THE MATHEMATICAL SEMANTICS OF ALGOL 60

by

Peter Mosses

Technical Monograph PRG-12

January, 1974.

Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford.

Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford.

ABSTRACT

This paper describes the programming language ALGOL 60 (omitting own declarations) by using the Scott-Strachey mathematical semantics. A separate commentary on this description is provided, including an indication of the correspondence between the semantic description language and the λ-calculus.

Familiarity with previous publications on mathematical semantics, e.g. [6,8,10,13], and with the λ-calculus, is assumed.

CONTENTS

[The commentary is bound separately.]

REFERENCES

[1] Allen, C.D.; Chapman, D.N.; & Jones, C.B., *A Formal Definition of ALGOL 60*, IBM U.K. Technical Report TR12.105.

[2] Knuth, D., *The Remaining Trouble Spots in ALGOL 60*, Comm. ACM 10 (1967), pp. 611-618.

[3] Landin, P.J., *A Correspondence Between ALGOL 60 and Church's Lambda-Notation*, Comm. ACM 8 (1965), pp. 89-101, 158-165.

[4] Lauer, P., *Formal Definition of ALGOL 60*, IBM Lab. Vienna, Technical Report TR.25.088 (1968)

[5] Naur, P. (Ed.) *Revised Report on the Algorithmic Language ALGOL 60*, Comm. ACM 6 (1963), pp.1-17.

*[6] Scott, D., *Outline of a Mathematical Theory of Computation*, Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems (1970), pp. 169-176.

[7] Scott, D., *Lambda Calculus and Recursion Theory*, Private Communication.

*[8] Scott, D; Strachey, C. *Towards a Mathematical Semantics for Computer Languages*, Proc. Symposium on Computers and Automata. Microwave Institute Symposia Series 21, Polytechnic Institute of Brooklyn.

[9] Scott, D; Strachey, C. *Data Types as Lattices*, (in preparation).

*[10] Strachey, C. *Varieties of Programming Language*, Proc. International Computing Symposium, Cini Foundation, Venice (1972), pp.222-233.

[11] Strachey, C. *An Abstract Model for Storage*, (in preparation).

[12] Strachey, C.; Wadsworth, C.P. *Continuations: A Mathematical Semantics with Full Jumps*, Programming Research Group Technical Monograph PRG-11.

[13] Tennent, R.D., *Mathematical Semantics of Programming Languages* Technical Report 73-15 (May 1973) Dept. Computing & Information Science, Queen's University, Kingston, Ontario.

[14] Utman, R.E. (Chairman ASA X3.4.2) *Suggestions on ALGOL 60 (Rome) Issues*, Comm. ACM 6 (1963) pp.20-23.

* also available as a Programming Research Group Technical Monograph.

INTRODUCTION

This paper presents the 'semantic clauses' of ALGOL 60, using the methods developed at Oxford by Professor C. Strachey and others. The language described is that specified in the Revised Report on ALGOL 60 [5] (referred to below as "the Report"), except that 'own' declarations have been omitted - this will be discussed below.

The dividing lines between syntax and semantics, and semantics and implementation, are rather hazy - especially those between the latter two. The policy taken here has been to define primitive operations, such as *ApplyFn* and *Jump*, in a minimal fashion, and to give only axioms about the store-management functions. An implementation of this semantics could stipulate new definitions of these operations, but should preserve any theorems deducible from the original definitions and axioms (i.e. under some suitable formalism, e.g. that of the language LAMBDA [7]).

The mathematics and the comments upon it are presented separately, with the aim of exhibiting the structure of the semantic functions more clearly. In the commentary, ¶... refers to a section of the Report. The commentary on a function is headed by the name of that function, and an index is given to all functions, together with an indication of their types.

As in any large program before 'debugging', there will probably be several syntactical and semantical errors in this description. However, the author hopes soon to have a 'compiler' for semantic descriptions, the use of which should increase one's degree of belief in their correctness - this project is to form part of the author's thesis, to be submitted in supplication for the degree of D.Phil.

For the mathematical justification of the approach used here, see [6, 8, 9, 10, 11]. Also of interest as tutorial papers, in using and understanding semantic clauses, are [12, 13].

In connection with the omission of 'own' declarations, see [2, 14]. The doubts expressed in [14], about the lack of initialisation of 'own' identifiers, seem well-founded, as the semantics of the ALGOL 60 construction is very untidy. A more natural construction might be to allow initialised definitions in procedure headings, so that the scope of the definition is the body of the procedure, whilst its extent is the same as that of the procedure identifier. This suggestion was made by Landin in [3], and can be incorporated into the given syntax and semantics at almost no cost.

This report is here put forward less as 'the last word' on ALGOL 60 semantics, than as an experiment in using the Scott-Strachey semantic method to describe practical programming languages.

Any comments on the report, or suggestions for its improvement, will be very welcome.

ACKNOWLEDGEMENTS

The original inspiration for this report came from reading [1] and [3], as it was felt that a shorter and less algorithmic description of ALGOL 60 could be formulated in the Scott-Strachey semantics.

Many thanks are due to the members of the Programming Research Group, Oxford University, who studied earlier versions of this report and made many helpful comments.

This report was written whilst the author was being supported by an SRC Research Studentship.

SYNTAX

```
Prog → Sta

Sta  → begin DecL DefL StaL end
     → begin StaL end
     → if Exp then Sta1 else Sta2
     → Ide :  Sta
     → goto Exp
     → Var := AssL
     → for Var := ForL do Sta
     → Ide(ExpL)
     → Λ

StaL → Sta ; StaL
     → Sta

DecL → Dec {; Dec}*| Λ

Dec  → Type IdeL
     → Type IdeL[BdsL]

IdeL → Ide {, Ide}*

BdsL → Bds {, Bds}*

Bds  → Exp1 : Exp2

DefL → Def {; Def}* | Λ

Def  → switch Ide := ExpL
     → Type Ide(ParL); Sta

ParL → Par {,Par}* | Λ

Par  → Type Ide name
     → Type Ide value
```

```
Type → real | integer | boolean
     → array | Type array
     → procedure | Type procedure
     → label | string | switch

AssL → Var := AssL
     → Exp

ForL → For {, For}*

For  → Exp
     → Exp₁ while Exp₂
     → Exp₁ step Exp₂ until Exp₃

ExpL → Exp {, Exp}* | Λ

Exp  → if Exp₁ then Exp₂ else Exp₃
     → Exp₁ Op Exp₂
     → Op Exp
     → Ide(ExpL)
     → Ide[ExpL]
     → Ide
     → Const
     → Str
     → (Exp)

Var  → Ide[ExpL]
     → Ide

Op   → LogOp
     → RelOp
     → NumOp
```

$$\text{LogOp} \to \equiv \mid \supset \mid \vee \mid \wedge \mid \neg$$

$$\text{RelOp} \to < \mid \leqslant \mid = \mid \neq \mid \geqslant \mid >$$

$$\text{NumOp} \to + \mid - \mid \times \mid / \mid \div \mid \uparrow$$

Const → true | false
→ P INT
→ P REAL

Str → P STRING

Ide → P IDE

DOMAINS

(i) Standard Domains:

I (identifiers)
N (integers)
O (empty domain)
Q (strings)
T {true, false}

(ii) Syntactic Domains:

AssL
Bds
BdsL
Const
Dec
DecL
Def
DefL
El = Bds + Dec + Def + Exp + Ide + Par
Exp
ExpL
For
ForL
IDE (undefined)
Ide
IdeL
INT (undefined)
List = BdsL + DecL + DefL + ExpL + IdeL + ParL
LogOp
NumOp
Op
Par
ParL
Prog

REAL (undefined)
RelOp
Sta
StaL
Str
STRING (undefined)
Type
Var

(iii) Semantic Domains:

ActiveFn	=	*MakeActiveFn*(*ResLocn*:Locn, *Fn*:Fn)
Area		(indicating locations in use)
Array	=	*MakeArray*(*BdsL*:Bds*, *LocnL*:Locn*)
Bds	=	*MakeBds*(*LBd*:N, *UBd*:N)
C	=	$S \to S$
D	=	Locn + Array + Switch + Fn + ActiveFn + Rt + Label + String + Name
Den	=	⟨D, Typ⟩
E	=	D + V + Bds
Fn	=	Param* $\to$ W
G	=	$C \to C$
K	=	$E \to C$
Label	=	*MakeLabel*(*ProperArea*:Area, *Code*: C)
Locn		(addresses of real,integer and boolean values)
M	=	{"ev", "jv", "lv", "rv"}
Map		(associating locations with values)
Name	=	$M \to W$
Param	=	Typ $\to$ M $\to$ W
R		(real numbers)
Rt	=	Param* $\to$ G
S	=	*MakeS* (*SArea*:Area, *SMap*:Map)
String	=	(ALGOL 60 strings)
Switch	=	$N \to W$

$$
\begin{aligned}
\mathrm{Typ} &= \mathrm{Typ}_1 + \mathrm{Typ}_2 + \ldots + \mathrm{Typ}_7 \\
\mathrm{Typ}_1 &= \mathit{MakeTyp}(\mathit{Main}{:}\mathrm{X}_1,\ \mathit{Qual}{:}0) \\
\mathrm{Typ}_2 &= \mathit{MakeTyp}(\mathit{Main}{:}\mathrm{X}_2,\ \mathit{Qual}{:}\mathrm{Typ}_1) \\
\mathrm{Typ}_3 &= \mathit{MakeTyp}(\mathit{Main}{:}\mathrm{X}_3,\ \mathit{Qual}{:}0) \\
\mathrm{Typ}_4 &= \mathit{MakeTyp}(\mathit{Main}{:}\mathrm{X}_4,\ \mathit{Qual}{:}\mathrm{Typ}_1) \\
\mathrm{Typ}_5 &= \mathit{MakeTyp}(\mathit{Main}{:}\mathrm{X}_5,\ \mathit{Qual}{:}0) \\
\mathrm{Typ}_6 &= \mathit{MakeTyp}(\mathit{Main}{:}\mathrm{X}_6,\ \mathit{Qual}{:}\mathrm{Typ}_1+\mathrm{Typ}_2+\mathrm{Typ}_3+\mathrm{Typ}_4+\mathrm{Typ}_5) \\
\mathrm{Typ}_7 &= \mathit{MakeTyp}(\mathit{Main}{:}\mathrm{X}_7,\ \mathit{Qual}{:}\mathrm{Typ}_4) \\
\mathrm{U} &= \mathrm{I} \to \mathrm{Den} \\
\mathrm{V} &= \mathrm{N} + \mathrm{R} + \mathrm{T} \\
\mathrm{W} &= \mathrm{K} \to \mathrm{C} \\
\mathrm{X} &= \mathrm{X}_1 + \mathrm{X}_2 + \ldots + \mathrm{X}_7 \\
\mathrm{X}_1 &= \{\text{"real"}, \text{"integer"}, \text{"boolean"}, \text{"num"}\} \\
\mathrm{X}_2 &= \{\text{"array"}\} \\
\mathrm{X}_3 &= \{\text{"label"}\} \\
\mathrm{X}_4 &= \{\text{"fn"}\} \\
\mathrm{X}_5 &= \{\text{"rt"}, \text{"string"}, \text{"switch"}\} \\
\mathrm{X}_6 &= \{\text{"name"}\} \\
\mathrm{X}_7 &= \{\text{"active"}\}
\end{aligned}
$$

(iv) Denotation Domains of Bound Variables:

α : Locn

β : T

γ : G

δ : D

ε : Basic

ζ - untyped

η : Area

θ : C

ι : I

κ : $K + [E^* \rightarrow C]$

(λ)

μ : M

ν : N

ξ : N + R

(o)

π : Param

ρ : U

σ : S

τ : Typ

υ : $M \rightarrow W$

ϕ - untyped

χ : X

ψ : Bds

ω : W

t denotes a "deduction tree" belonging to a syntactic domain.

SEMANTIC FUNCTIONS

```
compiler λt:Prog. λρ₀. λθ₀.
          let τ₁ = MakeTyp("fn", MakeTyp("real", ?)) in
          let τ₂ = MakeTyp("fn", MakeTyp("integer", ?)) in
          let ρ₁ = ρ₀[Abs/τ₁/id"abs"]
                    [Sign/τ₂/id"sign"]
                    [Sqrt/τ₁/id"sqrt"]
                    [Sin/τ₁/id"sin"]
                    [Cos/τ₁/id"cos"]
                    [Arctan/ τ₁ / id"arctan"]
                    [Ln/τ₁/id"ln"]
                    [Exp/τ₁/id"exp"]
                    [Entier/τ₂/id"entier"]
          in
          switch labelof t in
          §
          case"Sta":𝔇⟦Sta⟧ρ₁θ₀
          ∮

def𝔇⟦t:Sta⟧ρθ =
     let ⟨ι*,τ*⟩ = ⟨𝒥*_lab⟦t⟧,𝒯*_lab⟦t⟧⟩ in
        Area ‖
          λη. ℭ⟦t⟧ρ[ (fix δ*. 𝔊⟦t⟧ρ[δ*/τ*/ι*]ηθ) /τ*/ι*] ‖ θ

def ℭ*⟦t:StaL⟧ρθ = switch labelof t in
§
case"Sta ; StaL": ℭ⟦Sta⟧ρ ‖ ℭ*⟦StaL⟧ρ ‖ θ
case"Sta":        ℭ⟦Sta⟧ρθ
∮
```

```
def 𝔈⟦t:Sta⟧ρθ = switch labelof t in
§
case"begin DecL DefL StaL end":
    let ⟨ι*_1,τ*_1⟩ = ⟨𝔍*_dec⟦DecL⟧, 𝔗*_dec⟦DecL⟧⟩ in
    let ⟨ι*_2,τ*_2⟩ = ⟨𝔍*_def⟦DefL⟧, 𝔗*_def⟦DefL⟧⟩ in
    let ⟨ι*_3,τ*_3⟩ = ⟨𝔍*_lab⟦StaL⟧, 𝔗*_lab⟦StaL⟧⟩ in
    Indistinct(ι*_1 cat ι*_2 cat ι*_3) → ?,
          Area ‖
           λη_1.𝔇*⟦DecL⟧ρ[?/ ?/ ι*_1cat ι*_2 cat ι*_3] ‖
            λδ*_1. Area ‖
             λη_2. let ρ_1 = ρ[δ*_1/τ*_1/ι*_1] in
                   let θ_1 = SetArea(η_1){θ} in
                   𝔈*⟦StaL⟧ρ_1[(fix δ*. let ρ_2 = ρ_1[δ*/τ*_2catτ*_3/ι*_2catι*_3] in
                                         ℋ*⟦DefL⟧ρ_2 cat 𝔊*⟦StaL⟧ρ_2η_2θ_1)
                              / τ*_2 cat τ*_3 / ι*_2 cat ι*_3] ‖ θ_1

case"begin StaL end": 𝔈*⟦StaL⟧ρθ

case"if Exp then Sta_1 else Sta_2":
   ℛ⟦Exp⟧ρ"boolean" {λβ. β → 𝔈⟦Sta_1⟧ρθ, 𝔈⟦Sta_2⟧ρθ}

case"Ide: Sta":  let ⟨δ,τ⟩ = ρ⟦Ide⟧ in Hop(δ)

case"goto Exp": 𝔍⟦Exp⟧ρ"label" ‖ λδ. Jump(δ)

case"Var := AssL":
    let χ = Main(𝔗_var⟦Var⟧ρ) in 𝔄⟦t⟧ρχ⟨⟩ ‖ θ

case"for Var := ForL do Sta":
    let τ = 𝔗_var⟦Var⟧ρ in Mainτ = "boolean" → ?,
    𝔉*⟦ForL⟧ρ(Mainτ)(𝔙⟦Var⟧ρτ)(𝔓⟦Sta⟧ρ) ‖ θ

case"Ide(ExpL)":
    Coerce(ρ⟦Ide⟧)(MakeTyp("rt",?))"ev" ‖
     λδ. ApplyRt(δ)(𝔘*⟦ExpL⟧ρ){θ}

case"Λ":  θ
$
```

```
def 𝒟*⟦t:DecL⟧ρκ = Π(𝔛_2⟦t⟧(λt_1.𝒟⟦t_1⟧ρ)) ‖ κ

def 𝒟⟦t:Dec⟧ρκ = switch labelof t in
§
case"Type IdeL":
     let τ = 𝒯⟦Type⟧ in Π(𝔛_1⟦IdeL⟧(λt_1. Newτ)) ‖ κ
case"Type IdeL[BdsL]":
     let τ = 𝒯⟦Type⟧ in
    ℬ⟦BdsL⟧ρ ‖ λψ*. Π(𝔛_1⟦IdeL⟧(λt_1. NewArrayτψ*)) ‖ κ
§

defℋ*⟦t:DefL⟧ρ = 𝔛_1⟦t⟧(λt_1.ℋ⟦t_1⟧ρ)

def ℋ⟦t:Def⟧ρ = switch labelof t in
§
case"switch Ide := ExpL":
     let ω* = 𝔛_1⟦ExpL⟧(λt_1. 𝒥⟦t_1⟧ρ"label") in λν. ω*↓ν
case"Type Ide(ParL); Sta":
     switch labelof "Type"of t in
     §
     case"procedure":
     λπ*. λθ.
      Area ‖
       λη. 𝒬*⟦ParL⟧π* ‖
        λδ*. 𝒫⟦Sta⟧ρ[δ*/𝒯*_par⟦ParL⟧/ 𝒥*_par⟦ParL⟧] ‖
         SetArea(η) ‖ θ
     case"Type procedure":
         let ⟨δ,τ⟩ = ρ⟦Ide⟧ in
         λπ*. λκ.
          Area ‖
           λη. New(Qualτ) ‖
            λα. let ⟨δ_1,τ_1⟩ = ⟨MakeActiveFn(α,δ), MakeTyp("active",τ)⟩ in
                𝒬*⟦ParL⟧π* ‖
                  λδ*. 𝒫⟦Sta⟧ρ[δ_1preδ*/ τ_1pre𝒯*_par⟦ParL⟧/ 𝒥⟦Ide⟧pre𝒥*_par⟦ParL⟧]‖
                   Contents(α) ‖
                     λβ. SetArea(η) ‖ κ(β)
     §
§
```

```
def 𝒬*⟦t:ParL⟧π*κ = ∏(𝔛₃⟦t⟧(π*)(𝒬)) ‖ κ

def 𝒬⟦t:Par⟧πκ = switch labelof t in
§
case"Type Ide name":  κ(π(𝒯⟦Type⟧))
case"Type Ide value":
      let τ = 𝒯⟦Type⟧ in
      Mainτ = "label" → π(τ)"jv" ‖ κ ,
      Mainτ = "array" → π(τ)"rv" ‖ λδ. CopyArrayδτ ‖ κ ,
                        π(τ)"rv" ‖ λε. Newτ ‖ λα. Setαε ‖ κ(α)
§

def 𝔖*⟦t:StaL⟧ρηθ = switch labelof t in
§
case"Sta ; StaL":  𝔖⟦Sta⟧ρη(ℭ*⟦StaL⟧ρθ) cat 𝔖*⟦StaL⟧ρηθ
case"Sta":  𝔖⟦Sta⟧ρηθ
§

def 𝔖⟦t:Sta⟧ρηθ = switch labelof t in
§
case"begin DecL DefL StaL end":  ⟨⟩
case"begin StaL end":  𝔖*⟦StaL⟧ρηθ
case"if Exp then Sta₁ else Sta₂":  𝔖⟦Sta₁⟧ρηθ cat 𝔖⟦Sta₂⟧ρηθ
case"Ide: Sta":  MakeLabel(η, ℭ⟦Sta⟧ρθ) pre 𝔖⟦Sta⟧ρηθ
case"goto Exp":
case"Var := AssL":
case"for Var := ForL do Sta":
case"Ide(ExpL)":
case"Λ":            ⟨⟩
§

def 𝒜⟦t:AssL⟧ρχα*θ = switch labelof t in
§
case"Var := AssL":  𝔏⟦Var⟧ρχ ‖ λα.𝒜⟦AssL⟧ρχ(α pre α*) ‖ θ
case"Exp":        ℛ⟦Exp⟧ρχ ‖ λε. SetMany(α)(ε) ‖ θ
§
```

$$\text{def } \mathcal{F}^*[\![t\text{:ForL}]\!]\rho\chi\upsilon\gamma\theta = \mathfrak{X}_4[\![t]\!](\lambda t_1.\ \mathcal{F}[\![t_1]\!]\rho\chi\upsilon\gamma) \parallel \theta$$

$\text{def } \mathcal{F}[\![t\text{:For}]\!]\rho\chi\upsilon\gamma\theta = \text{switch labelof t in}$

§

$$\begin{array}{l}
\text{case"Exp": } \upsilon\text{"lv"} \parallel \\
\quad \lambda\alpha.\ \mathcal{R}[\![\text{Exp}]\!]\rho\chi \parallel \\
\quad \lambda\xi.\ \mathit{Set}\alpha\xi \parallel \theta \\
\\
\text{case"}\text{Exp}_1 \text{ while Exp}_2\text{":} \\
\quad \text{fix } \theta'.\ \upsilon\text{"lv"} \parallel \\
\quad\quad \lambda\alpha.\ \mathcal{R}[\![\text{Exp}_1]\!]\rho\chi \parallel \\
\quad\quad \lambda\xi.\ \mathit{Set}\alpha\xi \parallel \\
\quad\quad \mathcal{R}[\![\text{Exp}_2]\!]\rho\chi \parallel \\
\quad\quad\quad \lambda\beta.\ \beta \rightarrow \gamma\{\theta'\},\ \theta \\
\text{case"}\text{Exp}_1 \text{ step Exp}_2 \text{ until Exp}_3\text{":} \\
\quad \upsilon\text{"lv"} \parallel \\
\quad \lambda\alpha.\ \mathcal{R}[\![\text{Exp}_1]\!]\rho\chi \parallel \\
\quad \lambda\xi_1.\ \mathit{Set}\alpha\xi_1 \parallel \\
\quad \text{fix } \theta'.\ \Pi(\upsilon\text{"rv"},\ \mathcal{R}[\![\text{Exp}_2]\!]\rho\chi,\ \mathcal{R}[\![\text{Exp}_3]\!]\rho\chi) \parallel \\
\quad\quad \lambda\langle\xi,\xi_2,\xi_3\rangle.\ \mathit{Finished}(\xi,\xi_2,\xi_3) \rightarrow \theta, \\
\quad\quad\quad \gamma\{\upsilon\text{"lv"} \parallel \\
\quad\quad\quad\quad \lambda\alpha'.\ \Pi(\upsilon\text{"rv"},\ \mathcal{R}[\![\text{Exp}_2]\!]\rho\chi) \parallel \\
\quad\quad\quad\quad \lambda\langle\xi',\xi_2'\rangle.\ \mathit{Set}(\alpha')(\mathit{Plus}(\xi',\xi_2')) \parallel \theta'\}
\end{array}$$

§

$$\begin{array}{l}
\text{def } \mathfrak{I}[\![t\text{:Ide}]\!] = \mathit{IdeVal}(\text{"IDE"of t}) \\
\text{def } \mathfrak{I}^*_{dec}[\![t\text{:DecL}]\!] = \mathfrak{X}_2[\![t]\!](\mathfrak{I}_{dec}) \\
\text{def } \mathfrak{I}_{dec}[\![t\text{:Dec}]\!] = \mathfrak{X}_1[\![\text{IdeL}]\!](\mathfrak{I}) \\
\text{def } \mathfrak{I}^*_{def}[\![t\text{:DefL}]\!] = \mathfrak{X}_1[\![t]\!](\mathfrak{I}_{def}) \\
\text{def } \mathfrak{I}_{def}[\![t\text{:Def}]\!] = \mathfrak{I}[\![\text{Ide}]\!] \\
\text{def } \mathfrak{I}^*_{par}[\![t\text{:ParL}]\!] = \mathfrak{X}_1[\![t]\!](\mathfrak{I}_{par}) \\
\text{def } \mathfrak{I}_{par}[\![t\text{:Par}]\!] = \mathfrak{I}[\![\text{Ide}]\!]
\end{array}$$

$\text{def } \mathfrak{I}^*_{lab}[\![t\text{:StaL}]\!] = \text{switch labelof t in}$

§

$$\begin{array}{ll}
\text{case"Sta ; StaL":} & \mathfrak{I}_{lab}[\![\text{Sta}]\!] \text{ cat } \mathfrak{I}^*_{lab}[\![\text{StaL}]\!] \\
\text{case"Sta":} & \mathfrak{I}_{lab}[\![\text{Sta}]\!]
\end{array}$$

§

```
def 𝒮_lab⟦t:Sta⟧ = switch labelof t in
§
case"begin DecL DefL StaL end":  ()
case"begin StaL end":            𝒮*_lab⟦StaL⟧
case"if Exp then Sta_1 else Sta_2":  𝒮_lab⟦Sta_1⟧ cat 𝒮_lab⟦Sta_2⟧
case"Ide: Sta":                  𝒮⟦Ide⟧ pre 𝒮_lab⟦Sta⟧
case"goto Exp":
case"Var := AssL":
case"for Var := ForL do Sta":
case"Ide(ExpL)":
case"Λ":                         ()
§

def 𝒯⟦t:Type⟧ = switch labelof t in
§
case"real":
case"integer":
case"boolean":  MakeTyp(labelof t, ?)
case"array":    MakeTyp("array", MakeTyp("real",?))
case"Type array":  MakeTyp("array",𝒯⟦Type⟧)
case"procedure":   MakeTyp("rt", ?)
case"Type procedure":  MakeTyp("fn",𝒯⟦Type⟧)
case"label":
case"string":
case"switch": MakeTyp (labelof t, ?)
§

def 𝒯*_dec⟦t:DecL⟧ = 𝔛_2⟦t⟧(𝒯_dec)
def 𝒯_dec⟦t:Dec⟧ = let τ = 𝒯⟦Type⟧ in 𝔛_1⟦IdeL⟧(λt'. τ)
def 𝒯*_def⟦t:DefL⟧ = 𝔛_1⟦t⟧(𝒯_def)
def 𝒯_def⟦t:Def⟧ = 𝒯⟦Type⟧
def 𝒯*_par⟦t:ParL⟧ = 𝔛_1⟦t⟧(𝒯_par)

def 𝒯_par⟦t:Par⟧ = switch labelof t in
§
case"Type Ide name":  MakeTyp("name", 𝒯⟦Type⟧)
case"Type Ide value": 𝒯⟦Type⟧
§
```

```
def 𝔗*lab⟦t:StaL⟧ = switch labelof t in
§
case"Sta ; StaL":  𝔗lab⟦Sta⟧ cat 𝔗*lab⟦StaL⟧
case"Sta":         𝔗lab⟦Sta⟧
§

def 𝔗lab⟦t:Sta⟧ = switch labelof t in
§
case"begin DecL DefL StaL end":  ⟨⟩
case"begin StaL end":            𝔗*lab⟦StaL⟧
case"if Exp then Sta1 else Sta2":𝔗lab⟦Sta1⟧ cat 𝔗lab⟦Sta2⟧
case"Ide: Sta":  MakeTyp("label", ?) pre 𝔗lab⟦Sta⟧
case"goto Exp":
case"Var := AssL":
case"for Var := ForL do Sta":
case"Ide(ExpL)":
case"Λ":                         ⟨⟩
§

def 𝔗var⟦t:Var⟧ρ = let⟨δ,τ⟩ = ρ⟦Ide⟧ in BasicTyp(τ)

def 𝔗res⟦t:Op⟧ = switch labelof t in
§
case"LogOp":
case"RelOp":  MakeTyp("boolean", ?)
case"NumOp":  MakeTyp("num", ?)
§

def 𝔗arg⟦t:Op⟧ = switch labelof t in
§
case"LogOp":  MakeTyp("boolean", ?)
case"RelOp":
case"NumOp":  MakeTyp("num", ?)
§
```

```
def 𝒯_const⟦t:Const⟧ = switch labelof t in

case"P REAL":   MakeTyp("real", ?)
case"P INT":    MakeTyp("integer", ?)
case"true"
case"false":    MakeTyp("boolean", ?)

def𝒱⟦t:Exp⟧ρτ₁μκ =
    let χ₁ = Main τ₁in
    switch μ in
    §
    case"ev":
         switch labelof t in
         §
         case"Ide":  Coerce(ρ⟦Ide⟧)τ₁μκ
         case"Str":  χ₁≠"string" → ? , κ(𝒮⟦Str⟧)
         $
    case"jv" :
         switch labelof t in
         §
         case"if Exp₁ then Exp₂ else Exp₃":
            ℛ⟦Exp₁⟧ρ"boolean"{λβ.β →𝒱⟦Exp₂⟧ρτ₁μκ, 𝒱⟦Exp₃⟧ρτ₁μκ}
         case"Ide[ExpL]":
             χ₁≠"label" → ?, Coerce(ρ⟦Ide⟧)(MakeTyp("switch",?))"ev" ‖
                                λδ.𝒩₁⟦ExpL⟧ρ ‖ λν. δ(ν){κ}
         case"Ide":  Coerce(ρ⟦Ide⟧)τ₁μκ
         $
    case"lv" :
         switch labelof t in
         §
         case"Ide[ExpL]":  Coerce(ρ⟦Ide⟧)(MakeTyp("array",τ₁))"ev" ‖
                               λδ.𝒩*⟦ExpL⟧ρ ‖ λν*. κ(Accessδν*)
         case"Ide":  Coerce(ρ⟦Ide⟧)τ₁μκ
         $
```

case"rv":

let $\kappa_1 = (\chi_1$="real"$\vee\ \chi_1$="integer"$) \rightarrow \kappa \circ \mathit{Transfer}\chi_1,\ \kappa$ in

switch labelof t in

§

case"if Exp_1 then Exp_2 else Exp_3":

$\mathcal{R}[\![\mathrm{Exp}_1]\!]\rho$"boolean"$\{\lambda\beta.\ \beta \rightarrow \mathcal{V}[\![\mathrm{Exp}_2]\!]\rho\tau_1\mu\kappa,\ \mathcal{V}[\![\mathrm{Exp}_3]\!]\rho\tau_1\mu\kappa\}$

case"Exp_1 Op Exp_2":

let $\langle\chi,\chi'\rangle = \langle \mathit{Main}(\mathcal{T}_{res}[\![\mathrm{Op}]\!]),\ \mathit{Main}(\mathcal{T}_{arg}[\![\mathrm{Op}]\!])\rangle$ in

$\sim \mathit{Good}\chi\chi_1\mu \rightarrow ?\ ,$

$\mathcal{T}(\mathcal{R}[\![\mathrm{Exp}_1]\!]\rho\chi',\ \mathcal{R}[\![\mathrm{Exp}_2]\!]\rho\chi') \parallel \kappa_1 \circ \mathcal{W}_2[\![\mathrm{Op}]\!]$

case"Op Exp":

let $\langle\chi,\chi'\rangle = \langle \mathit{Main}(\mathcal{T}_{res}[\![\mathrm{Op}]\!]),\ \mathit{Main}(\mathcal{T}_{arg}[\![\mathrm{Op}]\!])\rangle$ in

$\sim \mathit{Good}\chi\chi_1\mu \rightarrow ?\ ,$

$\mathcal{R}[\![\mathrm{Exp}]\!]\rho\chi' \parallel \kappa_1 \circ \mathcal{W}_1[\![\mathrm{Op}]\!]$

case"Ide(ExpL)":

$\mathit{Coerce}(\rho[\![\mathrm{Ide}]\!])(\mathit{MakeTyp}($"fn"$,\tau_1))\mu \parallel$

$\lambda\delta.\ \mathit{ApplyFn}(\delta)(\mathcal{U}^*[\![\mathrm{ExpL}]\!]\rho)\{\kappa_1\}$

case"Ide[ExpL]":

$\mathit{Coerce}(\rho[\![\mathrm{Ide}]\!])(\mathit{MakeTyp}($"array"$,\tau_1))\mu \parallel$

$\lambda\delta.\mathcal{N}^*[\![\mathrm{ExpL}]\!]\rho \parallel \lambda\nu^*.\ \mathit{Contents}(\mathit{Access}\delta\nu^*) \parallel \kappa_1$

case"Ide" :

$\mathit{Coerce}(\rho[\![\mathrm{Ide}]\!])\tau_1\mu\kappa_1$

case"Const":

$\sim \mathit{Good}(\mathit{Main}(\mathcal{T}_{const}[\![\mathrm{Const}]\!]))\chi_1\mu \rightarrow ?,\ \kappa_1(\mathcal{K}[\![\mathrm{Const}]\!])$

case"(Exp)":

$\mathcal{V}[\![\mathrm{Exp}]\!]\rho\tau_1\mu\kappa$

$\S\!\!\!|$

$\S\!\!\!|$

```
def 𝒥⟦t:Exp⟧ρχκ = 𝒱⟦t⟧ρ(MakeTyp(χ,?))"jv"κ
def ℒ⟦t:Var⟧ρχκ = 𝒱⟦t⟧ρ(MakeTyp(χ,?))"lv"κ
def ℛ⟦t:Exp⟧ρχκ = 𝒱⟦t⟧ρ(MakeTyp(χ,?))"rv"κ
def ℬ*⟦t:BdsL⟧ρκ = Π₀(𝒳₁⟦t⟧(λt₁.ℬ⟦t₁⟧ρ)) ‖ κ
def ℬ⟦t:Bds⟧ρκ = Π(𝒩⟦Exp₁⟧ρ 𝒩⟦Exp₂⟧ρ) ‖ κ∘MakeBds
def 𝒩*⟦t:ExpL⟧ρκ = Π₀(𝒳₁⟦t⟧(λt₁.𝒩⟦t₁⟧ρ) ‖ κ
def 𝒩⟦t:Exp⟧ρκ = ℛ⟦t⟧ρ"int"κ
def 𝒩₁⟦t:ExpL⟧ρκ = dimof t ≠ 1 → ?,𝒩⟦1 of t⟧ρκ
def 𝒰*⟦t:ExpL⟧ρ = 𝒳₁⟦t⟧(λt₁.𝒱⟦t₁⟧ρ)
def 𝒮⟦t:Str⟧ = StringVal("STRING"of t)

def 𝒦⟦t:Const⟧ = switch labelof t in
§
case"P REAL":  RealVal("REAL"of t)
case"P INT":   IntVal("INT"of t)
case"true":    true
case"false":   false
§

def 𝒲₁⟦t:Op⟧ε = switch labelof(1 of t) in
§
case"¬":  Not ε
case"+":  ε
case"-":  Negate ε
§

def 𝒲₂⟦t:Op⟧(ε,ε₁) = switch labelof(1 of t) in
§
case"≡":  Eqv(ε,ε₁)
case"⊃":  Imp(ε,ε₁)
case"∨ :  Or(ε,ε₁)
case"∧":  And(ε,ε₁)
case"<":  Lt(ε,ε₁)
case"≤":  Le(ε,ε₁)
case"=":  Eq(ε,ε₁)
case"≠":  Ne(ε,ε₁)
case"≥":  Ge(ε,ε₁)
case">":  Gt(ε,ε₁)
```

```
case"+":   Plus(ε,ε₁)
case"-":   Minus(ε,ε₁)
case"×":   Mult(ε,ε₁)
case"/":   RDiv(ε,ε₁)
case"÷":   IsIntε ∧ IsIntε₁ →
                let ε' = RDiv(ε,ε₁) in
                Mult(Signε', Entier(Absε')),
           ?
case"↑":   IsIntε₁ →
                Eq(Zero,ε₁) →
                     {Ne(Zero,ε) → One, ? },
                     {let ε' = Iter(Int(Absε₁))(λε₂. Mult(ε₂,ε))(One) in
                      Gt(Zero,ε₁) → ε', RDiv(One,ε')},
           IsRealε₁ →
                Eq(Zero,ε) →
                     {Gt(Zero,ε₁) → Real(Zero),?},
                Gt(Zero,ε) →
                      Exp(Mult(ε₁,Lnε)),
                ?,
           ?
$
```

$$\text{def } \mathfrak{X}_1[\![\text{t:List}]\!]\phi = \mathfrak{X}_2[\![\text{t}]\!](\lambda t_1.\ \langle\phi[\![t_1]\!]\rangle)$$

$$\text{def } \mathfrak{X}_2[\![\text{t:List}]\!]\ \phi = \mathit{CatMap}(\text{dimof t})(\lambda\nu.\ \phi[\![\nu \text{ of t}]\!])$$

$$\text{def } \mathfrak{X}_3[\![\text{t:ParL}]\!]\pi^*\phi = \text{dimof t} \neq \text{dimof } \pi^* \to ?,\ \mathit{CatMap}(\text{dimof t})(\lambda\nu.\ \langle\phi[\![\nu \text{ of t}]\!](\pi^*\!\downarrow\!\nu)\rangle)$$

$$\text{def } \mathfrak{X}_4[\![\text{t:ForL}]\!]\phi\theta = \mathit{Compound}(\text{dimof t})(\lambda\nu.\ \phi[\![\nu \text{ of t}]\!])(\theta)$$

AUXILIARY FUNCTIONS

(i) Defined:

```
def ApplyFn(δ:Fn)π*κ = δπ*κ
def ApplyRt(δ:Rt)π*θ = δπ*θ
def Areaκσ = κ(SArea(σ))(σ)
def BasicTyp(τ) = switch Mainτ in
§
case"name":
case"active":
case"fn":
case"array":  BasicTyp(Qualτ)
case"real":
case"integer":
case"boolean":  τ
default:  ?
§
def Coerce(δ,τ)τ_1μκ =
    let ⟨χ,τ'⟩ = ⟨Mainτ, Qualτ⟩ in
    let⟨χ_1,τ_1'⟩ = ⟨Mainτ_1, Qualτ_1⟩ in
    switch χ in
    §
    case"name":    δ(μ){λδ'. Coerce(δ',τ')τ_1μκ
    case"active":  μ="ev" ∨ μ="rv" → Coerce(Fnδ,τ')τ_1κ,
                   μ="lv" → Coerce(Locnδ,Qualτ')τ_1κ,?
    case"fn":      μ="ev" ∧ χ_1="fn" ∧ Good(Mainτ')(Mainτ_1')(μ) → κ(δ),
                   μ="ev" ∧ χ_1="rt" → κ(λπ*.λθ. δπ*{λε.θ}),
                   μ="rv" ∧ Good(Mainτ')(χ_1)μ → ApplyFn(δ)⟨⟩{κ}, ?
    case"array":  (μ="ev"∨ μ="rv")∧ χ_1="array" ∧ Good(Mainτ')(Mainτ_1')(μ)
                                                     → κ(δ), ?

    case"real":
    case"integer":
    case"boolean":  μ="lv" ∧ Goodχχ_1μ → κ(δ),
                    μ="rv" ∧ Goodχχ_1μ → Contentsδ{κ}, ?
```

```
    case"label":  μ="jv" ∧ χ₁="label" → κ(δ), ?
    case"rt":
    case"string":
    case"switch":  μ="ev" ∧ χ₁=χ → κ(δ), ?
    §

def Finished (ξ₁,ξ₂,ξ₃) = Lt(Mult(Minus(ξ₃,ξ₁), Sign(ξ₂)), Zero)
def Goodχχ₁μ = switch μ in
§
case"ev":
case"lv":  χ=χ₁
case"rv":  χ="boolean" → χ₁=χ,
           χ="integer" ∨ χ="real" → (χ₁="integer"∨χ₁="real"∨χ₁="num"),
                                      false
default:  false
§

def Hop(δ:Label) = Code(δ)
def Int(ξ) = Entier(Plus(ξ,Half))
def Jump(δ:Label) = SetArea(ProperAreaδ) ∥ Code(δ)
def SetAreaηθσ = θ(MakeS(η,SMapσ))
def SetManyα*εθ = Compound(dimof α*)(λν. Set(α*↓ν)(ε))(θ)
def Transferχξ =
    χ="real" → Realξ
    χ="integer" → Intξ, ?
```

(ii) Informally defined:

```
def CatMap(ν)(ϕ) = ϕ(1) cat ϕ(2) cat ... cat ϕ(ν)
def Compound(ν)(ϕ)(θ) = ϕ(1) ∥ ϕ(2) ∥ ... ∥ ϕ(ν) ∥ θ
def Indistinct (ι*) = let ν = dimof ι* in
                      (ι*↓1=ι*↓2) ∨ (ι*↓1=ι*↓3) ∨ ... ∨ (ι*↓1=ι*↓ν)
                      ∨             (ι*↓2=ι*↓3) ∨ ... ∨ (ι*↓2=ι*↓ν)
                                           . . . .
                                                      ∨ (ι*↓(ν-1)=ι*↓ν)
```

def $Inside\psi^*\nu^*$ = let $\nu' =$ dimof ν^* in

$$LBd(\psi^*\downarrow 1) \le \nu^*\downarrow 1 \le UBd(\psi^*\downarrow 1)$$
$$LBd(\psi^*\downarrow 2) \le \nu^*\downarrow 2 \le UBd(\psi^*\downarrow 2)$$
$$\ldots\ldots$$
$$LBd(\psi^*\downarrow \nu') \le \nu^*\downarrow \nu' \le UBd(\psi^*\downarrow \nu')$$

def $Iter(\nu)(\phi\text{:Basic}\to\text{Basic})(\varepsilon) = \phi(\phi(\ldots\phi(\varepsilon)\ldots))$
ν occurrences of ϕ.

def $\Pi\omega^*\kappa$ = let ν = dimof ω^* in
let p = $SomePermof1to(\nu)$ in
$\omega^*\downarrow p(1) \parallel \lambda\zeta_{p(1)}.\ \omega^*\downarrow p(2) \parallel \lambda\zeta_{p(2)}.\ \ldots\ \omega^*\downarrow p(\nu).\ \parallel \lambda\zeta_{p(\nu)}.$
$\kappa(\zeta_1,\zeta_2,\ldots,\zeta_\nu)$

def $\Pi_0\omega^*\kappa$ = let ν = dimof ω^* in
$\omega^*\downarrow 1 \parallel \lambda\zeta_1.\ \omega^*\downarrow 2 \parallel \lambda\zeta_2.\ \ldots\ \omega^*\downarrow\nu \parallel \lambda\zeta_\nu.\ \kappa(\zeta_1,\zeta_2,\ldots,\zeta_\nu)$

(iii) Restricting axioms:

We abbreviate as follows.

(a) Φ- eq E asserts that the argument of Φ is true, i.e. that
axiom $E[T/\Phi] = E[K/\Phi]$
where
$T = \lambda\beta.\ \beta\to I,\ ?$
$K = \lambda\beta.\ I$
$I = \lambda\sigma.\ \sigma$

(b) axiom $E_1 \leftrightarrow E_2$ denotes
axiom $\Pi\langle E_1,E_2\rangle = \Pi_0\langle E_1,E_2\rangle$ (i.e. E_1 and E_2 commute).

(c) Free variables are universally quantified over their domains.

Φ-eq $\quad New\tau \parallel \lambda\alpha.\ InArea\alpha \parallel \lambda\beta.\ \Phi(\beta)$

Φ-eq $\quad New\tau \parallel \lambda\alpha.\ Contents\alpha \parallel \lambda\varepsilon.\ \Phi(\varepsilon=?)$

Φ-eq $\quad InArea\alpha \parallel \lambda\beta.\ New\tau \parallel \lambda\alpha_1.\ \Phi(\beta \supset \alpha\neq\alpha_1)$

Φ-eq $\quad InArea\alpha \parallel \lambda\beta.\ Set\alpha\varepsilon \parallel Contents\alpha \parallel \lambda\varepsilon_1.\ \Phi(\varepsilon=\varepsilon_1)$

Φ-eq $\quad InArea\alpha \parallel \lambda\beta.\ Contents\alpha \parallel \lambda\varepsilon.\ Contents\alpha \parallel \lambda\varepsilon_1.\ \Phi(\varepsilon=\varepsilon_1)$

Φ-eq $\quad InArea\alpha \parallel \lambda\beta.\ NewArray\tau\psi^* \parallel \lambda\delta.\ \Phi(\beta \supset (Inside\psi^*\nu^* \supset Access\delta\nu^*\neq\alpha))$

Φ-eq $\quad NewArray\tau\psi^* \parallel \lambda\delta.\ \Phi(BdsL(\delta)=\psi^*)$

Φ-eq $\quad NewArray\tau\psi^* \parallel \lambda\delta.\ \Phi((Inside\psi^*\nu^* \wedge Inside\psi^*\nu_1^* \wedge Access\delta\nu^*=Access\delta\nu_1^*)$
$\qquad\qquad \supset \nu^*=\nu_1^*)$

Φ-eq $\quad NewArray\tau\psi^* \parallel \lambda\delta.\ InArea(Access\delta\nu^*) \parallel \lambda\beta.\ \Phi(Inside\psi^*\nu^* \supset \beta)$

Φ-eq $\quad NewArray\tau\psi^* \parallel \lambda\delta.\ Contents(Access\delta\nu^*) \parallel \lambda\varepsilon.\ \Phi(Inside\psi^*\nu^* \supset \varepsilon=?)$

Φ-eq $\quad InArea\alpha \parallel \lambda\beta.\ NewArray\tau\psi^* \parallel \lambda\delta.\ \Phi(\beta \supset (Inside\psi^*\nu^* \supset Access\delta\nu^*\neq\alpha))$

Φ-eq $\quad CopyArray\delta_1\tau \parallel \lambda\delta.\ \Phi(BdsL(\delta)=BdsL(\delta_1))$

Φ-eq $\quad CopyArray\delta_1\tau \parallel \lambda\delta.\ InArea(Access\delta\nu^*) \parallel \lambda\beta.\ \Phi(Inside\psi^*\nu^* \supset \beta)$

Φ-eq $\quad CopyArray\delta_1\tau \parallel \lambda\delta.\ Contents(Access\delta\nu^*) \parallel \lambda\varepsilon.\ Contents(Access\delta_1\nu^*) \parallel$
$\qquad \lambda\varepsilon_1.\ \Phi(Inside(BdsL(\delta))(\nu^*) \supset \varepsilon=Transfer(BasicTyp\tau)(\varepsilon_1))$

Φ-eq $\quad InArea\alpha \parallel \lambda\beta.\ CopyArray\delta_1\tau \parallel \lambda\delta.\ \Phi(\beta \supset (Inside(BdsL(\delta))(\nu^*) \supset$
$\qquad\qquad Access\delta\nu^*\neq\alpha))$

axiom $\quad \alpha\neq\alpha_1 \supset Contents\alpha \leftrightarrow \lambda\kappa.\ Set\alpha_1\varepsilon\{\kappa(?)\}$

axiom $\quad \alpha\neq\alpha_1 \supset Contents\alpha \leftrightarrow InArea\alpha_1$

axiom $\quad \alpha\neq\alpha_1 \supset Contents\alpha \leftrightarrow Contents\alpha_1$

INDEX

$\mathcal{T}_{const}[\![t:Const]\!]$	$= \tau$	19, C15
$\mathcal{T}_{dec}[\![t:Dec]\!]$	$= \tau^*$	17, C15
$\mathcal{T}^*_{dec}[\![t:DecL]\!]$	$= \tau^*$	17, C15
$\mathcal{T}_{def}[\![t:Def]\!]$	$= \tau$	17, C15
$\mathcal{T}^*_{def}[\![t:DefL]\!]$	$= \tau^*$	17, C15
$\mathcal{T}_{lab}[\![t:Sta]\!]$	$= \tau^*$	18, C15
$\mathcal{T}^*_{lab}[\![t:StaL]\!]$	$= \tau^*$	18, C15
$\mathcal{T}_{par}[\![t:Par]\!]$	$= \tau$	17, C15
$\mathcal{T}^*_{par}[\![t:ParL]\!]$	$= \tau^*$	17, C15
$\mathcal{T}_{res}[\![t:Op]\!]$	$= \tau$	18, C15
$\mathcal{T}_{var}[\![t:Var]\!]\rho$	$= \tau$	18, C15
$\mathcal{U}^*[\![t:ExpL]\!]\rho$	$= \pi^*$	21, C17
$\mathcal{V}[\![t:Exp]\!]\rho\tau\mu\kappa$	$= \theta$	19, C15
$\mathcal{W}_1[\![t:Op]\!]\varepsilon$	$= \varepsilon'$	21, C17
$\mathcal{W}_2[\![t:Op]\!](\varepsilon,\varepsilon_1)$	$= \varepsilon'$	21, C17
$\mathcal{X}_1[\![t:List]\!]\phi$	$= \omega^*$	22, C17
$\mathcal{X}_2[\![t:List]\!]\phi$	$= \omega^*$	22, C17
$\mathcal{X}_3[\![t:ParL]\!]\pi^*\phi$	$= \omega^*$	22, C17
$\mathcal{X}_4[\![t:ForL]\!]\phi\theta$	$= \theta'$	22, C17

$Abs\,\xi = \xi'$	-	-
$Access\,\delta\nu^* = \alpha$	-	C20
$And\,(\beta,\beta_1) = \beta'$	-	-
$ApplyFn\,(\delta:Fn)\pi^*\kappa = \theta$	23,	C18
$ApplyRt\,(\delta:Rt)\pi^*\theta = \theta'$	23,	C18
$Arctan\,\xi = \xi'$	-	-
$Area\,\kappa = \theta$	23,	C18
$BasicTyp\,\tau = \tau'$	23,	C18
$BdsL\,(\delta:Array) = \psi^*$	9,	-
$CatMap\,\nu\phi = \zeta^*$	24,	-
$Code\,(\delta:Label) = \theta$	9,	-
$Coerce\,(\delta,\tau)\tau_1\mu\kappa = \theta$	23,	C18
$Compound\,\nu\phi\theta = \theta'$	24,	-
$Contents\,\alpha\kappa = \theta$	-	C20
$CopyArray\,\delta\tau\kappa = \theta$	-	C20
$Cos\,\xi = \xi'$	-	-

$Entier\,\xi$	=	ν	-	-
$Eq(\xi,\xi_1)$	=	β	-	-
$Eqv(\beta,\beta_1)$	=	β'	-	-
$Exp\,\xi$	=	ξ'	-	-
$Finished(\xi_1,\xi_2,\xi_3)$	=	β	24,	C18
$Fn(\delta$:ActiveFn$)$	=	δ:Fn	9,	-
$Ge(\xi,\xi_1)$	=	β	-	-
$Good\,\chi\chi_1\mu$	=	β	24,	C18
$Gt(\xi,\xi_1)$	=	β	-	-
$Half$	=	ξ	-	-
$Hop(\delta$:Label$)$	=	θ	24,	C18
$IdeVal($t:IDE$)$	=	ι	-	C3
$Imp(\beta,\beta_1)$	=	β'	-	-
$InArea\,\alpha\kappa$	=	θ	-	C20
$Indistinct(\iota^*)$	=	β	24	-
$Inside\,\psi^*\nu^*$	=	β	25,	C20
$Int\,\xi$	=	ν	24,	C19
$IntVal($t:INT$)$	=	ν	-	C3
$IsInt\,\xi$	=	β	-	-
$IsReal\,\xi$	=	β	-	-
$Iter\,\nu\phi\varepsilon$	=	ε'	25	-
$Jump(\delta$:Label$)$	=	θ	24,	C19
$LBd(\psi)$	=	ν	9	-
$Le(\xi,\xi_1)$	=	β	-	-
$Ln\,\xi$	=	ξ'	-	-
$LocnL(\delta$:Array$)$	=	α^*	9	-
$Lt(\xi,\xi_1)$	=	β	-	-
$Main\,\tau$	=	χ	10	-
$MakeActiveFn(\delta$:Fn$,\alpha)$	=	δ	9	-
$MakeArray(\psi^*,\alpha^*)$	=	δ	9	-
$MakeBds(\nu,\nu_1)$	=	ψ	9	-
$MakeLabel(\eta,\gamma)$	=	δ	9	-
$MakeS(\eta,\phi$:Map$)$	=	σ	9	-
$MakeTyp(\chi,\tau)$	=	τ	10	-
$Minus(\xi,\xi_1)$	=	ξ'	-	-
$Mult(\xi,\xi_1)$	=	ξ'	-	-

$Ne(\xi,\xi_1)$	$= \beta$	-	-
$Negate\,\xi$	$= \xi'$	-	-
$New\,\tau\kappa$	$= \theta$	-	C20
$NewArray\,\tau\psi^*\kappa$	$= \theta$	-	C20
$Not\,\beta$	$= \beta'$	-	-
One	$= \nu$	-	-
$Or(\beta,\beta_1)$	$= \beta'$	-	-
$Plus(\xi,\xi_1)$	$= \xi'$	-	-
$ProperArea(\delta{:}\mathrm{Label})$	$= \eta$	9	-
$Qual\,\tau$	$= \tau'$	10	-
$RDiv(\xi,\xi_1)$	$= \xi'$	-	-
$Real\,\xi$	$= \xi'$	-	-
$RealVal(\mathrm{t{:}REAL})$	$= \xi$	-	C3
$ResLocn(\delta{:}\mathrm{ActiveFn})$	$= \alpha$	9	-
$SomePermof1to(\nu)$	$= \zeta{:}\mathrm{N}\rightarrow\mathrm{N}$	-	-
$SArea\,\sigma$	$= \eta$	9	-
$Set\,\alpha\varepsilon\theta$	$= \theta'$	-	-
$SetArea\,\eta\theta$	$= \theta'$	24,	C19
$SetMany\,\alpha^*\varepsilon\theta$	$= \theta'$	24,	C19
$Sign\,\xi$	$= \nu$	-	-
$Sin\,\xi$	$= \xi'$	-	-
$SMap\,\sigma$	$= \phi{:}\mathrm{Map}$	9	-
$Sqrt\,\xi$	$= \xi'$	-	-
$StringVal(\mathrm{t{:}STRING})$	$= \delta$	-	C3
$Transfer\,\chi\xi$	$= \xi'$	24,	C19
$UBd\,\psi$	$= \nu$	9	-
$Zero$	$= \nu$	-	-
$\prod\omega^*\kappa$	$= \theta$	25,	C19
$\prod_0\omega^*\kappa$	$= \theta$	25,	C19

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 IDE.

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.

Area

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'.

Map

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

R
String

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

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.
$\mathrm{A}, \mathrm{A}_1, \ldots$ / $\mathrm{A}^*, \mathrm{A}_1^*, \ldots$	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.

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

$a^*\downarrow i = a_i$

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

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

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

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

$$\begin{array}{ll} \mathrm{fxyz} = ((f(x))(y))(z) & \\ a \| b \| c = a\{b\{c\}\} & \end{array} \Bigg\} \quad \text{abbreviations to avoid a multitude of brackets}$$

Note: $\|$ is less binding than juxtaposition and $\rightarrow$, but does not terminate a λ-expression.

(e),{e} are used for parsing purposes, and to help readability.

$e(a_1,\ldots,a_m) = e\langle a_1,\ldots,a_m\rangle$

$[\![e]\!]$ - 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 Q to I

$$\rho[\delta/\tau/\iota] = \rho', \text{ where } \rho'[\![\iota']\!] = \begin{cases} \langle\delta,\tau\rangle & \text{if } \iota' = \iota \\ \rho[\![\iota']\!] & \text{if } \iota' \neq \iota \end{cases}$$

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

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

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

compiler e
def $\mathcal{A}[\![t]\!]\alpha\beta\gamma = e_1$
...
def $Abc\alpha\beta\gamma = e_n$
- 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 $\rho_1 = \rho_0[$...

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 $\mathcal{I}^*_{lab}$.

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

$\mathfrak{F}$ 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 $\rho[\![\text{Ide}]\!]$ here.

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

case"goto Exp":

$\mathcal{G}$ 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 $\mathfrak{C}$, case"begin DecL...".

def $\mathcal{H}$[[t:Def]]ρ=...

§

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":

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

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

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 $\mathcal{Q}$*[[t:ParL]]π*κ=...

$\mathcal{K}_3$ 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 $\mathcal{Q}$[[t:Par]]πκ=...

§

case"Type Ide name":

When used, the parameter will be coerced to $\mathcal{T}$[[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⟧$\rho\eta\theta$=...

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 $\mathfrak{J}$⟦t:Sta⟧$\rho\eta\theta$=...

§

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 $\mathfrak{T}_{var}[\![t\text{:Var}]\!]\rho$...
Used in $\mathfrak{C}$, case"Var := AssL", case"for Var := ...".

def $\mathfrak{T}_{res}[\![t\text{:Op}]\!]$
def $\mathfrak{T}_{arg}[\![t\text{:Op}]\!]$
Used for type-checking in $\mathfrak{V}$, case"Exp_1 Op Exp_2", case"Op Exp".

def $\mathfrak{T}_{const}[\![t\text{:Const}]\!]$
Used only in $\mathfrak{V}$, case"Const".

def $\mathfrak{V}[\![t\text{:Exp}]\!]\rho\tau_1\mu\kappa$=...
§
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 $\mathcal{T}_{arg}$.

§

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 $\mathcal{J}$⟦t:Exp⟧$\rho\chi\kappa$=...

def $\mathcal{L}$⟦t:Var⟧$\rho\chi\kappa$=...

def $\mathcal{R}$⟦t:Exp⟧$\rho\chi\kappa$=...

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

def $\mathcal{B}$*⟦t:BdsL⟧$\rho\kappa$=...

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

def $\mathcal{B}$...

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

The order of evaluation in $\mathcal{N}^*$ 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:ExpL]]$\rho\kappa$=...

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

def $\mathfrak{U}^*$[[t:ExpL]]ρ=...

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

def $\mathcal{S}$...

def $\mathfrak{K}$...

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

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

def $\mathfrak{X}_1$[[t:List]]ϕ=...

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:List]]ϕ=...

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

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

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

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

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

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*$(\delta,\tau)\tau_1\mu$=...

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 '$\uparrow$' 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.

*SomePermof*1*to*(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 suitable 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.