COMMENTARY ON THE MATHEMATICAL SEMANTICS OF ALGOL 60

by

Peter Mosses

[It is intended that this commentary be read in parallel with the semantic clauses.]

Contents:

SYNTAX - COMMENTARY

The grammar is written in an abbreviated BNF, with syntactic categories being denoted by words such as Prog, DecL. Subscripts to these words, as in Sta_1, do not distinguish different categories. Λ denotes the null category, and lexical categories, such as identifiers and numerals, which are not defined here, are prefixed by P, e.g. P IDf.

A star (*) indicates that the preceding category, or group of categories enclosed in braces ({,}), may be present zero or more times.

The grammar given is (very) ambiguous, but this doesn't matter here, as we shall use it only to *describe* deduction trees, and not to tell us how to form them. It was derived from an unambiguous grammar by combining categories to remove semantically irrelevant information, such as whether an expression Exp is a summand, multiplicand, or whatever. This caused a reasonable contraction in the size of the grammar, and in the number of syntactic categories.

Some additional transformations have been made to the original ALGOL 60 grammar. These should perhaps be expressed formally, but their descriptions are rather tedious, and need not detain us here. Informally, the transformations are:

(i) if Exp then Sta becomes if Exp then Sta else Λ;

(ii) Empty parameter lists are added to identifiers occurring as (procedure) statements, and to definitions of parameter-less procedures;

(iii) Parameter specifications are 'rationalised' to combine the type (which must be specified) with the formal parameter name and the name/value specification;

(iv) Declarations are sorted into two lists, DecL and DefL. DecL contains the non-recursive declarations of type and array identifiers, whereas DefL contains switches and procedures. The purpose of this will become apparent in the definition of $\mathfrak{C}$ in SEMANTIC FUNCTIONS.

(v) Comments are ignored, and parameter delimiters are denoted by commas.

Note that no attempt is made to specify any type matching at the syntactic level - this is done in the semantics, using the environment parameter ρ.

DOMAINS - COMMENTARY

(i) Standard Domains

These domains are associated with the interpretation of the meta-language (used in SEMANTIC and AUXILIARY FUNCTIONS), rather than that of the source language ALGOL 60. However, N and T are used here also as semantic domains for ALGOL integer numerals and booleans (true, false), respectively, so as to avoid the continual use of transfer-functions. I is a primitive domain, and its elements may be tested for equality only.

(ii) Syntactic Domains

Most of the syntactic domains correspond to categories of the same name in SYNTAX, and are specified by the grammar. The domains should be regarded as domains of "annotated deduction trees", in the words of [8], Ch.1. Here we shall take the annotation at a node of a tree, to be the string (in Q) of symbols on the right-hand-side of that production which was used in forming the node. The branches from the node belong to domains corresponding to other syntactic categories. To write out these domain definitions fully would give us, e.g.

```
DefL = node("Def{;Def}*")(Def pre Def*) + node("Λ")⟨⟩

Def  = node("switchIde:=ExpL")⟨Ide,ExpL⟩ +

       node("TypeIde(ParL);Sta")⟨Type,Ide,ParL,Sta⟩
```

(using some extra notation). The annotations at the nodes are used by the infix operator 'of', which is described in SEMANTIC FUNCTIONS-COMMENTARY: Meta-language.

List, and the domain of list elements El, are introduced to abbreviate the description of functionalities.

IDE, INT, REAL and STRING are, like the corresponding categories in SYNTAX, undefined here. They are to be evaluated in an implementation of these semantics by functions *IdeVal*:IDE→I; *IntVal*:INT→N; etc.

(iii) Semantic Domains

Some notation has been introduced here, so that the structure of a domain may be indicated without making arbitrary (and irrelevant) choices about the ordering of its component domains. E.g., consider

ActiveFn = *MakeActiveFn*(*ResLocn*:Locn,*Fn*:Fn)

This is meant to indicate that *MakeActiveFn* is the constructor function, and *ResLocn*,*Fn* are the corresponding selector functions, for elements of the domain ActiveFn.

<u>Area</u>

This domain might contain information about which locations (of Locn) are in use, i.e. have been supplied by an application of *New* (or of *NewArray*). The function *SetArea* 'reclaims' locations which are no longer accessible through program variables - this is not strictly necessary for ALGOL 60, and *SetArea* may be re-defined to have no effect on the store.

As the ALGOL 60 version of 'own' variables is not described in this semantics, further specification of the store S could give Locn the structure of a stack, and then Area would be just the 'top-of-stack pointer'.

<u>Map</u>

Like Area, Map is not further specified, although it is implicitly restricted by the 'axioms' of AUXILIARY FUNCTIONS, (iii).

<u>R</u>

<u>String</u>

These domains are not restricted. See ¶3.3.6 and ¶2.6.3.

$\underline{X}_1$, etc.

Literal strings are used to denote elements of these "known" finite domains. This enables basic symbols of the source language to be mentioned, without disturbing the lexical conventions of the meta-language.

(iv) Denotation Domains

These indications of the 'types' of bound variables are given only as an aid to the reader, and their mathematical significance is not exploited in this paper. The types are further indicated in the bound-variable lists of λ-expressions and defined functions.

With the aid of these denotation domains (of the metalanguage), the type of any function may be found from the INDEX, e.g. we get $\mathcal{A}$:[AssL→[U→[X→[Locn*→[C→C]]]]].

Note: For the purposes of this paper, each domain is assumed to include an "error found" element, denoted by '?'. A domain is in fact a lattice, in accordance with [8], etc., and the idea is for '?' to be incomparable with all elements of the domain (except with $\bot$ and $\top$, of course). '?' should be subscripted with an indication of its domain, but this is usually clear from the context and so is omitted in this semantics. It is convenient to be able to test x=?, where, for example, x might be the 'looking-up' of an identifier in an environment.

SEMANTIC FUNCTIONS - COMMENTARY

Meta-language:

The functions are defined using a variation of the 'semantic clauses' notation of Strachey (e.g. in [8, 12]). The main differences are the disappearance of some special operators, and the introduction of a more structured definitional form. The result of the latter has been to make the meta-language look much more like a programming language itself - the implementation of this language is to form part of the author's D.Phil. thesis. However, it should be stressed that the whole definition is just as mathematically-based and referentially transparent as before.

An informal guide to the metalanguage is given below. The reader is warned that the meta-language is still evolving, and that the variant used here is experimental.

$\alpha, \alpha_1, \alpha', t \ldots$	are bound variables
$\alpha^*, \alpha_1^* \ldots$	are bound variables denoting tuples. (N.B. Star (*) has no operational significance in this paper.)
$\mathcal{A}, \mathcal{A}_{abc}, \mathcal{A}_1, \ldots$	are semantic functions.
$\mathcal{A}^*, \mathcal{A}^*_{abc}, \ldots$	are semantic functions on a List.
A, $ABc123$	are auxiliary functions.
$A, A_1, \ldots$ $A^*, A_1^*, \ldots$	are semantic domains

? is the "error found" element (of the appropriate domain).

$\langle\rangle$ is the empty tuple

$\langle a_1, \ldots, a_m\rangle$ denotes a tuple, a* say, of known dimension.

$\text{dimof } a^* = m$

$a^* \downarrow i = a_i$

$a^* \text{cat} \langle b_1, \ldots, b_n\rangle = \langle a_1, \ldots, am, b_1, \ldots, b_n\rangle$

$e \text{ pre } t = \langle e\rangle \text{ cat } t$

$\lambda x.e$, $\lambda x{:}A.e$, $\lambda\langle x_1, x_2\rangle.e$ are λ-expressions, optionally typed.

fix x.e is the minimal fixed point of e with respect to the bound variable x. (An earlier notation was $Y(\lambda x.e)$.)

fxyz = ((f(x))(y))(z) }
a‖b‖c = a{b{c}} } abbreviations to avoid a multitude of brackets

Note: ‖ is less binding than juxtaposition and →, but does not terminate a λ-expression.

(e),{e} are used for parsing purposes, and to help readability.
$e(a_1,\ldots,a_m) = e(a_1,\ldots,a_m)$
⟦e⟧ - e must denote a deduction tree. See 'of' below.

If t denotes a deduction tree, then:
labelof t gives the annotation at t,
dimof t gives the number of branches from t (c.f. dimof α*),
ν of t gives the ν-th branch of t, and
"ABc"of t where 'ABc' is a syntax category, gives the 'correct' branch of t - this is deduced from the labelof t and takes any subscript on 'ABc' into account.

N.B. Throughout the definitions of the semantic functions, ⟦"ABc"of t⟧ is abbreviated to, simply, ⟦ABc⟧. The longer form is used when the denotation of the parameter deduction tree is not (literally) t.

id"abc" converts from O to I

$\rho[\delta/\tau/\iota] = \rho'$, where $\rho'⟦\iota'⟧ = \begin{cases} (\delta,\tau) & \text{if } \iota' = \iota \\ \rho⟦\iota'⟧ & \text{if } \iota' \neq \iota \end{cases}$

$\rho[\delta^*/\tau^*/\iota^*] = \rho[\delta_1/\tau_1/\iota_1][\delta_2/\tau_2/\iota_2]\ldots[\delta_n/\tau_n/\iota_n]$
where $\delta^* = (\delta_1,\ldots,\delta_n)$, etc., but only if $\iota_1,\ldots,\iota_n$ are distinct.

$$e_1 \to e_2, e_3 = \begin{cases} e_2 & \text{if } e_1 = \text{true} \\ e_3 & \text{if } e_1 = \text{false} \\ ? & \text{if } e_1 = ?_T \end{cases}$$

```
switch a in
§ case $b_{11}$:case $b_{12}$:...case $b_{1n_1}$:      $e_1$
  ...
  case $b_{m1}$:...case $b_{mn_m}$ :                  $e_m$
  default:                                            $e_{m+1}$
§
```

- the expressions b_{ij} are tested sequentially for equality with a, and if a match is found the result of the switch is the corresponding e_i. If no match is found, the result is e_{m+1}. The default case is in fact optional, and its omission is equivalent to specifying default:? Note the bracketting use of § and ∮.

let $x=e_1$ in
let $(y,z) = e_2$ in e_3
- non-recursive definition of local variables, equivalent to $(\lambda x.(\lambda(y,z).e_3)(e_2))(e_1)$.

compiler e
def𝒜⟦t⟧αβγ = e_1
...
def*Abc*αβγ = e_n
- the complete mutually-recursive definition of the semantic and auxiliary functions, specifying formal parameters. The scope of the functions includes e, the body of compiler, which is the main semantic function transforming a program's deduction tree into its mathematical value.

ALGOL 60 Semantic Functions:

compiler...

ρ_0 is to contain any input/output procedures, and extra system procedures.

let ρ_1=ρ_0[...

Here, the 'standard' procedures of ALGOL 60 are added to ρ_0. *Abs*, *Sign*, etc. are elements of Fn, hence also of D. Note that 'sin' may be re-declared to be something completely different, in the source program t.

case"Sta":...

A valid element of Prog has "Sta" as its label. $\mathfrak{P}$ deals with any labels occurring outside the outermost block of the program.

def$\mathfrak{P}$⟦t:Sta⟧$\rho\theta$ = ...

ι* is to be the list of label identifiers declared in t, but not inside any inner block. See $\mathfrak{I}^*_{lab}$.

τ* is to be the list of their types (all *MakeTyp*("label",?)). See $\mathfrak{T}^*_{lab}$.

$\mathfrak{G}$ gives a list of the corresponding entry points, incorporating in them η as the *ProperArea*.

fix is used, as labels are inherently recursive.

def$\mathfrak{C}$*⟦t:StaL⟧$\rho\theta$ = ...

Continuations are used, to compound the effects of the statements of a sequence whilst allowing jumps out of the statements. See [12] for a description of the general method of using continuations.

def$\mathfrak{C}$⟦t:Sta⟧$\rho\theta$ = ...

This adds the effect of a single statement, to that of θ.

§

case"begin DecL...

This is, thankfully, the most complicated case. It would be even worse without the assumed re-ordering of the declarations into the two lists DecL and DefL.

Note that array bounds in DecL are not simply evaluated in ρ (see $\mathfrak{D}$*,$\mathfrak{D}$). This is to conform with ¶5 and ¶5.2.4.2, in that

```
        ...
        integer n; n:=10;
        begin array A[1:n]
                procedure n(x);...
        ...
```

is not to be allowed.

$\lambda\eta_1.\mathcal{D}^*\ldots$

The area is found so that, on a normal exit from the block, locations which have become inaccessible through program variables may be 'garbage-collected' using *SetArea*.

$\lambda\eta_2$.let...

The area η_2 is incorporated into the values of labels, to enable 'garbage-collection' after jumping out of an inner block. See *Jump*.

case"begin StaL end":

Here, begin and end are used only as brackets, and do not affect meaning or scopes.

case"if Exp then...

The Exp is evaluated 'first'. Note that the effect of

if Exp then Sta else Λ;

is not necessarily null when the value of Exp is false (in T), in contrast to ¶4.5.3.2. It should not be considered a disadvantage of the semantic clauses, that one cannot easily describe in them (without explicitly copying σ) the semantics given in ¶4.5.3.2, which requires the reversibility of any side-effects occasioned by the evaluation of Exp.

case"Ide: Sta":

It can be seen that when $\mathcal{C}$ is applied to t:Sta and ρ, all the labels declared in t will have been added to ρ already. Hence the continuation from the label, which forms part of the value of a label, may be found from ρ⟦Ide⟧ here.

Hop is like *Jump*, but omits the (unnecessary) resetting of the store area.

case"goto Exp":

$\mathcal{J}$ evaluates a designational expression.

case"Var := AssL":

The type of Var is "manifest", i.e. ascertainable without applying the program to a store σ - without "running" the program. $\mathcal{A}$ insists that all the left-parts of AssL are of the same type as Var; and $\mathcal{R}$, when called from $\mathcal{A}$, inserts a transfer function, converting the expression to this type.

case"for Var := ForL do Sta ":

Again the type of Var is manifest, and must here be arithmetic. Var is "called by name" - note that $\mathcal{V}[\![\text{Var}]\!]\rho\tau$ has not been applied to κ or σ. *Main* selects part of a (structured) type, as does *Qual* later.

case"Ide(ExpL)":

Note that *Coerce* allows Ide to be a function designator - see [2], Correction 4.

case"Λ":

A dummy statement adds nothing to the continuation parameter θ.

§

def $\mathcal{D}^*[\![$t:DecL$]\!]\rho\kappa$=...

$\mathfrak{X}_2[\![t]\!]\phi$ maps elements of t:List with ϕ.

$\Pi(\omega^*)\kappa$ evaluates the ω_i in an unspecified order, and applies κ to the (possibly) re-sorted list of results.

def $\mathcal{D}[\![$t:Dec$]\!]\rho\kappa$=...

§

case"Type IdeL":

Declaration of type identifiers.

case"Type IdeL[BdsL]":

Declaration of array identifiers. Note that BdsL is only evaluated once.

§

def $\mathcal{H}^*[\![$t:DefL$]\!]\rho$=...

This function produces a tuple of switches, routines and functions, to be added to an environment. See $\mathcal{C}$, case"begin DecL...".

def $\mathcal{H}[\![t{:}\mathrm{Def}]\!]\rho$=...

§

case"switch Ide := ExpL":

Expressions in ExpL are evaluated only after they are selected by a use of the switch.

case"Type Ide (ParL); Sta":

§

case"procedure":

$\mathfrak{Q}^*$ sets up call-by-value parameters.

$\mathfrak{P}$ sets up labels and calls $\mathfrak{C}$.

Area is found to facilitate re-use of locations which have become inaccessible, after a normal return from the procedure body.

case"Type procedure":

The location α will be set when Ide (above) appears as the left-part of an assignment statement in Sta. The type of δ is tagged with "active" to distinguish the function designator inside and outside Sta.

§§

def $\mathfrak{Q}^*[\![t{:}\mathrm{ParL}]\!]\pi^*\kappa$=...

$\mathfrak{X}_3$ checks that there is the same number of actual parameters in π^*, as formal parameters in t.

Π sets up the parameters in some unspecified order.

def $\mathfrak{Q}[\![t{:}\mathrm{Par}]\!]\pi\kappa$=...

§

case"Type Ide *name*":

When used, the parameter will be coerced to $\mathfrak{T}[\![\mathrm{Type}]\!]$, see $\mathcal{V}$.

Note that the Type has to be specified. This implies that one cannot write, e.g., the following (new?) horror:

```
        integer procedure f; f := next;
        integer procedure g; if next=1 then g:=next  ;
        procedure h(x);     x;
           h(if next=2 then f else g);
```

which, by ¶4.7.3.2, is equivalent to

```
        if next=2 then f else g; .
```

Thus, although an arbitrary expression may not stand alone as a statement, a conditional expression has become, through the call by name mechanism, a conditional statement!

Incidentally, one might perhaps invoke ¶5.4.4 to invalidate the above example. This illustrates what seems to be the cause of several ALGOL 60 ambiguities: the prescription of several clashing universal rules, with no indication of the intended order of their application. Note that this problem does not occur in the mathematical semantics.

case"Type Ide **value**":

CopyArray inserts transfer functions between real and integer values, if necessary. This is so that subscripted variables may conform to ¶5.1.3, and to allow system routines to accept real or integer arrays indifferently.

§

def $\mathfrak{J}^*[\![t:StaL]\!]\rho\eta\theta=\ldots$

This function gives a tuple of the label values declared in t. Although it takes a continuation θ, it is not applied to the store.

def $\mathfrak{G}[\![t:Sta]\!]\rho\eta\theta=\ldots$

§

case"begin DecL...

Label scopes do not extend out of a block.

case"begin StaL end":

A compound statement does not restrict label scopes.

case"if Exp then...

Jumps may be made into the arms of a conditional statement.

case"Ide: Sta":

Each label is constructed from the local area η, and the continuation through the rest of the program. In fact the latter is usually just the continuation to the next label, followed by a *Hop* - see $\mathfrak{C}$.

case"goto Exp":

case"Var := AssL":

case"for Var := ForL do Sta":

Jumps into a for-statement are prohibited by restricting the scopes of the labels in Sta. This is slightly at variance with ¶4.6.6.

case"Ide(ExpL)":

case"Λ":

Note that label values are not extracted from procedure declarations.

§

def $\mathcal{A}$⟦t·AssL⟧ρχα*θ=...

χ is the type to which the left-parts must conform, and α* accumulates the locations found by evaluating the left-parts.

§

case"Var := AssL":

The left-parts are evaluated in left-to-right order.

case"Exp":

The right-part is evaluated, and the (coerced) value is assigned to all the previously-found locations in α*.

§

def $\mathcal{F}$*⟦t:ForL⟧ρχυγθ=...

Contrary to the Report, 'the' controlled variable is not undefined after exit due to exhaustion of the for-list. To make it undefined would need another evaluation of 'the' variable, which might be of significance if it is a subscripted variable. See [2], Ambiguity

def $\mathcal{F}$⟦t:For⟧ρχυγθ=...

§

case"Exp":

case"Exp_1 while Exp_2":

case"Exp_1 step Exp_2 until Exp_3":

The "conservative" interpretation of the Report.

An alternative is to use υ to evaluate α in $\mathcal{F}$*, omitting υ"lv" ‖ λα. throughout, and replacing υ"rv" by *Contents* α. Then *the location* α may be set to be undefined after a controlled exit (in $\mathcal{F}$*

§

def $\mathfrak{I}$ ...
def $\mathfrak{I}^{*}_{dec}$ ... c.f. $\mathfrak{D}^{*}$
def $\mathfrak{I}_{dec}$...
def $\mathfrak{I}^{*}_{def}$... c.f. $\mathfrak{H}^{*}$
def $\mathfrak{I}_{def}$...
def $\mathfrak{I}^{*}_{par}$... c.f. $\mathfrak{Q}^{*}$
def $\mathfrak{I}_{par}$...
def $\mathfrak{I}^{*}_{lab}$... c.f. $\mathfrak{G}^{*}$
def $\mathfrak{I}_{lab}$...

def $\mathfrak{T}$...
def $\mathfrak{T}^{*}_{dec}$... c.f. $\mathfrak{I}^{*}_{dec}$
def $\mathfrak{T}_{dec}$...
def $\mathfrak{T}^{*}_{def}$... c.f. $\mathfrak{I}^{*}_{def}$
def $\mathfrak{T}_{def}$...
def $\mathfrak{T}^{*}_{par}$... c.f. $\mathfrak{I}^{*}_{par}$
def $\mathfrak{T}_{par}$...
def $\mathfrak{T}^{*}_{lab}$... c.f. $\mathfrak{I}^{*}_{lab}$

def $\mathfrak{T}_{var}⟦t{:}\mathrm{Var}⟧\rho$...
Used in $\mathfrak{C}$, case"Var := AssL", case"for Var := ...".

def $\mathfrak{T}_{res}⟦t{:}\mathrm{Op}⟧$
def $\mathfrak{T}_{arg}⟦t{:}\mathrm{Op}⟧$
Used for type-checking in $\mathfrak{V}$, case"Exp_1 Op Exp_2", case"Op Exp".

def $\mathfrak{T}_{const}⟦t{:}\mathrm{Const}⟧$
Used only in $\mathfrak{V}$, case"Const".

def $\mathfrak{V}⟦t{:}\mathrm{Exp}⟧\rho\tau_1\mu\kappa$ = ...
§
case"ev":
This mode is used when an element of
Array + Switch + Fn + Rt + String
is required.

case"jv":
Used for designational expressions, giving an element of Label.
case"lv":
Used on left-part expressions, giving a result in Locn.

case"rv":

Transfer χ_1 will only be inserted at the outermost level of an expression, see $\mathfrak{I}_{arg}$.

§

case"if Exp$_1$ then Exp$_2$ else Exp$_3$":

The program will *not* 'fail at run-time' if, say, Exp$_2$ is of the wrong type, but only Exp$_3$ is used.

case"Exp$_1$ Op Exp$_2$":

The Report leaves unspecified the order of evaluation of operands - so does the use of Π here.

case"Op Exp":

Op will be +, - or $\neg$ (logical negation).

case"Ide(ExpL)":

Parameter-less function designators are catered for by case"Ide", below.

case"Ide[ExpL]":

Coerce allows a real array to be used when an integer value is wanted, and vice versa, but does not insert the transfer function itself.

case"Ide":

Here we deal with parameter-less function designators, as well as with simple variables.

case"Const":

This gives the value associated with a numeral or logical value.

case"(Exp)":

Note that this case only appears under case"rv".

§§

def $\mathfrak{F}[\![t{:}\mathrm{Exp}]\!]\rho\chi\kappa=\ldots$

def $\mathfrak{L}[\![t{:}\mathrm{Var}]\!]\rho\chi\kappa=\ldots$

def $\mathfrak{R}[\![t{:}\mathrm{Exp}]\!]\rho\chi\kappa=\ldots$

These functions just abbreviate standard calls of $\mathcal{V}$.

def $\mathfrak{B}^*[\![t{:}\mathrm{BdsL}]\!]\rho\kappa=\ldots$

Π_0 evaluates a list in left-to-right order, see ¶4.2.3.

def $\mathcal{B}$...

def $\mathcal{N}^\star[\![t{:}\mathrm{ExpL}]\!]\rho\kappa$= ...

The order of evaluation in $\mathcal{N}^\star$ is again left-to-right - assuming that ¶4.2.3.1 applies to variables in arithmetic expressions as well.

def $\mathcal{N}$...

def $\mathcal{N}_1[\![t{:}\mathrm{ExpL}]\!]\rho\kappa$= ...

This function is used to evaluate a switch designator, which may have only one "subscript".

def $\mathcal{U}^\star[\![t{:}\mathrm{ExpL}]\!]\rho$= ...

The actual parameters in t are partially evaluated in the correct environment ρ.

def $\mathcal{S}$...

def $\mathcal{K}$...

def $\mathcal{W}_1$...

def $\mathcal{W}_2$...

def $\mathfrak{X}_1[\![t{:}\mathrm{List}]\!]\phi$= ...

This is like $\mathfrak{X}_2$, but ϕ, when applied to an element of t, gives only a single value, not a tuple.

def $\mathfrak{X}_2[\![t{:}\mathrm{List}]\!]\phi$= ...

ϕ is applied to each element of t, and the resulting tuples are concatenated.

def $\mathfrak{X}_3[\![t{:}\mathrm{ParL}]\!]\pi^\star\phi$= ...

This function is used only in $\mathcal{Q}^\star$.

def $\mathfrak{X}_4[\![t{:}\mathrm{ForL}]\!]\phi\theta$= ...

Used only in $\mathcal{F}^\star$.

AUXILIARY FUNCTIONS - COMMENTARY

(i) Defined Functions

To some extent these functions are defined rather arbitrarily. However, the attempt has been made to keep them as simple as possible.

def *ApplyFn*...
def *ApplyRt*...

def *Area*$\kappa\sigma$=...

This, and *SetArea*, are the only defined functions which need to manipulate σ explicitly. Note that *SMap*(σ) is not duplicated. The copying of *SArea*(σ) could be justified by formulating a model for storage for ALGOL 60, in which Locn=N, Area=N and Map=N→V, and by defining *New*, *Contents*, *InArea*, etc. to satisfy the restricting axioms of (iii).

def *BasicTyp*...

def *Coerce*(δ,τ)$\tau_1\mu$=...

This function deals with most of the type-checking on identifiers, and effects the various coercions specified by the Report. The only point of divergence from (one reasonable interpretation of) the Report is in connection with "active" functions, i.e. function designators inside the definition of that same function. It is caused by the fact that the semantics presented here give ¶4.7.3.2 precedence over ¶5.4.4 (which is incorrect anyway - [2], Correction 4). Briefly, a routine r(f) may specify f to be, e.g., an integer, called by name, and then proceed to assign to it. ¶5.4.4 indicates that r(g) may be called from inside the definition of g - for substitution of the body of r will give a legal ALGOL 60 program!

def *Finished*...
def *Good*...

def *Hop*(δ:Label)=...

This function is used to effect a *Jump*, when it is known that the area will not need changing.

def *Int*...

def *Jump*(δ:Label)=...

The incorporation of the local store area into label values facilitates 'garbage collection'. See *Area*.

def *SetArea*$\eta\theta\sigma$=...

See DOMAINS-COMMENTARY, (iii), Area.

def *SetMany*$\alpha^*\varepsilon\theta$=...

The order of setting is irrelevant.

def *Transfer*$\chi\varepsilon$=...

This function is called only from $\mathcal{V}$, case"rv". It is needed because the types of expressions involving '↑' may not be ascertainable before 'running' the program.

(ii) Informally defined

A looser notation is used here, as we are not concerned with the implementation of these functions. The only point of note is:

def $\prod\omega^*\kappa$=...

This operator was introduced to describe some features of ALGOL 60 which are intentionally (?) left unspecified by the Report, e.g. the order of evaluation of the operands in an expression.

SomePermof1to(ν) gives an unspecified permutation of $1,2,\ldots,\nu$; and successive applications of this function should be regarded as giving (possibly) distinct permutations. Hence a degree of arbitrariness cannot be eliminated from the semantic value of a source-language program, when that program 'depends' on an unspecified part of ALGOL 60.

def $\prod_0\omega^*\kappa$=...

This version of $\prod$ evaluates the elements of ω^* in left-to-right order.

(iii) Restricting axioms

The functions restricted are as follows.

Access$\delta\nu^*=\alpha$

δ is an array, and ν^* is a subscript list. The array contains its bounds-list, which acts as its "dope vector".

Contents$\alpha\kappa=\theta$

κ is applied to that element of V which is currently associated with α by the store.

CopyArray$\delta\tau\kappa=\theta$

A new array, with the same bounds-list as δ, is produced, and its locations are set to the contents of the locations of δ, these values being transferred to *Main*(τ).

InArea$\alpha\kappa=\theta$

κ is applied to the result of testing whether or not α is in the current area of the store. This function is redundant in ALGOL 60 without own declarations, as extent is the same as scope.

Inside$\psi^*\nu^*=\beta$

This function checks that subscripts are within array bounds.

New$\tau\kappa=\theta$

κ is applied to an unused location, suitable for contents described by type τ.

NewArray$\tau\psi^*\kappa=\theta$

κ is applied to an array, constructed from a suitahle number of unused locations and the bounds-list ψ^*.

Note: The form of the axioms is new, and not entirely satisfactory. However, it was thought better to include this section with the ALGOL 60 description, rather than to omit it, or wait until a better formalism is found.