The Essence of Algol[1]

John C. Reynolds
Syracuse University, Syracuse, NY, U.S.A.

Abstract

Although Algol 60 has been uniquely influential in programming language design, its descendants have been significantly different than their prototype. In this paper, we enumerate the principles that we believe embody the essence of Algol, describe a model that satisfies these principles, and illustrate this model with a language that, while more uniform and general, retains the character of Algol.

1. The Influence of Models of Algol

Among programming languages, Algol 60 [1] has been uniquely influential in the theory and practice of language design. It has inspired a variety of models which have in turn inspired a multitude of languages. Yet, almost without exception, the character of these languages has been quite different than that of Algol itself. To some extent, the models failed to capture the essence of Algol and gave rise to languages that reflected that failure.

One main line of development centered around the work of P. J. Landin, who devised an abstract language of applicative expressions [2] and showed that Algol could be translated into this language [3]. This work was influenced by McCarthy's Lisp [4] and probably by unpublished ideas of C. Strachey; in turn it led to more elaborate models such as those of the Vienna group [5]. Later many of its basic ideas, often considerably transformed, reappeared in the denotational semantics of Scott and Strachey [6].

In [2], after giving a functional description of applicative expressions, Landin presented a state-transition machine, called the SECD machine, for their evaluation. Then in [3] he extended applicative expressions to "imperative applicative expressions" by introducing assignment and a label-like mechanism called the J-operator. The imperative applicative expressions were not described functionally, but by an extension of the SECD machine called the "sharing machine." In later models, such as that of the Vienna group, sharing was elucidated by introducing a state component usually called the "store" or "memory."

---

[1]Work supported by National Science Foundation Grant MCS-8017577 and U.S. Army Contract DAAK80-80-C-0529. First appeared in J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Proceedings of the International Symposium on Algorithmic Languages, Amsterdam, October 1981. North-Holland, Amsterdam. Reprinted in *Algol-like Languages*, ed. P. W. O'Hearn and R. D. Tennent, vol. 1, pp. 67–88, Birkhäuser, 1997.

For our present concerns, three aspects of Landin's model are especially significant. First, the variety of values that can be assigned to variables is the same as the variety that can be denoted by identifiers or passed as parameters. Landin does not emphasize this fact; it is simply a direct consequence of the typelessness of imperative applicative expressions. Second, no distinction is made between assignments to variables and assignments to locations embedded within data structures. Again, this is inherent in the nature of the model, in which variables themselves are locations embedded within the data structures of the sharing machine.

Finally, since operands are evaluated before operators, the basic method of parameter passing is call by value, and call by name is described in terms of call by value using parameterless functions (in contrast to the Algol 60 report [1], where call by value is described in terms of call by name using appropriately initialized local variables). This approach apparently stems from the view that undefined values do not "exist," so that a function cannot map an undefined value into a defined value (as in Lisp, where the conditional must be regarded as a special form rather than a function). This is in contrast with the more recent view of Scott that an undefined value is as legitimate as any other; its only peculiarity is being least in a partial ordering that must be respected by functions.

Directly or indirectly, Landin's model was the basis for a number of programming languages, including his own Iswim [7], Evans and Wozencraft's Pal [8], and my Gedanken [9]. Less obviously, the model influenced Algol 68 [10], despite the significant distinction that this language is highly typed. All of these languages inherited from the model the characteristics described above: anything that can be passed as a parameter can be assigned to a variable, there is no fundamental distinction between assignments to variables and to components of data structures, and call by value is either the only or the basic mode of parameter transmission.

As a consequence, all of these languages are significantly different from Algol; in certain respects they are closer to the spirit of Lisp. They are all subject to the criticism of references made by Hoare [11]. (Strictly speaking, only Algol 68 and Gedanken use the reference concept, but Hoare's criticism is equally applicable to the sharing or L-value approach used in Iswim and Pal.)

Moreover, except for Algol 68, none of these languages obey a stack discipline. It would require a clever compiler to make any use of a stack during program execution, and even then it would be difficult for a programmer to foresee when such use would occur.

In Algol 68, a stack discipline is obtained by imposing the restriction that a procedure value becomes undefined upon exit from any block in which a global variable of the procedure is declared. However, this restriction is imposed for the specific purpose of rescuing the stack; a stack discipline is not a natural consequence of the basic character of the language.

Another line of development stemming from Algol 60 has led to languages such as Pascal [12] and its descendants, e.g. Euclid [13], Mesa [14], and Ada [15], which are significantly lower-level than Algol. Each of these languages seriously restricts the block or procedure mechanism of Algol by eliminating features such as call by name, dynamic arrays, or procedure parameters.

I am not familiar enough with the history of these languages to do more than speculate about the influence of models. However, a desire to be "closer to the machine" than Algol 60 seems evident from the abandonment of features requiring inefficient or "clever" implementations. In this respect, implementations themselves can be thought of as models influencing language design.

In addition, the influence of program-proving formalisms, particularly the work of Hoare [16], is clear. An axiomatic definition of Pascal [17] seems to have influenced that language, and the axiomatization of Euclid [13] was a major goal of its design.

Since Hoare's treatment of procedures [18] does not encompass call by name, procedure parameters, or aliasing, it may account for the weakening of the procedure mechanism in some of these languages. Certainly the view of procedures given by this kind of axiomatization is profoundly different than the copy rule.

2. Some Principles

The preceding somewhat biased history is intended to motivate a new model that I believe captures the essence of Algol and can be used to develop a more uniform and general "Idealized Algol" retaining the character of its prototype. Although its genesis lies in the definition of the simple imperative language given in [19], the crux of the model is a treatment of procedures and block structure developed by F. J. Oles and myself.

This paper only describes the basic nature of the model, and it avoids the mathematical sophistication, involving universal algebra and category theory, that is needed to reveal its elegance. A complete and mathematically literate description is given in [20].

It should be emphasized that the description of "Idealized Algol" in this paper is extremely tentative and only intended to illustrate the model.

Before delving into the details, we state the principles that we believe embody the essence of Algol:

1. Algol is obtained from the simple imperative language by imposing a procedure mechanism based on a fully typed, call-by-name lambda calculus.

In other words, Landin was right in perceiving the lambda calculus underlying Algol, but wrong in embracing call by value rather than call by name.

The qualification "fully typed" indicates agreement with van Wijngaarden that all type errors should be syntactic errors, and that this goal requires a syntax with an infinite number of phrase classes, themselves possessing grammatical or (more abstractly) algebraic structure. (I believe that this characteristic will be the most influential and long lasting aspect of Algol 68.) The failure of this property for Algol 60 is a design mistake, not part of its essence.

When carried to the extreme, this principle suggests that the lambda calculus is the source of all identifier binding. More precisely, except for syntactic sugar (language constructs that can be defined as abbreviations in terms of more basic constructs, as the **for** statement is defined in the Algol 60 Report), the only binding mechanism should be the lambda expression.

2. There are two fundamentally different kinds of type: *data types*, each of which denotes a set of values appropriate for certain variables and expressions, and *phrase types*, each of which denotes a set of meanings appropriate for certain identifiers and phrases.

This syntactic distinction reflects that fact that in Algol values (which can be assigned to variables) are inherently different from meanings (which can be denoted by identifiers and phrases, and passed as parameters). Thus Algol-like languages contradict the principle of completeness [9].

Moreover, in Algol itself data types are limited to unstructured types such as **integer** or **Boolean**, while structuring mechanisms such as procedures and arrays are only applicable to phrase types.

3. The order of evaluation for parts of expressions, and of implicit conversions between data or phrase types, should be indeterminate, but the meaning of the language, at an appropriate level of abstraction, should be independent of this indeterminacy.

By "appropriate" we mean a level of abstraction where overflow and roundoff are ignored and termination with an error message is regarded as equivalent to nontermination. This principle prohibits expressions with side effects such as assignments to nonlocal variables or jumps to nonlocal labels, but not expressions that cause error stops.

If types are described grammatically, the indeterminacy of implicit conversions will cause ambiguity. For example, in a context calling for a real expression, 3 + 4 might be parsed as either

```
               <real exp>                                  <real exp>
                   |                                        /  |  \
             <integer exp>                 or      <real exp>  +  <real exp>
              /    |    \                               |              |
    <integer exp>  +  <integer exp>              <integer exp>   <integer exp>
```

Except for overflow and (with unfortunate hardware) roundoff, both parses should have the same meaning.

4. Facilities such as procedure definition, recursion, and conditional and case constructions should be uniformly applicable to all phrase types.

This principle leads to procedures whose calls are procedures, but under a call-by-name regime such procedures do not violate a stack discipline in the way that, for example, function-returning functions in Gedanken violate such a discipline. More interestingly, this principle leads to conditional variables and procedures whose calls are variables; indeed arrays can be regarded as a special case of the latter.

5. The language should obey a stack discipline, and its definition should make this discipline obvious.

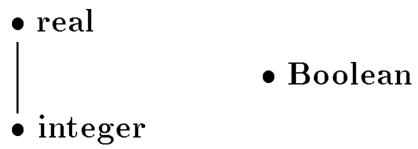
Almost any form of language definition can be divided into primary and secondary parts, e.g.

	Primary	Secondary
Denotational Semantics	Domain equations	Semantic equations
Algebraic Semantics	Definition of the target algebra carrier	Definition of the target algebra operations
Operational Semantics	Definition of the set of states of the interpreter	Definition of the state-transition function

By "should make the stack discipline obvious" we mean that the stack discipline should be a consequence of the primary part of the language definition. Specifically, the primary part should show that the execution of a command never changes the "shape" of the store, i.e. the aspect of the store that reflects storage allocation.

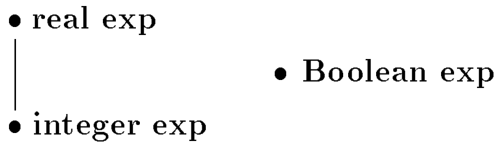
3. Data Types and Expressions

To stay close to Algol 60, we take {**integer, real, Boolean**} as the set of data types. To introduce an implicit conversion from **integer** to **real**, we define the partial ordering



and say that τ is a subtype of τ' when $\tau \leq \tau'$.

For each data type τ there is a phrase type τ **exp**(ression), and these phrase types inherit the subtype relation of the data types:



When $\theta \leq \theta'$ we again say that θ is a subtype of θ', now meaning that any phrase of type θ can appear in any context requiring a phrase of type θ', e.g. any integer expression can occur in any context requiring a real expression.

A *type assignment* is a function from some finite set of identifiers to phrase types. To describe the syntax of our language we will use phrase-class names of the form $<\theta, \pi>$, where θ is a phrase type and π is a type assignment, to denote the set of phrases P such that

1. The identifiers occurring free in P belong to the domain of π.

2. When its free identifiers are given the phrase types indicated by π, P has phrase type θ.

We will describe syntax by production schemas (in the spirit though not the notation of van Wijngaarden) in which the metavariables τ, θ, π, and ι range over data types, phrase types, type assignments, and identifiers, respectively. A fragment of an appropriate syntax for expressions is

$$<\theta, \pi> ::= <\theta', \pi> \qquad \text{when } \theta' \leq \theta$$

$$<\theta, \pi> ::= \iota \qquad \text{when } \iota \in \mathrm{dom}(\pi) \text{ and } \pi(\iota) = \theta$$

$$<\text{integer exp}, \pi> ::= 0 \mid 1 \mid <\text{integer exp}, \pi> + <\text{integer exp}, \pi>$$

$$<\text{real exp}, \pi> ::= 0.5 \mid <\text{real exp}, \pi> + <\text{real exp}, \pi>$$

$$\begin{aligned} <\text{Boolean exp}, \pi> ::= {} & \mathbf{true} \mid \mathbf{false} \mid <\tau\ \text{exp}, \pi> = <\tau\ \text{exp}, \pi> \\ & \mid <\text{Boolean exp}, \pi>\ \mathbf{and}\ <\text{Boolean exp}, \pi> \end{aligned}$$

(Here $\mathrm{dom}(\pi)$ denotes the domain of the type assignment π.) This is an abstract syntax to the extent that precedence considerations are ignored. One could "concretize" it by adding parentheses around the right side of each production, but a realistic concrete syntax would require far fewer parentheses. In fact, we will use fewer parentheses in our examples of programs, trusting the reader's intuition to supply the missing ones sensibly.

In the first two production schemas θ ranges over all phrase types, not just the types of expressions introduced so far. The first schema shows the purpose of the subtype relationship. The second shows that an identifier assigned some phrase type can always be used as a phrase of that type.

In accordance with Principle 3, the syntax is ambiguous (aside from parenthesization considerations), but this ambiguity must not result in ambiguous meanings. An appropriate method for insuring this requirement is described in [19]; it requires that the syntax possess a property that might be called "minimal typing":

> For any phrase P and type assignment π, if there is any θ such that $P \in <\theta, \pi>$, then there is a *minimal* θ_0 such that $P \in <\theta, \pi>$ if and only if $\theta_0 \leq \theta$.

(When a phrase-class name is used as a set it stands for the set of all phrases that can be derived from that phrase-class name.)

To prohibit expressions with side effects, we will forbid any occurrence of commands within expressions (except in vacuous contexts such as parameters of constant procedures) and insist that the bodies of function procedures be expressions. Actually, this is unnecessarily Draconian; one would like to permit block expressions, as in Algol W [22], but restricted to avoid side effects. However, this topic is beyond the scope of this paper.

4. The Simple Imperative Language

The next step is to introduce variables for each data type. But here we encounter a surprising complication. As a consequence of Principle 4, we want to have conditional variables. For example, when n is an integer variable and x is a real variable, we should be able to write **if** p **then** n **else** x on either side of an assignment statement. When used on the right side, this phrase must be considered as a real expression, since when p is false it can produce a noninteger value. But when used on the left side, it must be considered an integer variable, since when p is true it cannot accept a noninteger value. Thus there are variables that accept a different data type than they produce.

The first step in dealing with this situation is to realize that, in addition to variables, which accept and produce values, and expressions, which only produce values, it is natural to introduce phrases called *acceptors*, which only accept values. Thus for each data type τ, we will have the phrase type τ **acc**(eptor). The subtype relation for acceptors is the dual of the subtype relation for data types. For example, since **integer** is a subtype of **real**, integer values can be implicitly converted into real values, so that a real acceptor can be used in any context requiring an integer acceptor, i.e. **real acc** $\leq$ **integer acc**.

The second step is to categorize variables separately by the data types that they accept and produce. Thus for each pair of data types τ_1 and τ_2, we have the phrase type τ_1 (accepting) τ_2 (producing) **var**(iable), which is a subtype of both τ_1 **acc** and τ_2 **exp**. Among variables themselves, $\tau_1\tau_2$ **var** is a subtype of $\tau_1'\tau_2'$ **var** when τ_1 **acc** is a subtype of τ_1' **acc** and τ_2 **exp** is a subtype of τ_2' **exp**; i.e. when the data types satisfy $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$.

The case construction for variables raises the same difficulty as the conditional. But a further problem arises if the empty construction **case** n **of** () is permitted. Of course, it would not be unreasonable to prohibit this construction, but it is consistent to view it as a phrase whose phrase type **univ**(ersal) is a subtype of all phrase types. All phrases of this type have the meaning "undefined," which implicitly converts into the undefined element $\perp$ of the domain of meanings of any other phrase type.

The only other phrase type needed to describe the simple imperative language is **comm**(and). (Throughout this paper, we will speak of commands rather than statements.) In summary, the phrase types of the simple imperative language, which we will call *primitive* phrase types, are

$$\tau\ \mathbf{exp} \qquad \tau\ \mathbf{acc} \qquad \tau_1\tau_2\ \mathbf{var} \qquad \mathbf{comm} \qquad \mathbf{univ}$$

and the subtype relation is the least partial ordering such that

$\tau \leq \tau'$ implies $\tau\ \mathbf{exp} \leq \tau'\ \mathbf{exp}$

$\tau' \leq \tau$ implies $\tau\ \mathbf{acc} \leq \tau'\ \mathbf{acc}$

$\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$ implies $\tau_1\tau_2\ \mathbf{var} \leq \tau_1'\tau_2'\ \mathbf{var}$

$\tau_1\tau_2\ \mathbf{var} \leq \tau_1\ \mathbf{acc}$

$\tau_1\tau_2\ \mathbf{var} \leq \tau_2\ \mathbf{exp}$

$\mathbf{univ} \leq \theta$

An appropriate syntax is:

$$
\begin{aligned}
<\mathrm{comm}, \pi> ::= {} & \mathbf{skip} \mid <\tau\,\mathrm{acc}, \pi> := <\tau\,\mathrm{exp}, \pi> \\
& \mid <\mathrm{comm}, \pi> ; <\mathrm{comm}, \pi> \\
& \mid \mathbf{while}\ <\mathrm{Boolean\ exp}, \pi>\ \ \mathbf{do}\ <\mathrm{comm}, \pi> \\
<\theta, \pi> ::= {} & \mathbf{if}\ <\mathrm{Boolean\ exp}, \pi>\ \mathbf{then}\ <\theta, \pi>\ \mathbf{else}\ <\theta, \pi> \\
& \mid \mathbf{case}\ <\mathrm{integer\ exp}, \pi>\ \mathbf{of}\ (<\theta, \pi>, \ldots, <\theta, \pi>)
\end{aligned}
$$

In the last two lines, θ stands for any phrase type, including the nonprimitive types to be introduced later. The minimal typing property holds for these productions if, in the partial ordering of phrase types, every finite set with an upper bound has a least upper bound. In fact, the achievement of this property for primitive phrase types was the real goal of the arguments about acceptors, variables, and **univ** at the beginning of this section.

For mathematical simplicity, it is tempting to make the partial ordering of phrase types into a lattice by introducing a phrase type **ns** (nonsense), of which all phrase types are subtypes. However, although a nonsense type simplifies certain theoretical techniques, as in [19], it is not germane to the purposes of this paper.

A complete semantic definition of the simple imperative language is given in [19]; here we will only delineate the basic nature of such a definition by giving its domain equations. For each phrase type θ, there is a domain of meanings D_θ, and for each type assignment π, there is a domain of environments Env_π, which is the product $\prod_{\iota \,\in\, \mathrm{dom}(\pi)} D_{\pi(\iota)}$ of the domains for the type of each identifier in $\mathrm{dom}(\pi)$. Then for each phrase class $<\theta, \pi>$ there is a semantic function from phrases to environments to meanings, i.e. $\mu_{\theta,\pi} \in <\theta, \pi> \to (\mathrm{Env}_\pi \to D_\theta)$.

For direct semantics $D_{\mathbf{comm}}$ is a domain of state transitions, i.e. $S \to S_\perp$, where S is the set of states of the store (hereafter simply called states), and $S_\perp$ indicates the formation of a domain by adding an undefined element $\perp$ (denoting nontermination) to the set S. Similarly $D_{\tau\,\mathbf{exp}}$ is $S \to (V_\tau)_\perp$, where $V_{\mathbf{integer}}$ is the set of integers, $V_{\mathbf{real}}$ is the set of real numbers, and $V_{\mathbf{Boolean}}$ is the set $\{\mathbf{true},\ \mathbf{false}\}$.

There are two ways of treating variables. The more conventional is to say that, for each data type, a state has a component mapping an appropriate set of "L-values" (or "names" or "references" or "abstract addresses") into values of that data type, i.e.

$$S = (L_{\mathbf{integer}} \to V_{\mathbf{integer}}) \times (L_{\mathbf{real}} \to V_{\mathbf{real}}) \times (L_{\mathbf{Boolean}} \to V_{\mathbf{Boolean}}).$$

Then $D_{\tau\,\mathbf{var}}$ is $S \to (L_\tau)_\perp$.

A preferable approach, however, avoids any commitment to a notion such as L-values or references, and more clearly reveals the relationship among variables, acceptors, and expressions. One regards the meaning of an acceptor as a function mapping each value into the state transformation caused by assigning that value to the acceptor, so that $D_{\tau\,\mathbf{acc}} = V_\tau \to (S \to S_\perp)$. Then the meaning of a variable is a pair of functions describing its meanings in its dual roles of acceptor and expression, so that $D_{\tau_1\tau_2\,\mathbf{var}} = D_{\tau_1\,\mathbf{acc}} \times D_{\tau_2\,\mathbf{exp}}$. The implicit conversion functions from variables to acceptors and expressions are the projections from $D_{\tau_1\tau_2\,\mathbf{var}}$ to $D_{\tau_1\,\mathbf{acc}}$ and $D_{\tau_2\,\mathbf{exp}}$.

These two views of variables provide a nice example of the way in which formal definition can influence language design. As long as we do not impose any structure involving L-values or references upon states, there is no danger of defining anything, such as call by reference, that involves these concepts. On the other hand, the more abstract approach opens the door to features, such as doublets in Pop-2 [21] or implicit "references" in Gedanken [9], that define a variable by giving arbitrary procedures for accepting and producing values.

In fact, the more abstract treatment of variables makes no commitment at all to the structure of states; S is a parameter of the semantics that can sensibly stand for any set at all. To emphasize this generality, we make S an explicit argument of D_θ and Env_π, and regard the semantics of a phrase as a family of functions, indexed by S, from environments to meanings:

$$\text{If } P \in <\theta, \pi> \text{ then } \mu_{\theta,\pi}(P)(S) \in \ \mathrm{Env}_\pi(S) \to D_\theta(S),$$

where

$$\mathrm{Env}_\pi(S) = \prod_{\iota \in \mathrm{dom}(\pi)} D_{\pi(\iota)}(S),$$

$$D_{\mathbf{comm}}(S) = S \to S_\perp,$$

$$D_{\tau\,\mathbf{exp}}(S) = S \to (V_\tau)_\perp,$$

$$D_{\tau\,\mathbf{acc}}(S) = V_\tau \to D_{\mathbf{comm}}(S),$$

$$D_{\tau_1\tau_2\,\mathbf{var}}(S) = D_{\tau_1\,\mathbf{acc}}(S) \times D_{\tau_2\,\mathbf{exp}}(S).$$

However, although the semantics of a phrase is a family of environment-to-meaning functions, the members of this family must bear a close relationship to one another. Roughly speaking, whenever a state set S can be "expanded" into another state set S', the semantics of a phrase for S must be related to its semantics for S'.

To make the notion of expansion precise, we first introduce some useful notation:

Identity and composition of functions

We write I_S for the identity function on S, and $\cdot$ for functional composition in diagrammatic order (so that $(f \cdot g)(x) = g(f(x))$).

Strict extension

When $f \in S \to S'_\perp$, we write $f_{\perp\!\!\perp}$ for the $\perp$-preserving extension of f to $S_\perp \to S'_\perp$. When $f \in S \to S'$, we write $f_\perp$ for the $\perp$-preserving extension of f to $S_\perp \to S'_\perp$.

Identity and composition of state-transition functions

We write J_S for the identity injection from S to $S_\perp$. When $f, g \in S \to S_\perp$, we write $f * g$ for $f \cdot (g_{\perp\!\!\perp}) \in S \to S_\perp$.

Diagonalization

We write D_S for the continuous function from $S \to S \to S_\perp$ to $S \to S_\perp$ such that $D_S(h)(\sigma) = h(\sigma)(\sigma)$.

In the last definition (and later in this paper) we assume that $\to$ is right associative and that function (and procedure) application is left associative.

Then we define an *expansion* of S to S' to be a pair $\langle g, G\rangle$ of functions $g \in S' \to S, G \in (S \to S_\perp) \to (S' \to S'_\perp)$ such that

1. G is continuous and $\perp$-preserving.

2. $G(J_S) = J_{S'}$.

3. When $f_1, f_2 \in S \to S_\perp$, $G(f_1 * f_2) = G(f_1) * G(f_2)$.

4. When $f \in S \to S_\perp$, $g \cdot f = G(f) \cdot (g_\perp)$.

5. When $h \in S \to S \to S_\perp$, $G(D_S(h)) = D_{S'}(g \cdot h \cdot G)$.

Intuitively, g maps each state in S' into the member of S that is "embedded" within it, while G maps each state-transition function in $S \to S_\perp$ into the state-transition function in $S' \to S'_\perp$ that "simulates" it.

More precisely, an expansion of S to S' induces, for each phrase type θ, a function in $D_\theta(S) \to D_\theta(S')$ that maps meanings appropriate to S into meanings appropriate to S'. If we write $D_\theta(\langle g, G\rangle)$ for the function in $D_\theta(S) \to D_\theta(S')$ induced by $\langle g, G\rangle$, then

$$D_{\mathbf{comm}}(\langle g, G\rangle) = G,$$

$$D_{\tau\,\mathbf{exp}}(\langle g, G\rangle)(e \in S \to (V_\tau)_\perp) = g \cdot e,$$

$$D_{\tau\,\mathbf{acc}}(\langle g, G\rangle)(a \in V_\tau \to D_{\mathbf{comm}}(S)) = a \cdot G,$$

$$D_{\tau_1\tau_2\,\mathbf{var}}(\langle g, G\rangle)(\langle a, e\rangle) = \langle D_{\tau_1\,\mathbf{acc}}(\langle g, G\rangle)(a), D_{\tau_2\,\mathbf{exp}}(\langle g, G\rangle)(e)\rangle.$$

By pointwise extension, an expansion of S to S' induces, for each type assignment π, a function in $\mathrm{Env}_\pi(S) \to \mathrm{Env}_\pi(S')$ that maps environments appropriate to S into environments appropriate to S'. If we write $\mathrm{Env}_\pi(\langle g, G\rangle)$ for the function in $\mathrm{Env}_\pi(S) \to \mathrm{Env}_\pi(S')$ induced by $\langle g, G\rangle$, then

$$\mathrm{Env}_\pi(\langle g, G\rangle)(\eta \in \mathrm{Env}_\pi(S))(\iota) \;=\; D_{\pi(\iota)}(\langle g, G\rangle)(\eta(\iota)).$$

We can now state the fundamental relationship between the semantics of a phrase for different state sets: If P is a phrase in $<\theta, \pi>$ and $\langle g, G\rangle$ is an expansion of S to S', then

$$\mu_{\theta,\pi}(P)(S) \cdot D_\theta(\langle g, G\rangle) \;=\; \mathrm{Env}_\pi(\langle g, G\rangle) \cdot \mu_{\theta,\pi}(P)(S').$$

(In fact, properties (1) to (5) of expansions are sufficient to make this relationship hold for all phrases of the simple imperative language.)

As shown in [20], this development can be described succinctly in the language of category theory. State sets and expansions form a category Σ, with $\langle I_S, I_{S\to S_\perp}\rangle$ as the identity on S and $\langle g, G\rangle \cdot \langle g', G'\rangle = \langle g' \cdot g, G \cdot G'\rangle$ as composition. Then each D_θ and Env_π is a functor from Σ to the category *Dom* of domains and continuous functions, and the fundamental relationship given above is that $\mu_{\theta,\pi}(P)$ is a natural transformation from Env_π to D_θ.

5. Procedures and Their Declarations

To provide procedures, we introduce a binary operation $\to$ upon phrase types. A phrase of type $\theta_1 \to \theta_2$ denotes a procedure whose calls are phrases of type θ_2 containing an actual parameter of type θ_1. Multiple parameters will be treated by Currying, i.e.

$$P(A_1, \ldots, A_n) \quad \text{means} \quad P(A_1) \cdots (A_n)$$

and

$$\lambda(F_1 \colon \theta_1, \ldots, F_n \colon \theta_n).\, B \quad \text{means} \quad \lambda F_1 \colon \theta_1. \; \cdots \; \lambda F_n \colon \theta_n.\, B.$$

This way of desugaring multiple parameters is sufficiently well known that we will not formalize it.

Thus what would conventionally be called a proper procedure (or τ function procedure) accepting parameters of types $\theta_1, \ldots, \theta_n$ is regarded as a phrase of type $\theta_1 \to \cdots \to \theta_n \to$ **comm** (or $\theta_1 \to \cdots \to \theta_n \to \tau$ **exp**). Note that parameterless proper procedures are simply commands (as was recognized in Algol W [22], where an actual parameter of this type could be any command), and parameterless function procedures are simply expressions (which is a natural and pleasant consequence of call by name).

It is easy to see that if $\theta_2 \leq \theta_2'$, then $\theta_1 \to \theta_2 \leq \theta_1 \to \theta_2'$. Less obviously, if $\theta_1' \leq \theta_1$, then $\theta_1 \to \theta_2 \leq \theta_1' \to \theta_2$. For example, since an integer expression can appear in any context requiring a real expression, a proper procedure accepting a real expression can also accept any integer expression and is therefore meaningful in any context requiring a proper procedure accepting an integer expression. Thus **real exp** $\to$ **comm** $\leq$ **integer exp** $\to$ **comm**.

In summary, the set of phrase types is the smallest set containing the primitive phrase types and closed under the binary operation $\to$. Its subtype relation is the least partial ordering satisfying the properties given earlier plus

$$\theta_1' \leq \theta_1 \text{ and } \theta_2 \leq \theta_2' \text{ implies } \theta_1 \to \theta_2 \leq \theta_1' \to \theta_2'.$$

In brief, $\to$ is antimonotone in its first operand and monotone in its second operand.

A suitable syntax for application (procedure statements and function designators), abstraction (lambda expressions), and least fixed-points (recursion) is

$$<\theta_2, \pi> ::= <\theta_1 \to \theta_2, \pi>(<\theta_1, \pi>)$$

$$<\theta_1 \to \theta_2, \pi> ::= \lambda\iota \colon \theta_1. <\theta_2, [\pi \mid \iota : \theta_1]>$$

$$<\theta, \pi> ::= \mathbf{rec} <\theta \to \theta, \pi>$$

Here $[\pi \mid \iota : \theta_1]$ denotes the type assignment similar to π except that it maps ι into θ_1, i.e.

$$\mathrm{dom}([\pi \mid \iota : \theta_1]) = \mathrm{dom}(\pi) \cup \{\iota\},$$

$$[\pi \mid \iota : \theta_1](\iota) = \theta_1,$$

$$[\pi \mid \iota : \theta_1](\iota') = \pi(\iota') \text{ when } \iota' \neq \iota.$$

For later developments, it will be convenient to extend this notation by using the following abbreviations:

$$[\pi \mid \iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n] \;=\; [\cdots[\pi \mid \iota_1 : \theta_1]\cdots \mid \iota_n : \theta_n],$$
$$[\iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n] \;=\; [\epsilon \mid \iota_1 : \theta_1 \mid \cdots \mid \iota_n : \theta_n],$$

where ϵ is the type assignment with empty domain. Note that the latter form can be used to notate any type assignment explicitly.

The obvious approach to semantics is to take the meanings of phrases of type $\theta_1 \rightarrow \theta_2$ to be continuous functions from meanings of phrases of type θ_1 to meanings of phrases of type θ_2, i.e. $D_{\theta_1 \rightarrow \theta_2}(S) = D_{\theta_1}(S) \rightarrow D_{\theta_2}(S)$. However, when we consider variable declarations in the next section we will find that this approach conflicts with Principle 5.

Even in the absence of a definite semantics, meaning can be clarified by equivalences. We write $P \equiv_{\theta,\pi} Q$ to indicate that $\mu_{\theta,\pi}(P) = \mu_{\theta,\pi}(Q)$, i.e. that P and Q have the same meaning when regarded as phrases in $<\theta, \pi>$.

First we have the standard equivalences of the (typed) lambda calculus. If $P \in <\theta_2, [\pi \mid \iota{:}\, \theta_1]>$ and $Q \in <\theta_1, \pi>$, then

$$(\lambda\iota{:}\, \theta_1.\, P)(Q) \;\equiv_{\theta_2,\pi}\; P|_{\iota \rightarrow Q}, \qquad \text{(beta reduction)}$$

where $P|_{\iota \rightarrow Q}$ denotes the result of substituting Q for the free occurrences of ι in P, with appropriate renaming of bound identifiers in P. If $P \in <\theta_\iota \rightarrow \theta_2, \pi>$ and $\iota \notin \mathrm{dom}(\pi)$, then

$$\lambda\iota{:}\, \theta_1.\, (P(\iota)) \;\equiv_{\theta_1 \rightarrow \theta_2,\pi}\; P. \qquad \text{(eta reduction)}$$

Next, an obvious equivalence describes the fixed-point property of **rec**. If $P \in <\theta \rightarrow \theta, \pi>$, then

$$\mathbf{rec}\ P \;\equiv_{\theta,\pi}\; P(\mathbf{rec}\ P).$$

Finally, two equivalences relate procedures to the conditional construction. If $P \in <\text{Booleanexp}, \pi>$, $Q, R \in <\theta_1 \rightarrow \theta_2, \pi>$ and $S \in <\theta_1, \pi>$, then

$$(\mathbf{if}\ P\ \mathbf{then}\ Q\ \mathbf{else}\ R)(S) \;\equiv_{\theta_2,\pi}\; \mathbf{if}\ P\ \mathbf{then}\ Q(S)\ \mathbf{else}\ R(S).$$

If $P \in <\text{Booleanexp}, \pi>$, $Q, R \in <\theta_2, [\pi \mid \iota{:}\, \theta_1]>$, and $\iota \notin \mathrm{dom}(\pi)$, then

$$\lambda\iota{:}\, \theta_1.\, \mathbf{if}\ P\ \mathbf{then}\ Q\ \mathbf{else}\ R \;\equiv_{\theta_1 \rightarrow \theta_2,\pi}\; \mathbf{if}\ P\ \mathbf{then}\ \lambda\iota{:}\, \theta_1.\, Q\ \mathbf{else}\ \lambda\iota{:}\, \theta_1.\, R.$$

For the declaration of procedures, we prefer the **let** and **letrec** notation of Landin [3] to that of Algol 60; it is uniformly applicable to all phrase types (not just procedures), it distinguishes clearly between nonrecursive and recursive cases, and it doesn't make declarations look like commands. The syntax is

$$\begin{aligned}<\theta,\pi> ::= {} & \mathbf{let}\ \iota_1\ \mathbf{be}\ <\theta_1,\pi>\ \ \&\ \cdots\ \&\ \ \iota_n\ \mathbf{be}\ <\theta_n,\pi>\ \mathbf{in}\ <\theta,\pi'> \\ & \mid \mathbf{letrec}\ \iota_1:\theta_1\ \mathbf{be}\ <\theta_1,\pi'>\ \ \&\ \cdots\ \&\ \ \iota_n:\theta_n\ \mathbf{be}\ <\theta_n,\pi'>\ \mathbf{in}\ <\theta,\pi'>\end{aligned}$$

where $\pi' = [\pi \mid \iota_1\colon\theta_1 \mid \cdots \mid \iota_n\colon\theta_n]$. (Note that the types $\theta_1,\ldots,\theta_n$ must be stated explicitly for **letrec** but not **let**.)

This notation can be defined as syntactic sugar in terms of application, abstraction, and **rec**. The nonrecursive **let** construction is straightforward. If $P_1 \in <\theta_1,\pi>,\ldots,P_n \in <\theta_n,\pi>$, and $P \in <\theta,\pi'>$, then

$$\begin{aligned}&\mathbf{let}\ \iota_1\ \mathbf{be}\ P_1\ \ \&\ \cdots\ \&\ \ \iota_n\ \mathbf{be}\ P_n\ \mathbf{in}\ P\ \ \equiv_{\theta,\pi} \\ &\qquad (\lambda\iota_1\colon\theta_1.\ \cdots\ \lambda\iota_n\colon\theta_n.\ P)(P_1)\cdots(P_n)\end{aligned}$$

This equivalence can be used to remove all occurrences of **let** from a program without changing its meaning. Although it is formally similar to the equivalence given by Landin [3], it has a different import since call by name is being used instead of call by value. For example, if E is an integer expression, then **let** x **be** E **in** 3 has the same meaning as $(\lambda x\colon \mathbf{integer\ exp}.\,3)(E)$ which, by beta reduction, has the same meaning as 3, even when E is nonterminating. If x and y are integer variables, **let** z **be** x **in** $(x := 4; y := z)$ has the same meaning as $(\lambda z\colon \mathbf{integer\ integer\ var}.\,(x := 4; y := z))(x)$ which, by beta reduction, has the same meaning as $x := 4; y := x$.

To treat the recursive **letrec** construction, we will first define the nonmultiple case and then show how multiple declarations can be reduced to this case. For the nonmultiple case we follow Landin: If $P_1 \in <\theta_1,[\pi \mid \iota_1\colon\theta_1]>$ and $P \in <\theta,[\pi \mid \iota_1\colon\theta_1]>$, then

$$\mathbf{letrec}\ \iota_1\colon\theta_1\ \mathbf{be}\ P_1\ \mathbf{in}\ P \equiv_{\theta,\pi} (\lambda\iota_1\colon\theta_1.\ P)(\mathbf{rec}\ \lambda\iota_1\colon\theta_1.\ P_1).$$

For the multiple case we give an equivalence, suggested by F. J. Oles, that avoids the use of products of phrase types. If $P_1 \in <\theta_1,\pi'>$, ..., $P_n \in <\theta_n,\pi'>$, and $P \in <\theta,\pi'>$, where $\pi' = [\pi \mid \iota_1\colon\theta_1 \mid \cdots \mid \iota_n\colon\theta_n]$, then

$$\begin{aligned}&\mathbf{letrec}\ \iota_1:\theta_1\ \mathbf{be}\ P_1\,\&\ \cdots\ \&\,\iota_n:\theta_n\ \mathbf{be}\ P_n\ \mathbf{in}\ P\ \ \equiv_{\theta,\pi} \\ &\qquad \mathbf{letrec}\ \iota_1:\theta_1\ \mathbf{be} \\ &\qquad\qquad (\mathbf{letrec}\ \iota_2:\theta_2\ \mathbf{be}\ P_2\,\&\ \cdots\ \&\ \iota_n:\theta_n\ \mathbf{be}\ P_n\ \mathbf{in}\ P_1) \\ &\qquad \mathbf{in}\ (\mathbf{letrec}\ \iota_2:\theta_2\ \mathbf{be}\ P_2\,\&\ \cdots\ \&\ \iota_n:\theta_n\ \mathbf{be}\ P_n\ \mathbf{in}\ P).\end{aligned}$$

6. Variable Declarations

To declare variables, we use the syntax

$$<\mathrm{comm}, \pi> ::= \mathbf{new}\ \tau\ \mathbf{var}\ \iota\ \mathbf{in}\ <\mathrm{comm}, [\pi \mid \iota : \tau\tau\, \mathrm{var}]>$$

(Note that declared variables always accept and produce the same data type.) However, since this construction involves binding we want to desugar it into a form in which the binding is done by a lambda expression. The solution is to introduce the more basic construction

$$<\mathrm{comm}, \pi> ::= \mathbf{newvar}(\tau)<\tau\tau\mathrm{var} \to \mathrm{comm}, \pi>$$

and to define

$$\mathbf{new}\ \tau\ \mathbf{var}\ \iota\ \mathbf{in}\ P \ \equiv_{\mathrm{comm},\pi}\ \mathbf{newvar}(\tau)\lambda\iota{:}\, \tau\tau\ \mathbf{var}.\, P,$$

where $P \in <\mathrm{comm}, [\pi \mid \iota : \tau\tau\ \mathrm{var}]>$.

Semantically, variable declarations raise a serious problem. The conventional approach is to permit the set S of store states to contain states with different numbers of L-values, and to define variable declaration to be an operation that adds an L-value to the state. For example, one might take a state to be a collection of strings of values for each data type

$$S = V^*_{\mathbf{integer}} \times V^*_{\mathbf{real}} \times V^*_{\mathbf{Boolean}},$$

and define the declaration of a τ variable to be an operation that adds one more component to the string of values of type τ.

The problem with this view is that it violates Principle 5 by obscuring the stack discipline. Execution of a command containing variable declarations permanently alters the shape of the store, i.e. the number of L-values or the length of the component strings. In effect, the store is a heap without a garbage collector, rather than a stack. It is hardly surprising that this kind of model inspired languages that are closer to Lisp than to Algol.

Our solution to this difficulty takes advantage of the fact that the semantics of a phrase is a family of environment-to-meaning functions for different sets of states. Instead of using a single set containing states of different shapes and regarding variable declaration as changing the shape of a state, we use sets of states with the same shape and regard variable declaration as changing the set of states. Specifically, if C is $\mathbf{new}\ \tau\ \mathbf{var}\ \iota\ \mathbf{in}\ C'$, then the semantics of C for a state set S depends upon the semantics of C' for the state set $S \times V_\tau$. Thus, since the semantics of C for S maps an environment into a mapping in $D_{\mathbf{comm}}(S) = S \to S_\perp$, it is obvious that executing C will not change the shape of a state.

To make this precise, suppose $C' \in <\mathrm{comm}, [\pi \mid \iota{:}\, \tau\tau\ \mathrm{var}]>$, so that $C \in <\mathrm{comm}, \pi>$. We first note that S and $S \times V_\tau$ are related by the expansion $\langle g, G\rangle$ in which g is the function from $S \times V_\tau$ to S such that $g(\langle \sigma, v \rangle) = \sigma$ and G is the function from $S \to S_\perp$ to $(S \times V_\tau) \to (S \times V_\tau)_\perp$ such that

$$G(c)(\langle \sigma, v \rangle) = \mathbf{if}\ c(\sigma) = \perp\ \mathbf{then}\ \perp\ \mathbf{else}\ \langle c(\sigma), v \rangle.$$

This expansion induces a function $\mathrm{Env}_\pi(\langle g, G \rangle)$ from $\mathrm{Env}_\pi(S)$ to $\mathrm{Env}_\pi(S \times V_\tau)$.

Let e be the function from $S \times V_\tau$ to $(V_\tau)_\perp$ such that $e(\langle \sigma, v \rangle) = v$, and a be a function from V_τ to $(S \times V_\tau) \to (S \times V_\tau)_\perp$ such that $a(v')(\langle \sigma, v \rangle) = \langle \sigma, v' \rangle$. Then $\langle a, e \rangle \in D_{\tau\tau\,\mathbf{var}}(S \times V_\tau)$ is an appropriate meaning for the variable being declared.

To obtain the meaning of $\mathbf{new}\ \tau\ \mathbf{var}\ \iota\ \mathbf{in}\ C'$ for the state set S and an environment $\eta \in \mathrm{Env}_\pi(S)$, we use $\mathrm{Env}_\pi(\langle g, G \rangle)$ to map η into $\mathrm{Env}_\pi(S \times V_\tau)$ and then alter the resulting environment to map ι into $\langle a, e \rangle$, obtaining

$$\eta' = [\mathrm{Env}_\pi(\langle g, G \rangle)(\eta) \mid \iota : <a, e>].$$

Then we take the meaning of C' for the state set $S \times V_\tau$ and the environment η', and compose this meaning, which is a state-transition function from $S \times V_\tau$ to $(S \times V_\tau)_\perp$, with a function that initializes the new variable to some standard initial value $\mathrm{init}_\tau \in V_\tau$, and a function which forgets the final value of the variable:

$$\begin{aligned}&\mu_{\mathrm{comm},\pi}(\mathbf{new}\ \tau\ \mathbf{var}\ \iota\ \mathbf{in}\ C')(S)(\eta) \ = \\ &\qquad (\lambda\sigma.\, \langle \sigma, \mathrm{init}_\tau \rangle) \cdot \mu_{\mathrm{comm},[\pi|\iota:\tau\tau\ \mathrm{var}]}(C')(S \times V_\tau)(\eta') \cdot (g_\perp).\end{aligned}$$

(Our unAlgol-like use of a standard initialization is the simplest way to avoid the abyss of nondeterminate semantics.)

However, this approach to variable declaration has a radical effect on the notion of what procedures mean that forces us to abandon the conventional idea that $D_{\theta_1 \to \theta_2}(S) = D_{\theta_1}(S) \to D_{\theta_2}(S)$. The problem is that variable declarations may intervene between the point of definition of a procedure and its point of call, so that the state set S' relevant to the call is different than the state set S at the point of definition, though there must be an expansion from S to S'.

As a consequence, a member p of $D_{\theta_1 \to \theta_2}(S)$ must be a family of functions describing the meaning of a procedure for different S'. Moreover, each of these functions, in addition to accepting the usual argument in $D_{\theta_1}(S')$ must also accept an expansion from S to S' that shows how the states of S are embedded in the richer states of S'.

As one might expect, the members of the family p must satisfy a stringent relationship (which can be expressed by saying that p is an appropriate kind of natural transformation). A precise definition is the following (where $\mathrm{expand}(S, S')$ is the set of expansions from S to S'): $p \in D_{\theta_1 \to \theta_2}(S)$ if and only if p is a state-set-indexed family of functions,

$$p(S') \in \ \mathrm{expand}(S, S') \times D_{\theta_1}(S') \to D_{\theta_2}(S'),$$

such that, for all $\langle g, G\rangle \in \text{expand}(S, S')$, $\langle g', G'\rangle \in \text{expand}(S', S'')$, and $a \in D_{\theta_1}(S')$,

$$D_{\theta_2}(\langle g', G'\rangle)(p(S')(\langle g, G\rangle, a)) = p(S'')(\langle g' \cdot g, G \cdot G'\rangle, D_{\theta_1}(\langle g', G'\rangle)(a)).$$

To make $D_{\theta_1 \to \theta_2}(S)$ into a domain, its members are ordered pointwise, i.e. $p_1 \sqsubseteq p_2$ if and only if $(\forall S')\, p_1(S') \sqsubseteq p_2(S')$.

Finally, we must say how an expansion from S to S' induces a function from $D_{\theta_1 \to \theta_2}(S)$ to $D_{\theta_1 \to \theta_2}(S')$: If $\langle g, G\rangle \in \text{expand}(S, S')$ and $p \in D_{\theta_1 \to \theta_2}(S)$, then $D_{\theta_1 \to \theta_2}(\langle g, G\rangle)(p) \in D_{\theta_1 \to \theta_2}(S')$ is the family p' of functions such that, for all S'', $\langle g', G'\rangle \in \text{expand}(S', S'')$, and $a \in D_{\theta_1}(S'')$,

$$p'(S'')(\langle g', G'\rangle, a) = p(S'')(\langle g' \cdot g, G \cdot G'\rangle, a).$$

A full description of this kind of semantics is presented in [20]; in particular abstraction and application are defined and the validity of beta and eta reduction is proved. This is done by showing that the above definition of $\to$ makes Dom^{Σ} (the category of functors and natural transformations from Σ to Dom) into a Cartesian closed category, which is an extremely general model of the typed lambda calculus.

Despite its apparent complexity, much of which is due to our avoidance of category theory in this exposition, this kind of semantics shows that our language is obtained by adding the typed lambda calculus to the simple imperative language in a way that imposes a stack discipline. The essential idea is that the procedure mechanism involves a "hidden abstraction" over a family of semantics indexed by state sets.

We suspect that this kind of hidden abstraction may arise in other situations where a formal language is extended by adding a procedural or definitional mechanism based on the lambda calculus. The generality of the idea is indicated by the fact that the definition of $\to$ and the proof that Dom^{Σ} is Cartesian closed do not depend upon the nature of the category Σ.

7. Call by Value

In the Algol 60 report, call by value is explained in terms of call by name by saying that a value specification is equivalent to a certain modification of the procedure body. In fact, however, this modification involves only the body and not the formal parameter list, so that it is equally applicable to commands that are not procedure bodies. In essence, call by value is really an operation on commands rather than parameters.

To capture this idea, we introduce the syntax

$$\text{<comm}, [\pi \mid \iota : \tau \text{ exp}]\text{>} ::= \tau \textbf{ value } \iota \textbf{ in } \text{<comm}, [\pi \mid \iota : \tau\tau \text{ var}]\text{>}$$

which is desugared by the equivalence

$$\tau \textbf{ value } \iota \textbf{ in } C \ \equiv_{\mathrm{comm},[\pi|\iota:\tau\ \mathbf{exp}]}$$

$$\textbf{new } \tau \textbf{ var } \iota' \textbf{ in } (\iota' := \iota\, ; (\lambda\iota{:}\,\tau\tau \textbf{ var.}\, C)(\iota')),$$

where $C \in$ <comm, $[\pi \mid \iota{:}\,\tau\tau$ var]> and $\iota' \notin \mathrm{dom}(\pi) \cup \{\iota\}$. (This is only a generalization of call by value for proper procedures; an analogous generalization for function procedures would require block expressions.)

Notice that τ **value** ι **in** C has a peculiar binding structure: the first occurrence of ι is a binder whose scope is C, yet this occurrence is itself free. (A similar phenomenon occurs in the conventional notation for indefinite integration.)

Call by result, as in Algol W [22], can obviously be treated similarly.

8. Arrays

Arrays of the kind used in Algol 60 can be viewed as procedures whose calls are variables. Thus an n-dimensional τ array is a phrase of type

$$\underbrace{\textbf{integer exp} \rightarrow \cdots \rightarrow \textbf{integer exp}}_{n \text{ times}} \rightarrow \tau\tau \textbf{ var}.$$

(Notice that this is a phrase type. If arrays were introduced as a data type, one could assign to entire array variables (as in APL) but not to their elements.)

The declaration of such arrays is a straightforward generalization of variable declarations, and can be desugared similarly. The details are left to the reader.

Unfortunately, this kind of array, like that of Algol, has the shortcoming that it does not carry its own bounds information. A possible solution is to introduce, for each $n \geq 1$, a phrase type $\mathbf{array}(n, \tau)$ that is a subtype of the type displayed above, and to provide bound-extraction operations that act upon these new phrase types. The concept of array in [28] could be treated similarly.

9. Labels

Since all one can do with a label ι is to jump to it, its meaning can be taken to be the meaning of **goto** ι. Thus labels can be viewed as identifiers of phrase type **comm**, and **goto** ι can simply be written as ι.

However, as suggested in Algol 68, labels denote a special kind of command, which we will call a *completion*, that has the property that it never returns control to its successor. If completions are not distinguished as a separate phrase type, it becomes difficult for either a human reader or a compiler to analyze control flow, particularly when procedure parameters denoting completions are only specified to be commands. To avoid this, we introduce **compl**(etion) as an additional phrase type that is a subtype of **comm** (so that completions can always be used as commands but not vice-versa).

Thus labels are identifiers of phrase type **compl**. Moreover, the production schemas for conditional and case constructions, procedure application, and recursion provide a variety of compound phrases of type **compl**. This variety can be enriched by the following syntax, in which various ways of forming commands are used to form completions:

$$
\begin{aligned}
&<\text{compl}, \pi> ::= <\text{comm}, \pi> \,;\, <\text{compl}, \pi> \\
&\qquad \mid \; \textbf{new}\ \tau\ \textbf{var}\ \iota\ \textbf{in}\ <\text{compl}, [\pi \mid \iota : \tau\tau\ \text{var}]> \\
&\qquad \mid \; \textbf{newvar}(\tau)\ <\tau\tau\ \text{var} \rightarrow \text{compl}, \pi> \\
&<\text{compl}, [\pi \mid \iota : \tau\ \text{exp}]> ::= \tau\ \textbf{value}\ \iota\ \textbf{in}\ <\text{compl}, [\pi \mid \iota : \tau\tau\ \text{var}]>
\end{aligned}
$$

Two more schemas suffice to describe commands and completions in which labels are declared in an Algol-like notation:

$$<\text{comm}, \pi> ::= <\text{comm}, \pi'> \,;\, \iota_1\colon <\text{comm}, \pi'> \,;\, \cdots \,;\, \iota_n\colon <\text{comm}, \pi'>$$

where $\iota_1, \ldots, \iota_n$ are distinct and $\pi' = [\pi \mid \iota_1\colon \text{compl} \mid \cdots \mid \iota_n\colon \text{compl}]$;

$$<\text{compl}, \pi> ::= <\text{comm}, \pi'> \,;\, \iota_1\colon <\text{comm}, \pi'> \,;\, \cdots \,;\, \iota_n\colon <\text{compl}, \pi'>$$

where $\iota_1, \ldots, \iota_n$ are distinct and $\pi' = [\pi \mid \iota_1\colon \text{compl}\,;\, \cdots \mid \iota_n\colon \text{compl}]$. Since these declarative constructions involve binding, we must desugar them into more basic forms. For this purpose, we introduce an escape operation that is a parameterless variant of Landin's J-operator [3].

$$<\text{comm}, \pi> ::= \textbf{escape}\ <\text{compl} \rightarrow \text{comm}, \pi>.$$

This operation can be described in terms of a conventional label declaration: If $P \in\ <\text{compl} \rightarrow \text{comm}, \pi>$ and $\iota \notin \text{dom}(\pi)$, then

$$\textbf{escape}\ P\ \equiv_{\text{comm},\pi}\ (P(\iota)\,;\, \iota\colon \textbf{skip}).$$

Our present goal, however, is the reverse. To describe label declarations in terms of escapes, we proceed in two steps. First, we describe a label-declaring command in terms of a label-declaring completion by adding a final jump to an enclosing escape: If $\iota_1, \ldots, \iota_n, \iota$ are distinct identifiers, $\pi' = [\pi \mid \iota_1\colon \text{compl} \mid \cdots \mid \iota_n\colon \text{compl}]$, $C_0, \ldots, C_n \in\ <\text{comm}, \pi'>$, and $\iota \notin \text{dom}(\pi)$, then

$$C_0\,;\iota_1\colon C_1\,;\cdots;\iota_n\colon C_n \;\equiv_{\mathrm{comm},\pi}$$

$$\textbf{escape}\ \lambda\iota\colon \mathrm{compl.}\,(C_0\,;\iota_1\colon C_1\,;\cdots;\iota_n\colon(C_n\,;\iota)).$$

Then we describe a label-declaring completion in terms of recursive definitions: If $\iota_1,\ldots,\iota_n$ are distinct identifiers, $\pi' = [\pi \mid \iota_1\colon \mathrm{compl} \mid \cdots \mid \iota_n\colon \mathrm{compl}]$, $C_0,\ldots,C_{n-1} \in \langle\mathrm{comm},\pi'\rangle$, and $K \in \langle\mathrm{compl},\pi'\rangle$, then

$$\begin{aligned}
&C_0\,;\iota_1\colon C_1\,;\cdots;\iota_n\colon K \;\equiv_{\mathrm{compl},\pi}\\
&\qquad \textbf{letrec}\ \iota_1\colon \mathrm{compl}\ \textbf{be}\ (C_1\,;\iota_2)\ \&\cdots\&\ \iota_{n-1}\colon \mathrm{compl}\ \textbf{be}\ (C_{n-1}\,;\iota_n)\\
&\qquad\qquad \&\ \iota_n\colon \mathrm{compl}\ \textbf{be}\ K\\
&\qquad \textbf{in}\ (C_0\,;\iota_1).
\end{aligned}$$

We have chosen to desugar the Algol notation for declaring labels because of its familiarity. Other, possibly preferable, notations can be treated similarly; for example, Zahn's event facility [29] can be described by escapes without recursion. Actually, the wisest approach might be to avoid all syntactic sugar and simply provide escapes.

Semantically, the introduction of labels requires a change from direct to continuation semantics, which will not be discussed here. In [20] it is shown that hidden abstraction on state sets can be extended to continuation semantics, though with a different notion of expansion.

10. Products and Sums

Although procedures and arrays are the only ways of building compound phrase types in Algol, most newer languages provide some kind of product of types, such as records in Algol W or class members in Simula 67 [26], and often some kind of sum of types, such as unions in Algol 68 or variant records in Pascal. In this section we will explore the addition of such mechanisms to our illustrative language.

Since we distinguish two kinds of type, we must decide whether to have products of data types or phrase types (or both). Products of data types would be record-like entities, except that one would always assign to entire records rather than their components. (Complex numbers are a good example of a simple product of data types.) On the other hand, products of phrase types are more like members of Simula classes than like records; one can never assign to the entire object, but only to components that are variables; other types of components, such as procedures, are also possible. In this paper, we will only consider products (and sums) of phrase types, thereby retaining the Algol characteristic that data types are never compound.

We must also decide between numbered and named products, i.e. between selecting components by an ordinal or by an identifier (i.e. field name). In this paper we will explore named products, since they are more commonly used than numbered products, and also since they are amenable to a richer subtype relationship.

To introduce named products of phrase types, we expand the set of phrase types to include

$$\mathbf{prod}\ \pi,$$

where π is a type assignment. Usually we will write products in the form $\mathbf{prod}[\iota_1 \colon \theta_1 \mid \cdots \mid \iota_n \colon \theta_n]$, where $\iota_1, \ldots, \iota_n$ are distinct identifiers. However, it should be understood that the phrase type denoted by this expression is independent of the ordering of the pairs $\iota_k \colon \theta_k$.

For a subtype ordering, one at least wants a component-wise ordering. But a more interesting and useful structure arises if we permit implicit conversions that drop components, e.g.

$$\begin{aligned} &\mathbf{prod}[\mathit{age} \colon \mathbf{integer\ exp} \mid \mathit{sex} \colon \mathbf{Boolean\ exp} \mid \mathit{salary} \colon \mathbf{integer\ var}] \\ &\qquad \leq \mathbf{prod}[\mathit{age} \colon \mathbf{integer\ exp} \mid \mathit{salary} \colon \mathbf{integer\ var}] \end{aligned}$$

In general, we have

$$\begin{aligned} &\mathbf{prod}\ \pi \leq \mathbf{prod}\ \pi' \text{ if and only if} \\ &\qquad \mathrm{dom}(\pi') \subseteq \mathrm{dom}(\pi) \text{ and } (\forall \iota \in \mathrm{dom}(\pi'))\pi(\iota) \leq \pi'(\iota). \end{aligned}$$

Next we introduce the syntax of phrases for constructing products and selecting their components:

$$<\mathbf{prod}[\iota_1 \colon \theta_1 \mid \cdots \mid \iota_n \colon \theta_n], \pi> ::= \langle \iota_1 \colon <\theta_1, \pi>, \ldots, \iota_n \colon <\theta_n, \pi> \rangle$$

where $\iota_1, \ldots, \iota_n$ are distinct identifiers, and

$$<\theta, \pi> ::= <\mathbf{prod}[\iota \colon \theta], \pi>.\ \iota$$

In the second production, notice that our subtype ordering permits us to write $[\iota \colon \theta]$ instead of $[\cdots \mid \iota \colon \theta \mid \cdots]$.

The semantics of products is explicated by the following equivalences: When $P_1 \in <\theta_1, \pi>, \ldots, P_n \in <\theta_n, \pi>$, $\iota_1, \ldots, \iota_n$ are distinct, and $1 \leq k \leq n$,

$$\langle \iota_1 \colon P_1, \ldots, \iota_n \colon P_n \rangle.\ \iota_k \ \equiv_{\theta_k, \pi}\ P_k.$$

When $P \in$ <prod π', π> and $\mathrm{dom}(\pi') = \{\iota_1, \ldots, \iota_n\}$,

$$\langle \iota_1 \colon (P.\, \iota_1), \ldots, \iota_n \colon (P.\, \iota_n) \rangle \ \equiv_{\mathrm{prod}\ \pi', \pi} \ P$$

We have mentioned that this kind of product is closely related to the class concept of Simula 67. In [25] it is shown that class declarations (in the reference-free sense of Hoare [27] rather than of Simula itself) can be desugared into constructions using such products.

Finally, we introduce named sums of phrase types. (Roughly speaking, type sums are disjoint unions.) We expand the set of phrase types to include

$$\mathbf{sum}\ \pi,$$

where π is a type assignment. The subtype relation is

$$\mathbf{sum}\ \pi \leq \mathbf{sum}\ \pi' \ \text{ if and only if}$$
$$\mathrm{dom}(\pi) \subseteq \mathrm{dom}(\pi') \ \text{and} \ \ (\forall \iota \in \mathrm{dom}(\pi))\ \pi(\iota) \leq \pi'(\iota).$$

In contrast to the situation with products, a subtype of a sum can contain fewer (rather than more) alternatives. To construct sums and to do case analysis, we introduce the syntax

$$<\!\mathrm{sum}[\iota : \theta], \pi\!> ::= \mathbf{tag}\ \iota \colon <\!\theta, \pi\!>$$
$$<\!\theta, \pi\!> ::= \mathbf{sumcase}\ \iota\ \mathbf{is}\ <\!\mathrm{sum}[\iota_1 \colon \theta_1 \mid \cdots \mid \iota_n \colon \theta_n], \pi\!>$$
$$\mathbf{in}\ (\iota_1 \colon <\!\theta, [\pi \mid \iota \colon \theta_1]\!>, \ldots, \iota_n \colon <\!\theta, [\pi \mid \iota \colon \theta_n]\!>)$$

where $\iota_1, \ldots, \iota_n$ are distinct identifiers.

Again, the semantics can be explicated by equivalences. When $\iota_1, \ldots, \iota_n$ are distinct, $1 \leq k \leq n$, $P_1 \in <\!\theta, [\pi \mid \iota \colon \theta_1]\!>, \ldots, P_n \in <\!\theta, [\pi \mid \iota \colon \theta_n]\!>$, and $A \in <\!\theta_k, \pi\!>$,

$$\mathbf{sumcase}\ \iota\ \mathbf{is}\ \mathbf{tag}\ \iota_k \colon A\ \mathbf{in}\ (\iota_1 \colon P_1, \ldots, \iota_n \colon P_n) \ \equiv_{\theta, \pi} \ \mathbf{let}\ \iota\ \mathbf{be}\ A\ \mathbf{in}\ P_k.$$

When $S \in$ <sum π', π> and $\mathrm{dom}(\pi') = \{\iota_1, \ldots, \iota_n\}$,

$$\mathbf{sumcase}\ \iota\ \mathbf{is}\ S\ \mathbf{in}\ (\iota_1 \colon \mathbf{tag}\ \iota_1 \colon \iota, \ldots, \iota_n \colon \mathbf{tag}\ \iota_n \colon \iota) \ \equiv_{\mathrm{sum}\ \pi', \pi} \ S.$$

Since **sumcase** is a binding operation, Principle 1 requires us to express it in terms of a construction in which the binding is done by lambda expressions. For this purpose, we introduce the idea of "source-tupling." Suppose $P_1, \ldots, P_n$ are procedures of phrase types $\theta_1 \to \theta, \ldots, \theta_n \to \theta$, respectively. Then $\mathbf{sourcetuple}(\iota_1 \colon P_1, \ldots, \iota_n \colon P_n)$ is a procedure of type $\mathbf{sum}[\iota_1 \colon \theta_1 \mid \cdots \mid \iota_n \colon \theta_n] \to \theta$ that will behave like P_k when applied to a parameter tagged with ι_k.

To make this precise we use the syntax

$$<\mathrm{sum}[\iota_1{:}\, \theta_1 \mid \cdots \mid \iota_n{:}\, \theta_n] \rightarrow \theta, \pi> ::= \\ \quad \mathbf{sourcetuple}(\iota_1{:} <\theta_1 \rightarrow \theta, \pi>, \ldots, \iota_n{:} <\theta_n \rightarrow \theta, \pi>)$$

where $\iota_1, \ldots, \iota_n$ are distinct identifiers.

Then **sumcase** is desugared by the following equivalence: If $\iota_1, \ldots, \iota_n$ are distinct, $S \in <\mathrm{sum}[\iota_1{:}\, \theta_1 \mid \cdots \mid \iota_n{:}\, \theta_n], \pi>$, $P_k \in <\theta, [\pi \mid \iota{:}\, \theta_k]>$ for $1 \leq k \leq n$, then

$$\mathbf{sumcase}\ \iota\ \mathbf{is}\ S\ \mathbf{in}\ (\iota_1{:}\, P_1, \ldots, \iota_n{:}\, P_n) \ \equiv_{\theta,\pi}$$

$$\mathbf{sourcetuple}(\iota_1{:}\, \lambda\iota{:}\, \theta_1.\, P_1, \ldots, \iota_n{:}\, \lambda\iota{:}\, \theta_n.\, P_n)(S).$$

It should be noted that sums of phrase types do not introduce any failure of typing such as the "mutant variable record problem" of Pascal, since one cannot change the tag of a sum by assignment. On the other hand, sums of data types would also avoid these problems since a branch on the tag of a value would not imply any assumption that a variable with that value would continue to possess the same tag. This suggests that the type-safety problem with sum-like constructions is due to a failure to distinguish data and phrase types.

11. Final Remarks

I have neglected the topic of program proving since I have discussed it elsewhere at length. Although Hoare's work on proving procedures is incompatible with call by name and procedure parameters, an alternative approach called specification logic appears promising. In [23] this logic is formulated for a subset of Algol W; in [24] it is given for a language closer to that described here.

Like Algol itself, our illustrative language raises problems of interference, i.e. variable aliasing and interfering side effects of statements and proper procedures. The language is rich enough that an assertion that two phrases do not interfere must be proved (as in specification logic) rather than derived syntactically. Several years ago in [25], I attempted to restrict a language like that described here to permit interference to be detected syntactically. Unfortunately, this work led to some nasty syntactic complications (described at the end of [25]) that have yet to be resolved. Still, I have hopes for the future of this approach.

Although this paper has dealt with nearly all the significant aspects of Algol 60, it has not gone much beyond the scope of that language. More for lack of understanding than space, I have avoided block expressions, user-defined types, polymorphic procedures, recursively defined types, indeterminate and concurrent computation, references, and compound data types.

It remains to be seen whether our model can be extended to cover these topics. Of course, some of them could reasonably be labelled unAlgol-like. But the essence of Algol is not a straitjacket. It is a conceptual universe for language design that one hopes will encompass languages far more general than its progenitor.

References

[1] P. Naur et al., Revised report on the algorithmic language Algol 60, *Comm. ACM* 6 (1) (January 1963) 1–17.

[2] P. J. Landin, A λ-calculus approach, in: L. Fox (Ed.), *Advances in Programming and Non-Numerical Computation* (Pergamon Press, Oxford, 1966) pp. 97–141.

[3] P. J. Landin, A correspondence between Algol 60 and Church's lambda-notation, *Comm. ACM* 8 (2,3) (February–March 1965) 89–101 and 158–165.

[4] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, *Comm. ACM* 3 (4) (April 1960) 184–195.

[5] P. Lucas, P. Lauer and H. Stigleitner, *Method and Notation for the Formal Definition of Programming Languages*, TR 25.087, IBM Laboratory Vienna. (June 1968).

[6] D. Scott and C. Strachey, Toward a mathematical semantics for computer languages, *Proc. Symposium on Computers and Automata*, Polytechnic Inst. of Brooklyn, Vol. 21, pp. 19–46. (Also: Technical Monograph PRG-6, Oxford Univ. Computing Lab., Programming Research Group (1971).)

[7] P. J. Landin, The next 700 programming languages, *Comm. ACM* 9 (3) (March 1966) 157–166.

[8] A. Evans, Pal—A language designed for teaching programming linguistics, *Proc. ACM 23rd Natl. Conf.*, 1968 (Brandin Systems Press, Princeton, NJ) pp. 395–403.

[9] J. C. Reynolds, Gedanken—A simple typeless language based on the principle of completeness and the reference concept, *Comm. ACM* 13 (5) (May 1970) 308–319.

[10] A. van Wijngaarden (ed.) et al., Revised report on the algorithmic language Algol 68, *Acta Informatica* 5 (1975) 1–236.

[11] C. A. R. Hoare, Recursive data structures, *Int. J. Comput. Information Sci.* 4 (2) (June 1975) 105–132.

[12] N. Wirth, The programming language Pascal, *Acta Informatica* 1 (1) (1971) 35–63.

[13] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell and G. J. Popek, Proof rules for the programming language Euclid, *Acta Informatica* 10 (1) (1978) 1–26.

[14] C. M. Geschke, J. H. Morris, and E. H. Satterthwaite, Early experience with Mesa, *Comm. ACM* 20 (8) (August 1977) 540–553.

[15] J. D. Ichbiah, J. C. Heliard, O. Roubine, J. G. P. Barnes, B. Krieg-Brueckner and B. A. Wichmann, Preliminary Ada reference manual and Rationale for the design of the Ada programming language, *SIGPLAN Notices* 14 (6) (June 1979).

[16] C. A. R. Hoare, An axiomatic basis for computer programming, *Comm. ACM* 12 (10) (October 1969) 576–580 and 583.

[17] C. A. R. Hoare and N. Wirth, An axiomatic definition of the programming language Pascal, *Acta Informatica* 2 (4) (1973) 335–355.

[18] C. A. R. Hoare, Procedures and parameters: An axiomatic approach, in E. Engeler (Ed.), *Symposium on Semantics of Algorithmic Languages*, Lecture Notes in Mathematics, Vol. 188 (Springer-Verlag, Berlin, 1971) pp. 102–116.

[19] J. C. Reynolds, Using category theory to design implicit conversions and generic operators, in: N. D. Jones (Ed.), *Semantics-Directed Compiler Generation*, Proceedings of a Workshop, Aarhus, Denmark, January 14–18, 1980, Lecture Notes in Computer Science, Vol. 94 (Springer-Verlag, Berlin, 1980) pp. 211–258.

[20] F. J. Oles, *A Category-Theoretic Approach to the Semantics of Programming Languages*, Ph.D. Dissertation (completed August 1982), Syracuse University.

[21] R. M. Burstall, J. S. Collins and R. J. Popplestone, *Programming in Pop-2* (Edinburgh University Press, 1971).

[22] N. Wirth and C. A. R. Hoare, A contribution to the development of Algol, *Comm. ACM* 9 (6) (June 1966) 413–432.

[23] J. C. Reynolds, *The Craft of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1981).

[24] J. C. Reynolds, *Idealized Algol and its Specification Logic*, Computer and Information Science, Syracuse University, Report 1–81 (July 1981).

[25] J. C. Reynolds, Syntactic control of interference, *Conference Record of the Fifth Annual Symposium on Principles of Programming Languages*, Tucson, January 1978 (Association for Computing Machinery, New York, 1978) pp. 39–46.

[26] O.-J. Dahl, B. Myhrhaug and K. Nygaard, *Simula 67 Common Base Language*, Norwegian Computing Centre, Oslo (1968).

[27] C. A. R. Hoare, Proof of correctness of data representations, *Acta Informatica* 1 (4) (1972) 271–281.

[28] E. W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[29] C. T. Zahn, A control statement for natural top-down structured programming, *Proceedings, Programming Symposium*, Paris, April 9–11, 1974, Lecture Notes in Computer Science, Vol. 19 (Springer-Verlag, Berlin, 1974) pp. 170–180.