two rules that precede it. However, we often have a situation in which we know that a specification statement $v, x : [P, Q]$ will include an assignment $x := E$. By letting it be the last statement in a sequential composition, we compute specification $v, x : [P, Q[x := E]]$ preceding it so that the combination is a proper refinement. By writing the combination as a single rule, the proxac system will compute and simplify predicate $Q[x := E]$. If we use the *Semicolon* rule instead, the author has to postulate this predicate and the system will verify its use in the subsequent refinement steps. The additional rule reduces the author's work.

In the module that contains the definition, we might want to prove that the more specific loop rule follows from the general *Loop* rule. Such a proof is given here.

rule $\mathit{UpLoop} : (v, i, \mathit{pre}, P, \mathit{from}, \mathit{to} \mid P \;\Rightarrow\; \mathit{from} \le \mathit{to} \;\triangleright$

$$
\begin{array}{ll}
 & v, i : [\mathit{pre}, P \wedge i = \mathit{to}] \\
\sqsubseteq & \{\;\; \mathit{Semicolon}[P := \mathit{pre}, Q := P \wedge \mathit{from} \le i \le \mathit{to}, R := P \wedge i = \mathit{to}, v := (v,i)] \;\;\} \\
 & v, i : [\mathit{pre}, P \wedge \mathit{from} \le i \le \mathit{to}];\; v, i : [P \wedge \mathit{from} \le i \le \mathit{to}, P \wedge i = \mathit{to}] \\
\sqsubseteq & \{\;\; \mathit{SemicolonAssignment}[x := i, E := \mathit{from}, P := \mathit{pre}, \\
 & \qquad\qquad\qquad\qquad\quad Q := P \wedge \mathit{from} \le i \le \mathit{to}] \;\;\} \\
 & v : [\mathit{pre}, P[i := \mathit{from}]];\; i := \mathit{from};\; v, i : [P \wedge \mathit{from} \le i \le \mathit{to}, P \wedge i = \mathit{to}] \\
\sqsubseteq & \{\;\; \mathit{Loop}[P := P \wedge \mathit{from} \le i \le \mathit{to}, \mathit{bf} := \mathit{to} - i, v := (v,i), b := i \ne \mathit{to}] \;\;\} \\
 & v : [\mathit{pre}, P[i := \mathit{from}]];\; i := \mathit{from}; \\
 & DO\;\; i \ne \mathit{to} \rightarrow v, i : [\; P \wedge \mathit{from} \le i < \mathit{to} \wedge \mathit{to} - i = BF, \\
 & \qquad\qquad\qquad\qquad\; P \wedge \mathit{from} \le i \le \mathit{to} \wedge \mathit{to} - i < BF] \\
 & OD \\
\sqsubseteq & \{\;\; \mathit{SemicolonAssignment}[\;\; P := P \wedge \mathit{from} \le i < \mathit{to}, x := i, E := i + 1, \\
 & \qquad\qquad\qquad\qquad\qquad Q := P \wedge \mathit{from} \le i \le \mathit{to} \wedge \mathit{to} - i < BF] \;\;\} \\
 & v : [\mathit{pre}, P[i := \mathit{from}]];\; i := \mathit{from}; \\
 & DO\;\; i \ne \mathit{to} \rightarrow\;\; v : [P \wedge \mathit{from} \le i < \mathit{to}, P[i := i + 1] \wedge \mathit{from} \le i < \mathit{to}]; \\
 & \qquad\qquad\qquad\quad i := i + 1 \\
 & OD)
\end{array}
$$

Usage of this long version of the *UpLoop* rule is identical to usage of the version listed in the module text. The external view of a rule with a calculational body is that of a rule with the body reduced to its first and last line with a connective deduced from the sequence of connectives. In this reduction, transitivity of $\sqsubseteq$ is essential. After a rule has been written it is shown in abbreviated form in the rules window so that it can be applied by a mouse click.

Notice that we have now given a proof of the correctness of the *UpLoop* rule. The mechanism for developing the proof is identical to the mechanism for refining a program.

4 An example of a program derivation

In this section we illustrate the use of the refinement rules to derive a program from its specification. The program is well-known and so is its derivation. Our focus of attention is the support given by the proxac system.

In some steps of the proof above (and in some steps of program derivations below, but not in any other earlier step), some variables of rules cannot be determined by pattern matching. As a result, the author of the text will need to give the proxac system hints regarding these unresolved variables. In this section, we indicate hints by underlining them.

The programming problem is known as the *maximum segment sum* problem (see [5]). Given is an array a of $N \ge 0$ integers. A segment of the array is a contiguous subsequence of the array. A segment has a segment sum, viz. the sum of all its array elements. The problem is to write a program to determine the maximum segment sum. We formalize the problem as

module *mss*
import *refine*
declare *QUANTIFIER* $\sum$
property $\mathit{INFIXOPERATOR}(\sum) = +$
declare *INFIX* 20 $\uparrow$
property $\mathit{ASSOCIATIVE}(\uparrow) \ \wedge \ \mathit{DUAL}(\uparrow) = \uparrow \ \wedge \ \mathit{IDEMPOTENT}(\uparrow)$
declare *QUANTIFIER* **MAX**
property $\mathit{INFIXOPERATOR}(\mathbf{MAX}) = \uparrow$
property $0 \leq N$
rule $\mathit{mss} : (n \mid 0 \leq n \leq N \triangleright \mathit{mss}.n = \mathbf{MAX}(j \mid 0 \leq j \leq n \triangleright \mathit{mes}.j))$
rule $\mathit{mes} : (j \mid 0 \leq j \leq N \triangleright \mathit{mes}.j = \mathbf{MAX}(i \mid 0 \leq i \leq j \triangleright \mathit{sum}.i.j))$
rule $\mathit{sum} : (i, j \mid 0 \leq i \leq j \leq N \triangleright \mathit{sum}.i.j = \sum(h \mid i \leq h < j \triangleright a.h))$
edit $s : [\mathit{true}, s = \mathit{mss}.N]$

We recognize the rules that we had in section 2. We use $\uparrow$ for the infix maximum operator. The problem is to write a program for computing $\mathit{mss}.N$, that is, a program that refines $s : [\mathit{true}, s = \mathit{mss}.N]$. We will need a loop, and this will lead to a specification statement in the loop body that contains $\mathit{mss}.n$ in the precondition and $\mathit{mss}.(n+1)$ in the postcondition. Given the calculation in section 2, we know that the latter can be rewritten as $\mathit{mss}.n \uparrow \mathit{mes}.(n+1)$ which means that we are tempted to introduce $\mathit{mes}.(n+1)$ in the loop invariant. However, upon termination of the loop, $n = N$, and $\mathit{mes}.(N+1)$ is undefined. We must, therefore, decrease by one the argument of mes and calculate $\mathit{mes}.(n+1)$ from $\mathit{mes}.n$ when needed. The programming problem can now be formalized as finding a refinement for $n, r, s : [\mathit{true}, s = \mathit{mss}.n \ \wedge \ r = \mathit{mes}.n \ \wedge \ n = N]$. If we had not noticed the problem with undefinedness of $\mathit{mes}.(N+1)$, we would have proceeded with $\mathit{mes}.(n+1)$ in the invariant. We would get stuck later on where a step cannot be justified because

$$0 \leq n < N \ \Rightarrow \ 0 \leq n+1 < N$$

cannot be established. We would not have been led into undefined results!

$$\begin{array}{ll}
 & s : [\mathit{true}, s = \mathit{mss}.N] \\
\sqsubseteq & \{ \ \mathit{Block}[v := s, \underline{w := (n, r)}, P := \mathit{true}, Q := s = \mathit{mss}.N] \ \} \\
 & \mathit{VAR}\ n, r\ \mathit{BEGIN}\ n, r, s : [\mathit{true}, s = \mathit{mss}.N]\ \mathit{END} \\
\sqsubseteq & \{ \ \mathit{StrengthenPost}[v := (n, r, s), P := \mathit{true}, Q := s = \mathit{mss}.N, \\
 & \qquad\qquad\qquad \underline{R := r = \mathit{mes}.N \wedge n = N}] \ \} \\
 & \mathit{VAR}\ n, r\ \mathit{BEGIN}\ n, r, s : [\mathit{true}, s = \mathit{mss}.N \wedge r = \mathit{mes}.N \wedge n = N]\ \mathit{END} \\
= & \{ \ n = N \ \} \\
 & \mathit{VAR}\ n, r\ \mathit{BEGIN}\ n, r, s : [\mathit{true}, s = \mathit{mss}.n \wedge r = \mathit{mes}.n \wedge n = N]\ \mathit{END}
\end{array}$$

Notice how the second step introduces $Q \wedge R$ in which R in turn is a conjunction. Since conjunction is associative, no parentheses surround R. We have found these kind of aspects instrumental in keeping down the amount of detail that the author has to deal with and, hence, the number of steps needed to complete a program derivation. We now focus attention on the latter specification statement and ignore the surrounding block. When doing so, the system keeps track of the context in which this narrowing of attention occurs.

```
   $n, r, s : [true, s = mss.n \wedge r = mes.n \wedge n = N]$
$\sqsubseteq$   { $UpLoop[v := (r,s), i := n, pre := true, P := s = mss.n \;\wedge\; r = mes.n,$
                 $\underline{from := 0}, to := N]$ }
   $r, s : [true, s = mss.0 \;\wedge\; r = mes.0]; n := 0;$
   $DO\ n \neq N \rightarrow r, s : [s = mss.n \;\wedge\; r = mes.n \;\wedge\; 0 \le n < N,$
                              $s = mss.(n+1) \;\wedge\; r = mes.(n+1) \;\wedge\; 0 \le n < N];$
                      $n := n + 1$
   $OD$
$\sqsubseteq$   {    $r, s : [true, s = mss.0 \;\wedge\; r = mes.0]$
        $=$     { $mss[n := 0]$ }
           $r, s : [true, s = mes.0 \;\wedge\; r = mes.0]$
        $\sqsubseteq$   { $SemicolonAssignment[v := r, \underline{x := s, E := r}, P := true,$
                                              $Q := s = mes.0 \;\wedge\; r = mes.0]$ }
           $r, s : [true, r = mes.0]; s := r$
        $=$   { $mes[j := 0]$ }
           $r, s : [true, r = sum.0.0]; s := r$
        $=$   { $sum[i := 0, j := 0]$ }
           $r, s : [true, r = 0]; s := r$
        $\sqsubseteq$   { $Assignment[v := s, \underline{x := r, E := 0}, P := true, Q := r = 0]$ }
           $r := 0; s := r$
     }
   $r := 0; s := r; n := 0;$
   $DO\ n \neq N \rightarrow r, s : [s = mss.n \;\wedge\; r = mes.n \;\wedge\; 0 \le n < N,$
                              $s = mss.(n+1) \;\wedge\; r = mes.(n+1) \;\wedge\; 0 \le n < N];$
                      $n := n + 1$
   $OD$
```

Notice that the above calculation contains a nested calculation. The step to replace $r, s : [true, s = mss.0 \;\wedge\; r = mes.0]$ by $r := 0;\ s := r$ consists of five steps by itself. Replacing them in the context where that specification statement occurs is justified by the monotonicity of sequential composition. We continue by narrowing attention to the specification statement in the loop body. We will need two more rules; they are not related to refinement but to ranges in quantifications. Since we have both ranges of the form $0 \le i \le j$ and of the form $i \le h < j$, we have two split rules.

rule $split : (x, y, z \triangleright (x \le y < z+1) = (x \le y < z \;\vee\; x \le y = z))$
rule $split : (x, y, z \triangleright (x \le y \le z+1) = (x \le y \le z \;\vee\; x \le y = z+1))$

We continue the refinement.

```
   $r, s : [s = mss.n \;\wedge\; r = mes.n \;\wedge\; 0 \le n < N,$
         $s = mss.(n+1) \;\wedge\; r = mes.(n+1) \;\wedge\; 0 \le n < N]$
$=$   { $mss[n := n+1]$ }
   $r, s : [s = mss.n \;\wedge\; r = mes.n \;\wedge\; 0 \le n < N,$
         $s = \mathbf{MAX}(j \mid 0 \le j \le n+1 \triangleright mes.j) \;\wedge$
         $r = mes.(n+1) \;\wedge\; 0 \le n < N]$
$=$   { $split[x := 0, y := j, z := n]$ }
```

$$
\begin{array}{ll}
 & r,s : [s = mss.n \;\wedge\; r = mes.n \;\wedge\; 0 \le n < N, \\
 & \quad s = \mathbf{MAX}(j \mid 0 \le j \le n \triangleright mes.j) \uparrow mes.(n+1) \;\wedge \\
 & \quad r = mes.(n+1) \;\wedge\; 0 \le n < N] \\
= & \{ \; mss \; \} \\
 & r,s : [s = mss.n \;\wedge\; r = mes.n \;\wedge\; 0 \le n < N, \\
 & \quad s = mss.n \uparrow mes.(n+1) \;\wedge\; r = mes.(n+1) \;\wedge\; 0 \le n < N] \\
= & \{ \; r = mes.(n+1) \; \} \\
 & r,s : [s = mss.n \;\wedge\; r = mes.n \;\wedge\; 0 \le n < N, \\
 & \quad s = mss.n \uparrow r \;\wedge\; r = mes.(n+1) \;\wedge\; 0 \le n < N] \\
\sqsubseteq & \{ \; \mathit{SemicolonAssignment}[v := r, \underline{x := s, E := s \uparrow r}, \\
 & \qquad P := s = mss.n \;\wedge\; r = mes.n \;\wedge\; 0 \le n < N, \\
 & \qquad Q := s = (mss.n) \uparrow r \;\wedge\; r = mes.(n+1) \;\wedge\; 0 \le n < N] \; \} \\
 & r,s : [s = mss.n \;\wedge\; r = mes.n \;\wedge\; 0 \le n < N, \\
 & \quad s \uparrow r = mss.n \uparrow r \;\wedge\; r = mes.(n+1) \;\wedge\; 0 \le n < N]; \\
 & s := s \uparrow r \\
= & \{ \quad r = mes.(n+1) \\
 & \quad = \quad \{ \; mes[j := n+1] \; \} \\
 & \quad\;\; r = \mathbf{MAX}(i \mid 0 \le i \le n+1 \triangleright sum.i.(n+1)) \\
 & \quad = \quad \{ \; split[x := 0, y := i, z := n] \; \} \\
 & \quad\;\; r = \mathbf{MAX}(i \mid 0 \le i \le n \triangleright sum.i.(n+1)) \uparrow (sum.(n+1).(n+1))) \\
 & \quad = \quad \{ \; sum[i := n+1, j := n+1] \; \} \\
 & \quad\;\; r = \mathbf{MAX}(i \mid 0 \le i \le n \triangleright sum.i.(n+1)) \uparrow 0 \\
 & \quad = \quad \{ \; sum[j := n+1] \; \} \\
 & \quad\;\; r = \mathbf{MAX}(i \mid 0 \le i \le n \triangleright \sum(h \mid i \le h < n+1 \triangleright a.h)) \uparrow 0 \\
 & \quad = \quad \{ \; split[x := i, y := h, z := n] \; \} \\
 & \quad\;\; r = \mathbf{MAX}(i \mid 0 \le i \le n \triangleright \sum(h \mid i \le h < n \triangleright a.h) + a.n) \uparrow 0 \\
 & \quad = \quad \{ \; sum[j := n] \; \} \\
 & \quad\;\; r = \mathbf{MAX}(i \mid 0 \le i \le n \triangleright sum.i.n + a.n) \uparrow 0 \\
 & \quad = \quad \{ \; \mathit{factor} \; \} \\
 & \quad\;\; r = (\mathbf{MAX}(i \mid 0 \le i \le n \triangleright sum.i.n) + a.n) \uparrow 0 \\
 & \quad = \quad \{ \; mes[j := n] \; \} \\
 & \quad\;\; r = (mes.n + a.n) \uparrow 0 \\
 & \} \\
 & r,s : [s = mss.n \;\wedge\; r = mes.n \;\wedge\; 0 \le n < N, \\
 & \quad s \uparrow r = mss.n \uparrow r \;\wedge\; r = mes.n + a.n \;\wedge\; 0 \le n < N]; \\
 & s := s \uparrow r \\
\sqsubseteq & \{ \; \mathit{Assignment}[v := s, \underline{x := r, E := (r + a.n) \uparrow 0}, \\
 & \qquad P := s = mss.n \;\wedge\; r = mes.n \;\wedge\; 0 \le n < N, \\
 & \qquad Q := s \uparrow r = mss.n \uparrow r \;\wedge\; r = mes.n + a.n \uparrow 0 \;\wedge\; 0 \le n < N] \; \} \\
 & r := (r + a.n) \uparrow 0; \; s := s \uparrow r
\end{array}
$$

The last-but-one step in the subcalculation is a step labeled *factor* and this is one of the built-in transformations. However, since we did not specify that addition distributes over maximum, the proxac system is unable to verify the correctness of this transformation and will print a question asking

$$
\text{Context implies: } \mathbf{MAX}(i \mid 0 \le i \le n \triangleright sum.i.n + a.n) = \\
\mathbf{MAX}(i \mid 0 \le i \le n \triangleright sum.i.n) + a.n \qquad ?
$$

The author of the text can decide to add the distribution property, to prove it, or to ignore the question.

When we widen the focus again from the specification statement in the loop body, we end up with the text

```
$VAR\ n, r$
$BEGIN$
  $r := 0;\ s := r;\ n := 0;$
  $DO\ n \neq N \rightarrow r := (r + a.n) \uparrow 0;\ s := s \uparrow r;\ n := n + 1\ OD$
$END$
```

and this program solves the problem at hand.

5 Conclusion

The total text of the program derivation is quite long, much longer than the program text itself. This observation is often used as an argument against the use of stepwise refinement or against formal methods. The derivation consists of a total of 31 steps, 8 of them being narrowing and widening the focus of attention. Of the remaining 23 steps, 16 steps require no hint at all. The 7 hints that had to be given have been underlined. These hints are the only input given to the system in addition to each mouse click that selects a rule and triggers its application. As a result, the total input is comparable in size to the resulting program and not to the derivation. One major benefit of using this system is that design decisions have been made explicit. Another major benefit is that all steps have been mechanically verified. We feel that this derivation shows that the use of a formal system for stepwise refinement of programs puts no extra burden on the programmer, and competes well with paper and pencil. Of course, we have used a set of rules that constitute the refinement calculus, but this is an investment that is amortized over the development of many programs. We have also written explicitly what the specification of the problem is. We don't think that a responsible programmer delivers a program without a specification, so this does not constitute extra work.

The transformation rules we have used are rather elementary. One can come up with more complicated rules that correspond to many steps in our present repertoire. This reduces the number of steps to complete a program derivation; however, the increase in the number of rules may make it harder to use them.

One can view the transformations as the commands of a programming language for formula manipulation. The transformations that we have described here, correspond to the elementary commands. We have used the notation of functions for describing those rules. By extending the notation with function composition, we construct composite transformation commands. By extending the notation with conditionals and a fixpoint operator, we obtain a complete programming language. These extensions allow us to construct what are sometimes called tactics. Tactics and their semantics are beyond the scope of the present paper.

6 Acknowledgement

The `proxac` editor can be found in `ftp` directory `jan/proxac` on `cs.caltech.edu.` A more detailed description can be found in [7] and an up-to-date version thereof can be found in the same `ftp` directory. Writing this editor and developing the notations used was, and is, a challenging undertaking. Diana Finley was instrumental in getting this project underway. Greg Davis contributed many ideas and helped get the program to the point where it actually became usable. My thanks go to both of them.

References

[1] R.J.R. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, University of Helsinki, 1978. Report A-1978-4.

[2] E.W. Dijkstra. A Constructive Approach to the Problem of Program Correctness. *BIT*, 8:174–186, 1968.

[3] E.W. Dijkstra. Notes on Structured Programming. In O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming.* Academic Press, 1971.

[4] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[5] D. Gries. A Note on the Standard Strategy for Developing Loop Invariants and Loops. *Science of Computer Programming*, 12:207–214, 1982.

[6] C. Morgan. *Programming from Specifications.* Series in Computer Science (C.A.R. Hoare, ed.). Prentice-Hall International, 1990.

[7] J.L.A. van de Snepscheut. JAN 183. Proxac: an Editor for Program Transformation. Technical Report CS 93-33, California Institute of Technology, 1993.

[8] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14:221–227, 1971.

[9] N. Wirth. *Systematic Programming.* Prentice-Hall, 1973.