stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 59/76 DECEMBER

L.G.L.T. MEERTENS

FROM ABSTRACT VARIABLE TO CONCRETE REPRESENTATION

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

AMS(MOS) subject classification scheme (1970): 68A05, 68A30

ACM-Computing Reviews-categories: 5.24, 4.22

From abstract variable to concrete representation *)

by

L.G.L.T. Meertens

ABSTRACT

The method of structured programming introduces a number of abstraction levels in the design of a program. The designer may use variables of any type that are meaningful to him, such as sets, in order to specify an abstract algorithm. When implementing this algorithm in a programming language, he faces the task to implement these variables with the data types available to him. This paper shows a technique, mainly presented informally by applying it to examples, for deriving the implementation of the abstract operations from an "interpretation function" which specifies the correspondence between an abstract variable and its concrete representation.

The technique does not depend on the choice of a particular programming language; for example, it may be used to implement an algorithm in languages with poor data structuring capabilities as FORTRAN or ALGOL 60.

Even though the development of a program may have been structured, this need not hold for the final program text: the abstract algorithm and the correspondence between the abstract variables and their representations have been lost. Some problems are discussed which are encountered in the design of a programming language which allows the textual expression of the abstraction levels.

KEY WORD & PHRASES: *abstract variable, concrete representation, structured programming, levels of abstraction, correctness proof, specification, interpretation function, data structure, efficiency, automatic data structure choice, abstract data type, uniform references.*

---

*) This report will be submitted for publication elsewhere.

1. VARIABLES AND ASSIGNMENT

The expressive power of algorithms transcends that of formulas because of two elements: repetition and variables. Both are essential. From a theoretical point of view, it may be argued that these elements are not fundamental, but, rather, special cases of two more fundamental principles: (recursive) procedures and parameters, with the correspondence

(backward) jump	—	(recursive) call
assignment	—	parameter passing.

This may be illustrated with the following two pieces of program:

```
k := 0;                    proc r = (int k) void :
while k ≤ 9                if k ≤ 9
do print (k);              then print (k);
   k +:= 1                      r (k + 1)
od                         fi;
                           r (0).
```

One may wonder to what extent the existence of special notations in most programming languages for repetition and variables is a relic from the machine code era, and to what extent it constitutes a recognition of their special role in the design of algorithms. A related question is whether the case against the unrestricted jump cannot be extended to the unrestricted assignment (see, e.g., WULF & SHAW [1]). This paper will bypass these questions; nevertheless it should be pointed out that much of what is stated here about variables applies to parameters as well.

Before proceeding, some notations have to be introduced.

The expression $A[\![v := e]\!]$ stands for the expression obtained by substituting the expression e for all occurrences of v in A. The treatment here is informal; may it suffice that this substitution has to be interpreted *intentionally* rather than *literally*.

If S is a set of elements of type T, and e has type T, we write $S \cup e$, rather than $S \cup \{e\}$, for S augmented with e.

The notation $\cup\, x :$ p (x) denotes the set of all x satisfying p (x); a more traditional notation would be $\{x \mid \text{p}\,(x)\}$.

If Ω is a dyadic operator, $v := v\,\Omega\,e$ is abbreviated to $v\,\Omega{:=}\, e$.

The set of consecutive integers $\cup i : a \leq i \leq b$, is denoted by $[a : b]$. Finally, the size (cardinality) of a set S is denoted by $\#S$.

The key to the notion of variable lies in the assignment, whose meaning can be expressed by the general rule

$$\{p[\![v := e]\!]\}\ v := e\ \{p\}.$$

In this rule, p stands for an assertion, v for a variable and e for an expression. The rule now states: if the assertion p has to hold afterwards, this can be accomplished by the assignment $v := e$, provided that v and e are chosen such that beforehand the assertion $p[\![v := e]\!]$ holds. This rule has proved to be a powerful instrument in programming.

An example: We want the following to hold for a variable S, whose value ranges over the sets:

$$\forall\ x \in S : Q(x).$$

Since the empty set ϕ contains no elements, we have, trivially,

$$\forall\ x \in \phi : Q(x).$$

By applying the instance of the assignment rule

$$\{\forall\ x \in \phi : Q(x)\}\ S := \phi\ \{\forall\ x \in S : Q(x)\},$$

we deduce that the goal is obtained by the assignment $S := \phi$.

2. ABSTRACT VARIABLES AND CONCRETE REPRESENTATION

In the above, a variable S was introduced to which a set can be assigned. If we want to exploit the power of the assignment to its full extent, we need the freedom to employ such variables. But most — even high level — programming languages have no sets among their primitive data types*. So the programmer who wants to implement an algorithm in which sets play a role, will in general have to design himself a representation, composed from the data types available to him. The same holds in general for all kinds of data types that can be used to express algorithms. For example, it is quite

---

* Although PASCAL (WIRTH [2]) has sets, these are exclusively subsets of predefined *finite* sets.

possible that an algorithm contains an assignment $z3 := z1 \times z2$, where $z1$, $z2$ and $z3$ are complex variables. In ALGOL 68 this notation may be kept, but for an ALGOL 60 program this has to be expressed differently, e.g. by

$$re3 := re1 \times re2 - im1 \times im2;$$
$$im3 := re1 \times im2 + im1 \times re2.$$

Even without the foregoing exposition one might guess that this piece of program computes the product of two complex numbers. We can say: the pair of real variables $\langle re3, im3 \rangle$ is a *concrete representation* of the *abstract variable* $z3$. The correspondence is given by $z = re + i \times im$, the so-called Cartesian representation. This is not the only possibility; a not unusual representation is the polar one, with $z = mod \times \exp(i \times arg)$. The ALGOL 60 program might well have had:

$$mod3 := mod1 \times mod2;$$
$$arg3 := arg1 + arg2.$$

3. STRUCTURED PROGRAMMING

Abstract variables are a natural result of the method of *top-down* or *structured* programming. The essence of this method lies in the use of a number of levels of abstraction. The notions *abstract* and *concrete*, as used above, are relative ones; for example, an integer, conceptually considered an atomic entity, will be implemented on a very concrete level as a conglomerate of two-valued variables. Luckily, we do not have to realize this in order to design, say, a better factorization algorithm.

The advantages of structured programming, as far as variables are concerned, are easily underestimated by those who have not (yet) tried to use this discipline of thought in order to guide their programming labour. In the abstract description of the algorithm variables may be used for all kinds of values that have a meaning to the designer of the algorithm: values that he can *think in*. This gives a freedom of expression, much greater than is offered by most current programming languages; it opens the road for writing algorithms whose correctness is easy to see or to prove. Meanwhile, the programmer has not yet committed himself at all as to the concrete representation; he still has the freedom of choosing that repre-

sentation which suits him well, either with respect to the ease of implementation, or with respect to the efficiency of the final program. Complex addition, e.g., is formulated more easily in terms of the Cartesian representation; on the other hand, the polar representation is more manageable for multiplication and much more so for taking the square root.

This way of doing offers at the same time the advantage that the correctness proof can be structured in complete analogy with program development: the correctness proof of the abstract algorithm need only be augmented with a proof of the correctness of the concrete representation. Because of this separation in two independently provable parts, the complexity of the whole proof is kept within reasonable bounds. A fully worked-out example, which possibly has more convincing power than the — out of necessity — toy-examples in this paper, can be found in HOARE [3].

4. INTERPRETATION AND ABSTRACT ASSIGNMENT

The correspondence between an abstract variable v_a and its concrete representation v_c (in general, a conglomerate of variables) can be specified by means of an *interpretation function* I, namely by $v_a = \mathrm{I}(v_c)$. (In the example of complex numbers we have $z = \mathrm{I}(\langle re, im \rangle)$, with $\mathrm{I}(\langle re, im \rangle) = re + \mathrm{i} \times im$.)

In the process of "concretization" of the algorithm, all occurrences of v_a have to disappear, in favour of v_c. A first step will be: the systematic replacement of v_a by $\mathrm{I}(v_c)$, by which, e.g., "if abs v_a > 1 then ..." is changed into "if abs $\mathrm{I}(v_c)$ > 1 then ...". This prescription fails, however, where assignments are involved. It is not possible to concretize $v_a := e_a$ by $\mathrm{I}(v_c) := \mathrm{I}(e_c)$, since $\mathrm{I}(v_c)$ is in general not a variable. Instead, we can try to concretize $v_a := e_a$ by an assignment of the form $v_c := e_c$. The question becomes: how must e_c be chosen? For this we call the assignment rule to our aid: the meaning of $v_a := e_a$ is the transformation of $p[\![v_a := e_a]\!]$ into p. We want to achieve, by a suitable choice of a concrete expression e_c:

$$\text{if } v_a = \mathrm{I}(v_c), \text{ then } \{p[\![v_a := e_a]\!]\}\ v_c := e_c\ \{p\},$$

or, equivalently,

$$v_a = \mathrm{I}(v_c) \wedge p[\![v_a := e_a]\!] \Rightarrow p[\![v_c := e_c]\!].$$

However, $p[\![v_c := e_c]\!]$ is meaningless, since p is defined in abstract terms

and does not contain v_c. We have to use $v_a = I(v_c)$ in order to be able to substitute an expression for v_c in p, namely by first replacing p by $p[\![v_a := I(v_c)]\!]$. So we have to obtain

$$v_a = I(v_c) \wedge p[\![v_a := e_a]\!] \Rightarrow p[\![v_a := I(v_c)]\!][\![v_c := e_c]\!],$$

which may be simplified to

$$v_a = I(v_c) \wedge p[\![v_a := e_a]\!] \Rightarrow p[\![v_a := I(e_c)]\!].$$

This implication holds for all p iff e_c is chosen such that $v_a = I(v_c)$ implies $e_a = I(e_c)$.

This hardly surprising result can also be derived in a different way which offers an interesting view. We might imagine that in the process of concretization the concrete representation does not so much *replace* the abstract variable, but that it comes *next to it*. (The abstract variable is, of course, superfluous in the final program, and therefore we will remove afterwards, when the program has been drawn up, all references to the abstract variable from the program text.) The assertion $v_a = I(v_c)$ then has to be kept invariant. Whenever the validity of $v_a = I(v_c)$ is lost by an assignment $v_a := e_a$, it has to be restored; we can try to use an assignment of the form $v_c := e_c$ to this purpose.

We want to have

$$\{v_a = I(v_c)\}\ v_a := e_a;\ v_c := e_c\ \{v_a = I(v_c)\}.$$

Working backwards according to the assignment rule, we obtain

$$\{e_a = I(e_c)\}\ v_a := e_a\ \{v_a = I(e_c)\}\ v_c := e_c\ \{v_a = I(v_c)\},$$

and the goal is obviously reached by choosing v_c such that $v_a = I(v_c)$ implies $e_a = I(e_c)$.

We now fix our attention upon the intermediate assertion $v_a = I(e_c)$. In general, the expression e_c will be a function of v_c, so $v_a = I(e_c) = I(f(v_c)) = J(v_c)$, with $J = I \circ f$, and we obtain

$$\{v_a = I(v_c)\}\ v_a := e_a\ \{v_a = J(v_c)\}.$$

This can be interpreted as a switch from an interpretation function I (which is merely based upon agreement) to a new interpretation function J. The assignment $v_c := e_c$ then takes care of the return to I.

Now, in general, there will be no hurry to return to I. As long as for

the crucial spots $v_a = I(v_c)$ holds, such an invariant may very well be temporarily broken in the parts in between. It is quite possible that $v_a = I(v_c)$ is an invariant of a loop, and that $v_a = J(v_c)$ is an invariant of an inner loop. Provided that we choose J such that $v_a = I(v_c)$ implies $e_a = J(v_c)$*, we can always apply the rule

$$\{v_a = I(v_c)\}\ v_a := e_a\ \{v_a = J(v_c)\}.$$

If we, as agreed, strike the assignment to the superfluous abstract variable from the final program text, we have left

$$\{v_a = I(v_c)\}\ \{v_a = J(v_c)\}.$$

We see here that the assignment $v_a := e_a$ has been implemented *without any corresponding action*, merely by switching interpretations. By shifting our abstract point of view, the *meaning* of the unaltered reality is changed, roughly like the image on our retina changes by viewing an object from a different angle.

During an assignment

$$\{v_a = J(v_c)\}\ v_c := e_c\ \{v_a = I(v_c)\}$$

we return to the original point of view, and in doing so we simultaneously move the object such that on the face of it — that is, on the abstract level — nothing happens.

Applied to a simple example: The abstract real variable a is represented by a pair, consisting of a real variable m and an integer variable e, where $I(\langle m, e\rangle) = m \times 2^e$. (This representation leaves the freedom to choose m such that $m = 0 \vee \frac{1}{2} \leq |m| < 1$, which may be profitable in certain cases.) The assignment $a \times := 2$ may then be realized by switching to such a J that $a = I(\langle m, e\rangle)$ implies $a \times 2 = J(\langle m, e\rangle)$. Now $a \times 2 = I(\langle m, e\rangle) \times 2 = m \times 2^e \times 2 = m \times 2^{e+1}$, so, obviously, $J(\langle m, e\rangle) = m \times 2^{e+1}$. If we want to return later on to I, this is possible with the assignment $e +:= 1$ (or, possibly, with $m \times := 2$).

---

* Such a J always exists. Since e_a is a function of v_a, we have $J(v_c) = e_a = g(v_a) = g(I(v_c))$, so $J = g \circ I$ is an obvious solution. In this way we also see that each function f satisfying $g \circ I = I \circ f$ provides a solution $e_c = f(v_c)$ for returning to I. However, it is in general much simpler to manipulate the assertions directly, than to solve the equation $g \circ I = I \circ f$ explicitly first.

Finally, be it remarked that the assignments $m \times := 2$; $e -:= 1$ leave the assertion $a = I(\langle m, e \rangle)$ invariant; on the abstract level nothing happens either. This type of assignments often occurs to restore invariants of the concrete representation (such as $m = 0 \vee \frac{1}{2} \leq |m| < 1$).

5. EXAMPLE: A RANDOM PERMUTATION

In order to further illustrate the idea of freedom of representation, we consider the problem of obtaining a random permutation of a number of given elements, e.g., letters. This can be performed by putting the elements in a hat, and taking them out again one by one, each time drawing at random from the elements still left in the hat:

```
hat := the elements to be permuted;
seq := ε {{ the empty sequence }};
while hat not empty
do draw at random an element el from hat;
   add el to seq
od
{seq contains the required permutation}.
```

Note that *hat* and *seq* together always contain just the elements to be permuted (except perhaps for the fleeting moment the drawn element *el* is floating between *hat* and *seq*).

An implementation in ALGOL 68 (assuming we want to permute the letters of the word *anagram*):

```
string hat := "anagram";
string seq := "",
while hat ≠ ""
do int r = entier (random × upb hat) + 1;
   char el = hat[r]; hat := hat[ : r-1] + hat[r+1 : ];
   seq +:= el
od.
```

Using the observation that the number of elements in *hat* and *seq* togeth-

er is invariant, we are led to a less natural representation:

Let n be the number of elements to be permuted (for *anagram*: $n = 7$). Let, moreover, s be a variable for an array of (fixed) length n, and let k be an integer variable.

Then *hat* is represented by the pair $\langle s, k \rangle$, namely with the interpretation $hat = (s[k+1], \ldots, s[n])$, where *the order of the elements is immaterial**. Furthermore, *seq* is also represented by $\langle s, k \rangle$, but this time with $seq = \langle s[1], \ldots, s[k] \rangle$, or, for short, $seq = s[1:k]$. The implementation then runs as follows:

```
int n = 7; [1 : n] char s := "anagram";
int k := 0;
{hat = (s[k+1], ... , s[n]) ∧ seq = s[1 : k]}
while k ≠ n
{hat = (s[k+1], ... , s[n]) ∧ seq = s[1 : k]}
do int r = entier (random × (n-k)) + k + 1; {{ k+1 ≤ r ≤ n }}
   char el := s[r]
   {hat = (s[k+1], ... , s[r-1], s[r+1], ... , s[n]) ∧ seq = s[1 : k]}
   s[r] := s[k+1]
   {hat = (s[k+2], ... , s[n]) ∧ seq = s[1 : k]}
   k +:= 1
   {hat = (s[k+1], ... , s[n]) ∧ seq = s[1 : k-1]}
   s[k] := el
   {hat = (s[k+1], ... , s[n]) ∧ seq = s[1 : k]}
od
{hat = (s[k+1], ... , s[n]) ∧ seq = s[1 : k]}
{{ so, since k = n, seq = s[1 : n] = s }}.
```

Note that the interpretation $hat = (s[k+1], \ldots, s[n]) \wedge seq = s[1:k]$

---

* So, e.g., $(a, a, b) = (a, b, a) \neq (a, b)$. Such a data type, for which the terms *bag* and *multiset* have been suggested, lies as it were between sets (for which $\{a, a, b\} = \{a, b, a\} = \{a, b\}$) and sequences (for which all three cases differ). It is a pity that bags, which often play an important role in algorithms, are treated so stepmotherly in mathematics. Examples of possible mathematical applications: the eigenvalues of a matrix, and the edges of a graph (if two vertices may be connected by several distinct edges).

does not hold continually inside the do-part, but that it does hold at the beginning and at the end; in other words, this representation is *invariant* with respect to the do-part, and, therefore, to the whole loop-clause. What was accomplished in the first implementation by $hat := hat[\,:r-1] + hat[r+1:\,]$ is brought about in this version by switching temporarily to another interpretation, namely $hat = (s[k+1], \ldots, s[r-1], s[r+1], \ldots, s[n])$, *without corresponding action.*

It is interesting to see how a part of this version may be programmed almost "automatically" on the basis of the assertions.

We want to get from the assertion

(p) $\quad hat = (s[k+1], \ldots, s[r-1], s[r+1], \ldots, s[n]) \wedge seq = s[1:k]$,

which clearly expresses the fact that the element $s[r]$ has just been lifted from hat, to the assertion

(q) $\quad hat = (s[k+1], \ldots, s[n]) \wedge seq = s[1:k-1]$,

which shows that the drawn element still has to be added to seq. It is not clear how to transform q directly into p, but a contribution is supplied by the substitution $[\![\, k +:= 1]\!]$. We then obtain $\{q[\![\, k +:= 1]\!]\}\ k +:= 1\ \{q\}$, so we now have a simpler problem, getting from p to $q[\![\, k +:= 1]\!]$, which may be written as

(t) $\quad hat = (s[k+2], \ldots, s[n]) \wedge seq = s[1:k]$.

The difference between p and t may be expressed slightly more explicitly by rewriting the assertions as, respectively,

(p') $\quad hat = (s[k+1], s[k+2], \ldots, s[r-1], s[r+1], \ldots, s[n]) \wedge seq = s[1:k]$

and

(t') $\quad hat = (s[k+2], \ldots, s[r-1], s[r], s[r+1], \ldots, s[n]) \wedge seq = s[1:k]$,

or, since the order of the $s[i]$ in hat is immaterial,

(t'') $\quad hat = (s[r], s[k+2], \ldots, s[r-1], s[r+1], \ldots, s[n]) \wedge seq = s[1:k]$.

It is now clear that t'' is transformed into p' by the substitution $[\![\, s[r] := s[k+1]]\!]$, so we have

$$\{p\}\ s[r] := s[k+1]\ \{t\}\ k +:= 1\ \{q\}.$$

To conclude this example, we will examine what is needed in either implementation of the abstract algorithm in order to put the elements from *seq* back in *hat* afterwards, so that executing the loop-clause once more would yield another random permutation.

As for the first implementation, this becomes

hat := *seq*; *seq* := "".

As for the second implementation, we want to get from *hat* = () ∧ *seq* = = *s*[1 : *n*] to *hat* = (*s*[1], ... , *s*[*n*]) ∧ *seq* = *s*[1 : 0], which, under invariant interpretation *hat* = (*s*[*k*+1], ... , *s*[*n*]) ∧ *seq* = *s*[1 : *k*], can be expressed as *k* = 0. Consequently, the required transfer is obtained by

k := 0.

6. REPRESENTATION AND EFFICIENCY

Above we have already hinted at the possibility of improving the efficiency of an algorithm by a suitable choice of representation. If we make the quite reasonable assumption that in ALGOL 68 implementations the time of string assignments increases proportionally with the length of the string assigned, the order of the process for the example just worked out has been reduced by the second implementation from n^2 to n. For a given operation some representations happen to allow a considerably more efficient implementation than some others, If, for example, we have a large set S whose elements are pairs consisting of a name and a number, an abstract operation such as

N := ∪ *nr* : ⟨*name*, *nr*⟩ ∈ S

(i.e., finding the numbers belonging to a given name) may be implemented much more efficiently if the elements of S are sorted by name in the concrete representation. This representation is used in telephone directories. In ALGOL 68 we might write

flex [1 : *n*] struct (string *name*, int *number*) *list*.

Of course, we must maintain the correspondence between S and its representation, which is specified by S = ∪ *list*[*i*] : *i* ∈ [1 : upb *list*]. But during

look-up the following assertion must also hold:

$$i \leq j \Rightarrow \textit{name} \text{ of } \textit{list}[i] \leq \textit{name} \text{ of } \textit{list}[j], \text{ for } i, j \in [1 : \text{upb } \textit{list}].$$

If care is taken that this assertion is an invariant of the algorithm, by "establishing" its validity at the initialization of *list* and by restoring it after each assignment to (one of) its elements, the validity of this invariant may be used in the correctness proof for the implementation of an operation such as looking up a name.

Here we see a new element: *an invariant which has no meaning in terms of the abstract variable is adjoined to the concrete representation.*

Another regularly occurring phenomenon is *adjoining variables to the concrete representation* in order to formulate invariants which do have a relation in meaning to the abstract variable. If we represent a vector v by $v = c[1 : n]$, and the norm $\|v\|$ has to be determined often, it may be sensible to introduce a variable *sc2* with invariant $sc2 = \sum_{i=1}^{n} c[i]^2$, which then may be written $\|v\| = \sqrt{sc2}$.

It might be argued that the ordinary programmer, who does not know about abstract variables and invariants, already does intuitively what is suggested in these reflections. The fact that programmers sometimes succeed in writing even large, complicated programs and next making them "operational" by debugging, indicates that they have made use, possibly without realizing so, of structuring techniques during programming. The point, however, is that the programmer is often simply compelled to choose a representation (with adjoined invariants) which makes possible an efficient program, but that this is also a *precarious* matter: he has, in the back of his mind, if not on paper, the *abstract* algorithm, and the task of establishing or restoring the adjoined invariants forms no part of it! If this task is performed incompletely, the program may still appear correct to him. An additional problem is that this type of error — which, according to experience, seems much more common than errors in the underlying abstract algorithm — gives rise to phenomena that do not facilitate tracing the error: the error will usually manifest itself at a completely different spot in the program from where it was made, namely at the concretely worked-out efficiently implemented abstract operation. Quite often it will not come to light at all during test runs.

The following conclusions are inescapable:

- It is imperative that the adjoined invariants be *explicitly* formulated as part of the specification of the concrete representation of an abstract variable.
- The check that these invariants are established and that they are maintained or restored at each assignment which threatens to destroy them must be a routine part of the correctness proof for the implementation.

In the case of adjoined variables, this may often be done as follows. Let v_+ stand for the adjoined variables, and p for the invariant to be maintained. We assume that v_+ is disjoint from v_c, so that v_+ does not occur in $I(v_c)$ and an assignment to v_+ does not threaten the invariant $v_a = I(v_c)$. Moreover, we assume that v_c does not occur in p, so p depends only on v_+ and v_a and is not threatened by an assignment to v_c. The only threat to p then comes from an abstract assignment $v_a := e_a$.

The simplest approach is found by assigning to v_+ *before* the assignment to v_a, as in

$$\{p\}\ v_+ := e_+;\ v_a := e_a\ \{p\}.$$

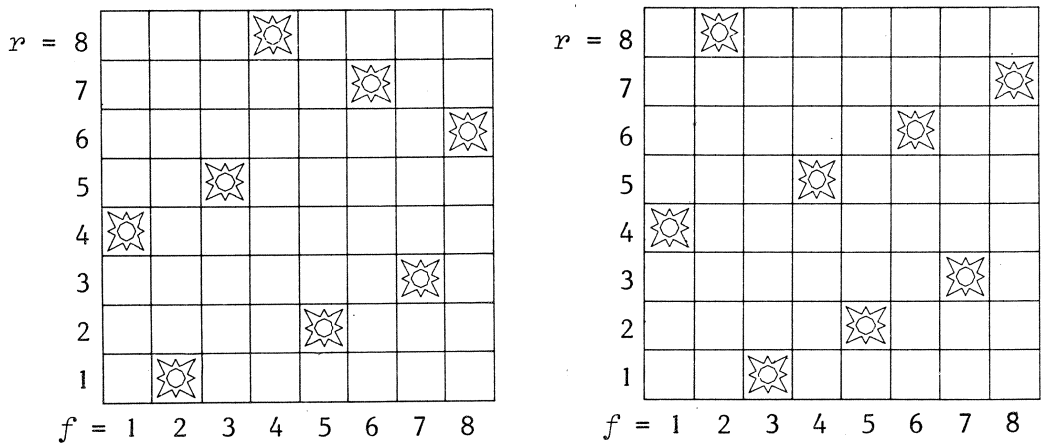
Applying the assignment rule backwards for the abstract assignment, this reduces to the problem of finding an expression v_+ to satisfy

$$\{p\}\ v_+ := e_+\ \{p[\![v_a := e_a]\!]\}.$$

7. EXAMPLE: THE EIGHT QUEENS

The eight queens problem (chosen more often as example [4 : 8]) runs as follows: give all configurations of eight queens on a chess-board in which none of the queens covers a square occupied by one of the other queens. In the figure below two configurations are shown: to the left a solution, and to the right a configuration which does not meet the requirements, since queen $\langle 2, 8\rangle$ covers square $\langle 7, 3\rangle$.

A configuration C may be considered a set of pairs $\langle f, r\rangle$, where f and r are the file and rank coordinates of the individual queens. The squares covered by a queen are characterized by the fact that one of the expressions f, r, $f + r$ (for \-diagonals) and $f - r$ (for /-diagonals) equals the corre-



sponding expressions for the square of that queen. If we define

$$OK(C) = \forall \langle f, r\rangle, \langle f', r'\rangle \in C : \langle f, r\rangle \neq \langle f', r'\rangle \Rightarrow f \neq f' \wedge r \neq r' \wedge f+r \neq f'+r' \wedge f-r \neq f'-r',$$

or, in plain English, no queen covers the square of another queen, then a solution C is characterized by $OK(C) \wedge \#C = 8$, with the tacit assumption that $f \in [1:8]$ and $r \in [1:8]$ for all $\langle f, r\rangle \in C$. It is clear that in an OK-configuration each file contains at most one queen, so that a solution has exactly one queen per file, which can be expressed as

$$F(C) = [1:8], \text{ where } F(C) = \cup f : \langle f, r\rangle \in C.$$

Note that a solution therefore satisfies $OK(C) \wedge F(C) = [1:\#C]$, which we abbreviate to $OKF(C)$.

If we have some C' with $OKF(C')$, $\#C' \geq 1$, then the configuration $C = \cup \langle f, r\rangle \in C' : f \neq \#C'$, which is obtained from C' by removing the queen from the rightmost non-empty file, will satisfy $OKF(C)$. We have, therefore, the following fact: given a method to generate, for some given value f, all OKF-configurations C with $\#C = f$, we also have a method to find all OKF-configurations C' with $\#C' = f+1$, since each can be obtained by adding an element $\langle f', r\rangle$ to some C, where $f' = f+1$. For $\#C = 0$ we already know the only possible configuration, $C = \phi$, which is also allowed, since we have $OKF(\phi)$.

We then obtain, in pseudo ALGOL 68 notation, the algorithm

config $C := \phi$; *extend*,

where *extend* is recursively defined by

```
proc extend = void :
{OKF(C)}
if #C = 8
then {OK(C) ∧ #C = 8}
     print(C)
else for ⟨f, r⟩ : OKF(C ∪ ⟨f, r⟩)
     do {OKF(C ∪ ⟨f, r⟩)}
        C ∪:= ⟨f, r⟩
        {OKF(C)}
        extend;
        C -:= ⟨f, r⟩
     od
fi.
```

Note that the correctness is partially based on the fact that the net effect of *extend* on C is nil. The simple inductive assertion method is not powerful enough to prove this, but it is easily proved, e.g., by induction on the recursion depth or by a generalization of the inductive assertion method (DE BAKKER & MEERTENS[9]).

The part

```
for ⟨f, r⟩ : OKF(C ∪ ⟨f, r⟩)
do .
   :
od .
```

can be worked out slightly further, using $f = \#C + 1$ and $r \in [1 : 8]$, to

```
int f = #C + 1;
for r to 8
do if OK(C ∪ ⟨f, r⟩)
   then .
        :
   fi .
od.
```

Here the OK- rather than the OKF-test may be used, since we know from OKF(C) that $F(C) = [1 : \#C]$, so $F(C \cup \langle f, r\rangle) = F(C) \cup f = [1 : \#C] \cup (\#C + 1) = [1 : (\#C + 1)]$. Moreover, since obviously $f \notin F(C)$, we know $\langle f, r\rangle \notin C$, so $\#(C \cup \langle f, r\rangle) = \#C + 1$.

For the concrete representation of C we can use

[1 : 8] int *rank*, int f,

with the interpretation $C = \mathrm{U}\ \langle i, rank[i]\rangle : i \in [1 : f]$, or, for short, $C = \mathrm{C}(f)$ (which implies $\#C = f$), but we will also use the interpretation $C = \mathrm{C}(f\text{-}1)$. The action $C \cup := \langle f, r\rangle$ is then simply implemented by assigning $rank[f] := r$ and switching simultaneously from the C(f-1)- to the C(f)-representation. We then obtain

```
[1 : 8] int rank, int f := 0
{C = C(f) { = C(0) = φ}}
proc extend = void :
if f = 8
then print ((rank, new line))
else {C = C(f)}
     f +:= 1
     {C = C(f-1)}
     for r to 8
     do if OK(C ∪ ⟨f, r⟩)
        then {C = C(f-1)}
             rank[f] := r
             {C = C(f)}
             extend
             {C = C(f)}
             {C = C(f-1)}
        fi
     od
     {C = C(f-1)}
     f -:= 1
     {C = C(f)}
fi;
extend.
```

Here we see another fine example of how an abstract assignment such as C -:= $\langle f, r\rangle$ is brought about, purely by switching from the interpretation $C = \mathrm{C}(f)$ to $C = \mathrm{C}(f\text{-}1)$, relying, of course, on the fact that $\langle f, r\rangle \in C$. The

interpretation $C = C(f\text{-}1)$ is an invariant of the loop-clause; the invariant $C = C(f)$ of *extend* therefore has to be restored afterwards by f -:= 1, which has no effect on the abstract level.

We are still left with the test $OK(C \cup \langle f, r\rangle)$. This is easily seen to be equivalent with

$$OK(C) \wedge (\forall \langle f', r'\rangle \in C : f \neq f' \wedge r \neq r' \wedge f+r \neq f'+r' \wedge f-r \neq f'-r').$$

Fortunately, we know that $OK(C)$ holds from the invariant of the abstract algorithm. We rewrite the remainder, distributing the quantifier, as

$$(\forall \langle f', r'\rangle \in C : f \neq f') \wedge (\forall \langle f', r'\rangle \in C : r \neq r') \wedge$$
$$\wedge (\forall \langle f', r'\rangle \in C : f+r \neq f'+r') \wedge (\forall \langle f', r'\rangle \in C : f-r \neq f'-r'),$$

which we abbreviate to

$$VER(f) \wedge HOR(r) \wedge$$
$$\wedge DIA1(f+r) \wedge DIA2(f-r).$$

$VER(f)$ is the question: $f \notin F(C)$, which, as we saw, may be answered in the affirmative. For $f, r \in [1 : 8]$ we have $f+r \in [2 : 16]$ and $f-r \in [-7 : 7]$, and in the hope to obtain an efficient implementation for the tests, we adjoin

$$[1 : 8] \text{ bool } hor, [2 : 16] \text{ bool } dia1, [-7 : 7] \text{ bool } dia2,$$

with invariants

$$hor[i] = HOR(i), \quad i \in [1 : 8],$$
$$dia1[i] = DIA1(i), \quad i \in [2 : 16] \text{ and}$$
$$dia2[i] = DIA2(i), \quad i \in [-7 : 7].$$

The test $OK(C \cup \langle f, r\rangle)$ can now be easily implemented as

$$hor[r] \wedge dia1[f+r] \wedge dia2[f-r],$$

but we have also taken on the task of maintaining the invariants.

For $C = \phi$ we find

$$hor[i] = HOR(i) = (\forall \langle f', r'\rangle \in \phi : i \neq r') = \text{true}, \quad i \in [1 : 8],$$

and, similarly, $dia1[i]$, $i \in [2 : 16]$ and $dia2[i]$, $i \in [-7 : 7]$.

For the assignment $C \cup := \langle f, r\rangle$ we obtain

$$HOR(i)[\![C \cup := \langle f, r\rangle]\!] =$$
$$= (\forall \langle f', r'\rangle \in C \cup \langle f, r\rangle : i \neq r') =$$
$$= ((\forall \langle f', r'\rangle \in C : i \neq r') \wedge i \neq r) =$$
$$= (HOR(i) \wedge i \neq r).$$

In order to maintain the invariant we therefore have to find a concrete assignment for

$$\{hor[i] = \mathrm{HOR}(i),\ i \in [1:8]\}\ v_+ := e_+\ \{hor[i] = (\mathrm{HOR}(i) \wedge i \neq r),\ i \in [1:8]\},$$

which succeeds with the choice $hor[i] \wedge := i \neq r,\ i \in [1:8]$, or, simply, $hor[r] :=$ false. For the other two invariants we find similarly $dia1[f+r] :=$ false and $dia2[f-r] :=$ false.

After the assignment $C -:= \langle f, r \rangle$, the invariants must once more be restored. For the same reason for which the simple assertion method failed — non-trivial recursion — this cannot be accomplished using analogous reasoning. Using recursion induction, however, it is quite simple. We take as inductive hypothesis that *extend* not only leaves C undisturbed, but also *hor*, *dia1* and *dia2*, so that it suffices to repair the "ostentatious" damage. Since the program point after "then", where the assignments will take place, is reachable only if $hor[r] \wedge dia1[f+r] \wedge dia2[f-r]$, the invariants can be restored by resetting $hor[r]$, $dia1[f+r]$ and $dia2[f-r]$ to true. The fully worked-out algorithm now becomes:

```
[1 : 8] int rank, int f := 0;
[1 : 8] bool hor, [2 : 16] bool dia1, [-7 : 7] bool dia2;
for i to 8 do hor[i] := true od;
for i from 2 to 16 do dia1[i] := true od;
for i from -7 to 7 do dia2[i] := true od;

proc extend = void :
if f = 8
then print ((rank, new line))
else f +:= 1
     for r to 8
     do if hor[r] ∧ dia1[f+r] ∧ dia2[f-r]
        then rank[f] := r; hor[r] := dia1[f+r] := dia2[f-r] := false;
             extend;
             hor[r] := dia1[f+r] := dia2[f-r] := true
        fi
     od;
     f -:= 1
fi;
extend.
```

8. AUTOMATIC CHOICE OF REPRESENTATION

At present, there is a trend in higher-level language design to introduce such modes (data types) as sets and sequences, and free the programmer from the task of choosing, specifying and implementing a concrete representation (SETL, see SCHWARTZ[10], or VERS2, see EARLEY[11]).

Some remarks on this development are in order. First of all, the value of sets, bags, sequences and trees in designing and specifying algorithms is so outstanding, that the language designer who addresses the problem of designing yet another general-purpose high-level programming language better have a good reason if he does not incorporate such modes. Secondly, automating the step from an algorithm using these modes to an efficient implementation has proved to be no child's play (LOW[12], SCHWARTZ[13], EARLEY[14]); the fear may be voiced that the time spent on optimizing during compilation will appear to be a (psychological?) threshold for acceptance. Finally, and this is the major point, it would be an illusion to think that even the most sophisticated system for automatic data representation choice will ever suffice to relieve the programmer from the task of specifying concrete representations. One example should serve to establish this point.

Suppose we have an algorithm which is concerned with a finite but very large set S of points in the Euclidean plane $\mathbb{R}^2$. The operation of determining, given a point $z_0 \in \mathbb{R}^2$, an element of S which minimizes the distance to z_0 is frequently performed, and so it has to be implemented efficiently. Nonetheless, the operation of merging two such sets must remain possible at reasonable cost. To these ends we can use the following representation:

We denote the square $\cup \langle x_1, y_1 \rangle : x \le x_1 < x+h \wedge y \le y_1 < y+h$ by $\square \langle x, y, h \rangle$.

The representation R = REPR(S, Q), where $Q = \square \langle x, y, h \rangle$, is then defined recursively as follows:

Case A: $S \cap Q = \phi$:

- R = empty;

Case B: $S \cap Q$ contains one point z:

- $R = z$;

Case C: $\#(S \cap Q) \ge 2$:

- R is the quintuple $\langle Q$, REPR(S, Q_1), REPR(S, Q_2), REPR(S, Q_3), REPR(S, Q_4)$\rangle$, where $Q_1 = \square \langle x, y, h/2 \rangle$, $Q_2 = \square \langle x+h/2, y, h/2 \rangle$, $Q_3 =$ $= \square \langle x, y+h/2, h/2 \rangle$ and $Q_4 = \square \langle x+h/2, y+h/2, h/2 \rangle$. (Note that Q_1, Q_2, Q_3 and Q_4 are mutually disjoint, and that $Q_1 \cup Q_2 \cup Q_3 \cup Q_4 = Q$.)

R is obviously a quaternary tree. If Q_0 is chosen sufficiently large, so that $S \subset Q_0$, we can take REPR(S, Q_0) as the representation of S. Under this representation one can use an efficient branch-and-bound method to determine an element minimizing the distance to a given point.

However, this representation has one disadvantage: if two points of S are very near each other, this is very likely to cause R to contain a long linear branch. This causes the amount of storage space occupied by R to be rather unstable and influences the efficiency unfavourably. (In fact, as is proved in VAN DE LUNE[15], even if the points of S are drawn at random from a uniform distribution over Q_0, the ratio between the expected number of nodes in R and $\#S$ does not tend to a limit as $\#S$ tends to infinity, but oscillates between positive bounds.) Therefore, we amend the representation in Case C as follows: if only one set $S \cap Q_{i_0}$ from the partitioning $S \cap Q_i$, $i \in [1:4]$, is non-empty, we take R = REPR(S, Q_{i_0}), thus retracting one edge in the tree. This retraction process is, of course, applied recursively. (This new representation can be viewed as a more concrete representation of the former one. Indeed, a programmer wishing to implement the latter representation would be well-advised to use the first representation as a stepping-stone.) Another important advantage of the latter representation is that we may consider REPR(S, Q_0) as being descended from an infinitely regressing sequence REPR(S, Q_{-1}), REPR(S, Q_{-2}), ..., with $Q_0 \subset Q_{-1} \subset Q_{-2} \subset \ldots$ and $Q_0 \cup Q_{-1} \cup Q_{-2} \cup \ldots = \mathbb{R}^2$. All elements in this sequence have identical (new) representations! This means that we need not bother with an initial choice of Q_0: if Q_0 no longer accomodates the current S , we simply move up the tree, changing the semi-abstract variable without corresponding concrete action.

Even a hypersophisticated compiler having this particular representation built-in — such a compiler might as well have *The* complete *Art of Computer Programming* built-in first — still cannot use it unless it "understands" the nature of the original problem, and especially the fact that a metric is involved. If it were not possible to give a simple expression for

$$\inf d(z_0, z) : z \in Q,$$

the whole branch-and-bound approach would fall flat. Moreover, the correctness critically depends on $x < x_0 < x+h$ implying $x < x+h/2 < x+h$, which seems a very reasonable property for digital arithmetic, and yet it cannot be derived

from the thirty axioms defining "proper" arithmetic postulated in VAN WIJNGAARDEN[16]. Understanding algorithms is as hard as any problem in mathematics, and each attempt at formalization merely gives rise to another undecidability result.

This digression is not meant to argue that automatic choice of data representations is not valuable. On the contrary, the more we can rely on the compiler to make a sensible choice in the simple cases with little to be gained, the more we can devote our ingenuity to the harder parts. Not only is reprogramming stacks for the hundredth time a tedious job, it is also begging for clerical errors due to waning attention. But a really (or very) high level programming language should nevertheless still cater for the programmer's need of specifying representations as a clearly discernible part of his program, not hidden in the murky details. It is then very natural to consider the "automatic" representation choice as a standard representation specification that can be overridden by the programmer.

9. HIGH-LEVEL PROGRAMMING LANGUAGES AND ABSTRACT VARIABLES

Programs are sometimes proclaimed to be well structured. If this has any meaning at all, it surely means that the structure of program development is reflected in the program text. Indeed, in order to understand a program text, it is necessary to understand the underlying abstract program, and it is very helpful if the program text is suggestive in this respect. The most important part of the documentation should be the abstract program, so a program text from which the abstract program is apparent may be termed self-documenting. One might even use the extent to which programming languages allow self-documenting programs as a measure of the height of their level.

If this principle is applied to abstract variables, one finds that high-level programming languages should allow the expression of a conglomerate of concrete variables as a single entity. This is in fact the essence of *records* as proposed by HOARE[17], and the above may be offered as an explanation after the fact of the fecundity of this proposal. Records have found their way into most recent programming languages; in ALGOL 68, e.g., they appear as *structured values*.

The ideal situation is that in which it is possible to separate the program in two parts: one part specifying the abstract algorithm, and one part

specifying the concrete implementation of the abstract variables and operations. (If more than two levels of abstraction are involved, this should, of course, be done recursively.) To some extent, this is possible in any high-level language. Alphard (WULF, LONDON & SHAW[18]) is centered around such a capability. Some diverse tools in various other languages that are suited to this purpose are the classes of SIMULA 67 (DAHL, MYHRHAUGH & NYGAARD[19]), the mode- and operation-definitions of ALGOL 68 (VAN WIJNGAARDEN & al.[20]), the — quite different — mode definitions and generic routines of EL1 (WEGBREIT[21]) and the clusters of CLU (LISKOV[22]); see also section 5 of MEERTENS[23]. The achievement of such separation through carefully controlled interfaces, as implemented in CDL2 and SLAN, is discussed in KOSTER[24].

For the ideal situation to be reached, several problems must be overcome. The remainder of this section is devoted to a summary of some of the problems for which no satisfactory solutions have yet been proposed if one adheres to rigid criteria of security and efficiency.

(a) *Several representations for one abstract mode.* It may well happen that an abstract algorithm contains several variables sharing one same mode (e.g., the mode "set of sequence of boolean"), but that completely distinct representations are called for. One approach is to adorn the abstract algorithm with pragmatic comments suggesting the particular representation desired, another is to discriminate in the specification section according to the identity of the abstract entity. In either case, alas, the neat boundary between abstract algorithm and representation specification is transgressed.

(b) *Cross-representing.* A change of (concrete) repesentation must occur when a value with one representation is assigned to a variable of the same abstract mode, but with a different representation (see under (a) above). Also, a representation change may be needed for "widening" from one abstract mode to a second one of which the former is, abstractly viewed, a submode (such as widening from "rational" to "real"). In either case the representation change is not expressible on the abstract level. It must therefore be performed "subcutaneously". For example, if we have a package specifying multi-precision real arithmetic and another package for multi-length integral arithmetic, we probably also need transfer functions between these data types. Now we could define something along the lines of

```
op widen = (int i) real :
if i < 0 then - widen - i
elif i = 0 then 0.0
else widen (i ÷ 2) × 2.0 + if odd i then 1.0 else 0.0 fi
fi,
```

but this is painfully expensive, and possibly inaccurate. The transfer function then must pierce the hulls supposedly defending the packages against illicit inquisition into their inwards.

(c) *Uniform references and overspecification.* The uniform reference condition (ROSS[25]) in connection with abstract variables requires that the syntactic form of operations on abstract variables depend on their abstract mode only, not on their representation. Even though a sparse matrix a may be implemented for some algorithm as two vectors of linear lists (one for the rows and one for the columns), indexing should still be written with a notation such as $a[i, j]$, but this must not commit the abstract algorithm to any specific concrete representation.

An approach is to consider this notation as syntactic sugar for the underlying construction ℵsub(a, i, j). The meaning of ℵsub may then be defined anew for each data type (cf. GESCHKE & MITCHELL[26]). Similarly, the privileges now conferred on some modes in the form of constant denotations, other special notations or coercions (implicit type conversions) must also be conferred on the programmer; otherwise, string z := "abc" would cease to be valid as soon as the programmer overrode the standard string representation.

The core of the problem is that of *overspecification*: on the abstract level the notation may already algorithmically specify the process beyond the point where the choice of concrete representation is left open. This problem is deeper than the mere notation of accessing operations. For example, the following piece of abstract program gives Warshall's method for computing the transitive closure of a boolean matrix:

```
for i from 1 to n
do a ∨:= a[ , i] × a[i, ] od.
```

A next step would be the specification of MATRIX ∨:= VECTOR1 × VECTOR2. One might quite naturally give the following definition as a first approximation:

```
(ref [1 : n, 1 : n] bool m) v:= ([1 : n] bool v1) × ([1 : n] bool v2) ⇐
for j from 1 to n
do if v1[j]
   then for k from 1 to n
        do if v2[k]
           then m[j, k] := true
           fi
        od
   fi
od.
```

But this makes it effectively impossible to specify a representation for sparse a and obtain the following sensible implementation:

```
ref line aji := (rows of a)[i];
while aji :≠: ref line (nil)
do ref line aik := (cols of a)[i];
   while aik :≠: ref line (nil)
   do insert (a, index2 of aji, index2 of aik);
      aik := next of aik
   od;
   aji := next of aji
od.
```

The problem is that the abstract program, in order to be less vague than "solve(*problem*)", must specify some flow of control, and the concrete representation may be chosen such as to make the actual flow of control more efficient. In the example, a better stepping-stone would be:

```
for j : v1[j], k : v2[k]
do m[j, k] := true od.
```

Even if such a clear intermediate specification is possible, it is not very realistic to expect the programmer to find it unless he has foreknowledge of the final concrete representation.

(d) *Generic mode constructors.* Once we have defined a representation for "bag of character", say, and defined all sorts of operations for this mode, it is a pity if we have to do this work all over again for "bag of integer

from 1 to N". It would be much better if we could give a mode as parameter to the definition of "bag of". This mechanism would also be useful for describing a standard representations specification section.

The desirability of such a feature is recognized, but no approach has been suggested which is not unreasonably restrictive and at the same time without run-time overhead. The problem is that operations on entities with generic modes can hardly be compiled until their actual modes are known (which, if the constructors are sufficiently general, cannot be determined statically anyhow). Maybe the solution here is to care less about this particular kind of inefficiency in view of the overall gain in efficiency one may expect to result from better programming techniques. An interesting and powerful approach is the treatment of modes in EL1 (WEGBREIT[21]).

REFERENCES

[1] WULF, W. & M. SHAW, *Global variable considered harmful*, SIGPLAN Notices 8, 2 (February 1973) 28-34.

[2] WIRTH, N., *The programming language PASCAL*, Acta Informatica 1 (1971) 35-63.

[3] HOARE, C.A.R., *Proof of a structured program: "The sieve of Eratosthenes"*, The Computer Journal 15 (1972) 321-325.

[4] WIRTH, N., *Program development by stepwise refinement*, Comm. ACM 14 (1971) 221-227.

[5] DIJKSTRA, E.W., *Notes on structured programming*, in: *Structured Programming*, O.-J. Dahl, E.W. Dijkstra & C.A.R. Hoare, APIC Studies in Data Processing 8, Academic Press, London, 1972.

[6] NAUR, P., *An experiment in program development*, BIT 12 (1972) 347-365.

[7] PLUM, T., *Random search on the 8-queens problem*, Software — Practice & Experience 4 (1974) 251-253.

[8] NEELY, P.M., *A new programming discipline*, Software — Practice & Experience 6 (1976) 7-27.

[9] DE BAKKER, J.W. & L.G.L.T. MEERTENS, *On the completeness of the inductive assertion method*, Journal of Computer and System Sciences 11 (1975) 323-357.

[10] SCHWARTZ, J.T., *On programming: An interim report on the SETL project. Installment 1: Generalities; Installment 2: The SETL language and examples of its use*, New York University, 1973.

[11] EARLEY, J., *High level operations in automatic programming*, Proceedings of a Symposium on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974) 50-59.

[12] LOW, J.R., *Automatic coding: choice of data structures*, CS-452, Stanford University, 1974.

[13] SCHWARTZ, J.T., *Automatic data structure choice in a language of very high level*, Comm. ACM 18 (1975) 722-728.

[14] EARLEY, J., *High level iterators and a method for automatically designing data structure representation*, (Journal of) Computer Languages 1 (1976) 321-342.

[15] VAN DE LUNE, J., *On the asymptotic behaviour of a sequence arising in computer science*, ZW 31/74, Mathematical Centre, 1974.

[16] VAN WIJNGAARDEN, A., *Numerical analysis as an independent science*, BIT 6 (1966) 66-81.

[17] HOARE, C.A.R., *Record handling*, ALGOL Bulletin 21.3.6 (1965), also in: *Programming Languages*, F. Genuys (ed.), Academic Press, New York, 1968.

[18] WULF, W.A., R.L. LONDON & M. SHAW, *Abstraction and verification in Alphard*, in: *New Directions in Algorithmic Languages 1975*, S.A. Schuman (ed.), IRIA, Rocquencourt, 1976.

[19] DAHL, O.-J., B. MYHRHAUGH & K. NYGAARD, *The Simula 67 common base language*, Norwegian Computing Centre, Oslo, 1968.

[20] VAN WIJNGAARDEN, A. & al. (Eds.), *Revised Report on the Algorithmic Language ALGOL 68*, Acta Informatica 5 (1975) 1-236.

[21] WEGBREIT, B., *The treatment of data types in EL1*, Comm. ACM 17 (1974) 251-264.

[22] LISKOV, B., *An introduction to CLU*, in: *New Directions in Algorithmic Languages 1975*, S.A. Schuman (ed.), IRIA, Rocquencourt, 1976.

[23] MEERTENS, L., *Mode and meaning*, in: *New Directions in Algorithmic Languages 1975*, S.A. Schuman (ed.), IRIA, Rocquencourt, 1976.

[24] KOSTER, C.H.A., *Visibility and types*, Proceedings of Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices 11, special issue; also FDT 8, 2 (1976) 179-190.

[25] ROSS, D.T., *Uniform referents: an essential property for a software engineering language*, in: *Software Engineering* 1, J.T. Tou (ed.), Academic Press, New York, 1970.

[26] GESCHKE, C.M. & J.G. MITCHELL, *On the problem of uniform references to data structures*, Proceedings International Conference on Reliable Software, SIGPLAN Notices 10, 6 (June 1975) 31-42.