stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 188/81 DECEMBER

L.G.L.T. MEERTENS

DEFINITION OF AN ABSTRACT ALGOL 68 MACHINE

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, 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.).

1980 Mathematics subject classification: 68B20

ACM-Computing Reviews-category: 4.12, 4.22

Definition of an abstract ALGOL 68 machine

by

L.G.L.T. Meertens

ABSTRACT

This report contains the definition of a machine-independent abstract machine, the "MIAM", whose code may serve as the target code for a portable ALGOL 68 compiler. Implementing ALGOL 68 with the MIAM entails two steps: implementing ALGOL 68 in terms of the MIAM, and implementing the MIAM in terms of an actual computer. This report defines only the "core" of the MIAM, which is sufficient to model all actions prescribed in the sections headed with "Semantics" of the Revised Report, with the exception of the widening coercions and the denotations.

KEY WORDS & PHRASES: ALGOL 68, abstract machine, compiler, portability

0. INTRODUCTION

This report contains the definition of the "MIAM" ("Machine-Independent Abstract Machine"), whose code may serve as the target code for a portable ALGOL 68 compiler. The philosophy that has governed the design of the MIAM, and notably the "cut principle", have been described in [1] and will not be repeated here. In making this definition available, it is hoped that it may also be instructive in the task of creating an ALGOL 68 compiler for a fixed target machine.

The definition given here, just as the ALGOL 68 Revised Report [3] (some familiarity with which is assumed), is not easy to read. Since the definition of a MIAM is a contract (although not with legal standing), it aims at a level of precision that is threatened by the informality required for an enjoyable exposition. Worse even, any reliance on whatever assumption, however reasonable by itself, as to how the MIAM will be used in translating ALGOL 68 programs, but that cannot be rigorously deduced from the definition proper, destroys the cut principle immediately.

However, some considerations as to why particular solutions have been chosen, suggestions for possible implementation approaches and other hopefully helpful remarks are incorporated in the text by placing them between "double pragmatic brackets", viz., "{{" and "}}". These remarks should in no way be construed to be part of the definition {{although they may be helpful to show the intended meaning in the case of shortcomings in the definition}}.

Implementing ALGOL 68 with the MIAM entails two steps: implementing ALGOL 68 in terms of the MIAM, and implementing the MIAM in terms of an actual computer. In order to reduce confusion, the term "translation" will be used for the former, and "realization" for the latter step.

The definition given in this report defines only the "core" of the MIAM. For a complete definition, a large number of relatively simple instructions have to be added, e.g., to deal with the numerous operations defined in the Standard Prelude. The core defined here is sufficient to model all actions prescribed in the sections headed with "Semantics" of the Re-

vised Report, with the exception of the widening coercions and the denotations.

One caveat is in order. The MIAM described here has not been tested in an actual effort to translate ALGOL 68, nor has it been realized on an actual computer. Such an effort is bound to bring problems to light that have not been foreseen in the design phase.

1. THE MACHINE AND THE PROGRAM

1.1. Tokens

a) A "token" is a primitive entity. Tokens that are named with different names are different.

{{Tokens serve as literals without inherent meaning or internal structure by virtue of which it is possible to discriminate certain entities.}}

1.2. Registers

a) A "register" is a variable for holding certain objects. {{No relation with hardware registers, if any, of the realization on an actual computer is assumed.}} Global registers of the MIAM are "T" and "S" for holding keys of locales, and "P" for holding a pointer to a process descriptor. If a register R is "set to" an object x, this means that R is made to hold a copy of x. If, subsequently, "R" is used in a context where an object is asked for, it stands for the object currently held in R.

1.3. Areas, pointers and models

{{Areas are the basic way to abstract from memory management. An area, once created, has a fixed (i.e., dynamically invariant) size. In the realization, areas will be modelled by contiguous segments of memory. These segments contain "hidden" fields; notably the scope and the model of the area, and presumably also the span, giving its size.}}

a) A "span" is a nonnegative integer. {{The term "span" is introduced to avoid confusion with the ALGOL 68 term "size", which has a totally different meaning. Since integers may be used in the MIAM for index calculations, they correspond to the integers of "size 0" (i.e., having the mode INT) from ALGOL 68.}}

b) An "area" has a span s and a "key" k, and is then composed of a sequence of s contiguous {{memory}} "cells", selected by "pointers", which are denoted k·0, k·1, ..., k·(s-1).

{{Areas come into being as the result of a GEN or EST-instruction. It is not further specified here what cells are, but in the realization they should correspond to the smallest units of memory that are adressable in an efficient way. On byte-oriented machines, this will be a byte. The key of an area and the pointer of a cell both correspond, in the realization, to its address. The main reason for maintaining a distinction between "keys" and "pointers" is that the operations: "given the pointer k·i, determine the corresponding address" and "given the pointer k·i, determine the key k" must both be efficiently realizable; the latter to make garbage collection tolerably efficient. A simple way is to represent the pointer k·i by a pair <address(k), address(k)+i>, although this requires more memory for a pointer. Under this realization scheme, a key may be represented more succinctly than a pointer.}}

c) Each cell is uniquely determined by its pointer, and vice versa, that is, $k_1 \cdot i_1$ and $k_2 \cdot i_2$ select the same cell if and only if $k_1 = k_2$ and $i_1 = i_2$.

{{The effect of garbage collection and compaction is transparent: although, in a realization of the MIAM, cells may be physically moved, the corresponding representations of keys and pointers are accordingly updated. This requires, of course, that the realization of the MIAM is able to keep track of all pointers.}}

d) The key of an area is said to "access" that area, and the pointers of an area are also said to "access" that area {{although the pointers point to sites in the area}}.

e) If p = k·i, where p is a pointer and k is a key, then p·j stands for k·(i+j); it is "required" {{1.7.1.f}} that $0 \leq i+j < s$, where s is the span of the area accessed by k.

f) Areas have a "scope", which is a nonnegative integer. The scope "of" a key or pointer is the scope of the area that it accesses {{and the scope of other objects is the largest (i.e., "newest") scope of component keys or pointers}}.

{{This scope is akin to the ALGOL 68 scope; it indicates the nesting of lifetimes of areas, and thereby of objects residing in the areas. By defining the scope of objects in terms of the scope of areas, it is sufficient to remember, in a realization, one scope per area, instead of per object, reducing the memory requirements. A price is paid in that it may be necessary to keep a whole area because of one object that may not be relinquished, as in the presumable translation of REF INT X = (HEAP [large] INT) [1].}}

g) A copy of an "object" {{1.5.a}} of a "type" {{1.4.a}} <k, a, l, u, m> "occupies" a "site", "appointed" by a pointer p = q·a which is "suitable" for that type, and the site then consists of the cells selected by q·l, ..., q·u.

{{The so-called "dynamic parts" are not considered part of the object. In the translation, they are represented by pointers.}} If q is k·i, where k is the key of an area with span s, then p is suitable for the type if i is a multiple of m, $0 \leq i+l$ and $i+u \leq s$. The object may be denoted, in a context where the type is known, by *p. {{If the type is not known, the denotation "*p" is ambiguous.}}

h) New copies may be made to occupy sites, thereby obliterating (parts of) former copies, if any. If x denotes an object of known type, x =* p stands for the action by which a new copy of the object x is made to occupy the site appointed by p; it is required {{1.7.1.f}} that p be suitable for x and that the scope of x be at least that of p. Moreover, if x is a label, it is required that x not be abortive {{1.7.1.d}}.

{{It can be shown that a key occupying a site, or held in T or S, must be the key of a locale (1.6.a). The GEN-instruction (3.3.a) for creating an area that is not a locale does not make its key available, and it is impossible to retrieve the key of an area from a pointer accessing it by means of MIAM-instructions.}}

i) The phrase "p appoints a site occupied by a copy of x" may be shortened to "p points to x".

j) The objects "contained in" an area are the objects pointed to by pointers of the area. {{Care should be taken not to confuse the pointers "of" an area, i.e., selecting a cell of the area, and the pointers "contained in" an area.}}

k) Apart from the "proper" pointers introduced in section 1.3.b, there exists a {{unique}} dummy pointer, the token "Nix", which does not access any area {{and thus does not select any cell}}, and which is unsuitable for any type. The scope of Nix is 0.

l) A "model" is a {{possibly empty}} set of pairs <t, d>, where t is one of the tokens "KEY" and "PTR", and d is an integer. Each area has a model, which may vary as a result of execution.

m) An area A is "reachable" if it is accessed by T or S {{1.6.b}}, if it is the C-locale {{1.6.c}}, if P {{1.5.2.c}} accesses the area, if the area contains a process descriptor whose token is Halted, or if some site in a reachable area A´ is occupied by the key of A or by some pointer accessing A. An area is reachable only if it is reachable by virtue of the previous sentence.

n) "Model conformance" holds if the following three criteria are met:

(i) The model of the C-locale {{1.6.c}} is empty;

(ii) The C-locale contains no pointers or keys accessing a different locale;

(iii) For each reachable area, other than the C-locale, there is a one-to-one correspondence between the pointers k•d of that area that point to a key (a pointer), and the elements <KEY, d> (<PTR, d>) of the model, with a possible exception for keys and pointers that are Nix or that access the C-locale.

{{If model conformance holds, this means that the sites of keys and pointers may be deduced from the model for purposes of garbage collection and area compaction. In the realization, a specialized representation of the sets that models are may be used. Because of the dynamic span determination of areas that are not locales, the collection of models that may play a role during execution is not finitely bounded. However, the corresponding models have a repetitive structure and may be represented in the realization by means of "hyper-models", by allowing, roughly, the abbreviation "and so on until the end of the area". The set of hyper-models that may play a role can then be kept finite and may be determined statically.}}.

o) The formula "m+d´", where m is a model and d´ is an integer, stands for the model {<t, d+d´> | <t, d> ∈ m}.

1.4. Types

{{Types in the MIAM are akin to ALGOL 68 modes. They are static attributes that allow the realization of efficient treatment of objects. The main difference with ALGOL 68 is that a given type "describes" the lay-out of a contiguous segment of memory of fixed size. Pointers have a common type that does not contain the type of the object pointed to. {{There is no such thing as type checking in the MIAM.}} However, in operations manipulating objects through the access provided by a pointer, the type of that object is always statically known.

The "philosophy" behind the type model used in the MIAM is as follows. In actual computers, there are certain privileged "primitive types" in terms of whose semantics the machine instructions are described. An efficient realization must make use of this fact whenever reasonably possible. Now it may occur that not all addresses are equally suited for storing objects of a given primitive type. For example, it may happen that integers may be operated upon efficiently only if they are stored at adresses that are a multiple of four. If the design of the MIAM does not recognize this fact of technology, efficient realization of the MIAM is out of the question. Now the translation phase should not be bothered by parameters of the target hardware: the MIAM code turned out for a particular ALGOL 68 program should be identical. The solution chosen is to assign types to objects of the MIAM in such a way that the realization may choose adresses for MIAM pointers suitable for the objects pointed to. Whether this is indeed possible for a given contraption depends on how reasonable its adress restrictions are. The model developed here will not cater for the case where integers may not be stored at adresses that are one more than some prime number. The assumptions used are:

(i) The hardware adresses suitable for a given primitive type P are of the form $a_P + n * m_P$, where n runs through the integers. (It is not assumed, of course, that there is an infinite number of suitable adresses; see under (ii).) The quantity m_P, the "modulus" of P, is at least one. Although this fact is not used, it is reasonable to choose a_P such that $0 \leq a_P < m_P$.

(ii) The hardware cells that will be occupied by a P object "at" address $a_P + n * m_P$ are $l_P + n * m_P$ through $u_P + n * m_P$, and if all of these cells physically exist, the adress is indeed suitable.

(iii) There exists a (least) common multiple M of all primitive type moduli (i.e., the set of moduli is finite).

In the realization, an area must always be "aligned" in such a way that the adress corresponding to its first cell, $k \cdot 0$, is a multiple of M. This ensures that a lay-out allowing efficient access to the objects in an area is possible.}}

a) A "type" is a quintuple <t, a, l, u, m>, where t is a token and a, l, u and m are integers. The "modulus" m is at least 1, and is a divisor of the "area modulus", denoted by "M". Moreover, l satisfies $0 \leq l < m$, and the "span" of the type, u-l+1, is at least 0.

{{If a = 1 for all primitive types, this property is inherited for composite types if the formulae given below are used.}}

b) The token of a type is either an "atomic type token" or the token "STRUCT". The names of the atomic type tokens correspond, one to one, to the terminal productions of ´tok´ {{2.1.h}}, after leading non-significant digits ´0´, if any, have been omitted from the constituent dec, if any {{see 2.1.m}}. If two types have the same atomic type token, then they are one and the same type.

{{Although each object has a type, there are no objects whose type has the token STRUCT. The latter kind of types may be used to model "composite objects", that, to the MIAM, and notably its realizer, exist only in the eye of the beholder (see also the remarks in section 1.5.a).

The function of the atomic type tokens is to allow two types to be different, even if all four characteristic numbers are equal, since some hardware may require different instructions for different primitive types, even if the abstract meaning of the instructions is the same.}}

c) "Tjoin(t_1, t_2)" and "Djoin(t_1, t_2)", where t_1 and t_2 are types, stand for a type t and an integer d, respectively, satisfying:

(i) if p is a pointer, suitable for t, then p is also suitable for t_1 and p·d is suitable for t_2;

(ii) the sites appointed by p for t_1 and by p·d for t_2 are disjoint, and are contained in the site appointed by p for t;

(iii) the token of t is STRUCT;

(iv) if t_1 is of the form <STRUCT, 0, 0, u, M>, then t is of the form <STRUCT, 0, 0, v, M>.

Tjoin and Djoin are {{deterministic}} functions of their arguments. Moreover, if Seq(n, t), where n is an integer $\geq$ 0 and t is a type, is defined inductively by

- Seq(0, t) = Type[G] {{2.2.j}};
- Seq(n+1, t) = Tjoin(Seq(n, t), t);

then there exists a function Shift, mapping types to integers, such that the value of Djoin(Seq(n+1, t), t) is Djoin(Seq(0, t), t)+n*Shift(t).

{{Formulae computing t and d, given $t_1 = \langle k_1, a_1, l_1, u_1, m_1 \rangle$ and $t_2 = \langle k_2, a_2, l_2, u_2, m_2 \rangle$, satisfying the requirements, are:

- let q_2 be $(u_1+m_2-l_2) \div m_2*m_2$ and
  let q_1 be $(q_2-(u_1+1-l_2)) \div m_1*m_1$;
- t = <STRUCT, a_1+q_1, l_1+q_1, u_2+q_2, LCM (m_1, m_2)>, where LCM stands for the Lowest Common Multiple;
- d = $(a_2+q_2)-(a_1+q_1)$.

The idea is that in the compound type the site of the t_2-object is shifted over q_2 cells to the right, being the least multiple of m_2 such that the shifted site is disjoint from the earliest possible site for the t_1-object. Next, the t_1-site is shifted to the right over q_1 cells, being the greatest multiple of m_1 that leaves the sites disjoint, in order to obtain a tight packing. This is not necessarily the tightest packing if the equality LCM(m_1, m_2) = max(m_1, m_2) does not hold for the moduli involved. More realistically, interchanging the order of the field sites might also give a tighter packing.

The "axiom" defined by means of the auxiliary function Seq means that sites for a sequence of objects of the same type are allocated equidistantly.}}

d) "Shift(t)", where t is a type, stands for the integer Djoin(Seq(n+1, t), t)-Djoin(Seq(n, t), t), where Seq is the function introduced in the previous section {{and Shift is the same function whose existence was postulated there}}.

{{A formula computing Shift(<k, a, l, u, m>), if the formulae for Tjoin and Djoin given above are used, is (u + m - l) + m * m. This function is useful for translating selection on multiple values.}}

1.5. Objects

a) An "object" is a "plain object" {{1.5.1.a}}, a key or a pointer, or a "descriptor" {{1.5.2.a, 1.5.2.b}}. Each object has a type and a scope. The token of the type of a key (a pointer) is "KEY" ("PTR").

{{The translator may model "composite objects" by composing them of a sequence of other objects. The MIAM proper does not "recognize" the existence of composite objects -- other than parallel action descriptors, whose internal structure, however, is inaccessible --, but provides all necessary equipment for the modelling, such as types for composite objects, being a function of the types of the components, determined with the "Tjoin" function. The site occupied by a "copy" of such a composite object is then occupied by a sequence of copies of its components, possibly leaving some cells unused in between. The "scope" of a composite objects is the largest of the scopes of its components.}}

1.5.1. Plain objects

a) A "plain object" is an integer, an "answer" (i.e., one of the tokens "Yes" and "No"), a "label" {{1.7.1.c}} or some "other plain object" {{e.g., a character or a real number}}. The token of the type of an integer (an answer, a label) is "INTO" ("ANS", "LAB"). The names of the tokens of the types of other plain objects correspond, one to one, to the terminal productions of ´tok´ obtained by adding productions for it by virtue of 2.1.i. The scope of a plain object is 0.

{{The definition and treatment of other plain objects are left open in this definition of the MIAM. For a contract between translation and realization, these have to be filled in, of course.}}

1.5.2. Descriptors

{{Parallel actions may be translated by means of parallel action and process descriptors. Starting from some primal process descriptor, a tree is descended of currently active processes. In the model given below, the branches of the tree correspond to pointers pointing in the direction from leaves to root. However, no explicit connection is given between a process descriptor and any parallel action descriptor created by the corresponding process (by the action of a SPAWN-instruction). A data structure realizing the tree must contain supplementary pointers to connect the tree; otherwise, the "If there exists" in the description of "Search Process" (1.7.2.c) could not be effected in a reasonable way. Also, the determination of "Spawner(p)" {{1.5.2.d}} requires implicit pointers pointing upwards in the tree.}}

a) A "parallel action descriptor" is an object, composed of a sequence of "process descriptors" {{1.5.2.b}} and a "parent" pointer {{which, if it is not Nix, points to a process descriptor <Spawned, l>}}. The site occupied by a copy of a parallel action descriptor is occupied by a sequence of copies of its components, possibly leaving some cells unused in between. The scope of a parallel action descriptor is the largest of the scopes of its components. A parallel action descriptor with n process descriptors has a type whose token has a name of the form "PARdec", where Val(dec) {{2.2.b}} is n.

b) A "process descriptor" is an object, composed of a token and possibly other objects; it is of one of the following four forms:

- <Running>;
- <Spawned, l>, where l is a label {{for continuation after all spawned processes have reached completion}};
- <Halted, sp, k_T, k_S, l>, where sp is a {{semaphore}} pointer pointing to an integer, k_T and k_S are keys and l is a label {{for resumption if the condition that caused the halting no longer applies}};
- <Complete>.

The scope of a process descriptor is the largest of the scopes of its components, where the tokens are assumed to have scope 0.

c) There is a register "P" holding a pointer which, if it is not Nix, points to a process descriptor {{the (unique) "<Running>" process descriptor}}.

d) "Spawner(p)", where p is a pointer pointing to a process descriptor, is the parallel action descriptor, pointed to by the pointer q, such that the site appointed by p is contained in the site appointed by q.

{{This definition is given in terms of pointers, since different parallel action descriptors might contain identical process descriptors.}}

1.6. Locales

a) An area may be a "locale".

{{Locales are created by an EST-instruction; other areas are created by a GEN-instruction. Locales are chained by a dynamic and a static (lexicographic) chain. Parallel action descriptors may built a tree form from these, otherwise linear, chains.}}

b) There exist two registers T and S that may hold keys of locales, that are then known as the "T-locale" and the "S-locale", respectively.

c) The "C-locale" is a {{standard}} locale, existing without explicit creation, whose scope is 0 and whose span is sufficiently large {{if realization permits}} to accomodate the static action prescribed by all of the CFILL-instructions {{3.2.f}}. "C" stands for the key of the C-locale. It is required that no pointers or keys accessing areas other than the C-locale are made to occupy sites contained in the C-locale.

{{The C-locale is the only locale whose scope is 0, and it is also the only area that exists without creation, not counting some fictitious locale(s) facilitating the semantic description.}}

1.7. Actions

1.7.1. The program

a) The program consists of a sequence of "instructions". The "execution" of the program consists of the execution of the instructions, one by one, starting with the first instruction, and ending, if the program is normally completed, with the last instruction. The execution of each instruction determines a successor, which is, unless otherwise specified, the next instruction in {{the sequence which is}} the program, or it results in "abortion" with an "error code" (an integer), whereupon no further instructions are executed.

b) For some instructions the execution is empty, apart from determining the next instruction as successor {{e.g., a LABEL instruction}}. Such instructions, may, however, influence the meaning of other instructions. This influence is execution-independent and must, therefore, be determined statically by performing once, in the textual order, the "static actions" prescribed for these instructions.

{{This holds, especially, for the CFILL-instructions and for the EST-FIN pairs. If the realization, in a second pass through the program to generate concrete code for the dynamic actions, should re-perform the static actions, the meaning remains the same. For example, the "meaning" of 4 in the COPY-instruction in

```
JOIN, A, INT, 4;
COPY, INTO, 666, &T4;
JOIN, 4, INT, 4;
```

is that Offset[4], that has been set in the first instruction, even though the following setting of Offset[4] is allowed and has a well-defined effect.}}

c) A "label" is the "valuation" {{2.2.a}} of a "lab" {{2.1.q}}. The scope of a label is 0, and its type is LAB.

d) There exists a special class of "abortive" labels, which have an error code.

e) "LABEL" {{3.4.c}} and "DOWN" {{3.5.d}} instructions are "labelled" with the valuation of their first argument {{which is execution independent}}. It is required that no two different instructions be labelled with the same label.

f) If, in this description, some condition is said to be "required", this means that the MIAM is not designed to be able to cope with the situation arising if the condition is not fulfilled.

{{It is certainly not the intention that a realization of the MIAM should check the requirements. Rather, the translation should generate a program whose execution cannot violate them.}}

1.7.2. Auxiliary actions

a) "Newkey(m, s, c)", where m is a model, s is a span and c is a scope, stands for the key yielded by the following action:
- it is required that model conformance holds {{1.3.n}};
- the action yields the key of a newly created area with model m, span s and scope c.

{{It may be helpful to know that this action is only prescribed by the instructions EST {{3.2.a}} and GEN {{3.3.a}}, and that prior to its invocation the execution of these instructions does not call for actions that might influence the occupancy of any area.}}

b) "Goto(l)", where l is a label, stands for the action whereby the instruction labelled with the label l, if any, is determined as successor to the instruction currently executed; otherwise, it is required that l be abortive and the action results in abortion {{1.7.1.a}} with the error code of l.

c) "Search Process" stands for the following action:
If there exists a pointer q, contained in a reachable area {{1.3.m}}, pointing to a process descriptor <Halted, sp, k_T, k_S, 1>, such that sp points to a nonnegative integer,
then

- <Running> =* q;
- P is set to q;
- T is set to k_T and S is set to k_S;
- Goto(1);

otherwise,

- Goto(a), where a is an abortive label with error code Deadlock.

d) "Discard Par(k)", where k is the key of a locale, stands for the following action:
If P and k access the same locale,
then

- let Spawner(P) {{1.5.2.d}} be the parallel action descriptor {{1.5.2.a}} <pp, pd_1, ... , pd_n>;
- P is set to pp;
- <Running> =* P;
- <Nix, <Complete>, ... , <Complete>> =* s {{which, in a realization of the MIAM, should be a dummy action}};
- Discard Par(k) {{again}};

otherwise,

- if k is not T, Discard Par(*k·u), where u is Offset[U] {{2.2.n}}.

2. ARGUMENTS

2.0. Notation

In the next section, a syntax definition method is used that is a variant of BNF. Non-terminal symbols are a sequence of lower case letters. A colon separates the left hand side of a rule from the right-hand side, and the alternatives are separated by a bar ("|"). All other marks are terminal symbols and stand for themselves. Blank spaces are not significant {{but are

inserted in the syntax rules in such a way that they help to increase legibility in the terminal productions if treated as terminal symbols}}.

2.1. Syntax

{{The following transcriptions may be helpful:

&	-pointer to	N	-No
*	-follow pointer	S	-S-locale
A	-Around-chain field	T	-T-locale
C	-C-locale	U	-Upon-chain field
E	-Established locale	X	-niX
G	-Generated area	Y	-Yes
H	-sHift		
I	-Indirect		

ans-answer	loc-locale
arg-argument	off-offset
dec-decimal	ptr-pointer
ins-inspectable (only)	rec-recipient pointer
int-integer	res-resident (copy)
jtp-joined type	sin-signed integer
lab-label	tok-token of type
lev-static level	typ-type
lit-literal	

}}

a) arg: lit | ins | rec

b) lit: sin | Y | N | Ldec | Adec | X | Htyp

c) Other productions for 'lit' may be added, together with rules for their valuation {{2.2.a}}. {{Presumably, these other productions correspond to denotations for the ALGOL 68 modes mirrored by INTsin with Val(sin) ≠ 0 and by additional productions for 'tok'; see 2.1.i.}}

d) sin: dec | -dec

e) ins: *Coff | *Toff | *Soff | *Ioff$_1$·off$_2$

f) rec: Coff | &Coff | Toff | &Toff | Soff | &Soff | Ioff$_1$·off$_2$

g) typ: tok | jtp

h) tok: G | E | KEY | PTR | ANS | INTsin | LAB | PARdec

i) Other productions for ´tok´ may be added. {{Presumably, other productions for ´tok´ are ´CHAR´, ´REALsin´, ´BITSsin´ and ´BYTESsin´, and ´CHANNEL´, ´BOOK´ and ´BUF´ if the approach from VAN VLIET[2] is taken for the translation.}}

j) jtp: U | A | dec

k) lev: dec

l) off: jtp | off+jtp

m) dec: a nonempty sequence of decimal digits

A dec may have leading digits 0; however, $0dec_1$ is considered entirely equivalent to dec_1 {{so L007 and L7, e.g., are one and the same lab}}.

n) res: ins | Coff | Toff | Soff

o) int: sin | Htyp | res

p) ans: Y | N | res

q) lab: Ldec | Adec | res

r) ptr: X | ins | rec

{{Auxiliary definition}}

s) loc: C | T | S

2.2. Valuation

a) The "valuation" of an arg determines an object {{generally during execution}}; it is denoted by Val(arg). A typ t determines statically a type, denoted by Type[t], and a model, denoted by Model[t]. Moreover, if t is a jtp, it determines an integer {{an "offset"}}, denoted by Offset[t]. In the static or dynamic requirements and actions, Type[t], Offset[t] and Model[t], where t is a jtp, have a meaning only if they have been set by the static action of a textually preceding instruction which has not been invalidated by a textually intervening instruction, and they have the mean-

ing as set by the textually last such instruction.

b) The valuations of the lits are {{execution independent and are}} determined as follows:

- Val(dec) is the integer whose decimal representation is dec;
- Val(-dec) is -Val(dec);
- Val(Y) is the answer Yes;
- Val(N) is the answer No;
- Val(Ldec) is the label "Ldec"; it is required that there be exactly one instruction labelled with "Ldec";
- Val(Adec) is an abortive label with error code Val(dec);
- Val(X) is the pointer Nix;
- Val(Htyp) is Shift(Type[typ]).

c) Offset[off+jtp] is Offset[off]+Offset[jtp].

d) Val(&Coff) is the pointer C·Offset[off].

e) Val(&Toff) is the pointer T·Offset[off].

f) Val(&Soff) is the pointer S·Offset[off].

g) Val(locoff) is *Val(&locoff) {{e.g., Val(T4) = *Val(&T4) = *T·Offset[4]}}.

h) Val($Ioff_1$·off_2) is the pointer p·Offset[off_2], where p is Val($Toff_1$); it is required that p be a pointer, other than Nix.

i) Val(*arg) is *Val(arg) {{; it is required that Val(arg) be a pointer, other than Nix}}.

j) Type[G] is <STRUCT, 0, 0, -1, 1> and Model[G] is the empty set.

k) Type[E] is <STRUCT, 0, 0, -1, M>, where M is the area modulus {{1.4.a}}, and Model[E] is the empty set.

l) Type[tok], where tok is not G or E, is the type whose token is named tok.

m) Model[tok] is {<tok, 0>} if tok is KEY or PTR, and the empty model {} otherwise.

n) Type[U] is Tjoin(Type[E], Type[KEY]), Offset[U] is Djoin(Type[E], Type[KEY]), and Model[U] is {<KEY, Offset[U]>}.

o) Type[A] is Tjoin(Type[U], Type[KEY]), Offset[A] is Djoin(Type[U], Type[KEY]), and Model[A] is {<KEY, Offset[U]>, <KEY, Offset[A]>}.

{{The effect is the same as would be obtained for decs u and a by
JOIN, E, KEY, u;
JOIN, u, KEY, a;}}

p) Type[dec], Offset[dec] and Model[dec] are defined if set by a JOIN, EST, MAX or FIN-instruction {{3.1.a, 3.2.a, 3.1.c, 3.2.b}}.

3. THE INSTRUCTIONS

3.0. Notation

In each instruction format given below, a lower-case letter, possibly adorned with a subscript or an apostrophe, stands for a non-terminal symbol for which a production rule is given in the lines following the instruction format. In the requirements and actions given for the instruction, they stand for the terminal productions by which they are replaced in the actual instruction. Production rules for different non-terminal symbols that have a common right-hand side may be replaced by one rule whose left-hand side consists of a list of the original left-hand sides.

3.1. Type instructions

a) JOIN, s, t, u;
s, t: typ
u: dec {{type}}

Static action:

• Type[u] is set to Tjoin(Type[s], Type[t]), Offset[u] is set to Djoin(Type[s], Type[t]), and Model[u] is set to the union of Model[s] and Model[t]+Offset[u] {{1.3.o}}.

{{If s and t accomodate the ALGOL 68 modes SS and TT, then u will accomodate STRUCT(SS f1, TT f2). If the argument Toff gives access to an object of the composite type u, then Toff gives access also, in a context where an object of type s is implied, to the first field, and Toff+u gives access to the second field. A structured mode with more than two fields, e.g., STRUCT(SS f1, TT f2, UU f3), may be handled by treating it as STRUCT(STRUCT(SS f1, TT f2) fx, UU f3) (or as STRUCT(SS f1, STRUCT(TT f2, UU f3) fy), which does not necessarily give the same lay out).

The typ G is a dummy that is useful to make uniform translation schemes in which the first (or the last) field does not have to be translated in a special way; Type[G] is defined as the type of a virtual object of zero span that can be accommodated at any site. G is also useful to create types that are equivalent to already given types, except that the token is STRUCT, as is required by the MAX-instruction.

The typ E forces alignment in the realization. It is especially useful for allocating sites in a locale (in translating an establishing-clause). Consider, for example, BEGIN SS x1; TT x2; UU x3; ... END. This can be handled as BEGIN STRUCT(SS f1, TT f2, UU f3) xx; ... END, but then the translation for accessing x1, say, as f1 OF xx, depends on the subsequent identifier-declarations. If, however, a dummy declaration "EE dummy" is assumed immediately following the BEGIN, where EE is treated as a mode accommodated by Type[E], the access for x1 as computed by the above scheme becomes independent of the sequel. Actually, the EST-instruction introduces not only EE alignment automatically, but also, for convenience, two explicit fields for keys, as in BEGIN EE dummy; KEY upon, around; ... END.}}

b) SJOIN, s, t, u;

s, t: typ

u: dec {{type}}

Static action:

● The same static action is performed as would be performed by JOIN, s, t, u;.

Dynamic Requirement:

● It is required that, for any non-empty site appointed by a pointer that is the result of valuating, during execution, an argument of the form &locjtp$_1$+...+jtp$_{n-1}$+u, where the value of Offset[u] is defined by virtue of an SJOIN-instruction, neither that site nor any part or component thereof, be appointed by any pointer contained in a reachable area {{1.3.m}}.

{{To grasp the usefulness of the requirement, it should be stressed that requirements of the MIAM, being clauses from a contract between the translator and the realizer, correspond to promises on the part of the first party. Since it is always possible to use a JOIN-instruction, the self-inflicted requirement of an SJOIN-instruction is a promise by the translator that no "alias" pointers will be set up appointing sites described with the offset u. This makes it possible for the realization to keep the corresponding objects in hardware registers, if this is desirable for optimization purposes, without global data-flow analysis to check the safety. In the general case, the translator can only make the promise after some degree of global analysis of the source text. This is not necessary, however, for the code emitted for anonymous counters needed to translate, e.g., various actions on multiple values, in which case the intended optimization may be quite profitable.}}

c) MAX, s, t;

s: jtp {{type}}

t: jtp {{type}}

Static requirement:

● Type[s] is of the form <STRUCT, 0, 0, u, M>, and Type[t] is of the form <STRUCT, 0, 0, v, M>.

Static action:

- Type[t] is set to <STRUCT, 0, 0, max (u, v), M>;
- any statically preceding settings of Offset[t] and Model[t] become invalid.

{{The MAX-instruction is useful for translating UNITED modes. By using the typ E, the required zeros and M can be forced. This is hardly a restriction, since each united object has to be allocated an area of its own in order to be able to set the model properly, and since areas have to be aligned anyway in the realization. This argumentation is invalid in cases like UNION(INT, REAL), where both variants give rise to an empty model, so some cells may remain unused because of the static requirement. Still, there is a good reason for always allocating a separate area for "united" objects: they may then be copied by simply copying the pointer yielded by the GEN-instruction. That this is the case does not follow from any particular property of the MIAM, but from the Semantics of ALGOL 68 itself.

Another important application of MAX-instructions is for accomodating sites for anonymous intermediate yields in a locale: the "working stack". The type of the locale may be treated as the union of all types for all intermediate stages the site lay out of the locale may be in.}}

3.2. Instructions concerned with locales

{{The following instructions use a "static level" with a "type number". As follows from the static requirements, this is redundant (but possibly helpful) information; however, a nesting of EST and FIN-instructions is thereby enforced.}}

a) EST, l, s;

l: lev {{static level}}

s: dec {{type}}

Static requirement:

- Val(l) is one more than the current static level.

Static action:

- the static level is set to Val(l);
- Type[s] is set to Type[A] and Model[s] is set to Model[A];
- the type number of the static level is set to Val(s).

Dynamic action:

- let c be the scope of the T-locale;
- let k be Newkey(Model[A], os, c+1) {{1.7.2.a}}, where os is the Offset[s] statically set at the corresponding (i.e., textually first following) instruction "FIN, l, s;"; {{with the same l and s}};
- S is set to T;
- T is set to k;
- S =* T.Offset[U] and S =* T.Offset[A].

{{This instruction is typically the first to be emitted in the translation of an establishing-clause. If the "upon" and the "around" locale do not coincide, one of the following instructions in the translation of an establishing clause will, presumably, be a SETS-instruction.}}

b) FIN, l, s;

l: lev {{static level}}

s: dec {{locale type}}

Static requirements:

- Val(l) is the current static level, and Val(s) is the corresponding type number;
- the T-locale does not contain a parallel action descriptor, not all of whose process descriptors are <Complete>;
- If Val(l) is 0, this instruction is the last instruction of the program.

Static action:

- Offset[s] is set to Djoin(Type[s], Type[G]) {{or, maybe, to obtain a multiple of M, to Shift(Type[s]) in the actual realization}};
- the static level is decreased by one.

Dynamic requirement:

• The dynamically last preceding EST-instruction not yet dynamically matched by a corresponding FIN-instruction is the statically corresponding EST-instruction.

Dynamic action:

• T is set to T.Offset[U];

• S is set to T.Offset[A] {{in which T is the newly set T}}.

{{Typically, some MAX-instructions may have intervened between the EST and FIN-instruction to set Type[s].}}

c) MOD, m;

m: jtp {{model}}

Dynamic action:

• the model of the T-locale is made to be Model[m].

{{This instruction is used to ensure that Model conformance holds prior to the execution of an EST- or GEN-instruction. It is not necessary, on translation, to issue a MOD-instruction for each change in the occupancy by keys or pointers; if no EST- or GEN-instruction may intervene between the execution of two MOD-instructions, the first one was superfluous. Care should be taken that all sites of keys and pointers indicated in the model have indeed been filled before an EST- or GEN-instruction is executed; for pointers this may be done by using Nix.}}

d) KEEP, p;

p: ptr

Requirements:

Let A be the area accessed by Val(p).

• A is not a locale {{i.e., A is created by a GEN-instruction}};

• The scope of the T-locale is greater than 0;

• The scope of A is the scope of the T-locale;

• the scope of any key or pointer, a copy of which occupies a site in A, is most the scope of the S-locale.

Dynamic action:

- the scope of A is made to be the scope of the S-locale.

{{The KEEP-instruction exists only for reasons of efficiency. Without this instruction, the multiple value yielded by the inner closed-clause in, e.g.,

BEGIN [] REAL x = ([large] REAL xx; ... ; xx); ... END

would have to be copied to a newly created area since its scope would be too large (i.e., numerically).}}

e) SETS, a;

a: arg {{key of locale}}

Dynamic action:

- S is set to Val(a).

f) CFILL, t, l, u;

t: tok

l: lit {{object of type Type[t]}}

u: off {{for object of type Type[t]}}

Static requirements:

- t is not G, E, KEY or some PARdec {{for which, anyway, no lits can be given}};
- no textually preceding CFILL-instruction has caused a copy to occupy the site appointed by C.Offset[u], nor any part or component thereof.

Static action:

- Val(l) =* C.Offset[u].

{{The filling of the C-locale is performed before execution starts; so an ins of the form *Coff may be used before the corresponding CFILL-instruction. For this to be meaningful, however, the (static) meaning of Offset[off] has to be the same in both instructions.}}

3.3. Instructions concerned with pointers

a) GEN, t, s, t´, a, r;

t, t´: typ {{possibly G}}

s: int {{number of t´ elements}}

a: arg {{key}}

r: rec {{for pointer}}

Dynamic requirement:

- let N be Val(s);
- N $\geq$ 0.

Dynamic action:

- let c be the scope of Val(a);
- let t_0 be Type[G] and let m_0 be Model[G] {{i.e., empty}};

For i from 1 to N:

- let t_i be Tjoin(t_{i-1}, Type[t´]) and let m_i be the union of m_{i-1} and Model[t´]+Djoin(t_{i-1}, Type[t´]);

- let u_0 be Tjoin(Type[E], Type[t]) and let n_0 be Model[t]+Djoin(Type[E], Type[t]);
- let u_1 be Tjoin(u_0, t_N) and let n_1 be the union of n_0 and m_N+Djoin(u_0, t_N);
- let k be Newkey(n_1, Djoin(u_1, Type[G]), c) {{1.7.2.a}};
- k·Djoin(Type[E], Type[t]) =* Val(r).

{{The computations are the same as would have been performed in

JOIN, G, t´, t_1;

JOIN, t_1, t´, t_2;

...

JOIN, t_{N-1}, t´, t_N;

JOIN, E, t, u_0;

JOIN, u_0, t_N, u_1;

which, however, cannot be performed statically if N is not known statically. See also the remarks about Model conformance in 3.2.c.}}

b) COPY, t, a, r;

t: tok

a: arg {{object of type Type[t]}}

r: rec {{for object of type Type[t]}}

Static requirement:

- t is not G, E or some PARdec.

Dynamic requirement:

- if the valuation Val(a), according to Section 2.2.a, is some *Val(b), then either Val(b) = Val(r), or the sites appointed by Val(b) and by Val(r) have no cells in common.

Dynamic action:

- Val(a) =* Val(r).

{{The copying of a composite object (see 1.5.a) has to be written out in terms of copying its primitive components. There is no way in which a parallel action descriptor can be copied.}}

c) DOT, t, p, d, r;

t: jtp

p: ptr

d: int

r: rec {{for pointer to object of type Type[t]}}

Dynamic action:

- Val(p)·Val(d) =* Val(r).

{{This action is generally only meaningful if p is derived from the result of a GEN-instruction, and d is the result of multiplying an integer with Val(Ht´), where t´ is the fourth argument of that GEN-instruction. Because of the commutativity of addition, field-selection on multiple values (e.g., given some [] COMPL zz, re OF zz), can easily be translated.}}

d) SCOPE, p, r;

p: ptr

r: rec {{for integer}}

Dynamic action:

- let c be the scope of Val(p);
- c =* Val(r).

e) IFIS, p, q, l;

p, q: ptr

l: lab

Dynamic action:

If Val(p) = Val(q)

then

- Goto(Val(l));

otherwise,

- no action.

f) IFISNT, p, q, l;

p, q: ptr

l: lab

Dynamic action:

If Val(p) = Val(q)

then

- no action;

otherwise,

- Goto(Val(l)).

3.4. Instructions concerned with control flow

a) INIT;

Static requirement:

- the instruction is the textually first instruction of the program.

Dynamic action:

• the static level is set to -1, and T and S are set to the key of the C-locale;

• P is set to a pointer to a process descriptor, appointing a site in a fictitious locale of scope 0;

• <Running> =* P.

b) IMAT, l;

l: dec {{line number}}.

Action: none {{but presumably this may be put to some diagnostic use}}.

c) LABEL, l;

l: Ldec

Static action:

• Val(l) is made to label the instruction.

d) GOTO, l;

l: lab

Dynamic action:

• Goto(Val(l)).

e) JUMP, v, l;

v: arg {{key}}

l: Ldec

Dynamic action:

• let k be T;

• T is set to Val(v);

• Discard Par(k);

• Goto(Val(l)).

f) UNL, b, l;

b: ans

l: Ldec

Dynamic action:

If Val(b) = Yes

then

- no action;

otherwise,

- Goto(Val(l)).

g) CASE, i, c, l, l_0, l_1, ... , l_n;

i: int

c: dec

l, l_0, ... : Ldec

Static requirement:

- Val(c) is n.

Dynamic action:

If 0 < Val(i) < n

then

- Goto(Val($l_{Val(i)}$));

otherwise,

- Goto(Val(l)).

3.5. Instructions concerned with parallel action descriptors

a) SPAWN, c, l_2, ... , l_n, l_{n+1}, r;

c: dec

l_2, ..., l_{n+1} : Ldec

r: rec {{for a parallel action descriptor of type Type[PARn]}}

Static requirement:

- Val(c) is n.

Dynamic action:

- let s be a parallel action descriptor <pp, p_1, ... , p_n> whose components are determined as follows:
  - pp is P;

- p_1 is <Running>;
- p_i, i = 2, ... , n, is <Halted, sp, T, S, Val(1)>, where sp is a pointer appointing a site in a fictitious locale of scope 0, occupied by a copy of a fixed, positive integer;
- <Spawned, Val(l_{n+1})> =* P;
- s =* Val(r);
- P is set to a pointer to p_1.

b) COMPLETE;

Dynamic requirement:

- *P is <Spawned, 1> for some label 1.

Dynamic action:

- let Spawner(P) {{1.5.2.d}} be <pp, p_1, ... , p_n>;
- <Complete> =* P;

If for some i, $1 \leq i \leq n$, p_i is not <Complete>, then

- Search Process {{1.7.2.c}};

otherwise,

- <Running> =* pp;
- P is set to pp;
- Goto(1).

c) UP, r;

r: rec {{for integer}}

Dynamic action:

- *Val(r)+1 =* Val(r).

d) DOWN, l, r;

l: Ldec

r: rec {{for integer}}

Static action:

- Val(l) is made to label the instruction.

Dynamic action:

If *Val(r) $\geq$ 1

then

- *Val(r)-1 =* Val(r);

otherwise,

- <Halted, Val(r), T, S, Val(1)> =* P;
- Search Process {{1.7.2.c}}.

3.6. Instructions concerned with simple arithmetic

a) ADD, i, j, r;

i, j: int

r: rec {{for integer}}

Dynamic action:

- Val(i)+Val(j) =* Val(r).

b) SUB, i, j, r;

i, j: int

r: rec {{for integer}}

Dynamic action:

- Val(i)-Val(j) =* Val(r).

c) NEG, i, r;

i: int

r: rec {{for integer}}

Dynamic action:

- -Val(i) =* Val(r).

d) MUL, i, j, r;

i: int

j: int

r: rec {{for integer}}

Dynamic action:

• Val(i)*Val(j) =* Val(r).

3.7. Instructions concerned with simple comparisons

a) IFc, i, j, l;

c: LT | LE | EQ | NE | GE | GT

i, j: int

l: lab

Dynamic action:

• let ⁑ be < ($\leq$, =, $\neq$, $\geq$, >) if c is LT (LE, EQ, NE, GE, GT);

If Val(i) ⁑ Val(j)

then

• Goto(Val(l));

otherwise,

• no action.

REFERENCES

[1] MEERTENS, L.G.L.T, On the definition of an abstract machine for a portable ALGOL 68 compiler, in Proc. Int. Conf. on ALGOL 68, (J.C. van Vliet & H. Wupper, eds), 97-117, Mathematical Centre Tracts 134, Mathematical Centre, Amsterdam, 1981.

[2] VAN VLIET, J.C., ALGOL 68 Transput, Part II, An Implementation Model, Mathematical Centre Tracts 111, Mathematical Centre, Amsterdam, 1979.

[3] VAN WIJNGAARDEN, A., & al. (eds), Revised Report on the Algorithmic Language ALGOL 68, Mathematical Centre Tracts 50, Mathematical Centre, Amsterdam, 1976.