

Programming by Expression Refinement: a Sequence of Examples

Joseph M. Morris

Department of Computing Science
University of Glasgow
Glasgow G12 8QQ
Scotland, UK

Abstract. We describe by a sequence of examples a method, which we call expression refinement, for deriving programs from specifications. The method consists of making specifications using a rich notation for expressions; the expressions are then subjected to value-preserving transformations in which the rich notations are replaced by more primitive ones that are part of the implementation language. This leads to a style of programming that makes much use of recursive functions rather than loops and invariant relations. The technique is part of a larger effort to develop a fully formal and practical calculus of programming. Initial experience suggests that expression refinement is potentially a more attractive approach for formal programming than one relying principally on traditional approaches based on refinement of statements.

1. Introduction

We take the view that a specification language is rather like a programming language except it has much fancier constructs and notations that facilitate ease of expression, although they may be impossible to implement. We regard a programming language as the implementable subset of the specification language. A specification language is richer than its constituent programming language in two regards: it has more expressive statements and richer data types. An example of an expressive but unimplementable statement is the nondeterministic assignment statement, which is a formal equivalent of "assign a value to variable x so that predicate P holds after the assignment" [1, 2, 3]. With respect to data, we admit in a specification language a much wider repertoire of operations than we are used to seeing in programming languages – such as all sorts of quantifiers – as well as a wider class of data types – such as sets, bags, and unbounded sequences. Programming consists in transforming a given specification by repeatedly subjecting small components of it to mathematical meaning-preserving transformations, until another specification emerges that is wholly expressed in the restricted notation of the programming language.

What we understand by "stepwise refinement" accords with this view, although its practitioners may not always recognise this because the surrounding specification language is informal, and the manipulations by which the program is arrived at are therefore limited in their formality. In so far as programming by stepwise refinement has been done with some formality, e.g. [4, 5, 6, 7], the preferred method has been to channel most of the manipulations through refinement of statements. A different way to go about programming is to shift the attention from the statements of the specification to the expressions, an approach that concerns us here: we illustrate a style of programming that puts the main thrust of the programming effort into rewriting expressions step by step until we have formulated them constructively. This is a style of programming — let's call it "expression refinement" — that leads to much use of recursive functions where otherwise we might have used loops.

We illustrate expression refinement by solving the following sequence of related programming problems:

- *The nearest smaller neighbour:* For each element in a given sequence of integers we are asked to compute its nearest smaller neighbour; to ensure well-definedness of the problem we postulate the presence of minus infinity at either end of the sequence.
- *The largest rectangular area in a histogram:* Compute the area of the largest rectangle that fits within the boundaries of a histogram viewed as a sequence of abutting fat bars of varying heights standing on a horizontal line.
- *The largest all-true submatrix:* For a given boolean matrix compute the area of the largest submatrix all of whose elements are *true*.

We are using the examples as a vehicle to illustrate our approach to calculating a program; the solutions as such are not of interest to us.

2. Basic notation

Most of the notations and heuristics we employ, and all the manipulations, are standard. We review them briefly here.

Function application is denoted by an infix dot which has the highest operator precedence. Finite sequences are regarded as partial functions on an initial segment of the natural numbers: a sequence b of length N has elements $b.0, b.1, \dots, b.(N-1)$. The subsequence of b between indices i and j , $0 \leq i \leq j \leq N$ and upper bound excluded, is denoted by $b(i..j)$ — $b(i..i)$ equals the empty sequence; $b(0..N)$ equals b ; and $b(i..j).0$ equals $b.i$ when $i < j$. The exclusion of the upper bound in the preceding makes for more convenient manipulations: for example, $b(i..j) = b(i..k) + b(k..j)$ where $+$ is sequence concatenation and $0 \leq i \leq k \leq j \leq N$. The domain of b is denoted by $\text{dom}.b$. For x an integer and b a sequence of integers we let $x \leq b$ stand for $(\forall i: i \in \text{dom}.b: x \leq b.i)$, and similarly for other relational operators. Trivially $x \leq \varepsilon$ where ε denotes the empty sequence, and $x \leq b+c \equiv x \leq b \wedge x \leq c$ for c also a sequence.

$(k: P.k: f.k)$ denotes the bag (multiset) containing an occurrence of $f.k$ for each k satisfying predicate $P.k$ (k is a dummy variable). $\text{min}.B$ yields the minimum of bag B of integers and equals ∞ if B is empty. Infix **min** yields the minimum of its integer arguments. A useful law is

$$\text{min}.(k: P.k: f.k) = \text{min}.(k: P.k \wedge Q.k: f.k) \text{ min}.(k: P.k \wedge \neg Q.k: f.k)$$

where $Q.k$ is another predicate; replacing an occurrence of the left-hand side of this equation with its right-hand side is called "range splitting". A similar sort of law is

$$\min.(k,l: P.k \wedge Q.k.l: f.k.l) = \min.(k: P.k: \min.(l: Q.k.l: f.k.l))$$

— its left-to-right application is called "nesting". For any j satisfying predicate $P.j$

$$\min.(k: P.k: k) = \min.(k: k \leq j \wedge P.k: k)$$

and

$$\min.(k: k=j \wedge P.k: f.k) = f.j$$

— left-to-right application is called "range contraction" and "one-point" respectively. We may supply a sequence b (of the right type) wherever a bag is expected, the argument being treated as though it were the bag ($i: i \in \text{dom}.b: b.i$). We define \max and **max** analogously to \min and **min**, respectively.

3. The nearest smaller neighbour

We first write down the specification for the nearest smaller neighbour problem, and then we'll explain it; for any natural N :

```

NSN0: c: sequence(N) of integer
      {b ∈ sequence(N) of integer}
      (||i: 0 ≤ i < N:
        [|
          ls: integer = i:integer. min.(j: 0 ≤ j ≤ i ∧ b.i ≤ b(j..i): j) •
          rs: integer = i:integer. max.(j: i < j ≤ N ∧ b.i ≤ b(i..j): j) •
          c.i := if ls.i=0 → rs.i
                [] ls.i≠0 →
                  if i-(ls.i-1) ≤ rs.i-i → ls.i-1
                  [] i-(ls.i-1) ≥ rs.i-i → rs.i
                fi
          fi
        ])
      )

```

The names and types of the given variables — here just sequence variable c of length N — are given above the horizontal line. The specification proper is given below the line and consists of assumptions and a statement of the desired effect. The assumption is given as a so-called assert statement, and here just says that b denotes a sequence of integers of length N ; by omission, the value of c is not significant and so c is being used as an output variable. The desired effect in the present case is given as a single concurrent assignment meaning "let $c.i$ have the value of the **if...fi** expression for each i in the range $0 \leq i < N$ ". Two abbreviations for parametrised integer expressions — ls and rs — are introduced with scope delimited by $[|$ and $]|$; in each case the parametrised expression denoted is sandwiched between an equality sign and a fat dot. For e an expression possibly containing free variable i , let $e(i \leftarrow k)$ denote e with all occurrences of i replaced by k . Formally the meaning of $[| f: integer = i:integer. e \bullet s|]$, where e stands for an expression and s stands for a statement possibly containing applications of f to an integer argument, is a copy of s with each $f.k$ in s replaced by $e(i \leftarrow k)$ — assuming no name clashes occur. Because the abbreviations have a local scope and so are not externally visible there is no obligation on the implementor in any way to preserve or implement the abbreviations. Consider rs : $rs.i$ is the maximum j such that each of $b.i, b.(i+1), \dots, b(j-1)$ is at least $b.i$. This is almost equivalent to saying that $rs.i$ is the least j greater than i such that $b.j < b.i$ — "almost" because in the case that there is no element smaller than $b.i$ to its right then $rs.i$ is N , and of course there is no $b.N$.

However, if we postulate the presence of a fictional minus infinity at N , and we do, then $b.(rs.i)$ is indeed the nearest element smaller than and to the right of $b.i$. ls is defined symmetrically, so that $b.(ls.i-1)$ is the nearest element smaller than and to the left of $b.i$, except that if $ls.i$ is zero then $b.i$ has no smaller element to its left. It should now be evident that the expression assigned to $c.i$ in the specification is the index of the element nearest to and smaller than the element at index i .

We could implement NSNO quite routinely: the N -fold \parallel is easily implemented with a loop, and we could replace the bodies of ls and rs with routine iterative calculations. But we sense that ls -values for successive elements are closely related, and similarly for rs -values, and we might exploit this to make a more efficient solution. The obvious strategy is to save the ls - and rs -values as we compute them, in the belief that they will help us to calculate later values efficiently, and the simple way to organise that is to calculate them all in advance:

```

NSN1:   $\parallel$   $x, y$ : sequence( $N$ ) of integer;
        ( $\parallel$   $i$ :  $0 \leq i < N$ :
           $\parallel$   $ls$ : integer =  $i$ : integer.  $\min.(j: 0 \leq j \leq i \wedge b.i \leq b(j..i): j) \bullet$ 
               $x.i := ls.i$ 
           $\parallel$ 
        );
        ( $\parallel$   $i$ :  $0 \leq i < N$ :
           $\parallel$   $rs$ : integer =  $i$ : integer.  $\max.(j: i < j \leq N \wedge b.i \leq b(i..j): j) \bullet$ 
               $y.i := rs.i$ 
           $\parallel$ 
        );
        ( $\parallel$   $i$ :  $0 \leq i < N$ :
           $c.i :=$  if  $x.i = 0 \rightarrow y.i$ 
               $\square$   $x.i \neq 0 \rightarrow$ 
                  if  $i - (x.i - 1) \leq y.i - i \rightarrow x.i - 1$ 
                   $\square$   $i - (x.i - 1) \geq y.i - i \rightarrow y.i$ 
                  fi
              fi
        )
     $\parallel$ 

```

The equivalence of the two specifications is evident. The problem is solved if we can find constructive equivalents of the function definitions. Let us concentrate on $ls.i$, beginning by writing down the elementary properties that follow immediately from its definition (it will turn out that L2 isn't used, but we don't know that yet):

L0:	$0 \leq ls.i \leq i$	$(0 \leq i < N)$
L1:	$b.i \leq b(ls.i..i)$	$(0 \leq i < N)$
L2:	$b.i \leq b(j..i) \equiv ls.i \leq j$	$(0 \leq j \leq i < N)$

```

     $ls.i$ 
="definition"
     $\min.(j: 0 \leq j \leq i \wedge b.i \leq b(j..i): j)$  (i)
="We're on the lookout for  $ls.(i-1)$  – so isolate the case  $j=i$  by range
splitting with predicate  $j=i$ "
     $\min.(j: 0 \leq j \leq i-1 \wedge b.i \leq b(j..i): j) \mathbf{min} \min.(j: j=i \wedge b.i \leq b(j..i): j)$ 
="one-point rule;  $b.i \leq b(i..i)$ "
     $\min.(j: 0 \leq j \leq i-1 \wedge b.i \leq b(j..i): j) \mathbf{min} i$ 
="To introduce  $i-1$  in place of the final  $i$  in  $b(j..i)$  we have to exclude the
case  $i=0$ ; when  $i=0$  the bag of the  $\min$ -term is empty"
    if  $i=0 \rightarrow i$ 

```

```

    [] i>0 → min.(j: 0≤j≤i-1 ∧ b.i ≤ b(j..i): j) min i
    fi
="definition of b.i ≤ b(j..i); the guard i>0 ensures b.(i-1) is defined"
    if i=0 → i
    [] i>0 → min.(j: 0≤j≤i-1 ∧ b.i ≤ b(j..i-1) ∧ b.i ≤ b.(i-1): j) min i
    fi
="min theory — when b.i > b.(i-1) the bag is empty"
    if i=0 → i
    [] i>0 → if b.i > b.(i-1) → i
                [] b.i ≤ b.(i-1) → min.(j: 0≤j≤i-1 ∧ b.i ≤ b(j..i-1): j) min i
    fi
fi
="min theory — i-1 is an element of the bag"
    if i=0 → i
    [] i>0 → if b.i > b.(i-1) → i
                [] b.i ≤ b.(i-1) → min.(j: 0≤j≤i-1 ∧ b.i ≤ b(j..i-1): j)
    fi
fi

```

Looking back through the derivation we see that the min-term above is the same as (i) with two occurrences of i in (i) — the first and third — replaced with $i-1$, and so we are invited to parametrise those two occurrences on our way to completing a recursive definition of $ls.i$. To this end, we distinguish the unchanging occurrence of i from the other two occurrences in (i) by defining, for $0 \leq k \leq i < N$,

$$h.i.k \equiv \min.(j: 0 \leq j \leq k \wedge b.i \leq b(j..k): j)$$

Obviously $ls.i = h.i.i$, and

```

    h.i.k
="repeating the derivation above, but in the presence of the extra parameter k"
    if k=0 → k
    [] k>0 → if b.i > b.(k-1) → k
                [] b.i ≤ b.(k-1) → min.(j: 0≤j≤k-1 ∧ b.i ≤ b(j..k-1): j)
    fi
fi
="definition of h"
    if k=0 → k
    [] k>0 → if b.i > b.(k-1) → k
                [] b.i ≤ b.(k-1) → h.i.(k-1)
    fi
fi

```

It is clear that the recursion above is well-founded. However, each invocation makes the absolute minimum amount of progress, and so we should press on looking for something better, knowing that we still have L0, L1, and L2 to exploit. We will try to refine $h.i.(k-1)$ further, assuming — see the guards in the preceding expression —

$$k > 0 \wedge b.i \leq b.(k-1)$$

```

    h.i.(k-1)
="definition"
    min.(j: 0≤j≤k-1 ∧ b.i ≤ b(j..k-1): j)

```

```

="We will try range contraction with  $j = ls.(k-1)$  for which we need the truth of
 $0 \leq ls.(k-1) \leq k-1 \wedge b.i \leq b(ls.(k-1)..k-1)$ : the first conjunct holds by
L0(with  $i:=k-1$ ), while the second holds by the guard  $b.i \leq b.(i-1)$  and
L1(with  $i:=k-1$ )"
  min.(j:  $0 \leq j \leq ls.(k-1) \wedge b.i \leq b(j..k-1)$ : j)
="sequence theory"
  min.(j:  $0 \leq j \leq ls.(k-1) \wedge b.i \leq b(j..ls.(k-1)) \wedge b.i \leq b(ls.(k-1)..k-1)$ : j)
="b.i  $\leq b(ls.(k-1)..k-1)$  by preceding hint but one"
  min.(j:  $0 \leq j \leq ls.(k-1) \wedge b.i \leq b(j..ls.(k-1))$ : j)
="definition of h"
  h.i(ls.(k-1))

```

We have shown

```

h.i.k = if k=0  $\rightarrow$  k
      [] k>0  $\rightarrow$  if b.i > b.(k-1)  $\rightarrow$  k
                  [] b.i  $\leq$  b.(k-1)  $\rightarrow$  h.i(ls.(k-1))
                  fi
      fi

```

Moreover, this recursive definition is well-founded. It follows from L0 and an easy inductive argument that the arguments of $h.i.k$ arising from the invocation $h.i.i$ satisfy $0 \leq k \leq i$. Hence we see from an inspection of the text that an evaluation of $h.i.i$ refers at most to those $ls.k$'s satisfying $0 \leq k < i$. This makes it attractive to calculate the $ls.i$'s in increasing order of i for then each $ls.k$ we need is available in x . An obvious minor optimisation is got by dropping the first parameter from h , making it available globally. We conclude that with loop invariant

$$0 \leq i \leq N \wedge x(0..i) = ls(0..i)$$

the first parallel assignment in NSN1 is implemented by

```

LS:  [[ i: integer;
      i:= 0;
      do i $\neq$ N  $\rightarrow$ 
        [[ h: integer = k: integer.
          if k=0  $\rightarrow$  k
          [] k>0  $\rightarrow$  if b.i > b.(k-1)  $\rightarrow$  k
                    [] b.i  $\leq$  b.(k-1)  $\rightarrow$  h.(x.(k-1))
                    fi
          fi •
          x.i:= h.i
        ]];
      i:= i+1
      od
    ]]

```

Reasoning about $rs.i$ just as we did for $ls.i$ leads to the following implementation of the second parallel assignment:

```

RS:  [[ i: integer;
      i:= N;
      do i $\neq$ 0  $\rightarrow$ 
        i:= i-1;
        [[ h: integer func =

```

```

        value k: integer;
        if k=N → k
        [] k<N → if b.i > b.k → k
                   [] b.i ≤ b.k → h.(y.k)
                   fi
        fi •
        y.i:= h.(i+1)
    ]]
od
]]

```

The implementation of the third parallel assignment is trivial, and so we arrive at the program:

```

NSN:  [[ x, y: sequence(N) of integer;
        LS;
        RS;
        [[ i: integer;
            i:= 0;
            do i≠N →
                c.i := if i-(x.i-1) ≤ y.i-i → x.i-1
                       [] i-(x.i-1) ≥ y.i-i → y.i
                fi;
                i:= i+1
            od
        ]]
    ]]

```

Our interest in the program is solely in its derivation, so as far as time complexity is concerned we'll just state without proof that it's linear in N . We can make some minor optimisations. A minor gain in execution time can be got by translating each recursion to an iteration; this is an absolutely standard transformation. Secondly, we can save execution time by calculating the y 's first, and then combining the calculations of the x 's and the c 's in one loop. We could then dispense with the x array altogether by saving the l 's in the y array as the r 's are consumed. But in almost any circumstance, these transformations aren't worth the bother.

4. The largest rectangular area under a histogram

Here is the specification of the second problem:

```

LRH0:  c: integer
        {b ∈ sequence(N) of natural}
        c:= max.(k,l,h: 0≤k<l≤N ∧ 0 ≤ h ≤ b(k..l): (l-k)*h)

```

We put the programming effort into formulating a constructive equivalent of the max-term. The standard move when faced with a max-expression with more than one dummy is to nest, and we'll do that. But first a question we should always ask: can we reduce the size of the bag? Well, for given indices k and l satisfying $0 \leq k < l \leq N$, the height of the maximum rectangle sitting on the base line between k and l , upper bound excluded, is $\min.b(k..l)$, which equals $b.i$ for some i in the domain of b . So the height of the maximum rectangle in the histogram must equal some $b.i$ *provided* N differs from 0. We feel that this property is evident, and so we won't labour it formally in the derivation below.

```

max.(k,l,h: 0≤k<l≤N ∧ 0 ≤ h ≤ b(k..l): (l-k)*h)

```

"range splitting to isolate the empty histogram"
 $\max.(k,l,h: 0 \leq k < l \leq N \wedge 0 \leq h \leq b(k..l): (1-k)*h) \mathbf{max} 0$ (ii)

"comments above"

$\max.(k,l,i: 0 \leq k \leq i < l \leq N \wedge b.i \leq b(k..l): (1-k)*b.i) \mathbf{max} 0$

"In electing to nest we have to consider which dummy to place at the outermost level; we choose i in order to preserve the symmetry between k and l, preserving symmetry generally being advantageous"

$\max.(i: 0 \leq i < N: \max.(k,l: 0 \leq k \leq i < l \leq N \wedge b.i \leq b(k..l): (1-k)*b.i)) \mathbf{max} 0$

Interlude. The standard move now is to nest again, but that destroys the symmetry we elected to preserve just one step back. Looking for another move we observe that $(1-k)*b.i$ is monotonic in l and antimonotonic in k for all i . This is worth pursuing because we know that \max has nice properties in the presence of monotonicity. For example, a standard law is that if $f.k$ is monotonic then

$$\max.(k: P.k: f.k) = f.(\max.(k: P.k: k)),$$

and if $f.k$ is antimonotonic then

$$\max.(k: P.k: f.k) = f.(\min.(k: P.k: k))$$

— empty bags excluded. We should formulate a similar rule for functions $f.k.l$ monotonic in l for all k , and antimonotonic in k for all l :

$$\max.(k,l: P.k \wedge Q.l: f.k.l) = f.(\min.(k: P.k: k)).(\max.(l: Q.l: l))$$

provided the bags are non-empty; we'll call a left-to-right application of this law "range extremising". *End of interlude.*

"sequence theory, preparing for range extremising"

$\max.(i: 0 \leq i < N: \max.(k,l: 0 \leq k \leq i < l \leq N \wedge b.i \leq b(k..i) \wedge b.i \leq b(i..l): (1-k)*b.i) \mathbf{max} 0$

"calculus"

$\max.(i: 0 \leq i < N: \max.(k,l: (0 \leq k \leq i \wedge b.i \leq b(k..i)) \wedge (i < l \leq N \wedge b.i \leq b(i..l)): (1-k)*b.i) \mathbf{max} 0$

"range extremising"

$\max.(i: 0 \leq i < N: (\max.(l: i < l \leq N \wedge b.i \leq b(i..l): l) - \min.(k: 0 \leq k \leq i \wedge b.i \leq b(k..i): k)) * b.i) \mathbf{max} 0$

"definitions of ls and rs from preceding section — how convenient!"

$\max.(i: 0 \leq i < N: (rs.i - ls.i)*b.i) \mathbf{max} 0$

The solution has been reduced to implementing

$$c := \max.(i: 0 \leq i < N: (rs.i - ls.i)*b.i) \mathbf{max} 0,$$

which we easily do with invariant

$$0 \leq n \leq N \wedge c = \max.(i: 0 \leq i < n: (rs.i - ls.i)*b.i) \mathbf{max} 0$$

```

LRH:  |[ x, y: sequence(N) of integer;
      LS;
      RS;
      |[ n: integer;
        n:= 0 || c:= 0;
        do n≠N →
          c:= c max (y.n-x.n)*b.n;
          n:= n+1
        od
      ]|
    ]|
  ]|

```

The program is linear in N.

In formulating $ls.i$ constructively in the earlier section, there was really no choice at any stage, whereas in the derivation above we made several significant choices. The first was to reduce the size of the bag before nesting, but that was hardly a surprising move and would surely present itself no matter what the formalism. The second decision was that at the point of nesting we did not place dummy i at the inner level with k and l together at the outer level, leading after a further step to the expression $\max.(k,l: 0 \leq k < l \leq N: (l-k) * \min.b(k..l)) \mathbf{max} 0$. That move turns out to do nothing but make life difficult by removing one degree of freedom. The final decision was to eschew the second opportunity to nest, in favour of exploiting monotonicity. Other decisions might have been taken, and other solutions arrived at. For example, consider the maximum rectangle as sitting on the base line between indices k and l for some k and l satisfying $0 \leq k < l \leq N$, upper bound excluded: the height h of the rectangle must satisfy $h > b.l$ or $l=N$, for otherwise we could extend the rectangle "on the right" in violation of our postulate that the rectangle is maximum. So instead of passing from (ii) to its successor above we could pass to

$$\max.(k,l,h: 0 \leq k < l \leq N \wedge (l=N \vee b.l \leq h \leq b(k..l)): (l-k)*h) \mathbf{max} 0,$$

and this also leads to a linear solution.

5. Largest all-true submatrix

We extend sequence notation to matrices: indices start at 0, for z a matrix $z.i.j$ selects an element of z , $z(i..j)(k..l)$ selects a submatrix, $z(i..j).k$ denotes a subsequence of column k , and $z.k(i..j)$ denotes a subsequence of row k , always excluding the upper bound in $(i..j)$. We allow boolean sequences and matrices to appear where a simple boolean is expected, its value being the conjunction of its elements.

Here is the specification of the third problem:

```

LTR0:  d: integer
      {z ∈ matrix(M,N) of bool}
      d:= max.(i,j,k,l: 0 ≤ i ≤ j ≤ M ∧ 0 ≤ k ≤ l ≤ N ∧ z(i..j)(k..l): (l-k)*(j-i))

```

Addressing the max-term:

$$\begin{aligned}
& \max.(i,j,k,l: 0 \leq i \leq j \leq M \wedge 0 \leq k \leq l \leq N \wedge z(i..j)(k..l): (l-k)*(j-i)) \\
= & \text{"nesting"} \\
& \max.(i: 0 \leq i \leq M: \\
& \quad \max.(j,k,l: i \leq j \leq M \wedge 0 \leq k \leq l \leq N \wedge z(i..j)(k..l): (l-k)*(j-i)) \\
&) \tag{iii}
\end{aligned}$$

Interlude. We get a strong sense of the preceding problem and so we'll try to exploit it. We'll proceed by trying to give the inner max-term above the same form as the max-term in LRH0. The crux term in (iii) is $z(i..j)(k..l)$, which an easy trick will reshape to what we need. First we describe the standard move in general terms. Let p be a boolean-valued sequence of length N ; then for $0 \leq m \leq n \leq N$

$$p(m..n) \equiv n \leq f.m \quad (\text{iv})$$

where function f is defined by

$$f.m \equiv \max.(i: m \leq i \leq N \wedge p(m..i): i)$$

Applying this device is quite common, for example, L2 of Section 3 can be viewed as an instance of the above. To re-write $z(i..j)(k..l)$, define

$$tt.m.n = \max.(j: m \leq j \leq M \wedge z(m..j).n: j) \quad (0 \leq m \leq M, 0 \leq n < N)$$

(Intuitively, $tt.m.n$ is the largest j such that in column n all the entries from row m to row $j-1$, inclusive, are *true*. Put another way, $tt.m$ can be viewed as describing the largest histogram of *true*s sitting on row m — regarding $(0,0)$ as "bottom-left".) Clearly

$$\text{L3: } m \leq tt.m.n \leq M \quad (0 \leq m \leq M, 0 \leq n < N)$$

and — compare with (iv) —

$$\text{L4: } z(m..j).n \equiv j \leq tt.m.n \quad (0 \leq m \leq j \leq M, 0 \leq n < N)$$

Hence:

$$\begin{aligned} & z(i..j)(k..l) \\ = & \text{"definition as predicate"} \\ & (\forall n: k \leq n < l: z(i..j).n) \\ = & \text{"L4(with } m:=i\text{)"} \\ & (\forall n: k \leq n < l: j \leq tt.i.n) \\ = & \text{"notation"} \\ & j \leq tt.i(k..l) \end{aligned}$$

In summary

$$\text{L5: } z(i..j)(k..l) \equiv j \leq tt.i(k..l) \quad (0 \leq i \leq j \leq M, 0 \leq k \leq l \leq N)$$

(Intuitively, L5 says that the rectangle described by its "bottom-left" corner (i,k) and "top-right" corner $(j-1,l-1)$, inclusive, is all *true*s iff it is included in the largest histogram of *true*s sitting on row i between columns k and $l-1$, inclusive.) The creation of L5 is the extent of the invention we need: the remainder follows inevitably — we only need to keep our eye on the max-term in LRH0 and reproduce its shape. The development from here on is mostly just manipulative detail, similar to what we have seen in the previous two problems, and the reader may feel content to follow the broad outline. *End of interlude.*

= "L5"

$$\begin{aligned} & \max.(i: 0 \leq i \leq M: \\ & \quad \max.(j,k,l: i \leq j \leq M \wedge 0 \leq k \leq l \leq N \wedge j \leq tt.i(k..l): (l-k)*(j-i)) \\ &) \end{aligned}$$

" $j \leq tt.i(k..l)$ implies $j \leq M$ by L3 when $k < l$, and otherwise the value of j is irrelevant"

$$\text{max.}(i: 0 \leq i \leq M: \\ \text{max.}(j, k, l: 0 \leq k \leq l \leq N \wedge i \leq j \leq tt.i(k..l): (l-k) * (j-i)) \\)$$

Interlude. Now only a change of dummy j to $j+i$ is needed. Define

$$tu.m.n = tt.m.n - m \quad (0 \leq m \leq M, 0 \leq n < N)$$

or equivalently

$$tu.m.n = \text{max.}(j: m \leq j \leq M \wedge z(m..j).n: j-m) \quad (0 \leq m \leq M, 0 \leq n < N)$$

(Intuitively, $tu.m.n$ is the length of the longest vertical run of *true*s sitting on position (m,n) ; so $tu.m$ describes the largest histogram of *true*s sitting on row m in a similar way to the encoding of the histogram in Section 4.)

End of interlude.

"change of dummy j to $j+i$ "

$$\text{max.}(i: 0 \leq i \leq M: \\ \text{max.}(j, k, l: 0 \leq k \leq l \leq N \wedge 0 \leq j \leq tu.i(k..l): (l-k) * j) \\)$$

"definition of g below"

$$\text{max.}(i: 0 \leq i \leq M: g.i)$$

where g is defined by

$$g.i = \text{max.}(j, k, l: 0 \leq k \leq l \leq N \wedge 0 \leq j \leq tu.i(k..l): (l-k) * j) \quad (0 \leq i \leq M)$$

In summary, we have only to compute the maximum of the $g.i$'s. Moreover, we know how to compute each $g.i$ efficiently because it's just the area of the maximum rectangle under $tu.i$. That leaves us with just one subproblem: how to compute the $tu.i$'s efficiently. We can discover that by the usual constructive rewriting: For $0 \leq n < N$

$$\begin{aligned} & tu.M.n \\ \text{"definition"} & \\ & \text{max.}(j: M \leq j \leq M \wedge z(M..j).n: j-M) \\ \text{"one-point rule"} & \\ & 0 \end{aligned}$$

For $0 \leq m < M$ and $0 \leq n < N$:

$$\begin{aligned} & tu.m.n \\ \text{"definition"} & \\ & \text{max.}(j: m \leq j \leq M \wedge z(m..j).n: j-m) \\ \text{"range splitting with predicate } j=m\text{"} & \\ & \text{max.}(j: m+1 \leq j \leq M \wedge z(m..j).n: j-m) \mathbf{max} \text{max.}(j: j=m \wedge z(m..j).n: j-m) \\ \text{"one-point"} & \\ & \text{max.}(j: m+1 \leq j \leq M \wedge z(m..j).n: j-m) \mathbf{max} 0 \\ \text{"}z(m..j).n \text{ as predicate"} & \\ & \text{max.}(j: m+1 \leq j \leq M \wedge z(m+1..j).n \wedge z.m.n: j-m) \mathbf{max} 0 \\ \text{"max theory"} & \end{aligned}$$

```

if z.m.n  $\rightarrow$   $\max.(j: m+1 \leq j \leq M \wedge z(m+1..j).n: j-m)$  max 0
[]  $\neg z.m.n \rightarrow 0$ 
fi
="max theory, definition of tu, tu-values are at least 0"
if z.m.n  $\rightarrow$  tu.(m+1).n + 1
[]  $\neg z.m.n \rightarrow 0$ 
fi

```

So, using obvious sequence notation (the upper bound is excluded in 0..N):

```

tu.M =  $\langle n: n \in 0..N: 0 \rangle$ 
tu.m =  $\langle n: n \in 0..N: \mathbf{if} z.m.n \rightarrow tu.(m+1).n+1 \ \square \ \neg z.m.n \rightarrow 0 \ \mathbf{fi} \rangle$    ( $0 \leq m < M$ )

```

The problem has been reduced to implementing $d := \max.(i: 0 \leq i \leq M: g.i)$. Because we have defined $tu.m$ in terms of $tu.(m+1)$ it makes sense to calculate the $g.i$'s in decreasing order of i . The invariant for the main loop has the standard form

$$0 \leq m \leq M \wedge b = tu.m \wedge d = \max.(i: m \leq i \leq M: g.i).$$

The inner loop calculates the new b and has invariant

$$0 \leq n \leq N \wedge b = tu.m(0..n) + tu.(m+1)(n..N)$$

Apart from variables b , d , m , and n , we see from LRH0 that we also need to introduce integer variable c . The program is:

```

LTR:  [| b: sequence(N) of integer; m, n, c, d: integer;
      m := M;
      n := 0; do n  $\neq$  N  $\rightarrow$  b.n := 0; n := n+1 od;
      LRH; d := c;
      do m  $\neq$  0  $\rightarrow$ 
        m := m-1;
        n := 0;
        do n  $\neq$  N  $\rightarrow$ 
          b.n := if z.m.n  $\rightarrow$  b.n+1
                []  $\neg z.m.n \rightarrow 0$ 
                fi;
          n := n+1
        od;
      LRH;
      d := d max c
      od
    ]|

```

The program takes time proportional to $M*N$. A worthwhile minor improvement is got by replacing LRH; $d := c$ with the equivalent $d := 0$.

6. Conclusion

We have engaged in an exercise of making programs by expression refinement. Our interest in this approach stems from a desire to make a fully formal calculus of programming in which we begin with a formal specification which we then subject to a sequence of mathematical transformations until we arrive at a program. The development of such calculi are of great importance for a variety

of reasons. The attempt to develop a formal calculus yields all sorts of insight into the nature of programming, and helps us to discover the kinds of reasoning we should try to bring to bear on programming problems. Experience suggests that if we can formulate and learn to apply the rules of programming then we can arrive at good solutions more easily and confidently. With the problem of the nearest smaller neighbour, for example, we arrived at an efficient solution with hardly a creative thought — the manipulations we engaged in are absolutely standard, and the same good solution would jump at anyone who has learned the formal vocabulary and its laws. This is not to claim that insight is never necessary, for we don't believe that, and indeed we had to be very sensible in our derivation of the final two programs. But the effect of formality is to reduce the amount of invention on several accounts. A formal approach has the great advantage of presenting to our faces the questions that should be asked in finding a solution: Here is a bag — might I reduce its size? Here are two dummies — should I nest, and how? And formality goes a fair way to prompting us with the answers; we find the formulae inviting us to make familiar moves that in their informal guise might look like invention: Apply range contraction! Appeal to the monotonicity of the function! In contrast, we often get stuck when programming informally because we can't even find the questions, let alone the answers. Most importantly of all, perhaps, a formal method becomes a repository of our programming experiences: Here is a kind of monotonic function I haven't met before, monotonic in one argument and antimonotonic in the other — can I accommodate it by combining laws I already know to make a new law that I will add to my stockpile?

Yet we should be modest in our claims, for formal programming as it stands is not universally applicable. Many formal approaches are to be admired more for their mathematical soundness than their practical utility, and even the best are limited in their domain of applicability. In large-scale software construction, for example, formal methods are in their infancy. However, if we believe that making large programs should benefit extensively from automated methods then the development of some formal methodology is a *sine qua non*.

Programming successfully in a style such as ours makes it necessary to learn a fair amount of notation and its rules of behaviour, and to become adept at choosing notations appropriate to the manipulations to be carried out. We have tried to illustrate how this operates in practice. The reader will perhaps have tired of the manipulations in places, and we grant that they can be tedious — after all the whole idea is to reduce to simple routine what should be routine. We add, however, that in deriving the examples above we have been particularly explicit about the manipulations, partly because we presumed they might not be so familiar to the reader, and partly because we wanted to give some indication of the extent to which our approach could be regarded as formal. In practice, we could perhaps afford to be less explicit.

Our main interest in pursuing expression refinement has been in developing a fully formal and practical calculus of programming. Our experience, albeit limited, is that it is potentially a more attractive approach for this purpose than one relying principally on statement refinement.

Acknowledgements. The alternative solution to the problem of the largest rectangle under a histogram, given at the end of section 4, is due to J. van der Woude, as is the idea of applying the solution of that problem to the problem of the largest all-true submatrix. I thank Robert Byrne for a critical reading.

References

1. Back RJR (1980) Correctness Preserving Program Refinements: Proof Theory and Applications. Tract 131, Mathematisch Centrum, Amsterdam

2. Morgan CC (July 1988) The specification statement. ACM TOPLAS 10: 403-419
3. Morris JM (1989) Programs from specifications. In Dijkstra EW (ed.) Formal Development of Programs and Proofs. Addison-Wesley, Reading (Mass.)
4. Backhouse R (1980) Program Construction and Verification. Prentice-Hall, Englewood Cliffs (N.J.)
5. Dijkstra EW (1976) A Discipline of Programming. Prentice-Hall, Englewood Cliffs (N.J.)
6. Gries D (1981) The Science of Programming. Springer, New York
7. Hehner ECR (1984) The Logic of Programming. Prentice-Hall, Englewood Cliffs (N.J.)