# MODELLING OF STORAGE PROPERTIES OF HIGHER LEVEL LANGUAGES

Kurt Walk

IBM Laboratory Vienna

## ABSTRACT

The role of storage in the characterization of higher level programming languages is discussed. Assignment, in particular, has significantly different meaning in different languages, which can hardly be understood without reference to an underlying model of storage. A general storage model is sketched which can be specialized to a model of ALGOL 68 or of PL/I storage. The same model is used to discuss language features allowing highly flexible data structures.

## INTRODUCTION

The term storage commonly is used for certain hardware constructions like core stores or tape units. In a more general sense, storage is a carrier of information, characterized by the ways in which this information may be changed and by the ways which give access to the information. In this paper we shall characterize storage as it is seen by the programmer using a higher level programming language. Although the above mentioned hardware constructions ultimately will realize the programmer's storage, we shall find peculiar properties of the latter.

To begin with, the role of storage in connection with programming languages shall be briefly reviewed. By computing we usually want to do something useful within a certain world of problems, be it hydrodynamics, linguistics, accounting, or pure mathematics. This makes it necessary to represent aspects of this external world in the computer, and to interpret the behaviour of the computer in terms meaningful to the external world. This problem of representation and interpretation is likely to be overlooked if computing is considered to be just a play with numbers, but it is crucial for the understanding of programming languages.

A programming language, being the intermediary between worlds of problems and the computer, is necessarily a compromise between: adequacy for types of problems, general applicability, and ease of interpretation by the computer. The type of the compromise determines the type of the language ("special purpose", "universal", "for the beginner", etc.), and the rich possibilities for finding different compromises account for the mass of languages that exist (Sammet(1970)).

In the following we shall seek to understand these compromises in the kind of storage underlying programming languages, i.e. in the ways of representing and updating the state of some problem world being operated on. We restrict our attention to "general purpose" languages, which means to languages like ALGOL or PL/I. It is not the intention, however, to fully or even correctly describe all storage properties of any one of these languages, but rather to find essential similarities and differences.

A general purpose language not being oriented toward a particular class of problems, the question is in which way one could get guidance in conceiving the general notion of a "state" to be represented in storage. We can expect to get hints from logic. A system of logic could characterize a "universe" by

a set of individuals

a set of properties and relations,

defining the set of questions one may ask about a state of affairs, like

does individual a have the property P ?

does the relation R hold between individuals a and b ?

The set of answers defines the state. Thus, we can gain some guidance from logic for a systematic design of information retrieval languages. This way has been followed very successfully (Codd(1970), see also Mealy(1967)).

However, logic deals with derivations from state descriptions, but not with their change. The concept of updating, i.e. the partial change of information is unique to information processing. Although, in turn, computation steps can be interpreted as steps of logical derivation, one can hardly expect to get guidance for the design of storage from there. The various forms of assignment, which are the means of updating of information in our present programming languages, can be understood from the need for an economic realization in the computer, rather than from a logical point of view. Correspondingly, there are many different notions of assignment, and furthermore, the dissimilarities, although significant, usually are not at all apparent from the various language descriptions. This situation is the motivation for the study of properties of storage and their mathematical formulation, hence of storage models.

We can realize by the computer (partial) functional relationships, i.e. unique (partial) mappings from one set of entities to another set of entities. We could tentatively describe the state of storage by a mapping f

$$f:L \overset{\sim}{\rightarrow} V$$

from a set L (of locations) to a set V (of values). We say that $l \in L$ has the value $v \in V$, or is associated with v, or has the property v, if

$$f(l) = v.$$

An assignment statement

$$l := v'$$

will transform f to a new function f' such that

$$f'(l) = v'$$
$$f'(l') = f(l') \text{ for all } l' \in L,\quad l' \neq l.$$

Let us consider a special example in order to see the problems we have not solved yet with this construction. We use a snatch of data as it might be contained in the personnel data file of a company. For each employee we want to record

the hiring date,
the history of positions held,
the employee he reports to.

These data could easily be expressed in some logical notation. We want to express them as mappings, however. We need

a set of employees,
a set of dates,
a set of position lists

and the mappings:

hiring date:employees $\overset{\sim}{\rightarrow}$ dates
pos.history:employees $\overset{\sim}{\rightarrow}$ position lists
reports to :employees $\overset{\sim}{\rightarrow}$ employees.

'A reports to B' means: B = reports to (A) in functional notation.

Now having only one mapping f to manipulate by a language, we are in trouble. Identifying locations L with employees, and having an appropriate set of values V, we could represent just one of the above mappings.

We are better off if we can use further mappings of a special nature, usually offered by programming languages. For our example we might call them

first :locations $\overset{\sim}{\rightarrow}$ locations
second:locations $\overset{\sim}{\rightarrow}$ locations
third :locations $\overset{\sim}{\rightarrow}$ locations

Using first, second, third for hiring date, pos.history, reports to, respectively, we are able to represent e.g.

A was hired 1 JULY 70
A held positions: 101 and 105
A reports to B

as shown by the diagram (we use $a \xrightarrow{g} b$ to mean $b = g(a)$):

$$\begin{tikzcd}
 & l_1 \arrow[r, "f"] & \text{1 JULY 70} \\
A \arrow[ur, "\text{first}"] \arrow[r, "\text{second}"] \arrow[dr, "\text{third}"] & l_2 \arrow[r, "f"] & \langle 101,105 \rangle \\
 & l_3 \arrow[r, "f"] & B
\end{tikzcd}$$

where A and B are identified with some members of L. The question for the hiring date of A, for example, is answered by

f(first(A)).

In fact, this way of representing information (exemplified by the above diagram) in essence is the way offered by general purpose languages. Among the questions which remain open are:

which kinds of access to information are provided ?
which of the mappings can be updated ?
which kind of updating is possible, e.g. extending the domain of f ?

There will be special answers given in the next sections. Before proceeding, we make the following observations:

(1) The mappings first, second, etc. above can be viewed as mappings from locations to component locations. This leads to the concept of a record (or structure, or aggregate). A record is a composite piece of information, used to record a certain number of properties of an entity (see e.g. Hoare(1968)).

(2) The range of the location $l_3$ is locations, i.e. we have included locations in the set of values. This leads to the concept of pointers, which are provided in different languages under different names, and with varying properties.

(3) The range of the location $l_2$ are lists. Lists are entities which themselves have components. Updating the mapping third can mean that the number of components of a value changes. Languages differ significantly in the ways they can handle entities with varying components, if they can handle them at all. This problem is discussed in section 3. in some detail.

(4) Identifiers used in languages are not necessarily to be identified with locations in the above sense. In languages like ALGOL 68 or PL/I there is another mapping, from identifiers to locations, which offers the possibility that two identifiers share the same location. Most important, cooperating processes (subroutines, coroutines) may have their own naming system, with partially shared locations.

## 1. SIMPLE LANGUAGES

### 1.1 A very simple language

Let us first conceive of a very simple language in which we dispose of

a set of identifiers,
a set of values (e.g. integers).
Storage is a partial mapping from identifiers to values:

$S_s$ :identifiers $\rightharpoonup$ values

Identifiers are used

(1) for retrieving values: an identifier x stands for $S_s(x)$.
(2) for changing $S_s$. An assignment statement
x := v
x being an identifier and v a value, changes $S_s$ to $S_s'$, where $S_s'$ is characterized by

$S_s'(x) = v$

$S_s'(y) = S_s(y)$ for $y \in$ identifiers, $y \neq x$.

Depending on the context, therefore, an identifier stands either for an element of the range of $S_s$(a value), or for an element of the domain of $S_s$. Following Strachey(1966), we call the entity an identifier stands for in the first case a right-hand value, in the second case a left-hand value (according to the occurrence at the right-hand or the left-hand side of the symbol :=). In the present example, the left-hand value of an identifier is simply the identifier itself.

Of course, we are unable to represent the data of the introductory example using this language, even if the set of values would include identifiers to be used as pointers.

It should be noted that this simple language must not be confused with a machine code. A machine code similarly would provide just a simple mapping, from addresses to values. But the power of a machine code derives from the fact that the set of addresses has got structure. The addresses are ordered by the relation

$a_{i+1} = a_i + 1$

which together with having addresses as values, gives the possibility of computing addresses. On the power of machine languages see Elgot(1964).

### 1.2 ALGOL 60

We shall not be concerned (also not in the following sections) with the problem of the scope of identifiers. Identifiers are assumed to be unique.

Storage $S_{60}$ of ALGOL 60 is similar to $S_s$. But with respect to the above simple language, there are two extensions of concern for our present consideration in ALGOL 60:

(1) We have to distinguish between simple and array identifiers. Simple identifiers are left-hand values. Array identifiers are mappings from ordered lists of integers (the subscripts) to subscripted identifiers, which are left-hand values. For example, the declaration

<u>real</u> <u>array</u> a [1:10]

introduces a as a mapping

a:the integers 1,2,...,10 $\longrightarrow$ a[1],a[2],...,a[10]

where the a[i] belong to the domain of $S_{60}$ (are left-hand values).

(2) Identifiers which are name parameters may be associated with syntactic constructions like expressions. This association, however, cannot be changed by assignment. Access and assignment to storage via name parameters involves indirect steps, which allows for certain sharing patterns. Let, for example

$x \xrightarrow{S_{60}} 5$

and y be a name parameter associated with x. The right-hand value of both x and y is 5, and an assignment to y will, as a side-effect, also change the value of x.

We do not have in ALGOL 60 identifiers and left-hand values as clearly separate sets, and left-hand values are not part of the range of $S_{60}$. There is consequently no direct way for representing the example data of the Introduction in ALGOL 60.1)

## 2. A STORAGE MODEL

### 2.1 A general storage model

In this section we shall sketch a storage model which more fully is given in Bekić,Walk(1970) and which is adequate for expressing storage properties of ALGOL 68 and PL/I. Each of these languages imposes specializations on the model, which will briefly be discussed in the following section. The model will be developed in two steps.

#### 2.1.1 A model with non-flexible locations

Storage, again, is viewed as a partial mapping from a set L of locations to a set V of values:

$S:L \overset{\sim}{\longrightarrow} V$

1) By a direct way we mean one which does not map the whole problem to numbers.

Each location $l \in L$ is characterized by a range, given by the function range($l$), which is the set of values which may be associated with $l$. This is an important restriction, which is made with regard to the realization one has in mind: a location, realized by some piece of physical storage, has a certain limited storage capacity.

In constrast to ALGOL 60, there are composite values and composite locations. A composite value $v$ has

an index-set $I$,
components $v_i$, $i \in I$.

The components $v_i$ may be elementary or, again, composite. Examples for composite values are arrays and structures, where the index-set is the set of subscript lists, and the set of selector identifiers, respectively.

A composite location has a range $R$, which has
an index-set $I$,
component ranges $R_i$.
The members of $R$ are composite values $v$ with
index-set $I$,
and components $v_i \in R_i, i \in I$.

A composite location, correspondingly, has component locations

$l_i$, $i \in I$
with range($l_i$) = $R_i$, for $i \in I$.

We must add a further restriction on the mapping from locations to its components. For example, let $l$ have two composite components $l_1$ and $l_2$, then no components of $l_1$ and $l_2$ must be identical, or have common components. To express these requirements, we introduce an independence relation between locations, written as $l$ indep $l'$, which has the following properties:

$l$ is not independent of $l$
if $l$ indep $l'$, then also $l'$ indep $l$
if $l$ indep $l'$, then also $l$ indep $l'_i$,
for all components $l'_i$ of $l'$.

Now we can state the requirement for components of composite locations $l$:

$l_i$ indep $l_j$, for $i \neq j$.

The storage function $S$ is constructed in the following way. We start with a function $S_o$ from a set $L_o$ of independent locations to values:

$$S_o : L_o \xrightarrow{\sim} V$$

which satisfies the condition that if the value $S_o(l)$ is defined, it is within the range of $l$. Storage $S$

$$S : L \xrightarrow{\sim} V$$

is the consistently complete extension of $S_o$ under the condition for composite locations $l$:

$S(l) = v$ if and only if $S(l_i) = v_i$

for all elements i of the index-set of v. This means that we have extended the domain of the storage function to the set L which is the set of locations $L_o$ plus all component locations, components of components, etc. We call L the active locations of S.

There are three operations on storage: allocation, freeing, and assignment.

Allocation is the inclusion of new locations in the set L of active locations of S. Allocation may be implied by the declaration of a variable in a programming language, or it may be performed by an explicit command, like the ALLOCATE statement in PL/I. Allocation is described as an operation on $S_o$. Let l be a location which is independent of all locations $L_o$ of $S_o$, then

$(\text{allocate } l)(S_o) = S'_o$

where

$S'_o : L_o \cup \{l\} \xrightarrow{\sim} V$

$S'_o(l)$ is undefined

$S'_o(l') = S_o(l')$ for all $l' \in L_o$.

The resulting storage S' is given as the extension of $S'_o$. Freeing a location $l \in L_o$ correspondingly is defined by

$(\text{free } l)(S_o) = S'_o$

where

$S'_o : L_o - \{l\} \xrightarrow{\sim} V$

$S'_o(l') = S_o(l')$ for $l' \in L_o - \{l\}$

Assignment of a value v to a location l of S, $v \in \text{range}(l)$, $l \in L$, is defined by

$(l := v)(S) = S'$

where

$S' : L \xrightarrow{\sim} V$

$S'(l) = v$

$S'(l') = S(l')$ for $l' \in L$, $l'$ indep $l$

It follows from these conditions, that an assignment to a location l will, as a side-effect, change the values of the component locations, as well as the values of the locations of which l is a component.

It is time now to reconsider the example given in the Introduction. Clearly, we will allocate composite locations with three components for all employees we keep on record. All goes well if the ranges of the component locations contain the values: dates, position-lists, locations, respectively (up to now we have not made any assumptions about the elementary values in V).

The difficult part is having lists of varying length in a range of values. Lists are composite values and thus require composite locations. However, the ranges of composite locations introduced above have a fixed index-set, allowing a fixed number of components only. The ways out in the present frame-work are: using composite locations which pre-determine a certain length of the lists they may contain, or realizing lists by chaining independent locations together with pointers. There are languages, however, which directly provide the flexibility we are looking for. The present storage model, therefore, is extended in the next section to describe this capability.

### 2.1.2 Storage with flexible locations

Flexible ranges are introduced as the union of alternative (elementary or composite) ranges:

$R = R' \cup R'' \cup \ldots$

An example of flexible ranges are ranges of ALGOL 68 flexible arrays, i.e. of arrays with unspecified bounds. They are conceived as the union of alternative ranges of arrays with fixed bounds (see the next section).

We assume that R can be uniquely decomposed into its alternative ranges, i.e. given R and a value $v \in R$ we can uniquely determine the alternative range of which v is a member.

A flexible location has a flexible range. For each composite alternative range R it has component locations:

$l_i^R$, for i element of the index-set of R,

range$(l_i^R) = R_i$, $l_i^R$ indep $l_j^R$ for $i \neq j$.

(Note that we do not require independence of component locations of different alternatives.)

Thus for each alternative a flexible location may have different structuring. Since the alternative is determined by the value, we need the current value of a location to determine its current component locations:

l, i, v determines $l_i$

instead of

l, i determines $l_i$

for composite location.

We have to update our notions of independence, of storage and of the operations on storage for flexible locations.

The condition that if l indep l', l is also independent of all components of l', also holds for flexible l'.

Storage again is constructed as the extension of a function $S_o$ of independent locations to values, but with the additional condition for flexible locations l, that if $S(l) = v$, then $S(l_i^R) = v_i$, where R is the current, composite alternative range determined by v.

As a consequence, the domain of S (the active locations), can be changed by an assignment to a flexible location, if the current alternative is changed by the value assigned. The alternative of a flexible location cannot be changed, however, by an assignment to one of its components. This is expressed by an additional condition for the assignment operation.

Let $(l:=v)(S) = S'$

and l' a flexible location containing l as a component (or subcomponent), then $S'(l')$ is an element of the current alternative range of l' as determined by the original value $S(l')$.

In particular, it is not possible to 'add' a new component to a flexible location by assignment to the component. A reorganization of a flexible location can be caused only by assignment to the entire location.

It should be clear that the realization of flexible location presents a tougher problem, compared with the realization of locations with a fixed structuring. The use of flexible locations in ALGOL 68 therefore is restricted by a number of special rules (e.g. components of flexible locations must not be passed to parameters) which make it easier to cope with the problem of the varying set of active locations in an implementation of the language.

## 2.2 A Glance at ALGOL 68 and PL/I

The storage model developed in the previous section is used for a discussion of storage concepts in ALGOL 68 and PL/I.

### 2.2.1 ALGOL 68

We have to characterize domain and range of the storage functions for ALGOL 68, which leads to the special storage function $S_{68}$. They are both determined by the ranges of values that can be associated with ALGOL 68 variables.

There are elementary ranges including e.g. the set of reals and the set of integers of a certain length. Locations are called 'names' in ALGOL 68 terminology (van Wijngaarden(Ed.)(1969)). Sets of names having a range within a certain set of ranges, also form elementary ranges (we do not need a closer characterization of ranges of names for the present discussion).

N-dimensional arrays with specified bounds, and structures form composite ranges. A composite array range is given by

an n-tuple of bound pairs $<l_1:u_1,\dots,l_n:u_n>$
a non-array component range R

A structure range is given by

an n-tuple of identifiers $<id_1,\dots,id_n>$
an n-tuple of component ranges $<R_1,\dots,R_n>$

The index-set of the array range is the set of n-tuples of integers $<i_1,\ldots,i_n>$ with $1_k \le i_k \le u_k$, $1 \le k \le n$, the index-set of the structure range is the set of identifiers $id_1,\ldots,id_n$.

Components of array values are in R, the $id_i$-components of structures in $R_i$.

Flexible ranges are formed by sets of n-dimensional arrays with arbitrary bounds (or some of the bounds being arbitrary). Flexible array ranges can uniquely be decomposed into (an infinite number of) alternative array ranges with fixed bounds. Unions of ranges, e.g. the union of integers and reals, are further instances of flexible ranges.

There are two kinds of mappings for variables in ALGOL 68: the mapping from (unique) identifiers to locations (or 'names'), and the mapping S68 from locations to values. The first mapping realizes the relation of identifiers 'possessing' locations, the second one of locations 'referring to' values (using the terminology of the ALGOL 68 Report). The 'possess' relation is established by identity declarations, the 'refer to' relation by assignment.

An identity declaration has the form

declarer identifier = expression

where the declarer denotes a class of values and where in the case of a variable declaration the value of the expression is a location. Locations are produced by 'generators'. Given the specification of a range, the value of a generator is a location having this range. The execution of the identity declaration makes the identifier possess the location, provided it is within the class of values denoted by the declarer.

Examples:

(1) The declaration of x:

<u>ref</u> <u>real</u> x = (<u>loc</u> <u>real</u> := 3.14)

<u>ref</u> <u>real</u> is a declarer denoting the set of locations whose range is the set of reals, <u>loc</u> <u>real</u> is a generator producing some location l whose range is the set of reals, and 3˙14 denotes a real value. Execution of the declaration produces the situation:

$$x \xrightarrow{\text{'possess'}} l \xrightarrow{S_{68}} 3.14$$

(2) The declaration of y:

<u>ref</u> <u>ref</u> <u>real</u> y = (<u>loc</u> <u>ref</u> <u>real</u> := (<u>loc</u> <u>real</u> := 3˙14))

<u>ref</u> <u>ref</u> <u>real</u> is a declarer denoting the class of locations whose range is the class of locations with range <u>real</u>. <u>loc</u> <u>ref</u> <u>real</u> and <u>loc</u> <u>real</u> are generators producing locations, say l and m, whose ranges are <u>ref</u> <u>real</u> and <u>real</u>, respectively.

The situation resulting from the execution of the declaration is:

$$\begin{tikzcd}
y \arrow[r, "\text{'possess'}"] & l \arrow[r, "S_{68}"] & m \\
 & m \arrow[r, "S_{68}"] & 3.14
\end{tikzcd}$$

The identifier y may stand for l, for m, or for 3˙14, depending on the context of its use. The corresponding rules are called 'coercion' rules which, however, will not be elaborated in this paper.

(3) The array declaration of z:

<u>ref</u> [1:2] <u>real</u> z = (<u>loc</u> [1:2] <u>real</u> := (3˙14,2˙72))

leads to

$$\begin{tikzcd}
z \arrow[r, "\text{'possess'}"] & l \arrow[r, "S_{68}"] & (3.14{,}2\ 72) \\
 & l_1 \arrow[r, "S_{68}"] & 3.14 \\
 & l_2 \arrow[r, "S_{68}"] & 2.72
\end{tikzcd}$$

where l is a composite location. The right-hand value of the subscripted identifier z[1], for example, is obtained in the steps

$z \rightarrow l \rightarrow l_1 \rightarrow 3.14.$

The execution of an assignment statement involves, first, the evaluation of the left- and right-hand side of the statement, resulting in a location and a value, and second, the association of the value with the location. This latter operation, provided that the value is within the range of the location, is precisely the assignment operation defined for the general model in section 2.1.

A slight extension of this model would be necessary to fully explain the operations of rowing, slicing and left-hand-side composing. These functions are dealt with in Bekić,Walk(1970).

## 2.2.2 PL/I

Again, we start with a discussion of the ranges of values for PL/I locations (see PL/I Language Specifications(1966)).

Besides arithmetic and string values, we have to mention pointers and offsets as classes of elementary values. Pointers and offsets are used as "pointers" in the sense given in the Introduction. They are not locations, however, but derivatives of locations, as described below.

Arrays and structures are composite ranges similar to those in ALGOL 68, but with an <u>ordered index-set</u>. The ordering is significant for the use of pointers. There are no flexible arrays in PL/I.

A new type of values are areas. Areas are storages, and can be explained as pairs

$<\mathcal{L}, S_A>$

where $\mathcal{L}$ is a set of locations and $S_A$ is a storage in the sense of section 2.1. $\mathcal{L}$ is the set of locations that potentially can be allocated in $S_A$, and thus accounts for the size of the area. An area range is a set of pairs $<\mathcal{L}, S_A>$ with fixed $\mathcal{L}$. We shall see below that area ranges are flexible.

There are proper variables and parameters in PL/I, which become associated with locations when they are declared, or by explicit allocate statements. This association is similar to the 'possess' relation in ALGOL 68. Interestingly, however, there is also the possibility to construct locations from two constituents: a pointer and the specification of a range. The syntactic facilities for this are pointer variables and based variables. It is because of this feature (and a similar feature for storage overlay called defining) that we have to specify additional properties for PL/I storage $S_{PL/I}$, not already implied by the general model. In particular, we have to investigate the relation between locations and component locations.

Pointers are introduced as derivatives of locations. There is a mapping[1)]

addr:locations $\longrightarrow$ pointers

addr is not a one-to-one mapping, i.e. pointers contain less information than locations. We need additional information, namely the specification of a range, to get a location from a pointer. There is a function

construct(p,R)

which for a pointer p and a range R uniquely determines a location l such that

addr(l) = p
range(l) = R

It follows that a pointer can "point to" any type of location.

Syntactically, there is a built-in function ADDR(ref), which returns the pointer addr(l) of the location l associated with the reference ref (by a reference we mean the identifier of a variable plus possibly subscripts and selector identifiers). Pointers may be assigned to pointer variables. A based variable is declared with a certain range, the identifier of a based variable stands for this range specification. Let P be a pointer variable

1 We are neglecting here the problem of 'non-connected' locations which arises in connection with cross-sections. See Bekić,Walk(1970).

whose value is p, and B a based variable whose declared range is R, then the syntactic construction

$$P \rightarrow B$$

has a 'left-hand value' (see section 1) which is $l = \text{construct}(p,R)$, and a 'right-hand value' which is $S_{PL/I}(l)$. If p was the pointer to the location of another variable, this gives the possibility of accessing the other variable's storage, possibly via a different "mask", i.e. range specification, as compared to the range specification of the other variable.

The question arises as to which relationships are defined between two values $l(S_{PL/I})$ and $l'(S_{PL/I})$, when $\text{addr}(l) = \text{addr}(l')$, but the ranges of l and l' are different.

First, there is no relationship defined in PL/I (it may be defined by an implementation, of course) if the ranges of l and l' are elementary. Second, however, for composite l and l' there may be identical component locations of l and l'. This means that PL/I imposes special properties on the mapping from locations to component locations.

As mentioned above, the index-set of a composite range is ordered in PL/I. Component locations of a location, consequently, are ordered. Let l be a composite location. Its i-th component location $l_i$ is given by the function

$$l_i = \text{map}(\text{addr}(l), \langle R_1, \ldots, R_{i-1} \rangle, R_i)$$

where $R_i$ are the component ranges of R. This means that a location $l'$, $\text{addr}(l') = \text{addr}(l)$, has the same component location $l_i$ if its range has the same component ranges up to i, independently of the component ranges "to the right of" the i-th component. This property is called <u>left-to-right equivalence</u>.

There are further properties of this mapping defined, which make further uses of pointers and based variables possible which otherwise are undefined.

Such further properties simply can be added as axioms for the function map, restricting the freedom of an implementation of PL/I storage mapping. Most important is "structure-independent" mapping, which is determined just by the ordered set of <u>elementary</u> constituents of composite ranges, independently of their hierarchical structuring. This allows access to composite values in storage with differing structural descriptions, given by the range specifications of based variables. We shall skip the formal definition of these storage mapping properties (see Bekić, Walk(1970)).

We finally discuss PL/I areas. One can use based variables to allocate locations in areas, like in the example:

```
DECLARE A AREA(100), B BASED FIXED, O OFFSET;
        ALLOCATE B IN (A) SET (O);
        O -> B = 5;
```

in which A is declared as area variable with size 100, and the based variable B is used to allocate a location of range FIXED (fixed point numbers), say $l_B$ in A. The pointer (or rather offset, as it is called for locations in areas) of this location automatically is stored in the offset variable O, which subsequently can be used to reconstruct the location by the reference $O \rightarrow B$. The location of the area variable, say $l_A$, can be viewed as a <u>flexible</u> location whose components come into existence by allocations in A. The above allocation creates a component location $l_{AB}$ of $l_A$, defined by

$$S_{PL/I}(l_{AB}) = S_A(l_B)$$

where $S_A$ is the storage defined by the value: $S_{PL/I}(l_A)$ of A.

There are again special properties defined for storages of area type, which easily can be given as restrictions on the general model, but which are disregarded in this paper.

It should have become clear, though, that PL/I much more than ALGOL 68 constrains the way in which storage is to be realized. It is particularly for languages like PL/I that a storage model is mandatory for an understanding of the various storage-dependent features.

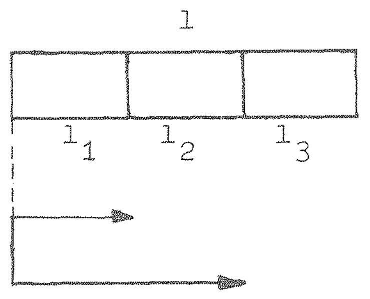
## 3. MORE FLEXIBILITY

Languages directly based on the storage model of the previous section are restricted with respect to the flexibility of the structuring of values that can be associated with locations. We first will discuss the reasons for this restriction, and then will see whether there are proposals which allow for more freedom in this respect.

The crucial point is the relation between locations and their components. For composite locations, we had:

$$l, i \text{ determines } l_i$$

which means that the existence and type of components is independent of the value assigned to locations. A structural change in storage S is possible only by allocation and freeing, i.e. by changing the domain of S or by using pointers to establish relations between locations. There is no independent allocation and freeing of component locations possible, however. The advantage of this concept is that it is realizable by <u>connected storage</u>: a composite location of the abstract model can be realized by a segment of adjacent (in the sense of the ordering mentioned in section 1.1) physical locations, where the component locations are identifiable by fixed offsets relative to the entire location:



The components thus are computable by simple address modification in a conventional computer. A further advantage is that the value of a composite location can be transferred block-wise e.g. to an external storage medium.

For flexible locations we had:

$l, v, i$ determines $l_i$

Here the structuring may depend on the stored value. We remember, however, the restricted possibilities for restructuring. Furthermore, flexible locations in general are not realizable by fixed blocks of connected storage. If no maximum size can be determined, the way out is to chain blocks of storage together with a pointer mechanism which is hidden to the programmer.

There are languages in which flexibility through a hidden pointer mechanism has been made a principle. The most prominent example for this is LISP. Although LISP can be understood withouth referring to its implementation, we expect that something can be learned for our present theme by looking into it.

## 3.1 A LISP-like system

Data in LISP, and also LISP expressions, are called S-expressions, which are either atoms or ordered pairs of S-expressions (McCarthy(1960)). We call the two components of an ordered pair the head and the tail, respectively. We show the way in which S-expressions are represented in storage, using a restricted version of the general storage model of section 2.1.

There are two kinds of locations: atomic locations and pair locations. The range of atomic locations is the set of atoms. Pair locations $l$ are composite, with two components $l_h$ and $l_t$. The range of $l_t$ and $l_h$ is:

elementary locations $\cup$ pair locations $\cup$ $\{NIL\}$

Example: Let ((A.B).C) represent an S-expression (a pair whose first component is the pair of A and B, whose second component is C). The following picture illustrates the storage representation of the S-expression, where $S_L$ stands for the storage function:

$$\begin{tikzcd}
 & & & l_h \arrow[r, "S_L"] & l_A \arrow[r, "S_L"] & A \\
 & l_h \arrow[r, "S_L"] & l' \arrow[ur] \arrow[dr] & & & \\
l \arrow[ur] \arrow[dr] & & & l_t \arrow[r, "S_L"] & l_B \arrow[r, "S_L"] & B \\
 & l_t \arrow[dr, "S_L"] & & & & \\
 & & l_C \arrow[r, "S_L"] & C & &
\end{tikzcd}$$

The S-expression is not stored as the value of <u>one</u> location, but its constituents are distributed among several independent locations. An expression can be identified, but not retrieved, by a single location.

There are the functions head and tail defined (called car and cdr in McCarthy (1960)) which return the head and tail, respectively, of an S-expression, and the function cons, which given two S-expressions produces the ordered pair having these S-expressions as components. These functions are realized such that their arguments are pointers, and that they return pointers representing S-expressions in storage:

$$(\text{head } l)(S_L) = S_L(l_h)$$
$$(\text{tail } l)(S_L) = S_L(l_t)$$

The function cons, besides giving a value, also changes $S_L$. Given two locations $l_1$, $l_2$ and $S_L$, we get the new storage by (see section 2.1):

$$\text{cons}(l_1,l_2)(S_L) = (\text{allocate } l;l_h:=l_1,l_t:=l_2)(S_L)$$ 1)

where l is some pair location not in the domain of $S_L$, and the value returned by cons is l.

Operations on data therefore may change the set of active locations in $S_L$. We note in passing that we have got only operations that increase the domain of $S_L$. This is of no concern in the theoretical model, in a realization one has to rely on a garbage collector to remove locations that are no longer used.

A variable to represent an S-expression is realized by associating the identifier of the variable with the location re-

1) We use (f;g)(x) to mean g(f(x)).

presenting the S-expression in storage, e.g. by a mapping 'denote':

$$x \xrightarrow{\text{denote}} l$$

Assignment of another S-expression, being represented by l' say, will change this association to

$$x \xrightarrow{\text{denote}} l'$$

Assignment, therefore, is not changing $S_L$, but the mapping 'denote'. The left-hand value of x is x itself.

Let us depict the way which leads from the location representing an S-expression to one of the atomic values:

$$l \rightarrow l_t \xrightarrow{S_L} l^1 \rightarrow l_t^1 \xrightarrow{S_L} l^2 \rightarrow \cdots\cdot \rightarrow l^i \xrightarrow{S_L} v^i$$

and compare this with a step from a flexible location to a component value:

$$l,S(l),i \rightarrow l_i \xrightarrow{S} v_i$$

In both cases, reference to stored values must be made in order to identify components. The latter case, however, relies on the existence of a composite value of the flexible location. The concept of composite values as values of locations is lost in the other model, the gain being a simpler type of storage: there is only one type of composite locations, with two components.

In the next section we shall present the model of a proposal following similar concepts. It uses locations with an arbitrary number of components, and reintroduces the concept of having locations as left-hand values of variables.

## 3.2 Data structures proposed by Standish

A system for declaring data structures and operations for their manipulation is proposed in Standish(1967). We shall give give a formal explanation of the basic lines of this proposal.

There are variables declared with descriptors, describing the class of entities[1] that can be associated with them. Descriptor identifiers can be defined to stand for descriptors, which allows for recursive descriptor definitions. A descriptor may have the following forms:

elementary descriptors describe classes of elementary values as in conventional programming languages, including "references" (to be explained as locations below);

1) The term 'entities' is used to avoid the term 'values' which has been used in a somewhat different sense in previous sections.

composite descriptors have the form $<d_1,\ldots,d_n>$, where the $d_i$ are descriptors. They describe entities whose components, respectively are described by $d_1,\ldots,d_n$ (We leave out for the present purpose particular component selectors and use the integers 1,...,n instead.);

iterative descriptors have the form repl d. They describe entities with an indefinite number of components described by d.

alternative descriptors have the form $(d_1|\ldots|d_n)$, where the $d_i$ are descriptors. They describe the union of the entities described by $d_1,\ldots,d_n$. This form is relevant only for range tests, but not for storage organization.

NIL describes the "empty data space".

There are also descriptor modifiers defined. They will be disregarded in the present context, as they do not concern the storage organization.

Like in the LISP system, composite entities are not the values of one (connected) location, but are realized by a pointer structure. Identifiers are associated with locations as left-hand values by a mapping 'denote'. This mapping is not changed by assignment.

We need the following types of locations:

elementary locations whose range is specified by elementary descriptors,

pointer locations whose range is the set of locations $\cup$ {NIL},

composite locations with a fixed number of components which are pointer locations,

flexible locations with an indefinite number of components which are pointer locations.

An elementary entity, say A, is represented in storage $S_{ST}$ as

$$l_A \xrightarrow{S_{ST}} A$$

where $l_A$ is an elementary location. We say that $l_A$ points to A.

A composite entity, say $e = (e_1,\ldots,e_n)$, $e_i$ being entities, is represented as

$$\begin{tikzcd}
 & l_1 \arrow[r, "S_{ST}"] & l_{e_1} & — \\
 & \cdots & & \cdots \\
l \arrow[uur, dash] \arrow[ur, dash, dashed] \arrow[dr, dash, dashed] \arrow[ddr, dash] & \vdots & & \\
 & \cdots & & \cdots \\
 & l_n \arrow[r, "S_{ST}"] & l_{e_n} & —
\end{tikzcd}$$

where l is composite or flexible according to whether e belongs to a composite or to an iterative descriptor. l is said to point to e. $l_1,\ldots,l_n$ are pointer locations. $l_{e_1},\ldots,l_{e_n}$ point to $e_1,\ldots,e_n$, respectively.

There is an operation cons(d,ap) on storage, where d is a descriptor and ap an argument part. An argument part is an elementary entity or a list of argument parts. cons(d,ap) represents ap in storage according to the description d. It is defined as follows under the assumption that ap has been tested to be representable with d, and that d is elementary, composite, or flexible:

```
    $cons(d,ap)(S_{ST}) = f(d,ap,l)(allocate\ l(S_{ST}))$
```

where l is a location not in the domain of $S_{ST}$ which is elementary, composite, or flexible according to whether d is elementary, composite, or iterative, and f is defined as

```
    $f(d,ap,l)(S_{ST}) =$
        if d is elementary:  $(l:=ap)(S_{ST})$
        if $d = <d_1,...,d_n>$, $ap = <ap_1,...,ap_n>$:
            $(allocate\ l^1;...;allocate\ l^n;l\ :=l^1;...;l_n:=l^n;$
                $f(d_1,ap_1,l^1);...;f(d_n,ap_n,l^n))(S_{ST})$
        if d = repl d', $ap = <ap\ ,...,ap_n>$:
            $(allocate\ l^1;...;allocate\ l^n;l:=<l^1,...,l^n>;$
                $f(d_1,ap_1,l^1);...;f(d_n,ap_n,l^n))(S_{ST})$
```

$l^1,...,l^n$ are new locations not yet in the domain of $S_{ST}$.

A variable, when declared, becomes associated with a left-hand value by a mapping 'denotes'. The left-hand value is a pointer location, whose value is the location pointing to the entity currently associated with the variable. This gives the possibility of sharing between variables:

$$\begin{tikzcd}
x \arrow[dr] & & \\
& l \arrow[r, "S_{ST}"] & l_e \\
y \arrow[ur] & &
\end{tikzcd}$$

Variables described as composite or iterative may be subscripted. The left-hand value of a reference $x[i_1,...,i_k]$ is given by:

```
    $lhv(l,<i_1...,i_k>)(S_{ST}) =$
      if k=0:  l
      otherwise:  $lhv(S_{ST}(l_{i_1}),<i_2,...,i_k>)(S_{ST})$
```

where: l = denotes(x). The right-hand value of a reference is given by $S_{ST}(l)$, if l is its left-hand value.

There is the possibility of syntactically controlling the number of indirect steps. If ref is a reference with left-hand value l, then the right-hand value of $\rightarrow$ ref is l. If r is the right-hand value of ref, then the right-hand value of <ref> is $S_{ST}(r)$.

The way from an identifier x to a component value v can be depicted as:

$$x \xrightarrow{\text{denotes}} l \xrightarrow{S_{ST}} l^1 \longrightarrow l_i^1 \xrightarrow{S_{ST}} l^2 \longrightarrow \dots \longrightarrow l^k \xrightarrow{S_{ST}} v$$

which shows the greater variability with respect to LISP. It is gained, of course, through partially abandoning the consequent simplicity of LISP. In particular, there are flexible locations again, though of a very special type.

There are various assignment operations defined. Let $l_1$ be the left-hand value of $ref_1$, and $l_2$ the right-hand value of $ref_2$. The assignment $ref_1 \leftarrow ref_2$ is defined by

$$(l_1 \leftarrow l_2)(S_{ST}) = (l_1 := l_2)(S_{ST})$$

Note that this assignment does not imply a data transfer, but just a pointer operation.

Overlay assignment $ref_1 \leftarrow\leftarrow ref_2$ is defined by

$$(l_1 \leftarrow\leftarrow l_2)(S_{ST}) = (S_{ST}(l_1) := S_{ST}(l_2))(S_{ST})$$

Note that if $ref_1$ is described as iterative, this operation may cause a restructuring of the flexible location $S_{ST}(l_1)$.

Copying of entities in storage (assignment does not imply copying) can be induced by copy functions. Let l point to an entity in storage.

The function copy(l) returns a new location l' with range(l') = range(l) and changes $S_{ST}$ to

$$(\text{copy } l)(S_{ST}) = (\text{allocate } l'; l' := S_{ST}(l))(S_{ST}).$$

The entire entity is copied by the function copy-all(l). It also returns a new location l'. Its operation on storage is defined by

$$\text{copy-all}(l)(S_{ST}) = (\text{allocate } l'; \text{cons-copy}(l', l))(S_{ST})$$

and

```
cons-copy(l',l)($S_{ST}$) =
    if $S_{ST}$(l) is elementary:  (l':=$S_{ST}$(l))($S_{ST}$)
    if $S_{ST}$(l) is composite or flexible with n components:
      (l':=$S_{ST}$(l);allocate $l^1$;...;allocate $l^n$;
       $l_1'$:=$l^1$;...;$l_n'$:=$l^n$;
       cons-copy($l^1$,$S_{ST}$($l_1$));...;cons-copy($l^n$,$S_{ST}$(l)))($S_{ST}$)
```

where $l^1, \dots, l^n$ are new locations with $\text{range}(l^i) = \text{range}(S_{ST}(l_i))$.

Note: the assignment $l':=S_{ST}(l)$ just serves the purpose of creating correctly the active component locations of l', if it is flexible.

REMARK

In conclusion, we point out a subject which is not elaborated at all in this paper, but which is significant for the economic representation and handling of data in storage. Following Wegner(1970), we call it the variability of information structures. Its investigation would shed light on a variety of features of programming languages, and may lead to an at least semi-quantitative treatment of storage problems.

Consider the relation

$$l \xrightarrow{S} v$$

where we had required that v is within a range R determined by l : R=range(l). Whatever the result of an updating of this relation may be, we know that a new value, v', again will be within R. Claiming that updating is costly, we can utilize this redundancy by associating not the value v with l, but a derivative of v which we call value representation vr. It is given by a function rep:

$$vr = rep(v,R)$$

with the condition that there exists a function val such that v can be retrieved:

$$v = val(vr,R).$$

The amount of information reduction that can be achieved depends on the size of the set of values and of R. Of course, the information about R still must be in l, but this is a relation which needs not to be updated.

In Lucas,Walk(1969) a storage model is described which is based on value representations, and which thus is closer to a real implementation than the model presented here.

Another relation whose variability is significant is that between identifiers and locations. In languages where identifiers are declared, they may be associated with locations only of the specific types described in the declaration. For example, the range of the locations may be fixed in a declaration. In the case of PL/I locations for example, we could instead of

$$id \xrightarrow{den} l$$

have

$$id \xrightarrow{den} p$$

$$id \xrightarrow{attr} R$$

where l can be reconstructed from the pointer p and the range R:

l = construct(p,R)

and attr is a relation which needs to be established only once.

Looking for relations which remain fixed during certain segments of computations is precisely the problem of implementations to find actions that can be performed once and for all during compile time, or at block activation time etc. It is the key also for understanding the difference between compiler and interpreter oriented languages.

## REFERENCES

Bekić,H.,Walk,K.(1970), Formalization of Storage Properties, to appear in Symposium on the Semantics of Algorithmic Languages, Springer Lecture Notes Series.

Codd,E.F.(1970), A Relational Model of Data for Large Shared Data Banks, Comm. ACM, Vol. 13, No. 6, pp. 377-387.

Elgot,C.C.,Robinson,A.(1964), Random Access, Stored Program Machines, An Approach to Programming Languages, J.ACM, Vol. 11, No. 4, pp. 365-399.

Hoare,C.A.R.(1968), Record Handling, in F.Genuys(Ed.), Programming Languages, Academic Press.

McCarthy,J.(1960), Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, Comm. ACM, Vol. 3, No. 4, pp. 184-195.

Mealy,G.H.(1967), Another Look at Data, FJCC, Conference Proceedings, Vol.      pp. 525-534.

PL/I Language Specifications(1966), IBM System Reference Library, Form No. C28-6571-4.

Sammet,J.E.(1970), Roster of Programming Languages, SIGPLAN Notices, Vol. 5, No. 8, pp. 19-25.

Standish,T.A.(1967), A Data Definition Facility for Programming Languages, Carnegie Institute of Technology, Pittsburgh, Pennsylvania.

Strachey,C.(1966), Towards a Formal Semantics, in T.B.Steel(Ed.), Formal Language Description Languages, North Holland.

Van Wijngaarden,A.(Ed.),Mailloux,B.J.,Peck,J.E.L.,Koster,C.H.A. (1968), Report on the Algorithmic Language ALGOL 68, Second printing by the Mathematisch Centrum, Amsterdam.

Wegner,P.(1970), The Variability of Computations, Technical Report, Center for Computers and Information Sciences, Brown University, Providence, Rhode Island.