Proof of a structured program: The Sieve of Eratosthenes

This paper provides a beautiful example of the development of a computer program from a version which works on abstract data types. Not only is there the obvious task of representing sets but also two different representations of integers are considered in order to develop an efficient program.

It is sometimes argued that such developments are published only for algorithms which are already known. An adequate rebuttal of this claim comes in Misra and Gries (1978), which was extended to an algorithm requiring sub-linear time in Pritchard (1981), both of which were stimulated by this paper. A survey of these algorithms is contained in Pritchard (1987).

This paper was submitted in March 1972 and published as [34]. Anyone who equates the term 'structured programming' with a narrow set of rules concerning the choice of control constructs might be surprised by the Appendix to this paper.

Abstract

This paper illustrates a method of constructing a program together with its proof. By structuring the program at two levels of abstraction, the proof of the more abstract algorithm may be completely separated from the proof of the concrete representation. In this way, the overall complexity of the proof is kept within more reasonable bounds.

9.1 Introduction

In a previous paper (Hoare, [16]) it was shown how a fairly rigorous proof technique could be applied to the development of a simple

C. A. R. Hoare, Proof of a structured program: The Sieve of Eratosthenes, *BCS Computer Journal* **15**(4), 321–5 (November 1972). This paper is republished by kind permission of the British Computer Society.

program, in order to inhibit the intrusion of programming and coding errors. The method involved making a careful formulation of what each part of the program is intended to do, and the invariants which it has got to preserve, before the program is coded. In general, it will then be intuitively obvious that the code is being written to meet its specification; but if confirmation is required, it is possible to extract formally (even mechanically, see Foley and Hoare [19]) the lemmas on which the correctness of the program depends, and give them a rigorous proof. Very often, these lemmas and proofs are both trivial and tedious, and it is to be hoped that the computer may in future be programmed to help in their extraction and verification (King 1969). But even in the absence of such aid, the most important lesson of this method of program construction is the clear statement of the purpose of each part of the program, and the assumptions which it needs to make in order to achieve this purpose. The formal derivation and proof of lemmas can be omitted if we have sufficient trust in our intuition.

The problem now arises, how can these techniques be applied to large programs, where they are so much more necessary? Their application even to small programs is already quite laborious, so their direct application to large programs is out of the question. The solution to this problem lies in the recognition (Dijkstra 1972c; Dahl 1972; Wirth 1971b) that a large program can and should be expressed as a small program written in a more powerful, higher-level programming language. This short program can then be constructed correctly using the same techniques as described in Hoare [16].

However, there remains the problem of 'implementing' the higher-level language in terms which the machine can understand, and of simultaneously proving that this implementation is correct. The programmer must select some efficient method of representation of the data used by the abstract program, and should formulate the function which maps the representation onto the abstract data which it represents. Then the various primitive operations on the abstract data must be implemented by procedures coded in a language closer to one that is comprehensible to the computer. Finally, it must be proved that each procedure operating on the representation has its intended effect on the abstract data which it represents.

The proof methods used in this paper are similar to those described and formalized in Milner (1971) and Hoare [32]. However, they are explained and justified rather informally, relying on intuition to verify that the lemmas quoted are indeed the ones that need to be proved in order to establish the correctness of the programs.

9.2 The abstract algorithm

The Sieve of Eratosthenes is an efficient method of finding all primes equal or less than a given integer N. The 'sieve' contains initially 2 and all odd numbers up to N; but numbers which are multiples of other numbers are gradually 'sifted out', until all numbers remaining in the sieve are prime.

The desired result of the algorithm is

$$\mathit{sieve} = \mathit{primes}(N)$$

where *sieve* is a set of integers and $\mathit{primes}(k)$ is the set of all primes up to and possibly including k.

Some important properties of the algorithm are that it never removes a prime from the sieve, and that it never adds to the sieve a number outside the range 2 to N. These properties may be formally defined by stating invariants of the algorithm – that is, facts which are true of the variables of the program thoughout its execution, except possibly in certain critical regions, or action clusters (Naur 1969), in which the individual variables of a structure are updated piecemeal. These invariants are:

$$\mathit{primes}(N) \subseteq \mathit{sieve} \subseteq \mathit{range}(N) \qquad (I)$$

where $\mathit{range}(k)$ is the set of numbers between 2 and k inclusive.

The sieve is said to be 'sifted' with respect to a number q if all multiples of q have been removed from it:

$$\mathit{sifted}(q) =_{\mathrm{df}} \forall\, n \in \mathit{sieve}(q \text{ divides } n \Rightarrow q = n)$$

The sieve is said to be 'sifted up to p' if all multiples of any prime less than p have been removed from it:

$$\mathit{sifted\ up\ to}(p) =_{\mathrm{df}} \forall\, q \in \mathit{primes}(p-1)(\mathit{sifted}(q))$$

We now prove the elementary lemmas on which the algorithm is based.

Lemma 1
$\mathit{sieve} = \{2\} \cup \{n \in \mathit{range}(N) \mid n \text{ is odd}\} \Rightarrow \mathit{sifted\ up\ to}\ (3)$

Proof No odd number is divisible by 2, so the only number in the sieve divisible by 2 is 2 itself. Therefore *sifted* (2), which implies the conclusion.

Lemma 2
$p^2 > N \wedge \mathit{sifted\ up\ to}\ (p) \wedge \mathit{primes}(N) \subseteq \mathit{sieve} \subseteq \mathit{range}(N) \Rightarrow \mathit{sieve} = \mathit{primes}(N)$

Proof Let $s \in sieve - primes(N)$ and let q be its smallest prime factor.

$\therefore$ $q^2 \leqslant s \leqslant N < p^2$ since s is a non-prime in $range(N)$

$\therefore$ $q \leqslant p - 1$

$\therefore$ $sifted(q)$ since $sifted\ up\ to(p) \wedge q$ is prime

$\therefore$ $s \notin sieve$ since q divides s (by hypothesis) and $q \neq s$

This contradiction shows that $sieve - primes(N)$ is empty. The conclusion follows from $primes(N) \subseteq sieve$.

The algorithm can be based on these two theorems:

```
begin p : integer;
  sieve := {2} ∪ {n ∈ range(N) | n is odd};
  p := 3;
  while p² ⩽ N do
    begin sift(p);
      p := next prime after (p)
    end
end
```

where $sift(p)$ is assumed to have the result

$$sifted(p)$$

and to preserve the invariance of

$$I \wedge sifted\ up\ to(p).$$

The correctness of the loop in preserving the invariance of $sifted\ up\ to(p)$ now depends on the lemma.

Lemma 3
$p^2 \leqslant N \wedge sifted\ up\ to(p) \wedge sifted(p) \Rightarrow sifted\ up\ to(next\ prime\ after(p))$

Proof Let

$$q \in primes\ (next\ prime\ after(p) - 1)$$

(1) if $q \leqslant p - 1$, $sifted(q)$ follows from $sifted\ up\ to(p)$
(2) if $q = p$, $sifted(q)$ follows from $sifted(p)$
(3) since q is prime it is impossible that

$$p < q \leqslant next\ prime\ after(p) - 1,$$

$sifted(q)$ follows in both possible cases, and consequently $sifted\ up\ to\ (next\ prime\ after(p))$.

Of course, this algorithm would be rather pointless unless there were some especially fast way of computing the next prime after p in the context in which it is required. Summarizing all facts known at this stage we get:

$$I \wedge p^2 \leqslant N \wedge sifted\ up\ to\ (next\ prime\ after(p)) \qquad (P)$$

Now we can rely on the theorem

Lemma 4
$P \Rightarrow \textit{next prime after}(p) = \textit{next after}\ (p, \textit{sieve})$
where *next after* (n, s) is the smallest element of s which is greater than n.

Proof Let

$$p' = \textit{next after}\ (p, \textit{sieve})$$

Since

$$\textit{primes}(N) \subseteq \textit{sieve}, \text{ and } p' \in \textit{sieve}$$

it follows that

$$p < p' \leqslant \textit{next prime after}(p)$$

To establish equality, it is sufficient to prove that p' is prime. Let r be the smallest prime factor of p', so $r \leqslant p'$.
Assume $r < p'$
then *sifted*(r) follows from $r < p' \leqslant \textit{next prime after}(p)$
and *sifted up to* (*next prime after* (p)).
$\therefore \quad r = p'$ since $p' \in \textit{sieve} \wedge r \textit{ divides } p'$
Therefore p' is equal to its smallest prime factor.

It remains to prove that *next after* $(p, \textit{sieve})$ actually exists; this follows from $p^2 \leqslant N$ and the fact that there is always a prime between p and p^2. This is a deep result in number theory, and will not be proved here.

A second obvious improvement in efficiency is to replace the test

$$p^2 \leqslant N$$

by

$$p \leqslant \textit{rootN}$$

where *rootN* has been precomputed as the highest integer not greater than *sqrt*(N).

9.3 Sifting

It remains to program the procedure *sift*(p). This procedure may assume the precondition that $p > 2$ is odd, and must produce the result that *sifted*(p). It must also preserve the invariants:

$$\textit{primes}(N) \subseteq \textit{sieve}$$
$$\textit{sieve} \subseteq \textit{range}(N)$$
$$\textit{sifted up to}(p)$$

The only change made to global variables by the operation of sifting is the repeated removal of an integer s from the sieve:

$$sieve := sieve - \{s\};$$

where s is not prime. Removal of non-prime elements from the sieve can never alter any of these invariants from true to false, as may be verified by the trivial lemmas:

Lemma 5
$s \notin primes(N) \wedge primes(N) \subseteq sieve \Rightarrow primes(N) \subseteq sieve - \{s\}$

Lemma 6
$sieve \subseteq range(N) \Rightarrow sieve - \{s\} \subseteq range(N)$

Lemma 7
$\forall q \in primes(p-1)\ \forall n \in sieve(q \text{ divides } n \Rightarrow q = n)$
$\quad\Rightarrow \forall q \in primes(p-1)\ \forall n \in (sieve - \{s\})(q \text{ divides } n \Rightarrow q = n)$

Proof After replacing $n \in (sieve - \{s\})$ by an equivalent form $n \in sieve \wedge n \notin \{s\}$, this is a tautology.

Now all that is necessary is to generate a sequence of multiples of p, and remove them from the sieve. However, it is known that $p > 2$, so that *sifted up to* (p) implies that *sifted*(2). Consequently, it is not necessary to remove *even* multiples of p, since they are already gone. Also all nonprimes less than p^2 will be gone as a result of *sifted up to* (p). We design the sifting process to preserve the invariance of

$$sifted\ up\ to(p) \wedge s \ \mathbf{mod}\ 2p = p \wedge s \geqslant 2 \wedge p \text{ is odd} \wedge p > 2 \wedge$$
$$\forall s' \in sieve \cap range(s-1)(p \text{ divides } s' \Rightarrow p = s')$$

Note that s **mod** $2p = p$ implies that s is non-prime, which we required to justify removal of s from sieve.

The program is:

```
sift: begin s, step: integer;
        step := p + p;
        s := p × p;
        while s ⩽ N do
          begin sieve := sieve − {s};
            s := s + step
          end
      end sift.
```

The correctness of the program depends on the following lemmas:

Lemma 8
p is odd $\Rightarrow (p \times p)$ **mod** $2p = p$

Lemma 9
$\mathit{sifted\ up\ to}\ (p) \wedge p > 2 \Rightarrow \ \forall s' \in \mathit{sieve} \cap \mathit{range}(p \times p - 1)$
$(p\ \mathit{divides}\ s' \Rightarrow p = s')$

Proof Let $s' \in \mathit{sieve} \cap \mathit{range}(p^2 - 1)$ and assume $p\ \mathit{divides}\ s'$. We shall prove that s' is prime, for then it follows immediately from $p\ \mathit{divides}\ s'$ that $s' = p$.

Let r be the smallest prime factor of s'.
There are two cases:

(1) if $r = s'$ then s' is prime anyway
(2) if $r^2 \leqslant s'$ then

$r^2 \leqslant s' < p^2$	since $s' \in \mathit{range}(p^2 - 1)$
$r \leqslant p - 1$	
$\mathit{sifted}(r)$	since $\mathit{sifted\ up\ to}(p)$
$\therefore\ r = s'$	since $r\ \mathit{divides}\ s' \in \mathit{sieve}$

hence s' is prime.

Lemma 10
s **mod** $2p = p \Rightarrow (s + 2p)$ **mod** $2p = p$

Lemma 11
$s \geqslant 2 \wedge p > 2 \Rightarrow s + 2p \geqslant 2$

Lemma 12
$\mathit{sifted}(2) \wedge s$ **mod** $2p = p \wedge s \geqslant 2 \wedge$
$\qquad \forall s' \in \mathit{sieve} \cap \mathit{range}(s - 1)(p\ \mathit{divides}\ s' \Rightarrow p = s')$
$\qquad \Rightarrow \ \forall s' \in (\mathit{sieve} - \{s\}) \cap \mathit{range}(s + 2p - 1)(p\ \mathit{divides}\ s' \Rightarrow p = s')$

Proof Consider s' satisfying the conditions
$s' \in (\mathit{sieve} - \{s\}) \wedge s' \leqslant s + 2p - 1 \wedge p\ \mathit{divides}\ s'$
There are three cases:

(1) if $s' < s$, then $p = s'$ follows from the antecedent
(2) $s' = s$ is impossible, since $s' \in \mathit{sieve} - \{s\}$
(3) if $s' > s$ then $s' = s + p$ since $s' \leqslant s + 2p - 1$ and p divides both s and s'

$\therefore\ s'$ **mod** $2p = 0$	since s **mod** $2p = p$
$\therefore\ 2\ \mathit{divides}\ s'$	
$\therefore\ s' = 2$	follows from $\mathit{sifted}(2)$

but this contradicts $s' > s \geqslant 2$. Thus case (3) is also impossible.

Lemma 13
$s > N \wedge sieve \subseteq range(N) \wedge$
$\quad \forall s' \in sieve \cap range(s-1)(p \text{ divides } s' \Rightarrow p = s') \Rightarrow sifted(p)$

Proof $sieve \subseteq range(N) \subseteq range(s-1)$
$\therefore \quad sieve \cap range(s-1) = sieve.$

9.4 Concrete representation

The program developed and proved in the previous section is abstract, in the sense that it assumes the possibility of operating on integers and sets of integers of arbitrary size (up to N). Now, if N is smaller than $2\uparrow wordlength - 1$ (where *wordlength* is the number of bits in the word of the computer), the possibility of operating on integers of this size will have been provided by the hardware of the computer. Furthermore, if $N \leqslant 2 \times wordlength$, the sieve itself may be represented as a single word of the computer store which has the nth bit set to one if $(2n + 1)$ is in the sieve. The operations on the set are probably built into the instruction code of the computer in the form of logical and shifting instructions.

However, the case which interests us is when N is rather large compared with the computer wordlength, but still rather smaller than the total number of bits in the main store of the computer. A figure of the order of 10 000 may be typical. It is, therefore, necessary to represent the sieve by means of an array of words, and it is now the responsibility of the programmer to find an efficient way of implementing the required set operations on this array. A word may be regarded as representing a subset of the positive integers less than the wordlength.

Unfortunately, the access and manipulation of an arbitrary bit held in an array of words is rather an inefficient operation, involving a division to find the right wordnumber, with the remainder indicating the bitnumber within the word. On machines with wordlength equal to a power of two this will be less inefficient than on other machines. But the solution which we adopt here is to represent the integers p, s, and *step* as *mixed radix* numbers. Each of them contains two components, a wordnumber and a bitnumber within the word. Furthermore, only *odd* numbers need be represented in the case of p and s, whereas only *even* numbers need be represented for the step.

Since we have to test p and s against N and *rootN*, it is as well to represent these in the same way as odd numbers. If they are even, this means subtracting one from them. The validity of this is due to the fact that

$$a \text{ is odd} \wedge b \text{ is even} \Rightarrow (a \leqslant b \equiv a \leqslant b - 1).$$

We therefore need to implement the arithmetic operations required on this curious representation of odd integers, as well as the required set operations on the sieve; and for this purpose we will use the concepts and notations described in Chapter 8, and declare the representations as Simula classes. A class is a declaration of a data structure (its local variables), together with declarations of all the procedures which operate on data of that structure. The body of each procedure is a critical region inside which invariants may be temporarily falsified. We can outline the requirements by showing the class declarations, but replacing all the procedure bodies by a comment stating what they are intended to do:

```
class eveninteger(a: oddinteger);
  begin wd, bit;
    initialize: comment initialize to $2 \times a$;
  end;
class oddinteger(n: integer);
  begin wd, bit: integer;
    procedure add(b: eveninteger);
    comment ': + b';
    boolean procedure eqless(c: oddinteger);
    comment ' $\leqslant c$';
    procedure nextafter(sve: sieveclass);
    comment ': = nextafter(itself, sve)';
    procedure square(c: oddinteger);
    comment ': = $c \times c$';
    comment initialize to $((n - 1) \div 2) \times 2 + 1$;
  end oddinteger;
class sieveclass(N: integer);
  begin constant $W = (N \div 2) \div wordlength$
    bitmap: array 0 .. W of word;
    procedure remove(s: oddinteger);
    comment ': $- \{s\}$';
    comment initialize to $\{2\} \cup \{n: 3 .. N \mid n \text{ is odd}\}$;
  end sieveclass.
```

In these declarations the type (or class) of a variable or a formal parameter is written after the declaratory occurrence of its name, and separated from it by a colon.

We can now recode the algorithm using class on these procedures instead of the statements which they are intended to simulate.

```
sieve: sieveclass(N);
begin constant rootN: oddinteger(sqrt(N));
  comment rootN := (sqrt(N) - 1) ÷ 2 × 2 + 1;
  constant oddN: oddinteger(N);
  comment oddN := (N - 1) ÷ 2 × 2 + 1;
  p: oddinteger(3); s: oddinteger(3);
  comment p := s := 3;
  while p.eqless(rootN) do
    comment p ⩽ rootN;
    begin step: eveninteger(p);
      comment step := 2 × p;
      s.square(p);
      comment s := p × p;
    while s.eqless(oddN) do
      comment s ⩽ oddN;
      begin sieve.remove(s);
        comment sieve := sieve - {s};
        s.add(step)
        comment s := s + step;
      end sloop;
    p.next after (sieve)
    comment p := nextafter(p, sieve);
    end ploop
end sieve;
```

The variable changed by each procedure call is written to the left of it, separated by a dot. The comment gives the intended effect.

First it is necessary to define the abstract objects in terms of their concrete representations. We introduce abbreviations:

$$number(w, b) =_{\mathrm{df}} 2 \times w \times wordlength + 2 \times b + 1$$
$$w(n) =_{\mathrm{df}} ((n - 1) \div 2) \div wordlength$$
$$b(n) =_{\mathrm{df}} ((n - 1) \div 2) \ \mathbf{mod} \ wordlength$$

We now have

$$w(number(w, b)) = w \qquad \text{provided } 0 \leqslant b < wordlength$$
$$b(number(w, b)) = b \qquad \text{provided } 0 \leqslant b < wordlength$$
$$number(w(n), b(n)) = ((n - 1) \div 2) \times 2 + 1$$
$$= \begin{cases} n & \text{if } n \text{ is odd} \\ n - 1 & \text{if } n \text{ is even} \end{cases}$$

Now we can define the abstract objects implemented by each class, in terms of the local variable of that class.

$$oddinteger = number\ (wd, bit)$$
$$eveninteger = number\ (wd, bit) + 1$$
$$sieveclass = \{2\} \cup \{n \in range(N) \mid n \text{ is odd} \wedge b(n) \in bitmap[w(n)]\}.$$

We also need the following invariant for both the *oddinteger* and the *eveninteger* classes.

$$0 \leqslant bit < wordlength \qquad (bitrange)$$

We now proceed to code the bodies of the procedures, and formulate the lemmas on which their correctness depends. The proofs (like the procedure bodies) are rather trivial, though also tedious. They do not depend on any results of number theory, and they are of a sort that might in the future be readily checked by a computer. They will therefore be omitted in the remainder of the paper. The important point to note is that all the proofs are independent of each other.

As a preliminary we code an operation to perform any necessary carry between the bit and the wordnumber:

```
carry: if $bit \geqslant wordlength$ then
  begin $bit := bit - wordlength$;
        $wd := wd + 1$
  end
```

The label indicates that this piece of code is known as *carry*. We prove that *carry* leaves unchanged the value of *number*(*wd*, *bit*).

This depends on the truth of

Lemma 1A

$$number(wd, bit) = 2 \times (\textbf{if } bit \geqslant wordlength \textbf{ then } wd + 1 \textbf{ else } wd) \times wordlength + 2 \times (\textbf{if } bit \geqslant wordlength \textbf{ then } bit - wordlength \textbf{ else } bit) + 1$$

Next we prove $0 \leqslant bit < 2 \times wordlength\ \{carry\}\ 0 \leqslant bit < wordlength$, that is, if the assertion before the left brace is true before performing *carry*, the assertion after the right brace is true afterwards.

This depends on the truth of

Lemma 1B

$$0 \leqslant bit < 2 \times wordlength \Rightarrow 0 \leqslant (\textbf{if } bit \geqslant wordlength \textbf{ then } bit - wordlength \textbf{ else } bit) < wordlength$$

This means that we can ensure the truth of *bitrange* by invoking *carry* after each change to the value of *bit*, provided this change does not take the *bit* outside the range

$$0 \leqslant bit < 2 \times wordlength.$$

```
(1) initialize $eveninteger\ (a: oddinteger)$:
    begin $wd := a.wd + a.wd$;
      $bit := a.bit + a.bit$;
      $carry$
    end
```

Lemma 2
$0 \leqslant a.bit < wordlength \Rightarrow 0 \leqslant a\,.\,bit + a.bit < 2 \times wordlength$

Lemma 3
$number\ (a.wd + a.wd,\ a.bit + a.bit) + 1 = 2 \times number(a.wd,\ a.bit)$

```
(2) initialize $oddinteger(n: integer)$:
    begin $wd := ((n-1) \div 2) \div wordlength$;
      $bit := ((n-1) \div 2)$ mod $wordlength$
    end
```

Lemma 4
$0 \leqslant ((n-1) \div 2) \textbf{ mod } wordlength < wordlength$

Lemma 5
$number(w(n),\ b(n)) = ((n-1) \div 2) \times 2 + 1$

```
(3) procedure $add(b: eveninteger)$;
      begin $wd: = wd + b.wd$;
        $bit := bit + b.bit + 1$;
        $carry$
      end
```

Lemma 6
$0 \leqslant bit,\ b.bit < wordlength$
$\Rightarrow 0 \leqslant bit + b.bit + 1 < 2 \times wordlength$

Lemma 7
$number(wd + b.wd,\ bit + b.bit + 1)$
$= number(wd,\ bit) + (number(b.wd,\ b.bit) + 1)$
Note: The proof method reminds us to add one to *bit* here.

```
(4) Boolean procedure $eqless(c: oddinteger)$;
      $eqless :=$ if $wd = c.wd$ then $(bit \leqslant c.bit)$
                 else $(wd < c.wd)$.
```

Lemma 8

$0 \leqslant bit,\ c.bit \leqslant wordlength$
$\Rightarrow (\textbf{if } wd = c.wd \textbf{ then } (bit \leqslant c.bit) \textbf{ else } (wd < c.wd)$
$\equiv number(wd, bit) \leqslant number(c.wd,\ c.\mathrm{bit}))$

```
(5)  procedure remove(s: oddinteger);
         bitmap[s.wd] := bitmap[s.wd] − {s.bit}
```

Let *bitmap′* be the result of executing this statement; i.e.

$$bitmap' = \lambda w:0..W(\textbf{if } w = s.wd \textbf{ then } bitmap[w] - \{s.bit\} \textbf{ else } bitmap[w])$$

Lemma 9

$0 \leqslant s.bit < wordlength$
$\Rightarrow \{2\} \cup \{n \in range(N) \mid b(n) \in bitmap'[w(n)]\}$
$= \{2\} \cup (\{n \in range(N) \mid b(n) \in bitmap[w(n)]\}$
$- \{number(s.wd, s.bit)\})$

```
(6)     procedure square(c: oddinteger);
          begin t: integer;
            t := 2 × c.bit × (c.bit + 1);
            bit := t mod wordlength;
            wd := 2 × c.wd × (c.wd × wordlength
               + 2 × c.bit + 1)
               + t ÷ wordlength
          end
```

Lemma 10

$number(c.wd, c.bit)^2$
$$= number(2 \times c.wd \times (c.wd \times wordlength + 2 \times c.bit + 1) + t \div wordlength,\ t \textbf{ mod } wordlength)$$
where
$$t = 2 \times c.bit \times (c.bit + 1)$$

Lemma 11

$0 \leqslant t \textbf{ mod } wordlength < wordlength$

Note: Here we need all the confidence that we can get in the correctness of the algebra.

```
(7)  initialize sieveclass:
         for w: 0.. W do bitmap[w] := {n | 0 ⩽ n < wordlength}
```

(i.e. for all w in the range 0 to W **do** ...).

The result of this **for** statement is to set *bitmap* to the constant function (each word of it is 'all ones'):

$$bitmap_0 = \lambda w:0..W(\{n \mid 0 \leqslant n < wordlength\})$$

Lemma 12
$\{2\} \cup \{n:3\,..\,N \mid n \text{ is odd}\} =$
$\{2\} \cup \{n:3\,..\,N \mid n \text{ is odd} \wedge b(n) \in \mathit{bitmap}_0[w(n)]\}$

```
(8) procedure next after (sve: sieveclass);
        with sve do
          begin this: word;
            this := bitmap[wd] ∩ {n | bit < n < wordlength};
            while this = empty do
              begin wd := wd + 1;
                this := bitmap[wd]
              end;
            bit := min(this); comment smallest member of this;
          end
```

The criterion of correctness is

$$(\mathit{number}(wd, bit) = n)\{\mathit{next\ after}\}R$$

where $R =_{\mathrm{df}} n < \mathit{number}(wd, bit) \in \mathit{sieve}$

$$\wedge\ \forall s(n < s < \mathit{number}(wd, bit) \Rightarrow s \notin \mathit{sieve})$$

The proof of this is left to the reader.

9.5 Conclusion

The concrete representation has now been proved to be a valid implementation of the operations required by the abstract program. The final concrete program may be obtained from the text of the program given in Sections 9.2 and 9.3, together with the text of the class declarations, using the simple substitution method implemented in Simula 67 and described in Dahl (1972) and Chapter 8.

In presenting the lemmas on which the correctness of each procedure is based, we have relied on intuition to relate the procedures to the lemmas. In fact, it is possible to formalize the relationship between program and lemma in such a way that the required lemma could be mechanically generated from the definitions of the invariants of the class, the definition of the representation function, and the definition of the operation which each procedure is intended to carry out.

It is interesting to compare the structured program developed here with the corresponding unstructured program displayed in the Appendix. It can be readily seen that a direct proof of the correctness of the unstructured version would be much more laborious.

The structuring of programs is of great assistance, not only in their proof, but also in their design, coding and documentation. I would expect that programming languages of the future may be designed to encourage and assist the structuring of programs, and perhaps even to enforce the observance of those disciplines necessary to avoid breakdown of structure.

For another example of stepwise development and proof of a program, see Jones (1971).

9.6 Appendix: The unstructured program

```
bitmap: array 0 .. W .. of word;

for w: 0 .. W do bitmap[w] := {n | 0 ≤ n < wordlength};
begin rootNwd, rootNbit, Nwd, Nbit, t, pwd, pbit: integer;
  t := (sqrt(N) − 1) ÷ 2;
  rootNwd := t ÷ wordlength;
  rootNbit := t mod wordlength;
  t := (N − 1) ÷ 2;
  Nwd := t ÷ wordlength;
  Nbit := t mod wordlength;
  pwd := 1 ÷ wordlength;
  pbit := 1 mod wordlength;
  while if pwd = rootNwd then pbit ≤ rootNbit
    else pwd < rootNwd do
    begin stepwd, stepbit, swd, sbit: integer;
      stepwd := pwd + pwd;
      stepbit := pbit + pbit;
      if stepbit ≥ wordlength then
        begin stepbit := stepbit − wordlength;
              stepwd := stepwd + 1
        end;
    t := 2 × pbit × (pbit + 1);
    sbit := t mod wordlength;
    swd := 2 × pwd × (pwd × wordlength + 2 × pbit + 1)
                                    + t ÷ wordlength;
    while if swd = Nwd then sbit ≤ Nbit
      else swd < Nwd do
      begin bitmap[swd] = bitmap[swd] − {sbit};
        swd := swd + stepwd;
        sbit := sbit + stepbit;
        if sbit ≥ wordlength then
          begin sbit := sbit − wordlength;
                swd := swd + 1
          end;
    **end** *sloop*;
    **begin** *this*: *word*;
      $this := bitmap[pwd] \cap \{n \mid pbit < n < wordlength\}$;
      **while** $this = \{\ \}$ **do**
        **begin** $pwd := pwd + 1$
          $this := bitmap[pwd]$
        **end**;
      $pbit := min(this)$
    **end** *nextafter*
  **end** *ploop*
**end** *sieve*
```