Preliminary Design of the Programming Language Forsythe

John C. Reynolds[1]

Carnegie Mellon University

June 21, 1988
CMU-CS-88-159

Abstract

This is a preliminary description of the programming language Forsythe, which is a descendent of Algol 60 intended to be as uniform and general as possible, while retaining the basic character of its progenitor.

---

[1]Research supported by NSF Grant CCR-8620191. A portion of the research was also sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB, Ohio. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any agency of the US Government.

The programming language Forsythe is a descendent of Algol 60 that is intended to be as uniform and general as possible, while retaining the basic character of its progenitor. The language is named after George E. Forsythe, founding chairman of the Computer Science Department at Stanford University. Among his many accomplishments, he played a major role in familiarizing American computer scientists (including the author) with Algol.

It must be emphasized that this is a preliminary description of Forsythe. As discussed in the final section of this report, there are likely to be significant changes in the language before it is implemented.

1. From Algol to Forsythe: An Evolution of Types

The long evolution which has led from Algol 60 to Forsythe is too complex to recount here in full detail. However, as an introduction, it is useful to outline the development of the heart of the language, which is its type structure. (In this introductory account, we retain the familiar notations of Algol 60, rather than using the novel notations of Forsythe.)

An essential characteristic of an Algol-like language is that the variety of entities that can be the value of a variable or expression is different from the variety of entities that can be the meaning of identifiers or phrases. We capture this characteristic by distinguishing two kinds of type (as in [7]):

- A *data type* denotes a set of values appropriate to a variable or expression.
- A *phrase type*, or more simply a *type*, denotes a set of meanings appropriate to an identifier or phrase.

In Algol 60, there are three data types: **integer**, **real**, and **boolean**. In Forsythe, we use more succinct names, **int**, **real**, and **bool**, and add a fourth data type, **char**, denoting the set of machine-representable characters.

To capture the existence of an implicit conversion from integers to reals, we define a partial order on data types called the *subtype* relation. We write $\delta \leq \delta'$, and say that δ is a *subtype* of δ' when either $\delta = \delta'$ or $\delta = \textbf{int}$ and $\delta' = \textbf{real}$, i.e.

In Algol 60, the phrase types are the entities, such as **integer**, **real array**, and **procedure**, that are used to specify procedure parameters. However, the phrase types of Algol

60 are not sufficiently refined to permit a compiler to detect all type errors. For example, in both

$$\textbf{procedure } \textit{silly}(x);\ \textbf{integer } x;\ y := x + 1$$

and

$$\textbf{procedure } \textit{strange}(x);\ \textbf{integer } x;\ x := x + 1$$

the formal parameter x is given the type **integer**, despite the fact that an actual parameter for ***silly*** can be any integer expression, since x is evaluated but never assigned to, while an actual parameter for ***strange*** must be an integer variable, since x is assigned to as well as evaluated.

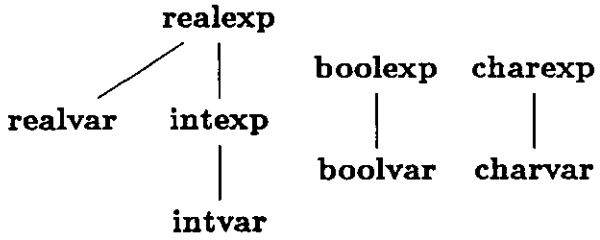
To remedy this defect, one must distinguish the phrase types **int**(eger) **exp**(ression) and **int**(eger) **var**(iable) (and similarly for the other data types), writing

$$\textbf{procedure } \textit{silly}(x);\ \textbf{intexp } x;\ y := x + 1$$

and

$$\textbf{procedure } \textit{strange}(x);\ \textbf{intvar } x;\ x := x + 1\ .$$

Like data types, phrase types possess a subtype relationship. Semantically, $\theta \leq \theta'$ means that there is an implicit conversion from meanings of type θ to meanings of type θ'. But the subtype relation can also be interpreted syntactically: $\theta \leq \theta'$ means that a phrase of type θ can be used in any context requiring a phrase of type θ'. Thus, since a variable can be used as an expression, **intvar** $\leq$ **intexp**, and similarly for the other data types. Moreover, since an integer expression can be used as a real expression, **intexp** $\leq$ **realexp**. In summary:



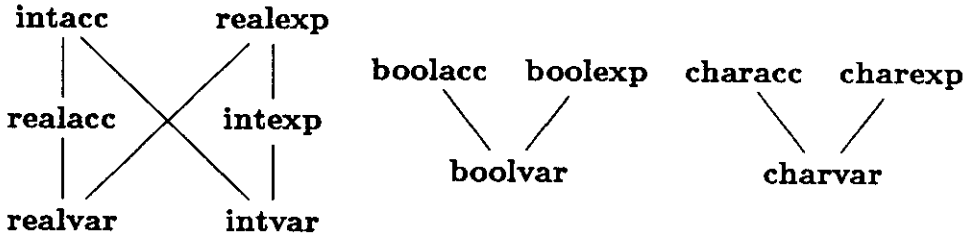
However, there is an unpleasant asymmetry here, which can be remedied by distinguishing, in addition to expressions, which can be evaluated but not assigned to, *acceptors*, which can be assigned to but not evaluated. Then, for example, we can write

$$\textbf{procedure } \textit{peculiar}(x);\ \textbf{intacc } x;\ x := 0$$

to indicate that ***peculiar*** assigns to its parameter but never evaluates it.

Clearly, **intvar** $\leq$ **intacc**, and similarly for the other data types. Moreover, **realacc** $\leq$ **intacc**, since an acceptor that can accept any real number can accept any integer. Thus

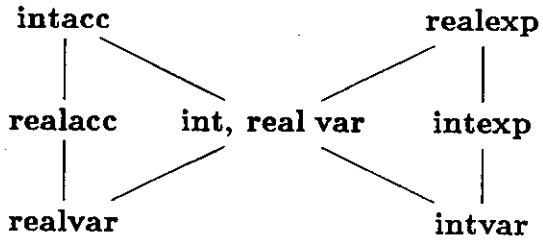
the subtype relation is



However, there is a further problem. In Forsythe, the conditional construction is generalized from expressions and commands to arbitrary phrase types; in particular one can construct conditional variables. Thus if p is a boolean expression, n is an integer variable, and x is a real variable, one can write

if p **then** n **else** x

on either side of an assignment command. But when this construction occurs on the right of an assignment, it must be regarded as a real expression, since p might be false, while when it occurs on the left of an assignment, it must be regarded as an integer acceptor, since p might be true. Thus the construction is an **int**(eger accepting), **real** (producing) **var**(iable), which fits into the subtype relation as follows:



Next, we consider the types of procedures. In Algol 60, when a parameter is a procedure, one simply specifies **procedure** (for a proper procedure), **integer procedure**, **real procedure**, or **boolean procedure**. But to obtain full compile-time typechecking, one must use more refined phrase types that indicate the number and type of parameters, e.g.

procedure(intexp, intvar)

to denote a proper procedure accepting an integer expression and an integer variable, or

real procedure(realexp)

to denote a real procedure accepting a real expression. (Note that this refinement introduces an infinite number of phrase types.)

These constructions can be simplified and generalized by introducing a binary type constructor $\rightarrow$ such that $\theta \rightarrow \theta'$ denotes the type of procedures that accept θ and produce

θ' or, more precisely, the type of procedures that accept a single parameter of type θ and whose calls are phrases of type θ'. For example, a real procedure accepting a real expression would have type **realexp** $\rightarrow$ **realexp**.

To describe proper procedures similarly, it is necessary to introduce the type **comm** to describe phrases that are commands (or in Algol jargon, statements). Then a proper procedure accepting an integer variable would have type **intvar** $\rightarrow$ **comm**.

To extend this formalism to procedures with several parameters, one might introduce a type constructor for products and regard, say, **procedure(intexp, intvar)** as (**intexp** $\times$ **intvar**) $\rightarrow$ **comm**. However, as we will see below, the product construction in Forsythe describes objects whose fields are selected by names rather than position. Thus multiple-parameter procedures are obtained by Currying rather than by the use of products.

For example, **procedure(intexp, intvar)** becomes **intexp** $\rightarrow$ (**intvar** $\rightarrow$ **comm**) or, more simply, **intexp** $\rightarrow$ **intvar** $\rightarrow$ **comm**, since $\rightarrow$ is right associative. In other words, a proper procedure accepting an integer expression and an integer variable is really a procedure accepting an integer expression whose calls are procedures accepting an integer variable whose calls are commands. Thus the call $p(a_1, a_2)$ is written $(p(a_1))(a_2)$ or, more simply, $p(a_1)(a_2)$, since procedure application is left associative. (In fact, if the parameters are identifiers or constants, one can simply write $p\, a_1\, a_2$).

In general, the type

$$\textbf{procedure}(\theta_1, \ldots, \theta_n)$$

becomes

$$\theta_1 \rightarrow \cdots \rightarrow \theta_n \rightarrow \textbf{comm}\,,$$

and, for each data type δ, the type

$$\delta\ \textbf{procedure}(\theta_1, \ldots, \theta_n)$$

becomes

$$\theta_1 \rightarrow \cdots \rightarrow \theta_n \rightarrow \delta\textbf{exp}\,.$$

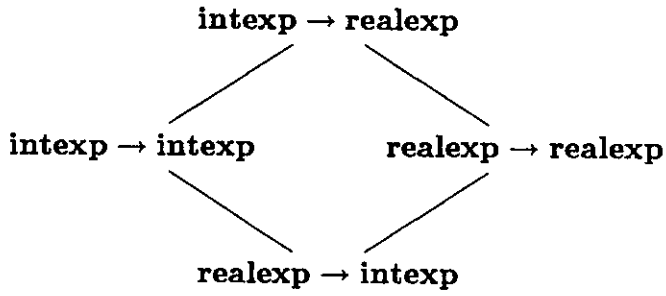
Moreover, this generalization includes the special case where $n = 0$, so that parameterless proper procedures are simply commands and parameterless function procedures are simply expressions. (Note that this simplification is permissible for call by name, but would not be for call by value, where parameterless procedures are needed - as in LISP - to postpone evaluation.)

To determine the subtype relation for procedural types, suppose $\theta_1' \leq \theta_1$ and $\theta_2 \leq \theta_2'$. Then a procedure of type $\theta_1 \rightarrow \theta_2$ can accept a parameter of type θ_1' (since this parameter can be converted to type θ_1) and its call can be converted from θ_2 to θ_2', so that the procedure also has type $\theta_1' \rightarrow \theta_2'$. Thus

$$\text{If } \theta_1' \leq \theta_1 \text{ and } \theta_2 \leq \theta_2' \text{ then } \theta_1 \rightarrow \theta_2 \leq \theta_1' \rightarrow \theta_2'\,,$$

i.e. $\rightarrow$ is antimonotone in its first operand and monotone in its second operand.

Thus, for example, since **intexp** $\leq$ **realexp**, we have



We have already seen that **comm**(and) must be introduced as a primitive phrase type. It is also useful to introduce a subtype of **comm** called **compl**(etion):

Essentially, a completion is a special type of command, such as a **goto** command, that never returns control.

The advantage of distinguishing completions is that control structure can be made more evident. For example, in

procedure *sqroot*$(x, y, error)$; **intexp** x; **intvar** y; **compl** *error*;
begin if $x < 0$ **then** *error*; C **end** ,

specifying *error* to be a completion makes it evident that C will never be executed when $x < 0$.

As mentioned earlier, Forsythe has a type constructor for named products. The essential idea is that the phrase type

$$(\iota_1 \colon \theta_1, \ldots, \iota_n \colon \theta_n)$$

is possessed by objects with fields named by the distinct identifiers $\iota_1, \ldots, \iota_n$, in which the field named ι_k has type θ_k. Note that the meaning of this phrase type is independent of the order of the $\iota_k \colon \theta_k$ pairs. We use the term "object" rather than "record" since fields need not be variables. For example, one could have a field of type **intvar** $\rightarrow$ **comm** which could be called as a proper procedure, but not assigned to. (Roughly speaking, objects are more like class members in Simula 67 than like records in Algol W.)

Clearly, the product constructor should be monotone:

If $n \geq 0$ and $\theta_1 \leq \theta'_1$ and ... and $\theta_n \leq \theta'_n$ then

$$(\iota_1 \colon \theta_1, \ldots, \iota_n \colon \theta_n) \leq (\iota_1 \colon \theta'_1, \ldots, \iota_n \colon \theta'_n) \, .$$

In fact, a richer subtype relationship is desirable, in which objects can be converted by "forgetting" fields, so that an object can be used in a context requiring a subset of its fields. This relationship (often called "multiple inheritance" [1]) is expressed by

$$\text{If } n \geq m \geq 0 \text{ and } \theta_1 \leq \theta'_1 \text{ and } \ldots \text{ and } \theta_m \leq \theta'_m \text{ then}$$

$$(\iota_1\colon \theta_1, \ldots, \iota_n\colon \theta_n) \leq (\iota_1\colon \theta'_1, \ldots, \iota_m\colon \theta'_m)\,.$$

At this point, we have summarized the type structure of Forsythe (then called "Idealized Algol") as it appeared in about 1981 [8]. Since then, the language has been generalized, and considerably simplified, by the introduction of conjunctive types [3].

The basic idea is to introduce a type constructor &, with the interpretation that a phrase has type $\theta_1 \,\&\, \theta_2$ if and only if it has both type θ_1 and type θ_2. This interpretation leads to the subtype laws

$$\theta_1 \,\&\, \theta_2 \leq \theta_1$$

$$\theta_1 \,\&\, \theta_2 \leq \theta_2$$

$$\text{If } \theta \leq \theta_1 \text{ and } \theta \leq \theta_2 \text{ then } \theta \leq \theta_1 \,\&\, \theta_2\,,$$

which assert that $\theta_1 \,\&\, \theta_2$ is a greatest lower bound of θ_1 and θ_2. (Note that the introduction of conjunction makes the subtype relation a preorder rather than a partial order, since one can have distinct types, such as $\theta_1 \,\&\, \theta_2$ and $\theta_2 \,\&\, \theta_1$, each of which is a subtype of the other. In this situation, we will say that the types are *equivalent*.)

We will see that conjunctive types provide the ability to define procedures with more than one type. For example

$$\textbf{procedure } \mathit{poly}(x);\ x \times x + 2$$

can be given the type (**intexp** → **intexp**) & (**realexp** → **realexp**). At present, however, the main point is that conjunction can be used to simplify the structure of types.

First, the various types of variables can be regarded as conjunctions of expressions and acceptors. For example, **intvar** is **intexp** & **intacc**, **realvar** is **realexp** & **realacc**, and **int, real var** is **realexp** & **intacc**.

Second, a product type with more than one field can be regarded as a conjunction of product types with single fields. Thus, instead of

$$(\iota_1\colon \theta_1, \ldots, \iota_n\colon \theta_n)\,,$$

one writes

$$\iota_1\colon \theta_1 \,\&\, \cdots \,\&\, \iota_n\colon \theta_n\,.$$

Note that the multiple inheritance relationship becomes a consequence of $\theta_1 \,\&\, \theta_2 \leq \theta_1$.

A final simplification concerns acceptors. The meaning of a δ acceptor a (for any data type δ) is completely determined by the meanings of the commands $a := e$ for all δ expressions e. Thus a has the same kind of meaning as a procedure of type δ**exp** $\rightarrow$ **comm**. As a consequence, we can regard δ**acc** as an abbreviation for δ**exp** $\rightarrow$ **comm**, and $a := e$ as an abbreviation for $a(e)$.

2. Types and the Subtype Relation

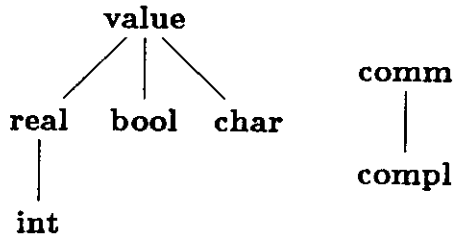
Having sketched its evolution, we can now define the type system of Forsythe precisely. The sets of data types, primitive (phrase) types, and (phrase) types can be defined by an abstract grammar:

$\delta ::=$ **int** | **real** | **bool** | **char** | **value**	(data types)
$\rho ::= \delta$ | **comm** | **compl**	(primitive types)
$\theta ::= \rho \mid \theta \rightarrow \theta \mid \iota{:}\theta \mid$ **ns** $\mid \theta \,\&\, \theta$	(types)

where the metavariable ι ranges over identifiers.

Here there are three changes from the previous section. Expression types are now named by their underlying data types; for example, **intexp** is now just **int**. A new data type **value** denotes the union of all other data types; its utility will become apparent later. Finally, a new phrase type **ns** (for "nonsense") has been introduced; it is possessed by all (parsable) phrases of the language, and can be viewed as the conjunction of the empty set of types.

The subtype relation $\leq_{\text{prim}}$ for primitive types is the partial order



For types, $\leq$ is the least preorder such that

$$\theta \leq \textbf{ns}$$

$$\theta_1 \,\&\, \theta_2 \leq \theta_1$$

$$\theta_1 \,\&\, \theta_2 \leq \theta_2$$

$$\text{If } \theta \leq \theta_1 \text{ and } \theta \leq \theta_2 \text{ then } \theta \leq \theta_1 \,\&\, \theta_2$$

$$\text{If } \rho \leq_{\text{prim}} \rho' \text{ then } \rho \leq \rho'$$

$$\text{If } \theta \leq \theta' \text{ then } \iota\!:\theta \leq \iota\!:\theta'$$

$$\text{If } \theta_1' \leq \theta_1 \text{ and } \theta_2 \leq \theta_2' \text{ then } \theta_1 \to \theta_2 \leq \theta_1' \to \theta_2'$$

$$\iota\!:\theta_1 \,\&\, \iota\!:\theta_2 \leq \iota\!:(\theta_1 \,\&\, \theta_2)$$

$$(\theta \to \theta_1) \,\&\, (\theta \to \theta_2) \leq \theta \to (\theta_1 \,\&\, \theta_2)$$

$$\mathbf{ns} \leq \iota\!:\mathbf{ns}$$

$$\mathbf{ns} \leq \theta \to \mathbf{ns} \ .$$

The first four relationships establish that **ns** is a greatest type and that $\theta_1 \,\&\, \theta_2$ is a greatest lower bound of θ_1 and θ_2. Note that we say "a" rather than "the"; since we have a preorder rather than a partial order, neither greatest types nor greatest lower bounds are unique. However, any greatest type must be equivalent to **ns**, and any greatest lower bound of θ_1 and θ_2 must be equivalent to $\theta_1 \,\&\, \theta_2$. We write $\theta \simeq \theta'$ when θ and θ' are equivalent, i.e. when $\theta \leq \theta'$ and $\theta' \leq \theta$.

The fact that **ns** is a greatest type and & is a greatest lower bound operator has the following consequences:

$$\theta_1 \,\&\, (\theta_2 \,\&\, \theta_3) \simeq (\theta_1 \,\&\, \theta_2) \,\&\, \theta_3$$

$$\theta \,\&\, \mathbf{ns} \simeq \theta$$

$$\mathbf{ns} \,\&\, \theta \simeq \theta$$

$$\theta_1 \,\&\, \theta_2 \simeq \theta_2 \,\&\, \theta_1$$

$$\theta \,\&\, \theta \simeq \theta$$

$$\text{If } \theta_1 \leq \theta_1' \text{ and } \theta_2 \leq \theta_2' \text{ then } \theta_1 \,\&\, \theta_2 \leq \theta_1' \,\&\, \theta_2'$$

$$\theta \leq \theta_1 \,\&\, \theta_2 \text{ iff } \theta \leq \theta_1 \text{ and } \theta \leq \theta_2 \ .$$

The next three relationships in the definition of $\leq$ assert that primitive types are related by $\leq_{\text{prim}}$, that the object-type constructor is monotone, and that $\to$ is antimonotone in its first operand and monotone in its second operand. The last four relationships have the following consequences:

$$\iota\!:(\theta_1 \,\&\, \theta_2) \simeq \iota\!:\theta_1 \,\&\, \iota\!:\theta_2$$

$$\theta \to (\theta_1 \,\&\, \theta_2) \simeq (\theta \to \theta_1) \,\&\, (\theta \to \theta_2)$$

$$\iota\!:\mathbf{ns} \simeq \mathbf{ns}$$

$$\theta \to \mathbf{ns} \simeq \mathbf{ns} \ .$$

The first two of these equivalences show that conjunction distributes with object constructors (modulo $\simeq$) and with the right side (but not the left) of $\to$. The last two equivalences are analogous laws for the conjunction of zero types.

It can be shown that every pair of types has a least upper bound (which is unique modulo $\simeq$). In particular, the following equivalences suffice to compute a least upper bound, $\theta_1 \sqcup \theta_2$, of any types θ_1 and θ_2:

$$\theta_1 \sqcup \theta_2 \simeq \theta_2 \sqcup \theta_1$$

$$\theta \sqcup \mathbf{ns} \simeq \mathbf{ns}$$

$$\theta_1 \sqcup (\theta_2 \,\&\, \theta_3) \simeq (\theta_1 \sqcup \theta_2) \,\&\, (\theta_1 \sqcup \theta_3)$$

$$\rho \sqcup \iota\colon\theta \simeq \mathbf{ns}$$

$$\rho \sqcup (\theta_1 \to \theta_2) \simeq \mathbf{ns}$$

$$\iota\colon\theta_1 \sqcup (\theta_2 \to \theta_3) \simeq \mathbf{ns}$$

$$\rho_1 \sqcup \rho_2 \simeq \rho_1 \sqcup_{\text{prim}} \rho_2 \text{ when } \rho_1 \sqcup_{\text{prim}} \rho_2 \text{ exists}$$

$$\rho_1 \sqcup \rho_2 \simeq \mathbf{ns} \text{ when } \rho_1 \sqcup_{\text{prim}} \rho_2 \text{ does not exist}$$

$$\iota\colon\theta_1 \sqcup \iota\colon\theta_2 \simeq \iota\colon(\theta_1 \sqcup \theta_2)$$

$$\iota_1\colon\theta_1 \sqcup \iota_2\colon\theta_2 \simeq \mathbf{ns} \text{ when } \iota_1 \neq \iota_2$$

$$(\theta_1 \to \theta_1') \sqcup (\theta_2 \to \theta_2') \simeq (\theta_1 \,\&\, \theta_2) \to (\theta_1' \sqcup \theta_2')\ .$$

Finally, we introduce names that abbreviate certain commonly occurring nonprimitive types. As discussed in the previous section,

$$\delta\mathbf{acc} \stackrel{\text{def}}{=} \delta \to \mathbf{comm}$$

(e.g. $\mathbf{intacc} \stackrel{\text{def}}{=} \mathbf{int} \to \mathbf{comm}$), and

$$\delta\mathbf{var} \stackrel{\text{def}}{=} \delta \,\&\, \delta\mathbf{acc}\ .$$

There are also abbreviations for commonly occurring types of sequences. In general, a sequence s of element type θ and length n is an entity of type $(\mathbf{int} \to \theta) \,\&\, \mathit{len}\colon\mathbf{int}$ such that the value of $s.\mathit{len}$ is n and the application $s\,i$ is well-defined for all integers i such that $0 \leq i < n$. (Of course, the proviso on definedness is not implied by the type of the sequence.) The following abbreviations denote specific kinds of sequences:

$$\delta\mathbf{seq} \stackrel{\text{def}}{=} (\mathbf{int} \to \delta) \,\&\, \mathit{len}\colon\mathbf{int}$$

$$\delta\mathbf{accseq} \stackrel{\text{def}}{=} (\mathbf{int} \to \delta\mathbf{acc}) \,\&\, \mathit{len}\colon\mathbf{int}$$

$$\delta\mathbf{varseq} \stackrel{\text{def}}{=} (\mathbf{int} \to \delta\mathbf{var}) \,\&\, \mathit{len}\colon\mathbf{int}\ .$$

In Algol terminology, a δ**varseq** is a one-dimensional δ array with a lower bound of zero and and upper bound one less than its length. A δ**seq** is a similar entity whose elements can be evaluated but not assigned to, and a δ**accseq** is a similar entity whose elements can be assigned to but not evaluated. For example, **charseq** is the type of string constants.

3. The Semantics of Types

To describe the meaning of types, we will employ some basic concepts from category theory. The main reason for doing so is that, by formulating succinct definitions in terms of a mathematical theory of great generality, we gain an assurance that our language will be uniform and general.

A second reason is that the abstract concept of a category establishes a bridge between intuitive and rigorous semantics. Intuitively, we think of a type as standing for a set, and an implicit conversion as a function from one such set to another. But since our language permits nonterminating programs, types must denote domains (i.e. complete partial orders with a least element) and implicit conversions must be continuous functions. Moreover, a further level of complication arises when one develops a semantics that embodies the block structure of Algol-like languages; then types denote functors and implicit conversions are natural transformations between such functors [8,5,6].

However, the choice between these three different views is simply a choice between three different "semantic" categories:

- SET — in which the objects are sets, and the set of morphisms $S \to S'$ is the set of functions from S to S'.
- DOM — in which the objects are domains, and $D \to D'$ is the set of continuous functions from D to D'.
- PDOM^{Σ} — in which the objects are functors from a category Σ of "store shapes" to the category PDOM of complete partial orders and continuous functions, and $F \to F'$ is the set of natural transformations from F to F'.

Therefore, if we formulate the semantics of types in terms of an arbitrary category, assuming only properties that are possessed by all three of the above categories (being Cartesian closed and possessing certain limits), then we can think about the semantics in the intuitive setting of sets and functions, yet be confident that our semantics makes sense in a more rigorous setting.

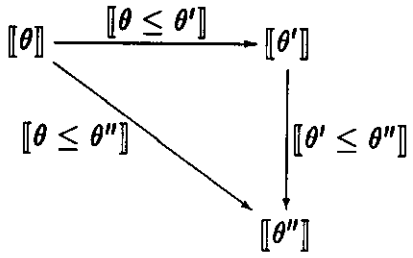
Thus we will define types in terms of an unspecified semantic category, while giving explanatory remarks and examples in terms of the particular category SET (or occasionally DOM).

For each type θ, we write $[\![\theta]\!]$ for the object (e.g. set) denoted by θ. Whenever $\theta \leq \theta'$, we write $[\![\theta \leq \theta']\!]$ for the implicit conversion morphism (e.g. function) from $[\![\theta]\!]$ to $[\![\theta']\!]$. Two requirements are imposed on these implicit conversion morphisms:

- For all types θ, the conversion from $[\![\theta]\!]$ to $[\![\theta]\!]$ must be an identity:

$$[\![\theta \leq \theta]\!] = I_{[\![\theta]\!]}.$$

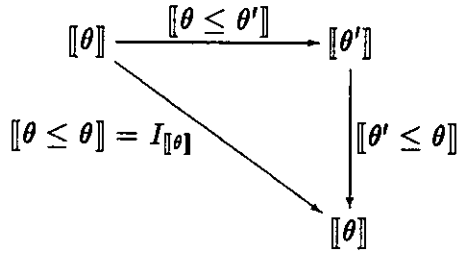
- Whenever $\theta \leq \theta'$ and $\theta' \leq \theta''$, the composition of $[\![\theta \leq \theta']\!]$ with $[\![\theta' \leq \theta'']\!]$ must equal $[\![\theta \leq \theta'']\!]$, i.e. the diagram



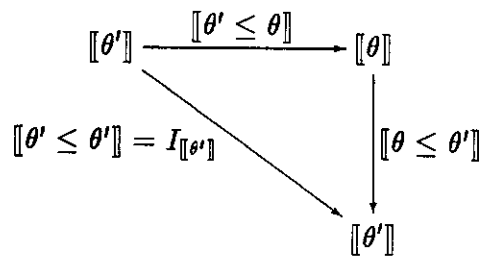
must commute.

These requirements coincide with a basic concept of category theory: $[\![-]\!]$ must be a functor from the preordered set of types (viewed as a category) to the semantic category.

The above requirements determine the semantics of equivalence. When $\theta \simeq \theta'$, the diagrams



and



both commute, so that $[\![\theta]\!]$ and $[\![\theta']\!]$ are *isomorphic*, which we denote by $[\![\theta]\!] \approx [\![\theta']\!]$. (Note, however, that nonequivalent types may also denote isomorphic objects.)

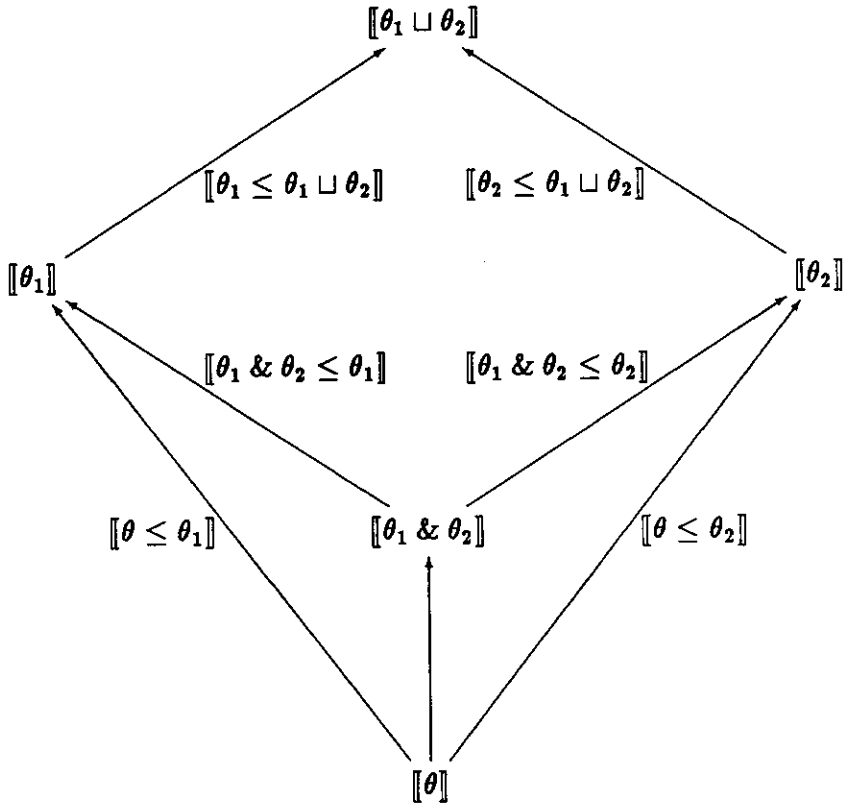
Next, we define (up to isomorphism) the meaning of each type constructor:

- Procedures — To define $\rightarrow$, we require the semantic category to be Cartesian closed, and define $[\![\theta \rightarrow \theta']\!]$ to be $[\![\theta]\!] \Rightarrow [\![\theta']\!]$, where $\Rightarrow$ denotes the exponentiation operation in the semantic category. In SET (DOM), $[\![\theta]\!] \Rightarrow [\![\theta']\!]$ is the set (domain) of all (continuous) functions from $[\![\theta]\!]$ to $[\![\theta']\!]$.

- Object Constructors — We define $[\![\iota\!:\theta]\!]$ to be an object that is isomorphic to $[\![\theta]\!]$.

- Nonsense — We define $[\![\mathbf{ns}]\!]$ to be a terminal object $\top$, i.e. an object such that, for any object s, there is exactly one morphism from s to $\top$. In SET or DOM a terminal object is a set containing one element. (Thus even nonsense phrases have a meaning, but they all have the same meaning.)

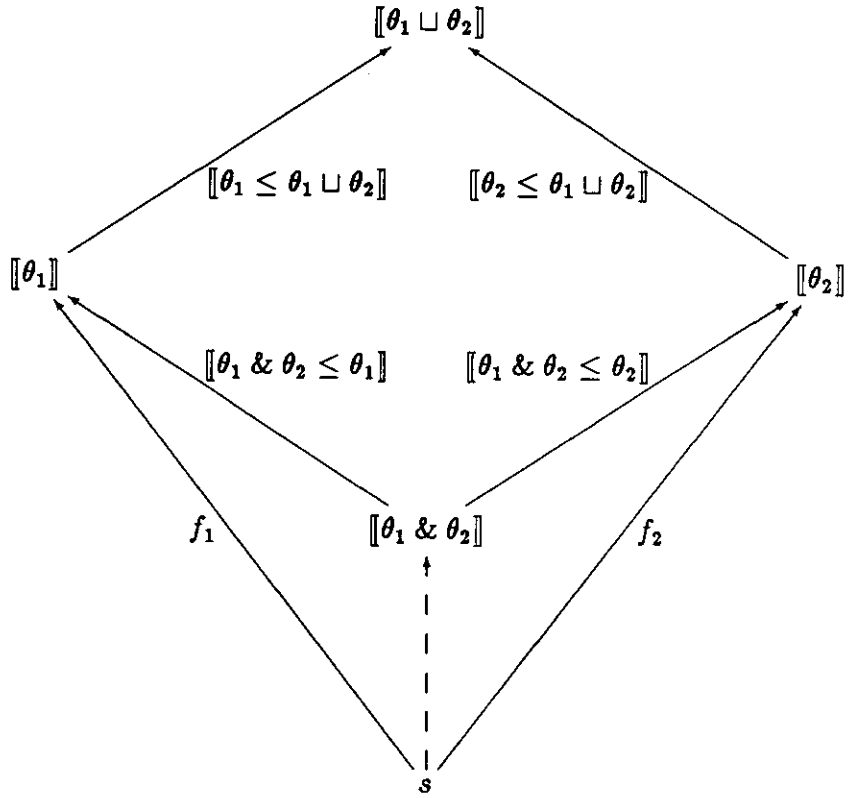
- Conjunction — Because of its novelty, we describe the meaning of conjunction in more detail than the other type constructors. Basically, the meaning of $\theta_1 \,\&\, \theta_2$ is determined by the meanings of θ_1, θ_2, and their least upper bound $\theta_1 \sqcup \theta_2$. From $\theta_1 \,\&\, \theta_2$, we can convert to θ_1 and from there to $\theta_1 \sqcup \theta_2$, or we can convert to θ_2 and from there to $\theta_1 \sqcup \theta_2$; clearly the two compositions of conversions should be equal. Moreover, whenever $\theta \leq \theta_1 \,\&\, \theta_2$, the composite composition from θ to $\theta_1 \,\&\, \theta_2$ to θ_1 should equal the direct conversion from θ to θ_1, and similarly for θ_2. In other words, in the diagram



the inner diamond must commute and, for all θ such that $\theta \leq \theta_1 \,\&\, \theta_2$, the two triangles must commute.

However, these requirements are not sufficient to determine $[\![\theta_1 \,\&\, \theta_2]\!]$. To strengthen them, we replace $[\![\theta]\!]$ by an arbitrary object s and $[\![\theta \leq \theta_1]\!]$ and $[\![\theta \leq \theta_2]\!]$ by any functions f_1 and f_2 that make the outer diamond commute, and we require the "mediating morphism" from s to $[\![\theta_1 \,\&\, \theta_2]\!]$ to be unique. Specifically, we define

$[\![\theta_1 \,\&\, \theta_2]\!]$ by requiring that, in the diagram



the inner diamond must commute and, for all objects s and morphisms f_1 and f_2 that make the outer diamond commute, there must be a unique morphism from s to $[\![\theta_1 \,\&\, \theta_2]\!]$ that makes the two triangles commute.

Clearly, this strengthening is something of a leap of faith. Thus it is reassuring that our definition coincides with a standard concept of category theory: we have defined $[\![\theta_1 \,\&\, \theta_2]\!]$ to be the *pullback* of $[\![\theta_1]\!]$, $[\![\theta_2]\!]$, and $[\![\theta_1 \sqcup \theta_2]\!]$ (which is unique up to isomorphism).

For sets or domains, the pullback is

$$[\![\theta_1 \,\&\, \theta_2]\!] \approx \left\{ \langle x_1, x_2 \rangle \;\middle|\; x_1 \in [\![\theta_1]\!] \text{ and } x_2 \in [\![\theta_2]\!] \text{ and } [\![\theta_1 \le \theta_1 \sqcup \theta_2]\!] x_1 = [\![\theta_2 \le \theta_1 \sqcup \theta_2]\!] x_2 \right\}.$$

(For domains, one must require all implicit conversion functions to be strict.) In other words, a meaning of type $\theta_1 \,\&\, \theta_2$ is a meaning of type θ_1 paired with a meaning of type θ_2, subject to the constraint that these meanings must convert to the same meaning of type $\theta_1 \sqcup \theta_2$.

The following are special cases of the definition of conjunction. Although we describe these cases in terms of SET and DOM, essentially the same results hold for any semantic category that is Cartesian closed and possesses the pullbacks necessary to define conjunction.

- If $\theta_1 \sqcup \theta_2 \simeq \mathbf{ns}$ then the constraint

$$[\![\theta_1 \leq \theta_1 \sqcup \theta_2]\!] x_1 = [\![\theta_2 \leq \theta_1 \sqcup \theta_2]\!] x_2$$

always holds, since both sides of the equation belong to the one-element set $[\![\mathbf{ns}]\!]$. Thus

$$[\![\theta_1 \,\&\, \theta_2]\!] \approx [\![\theta_1]\!] \times [\![\theta_2]\!] .$$

For example,

$$[\![\mathbf{intvar}]\!] = [\![\mathbf{int} \,\&\, (\mathbf{int} \to \mathbf{comm})]\!] \approx [\![\mathbf{int}]\!] \times [\![\mathbf{int} \to \mathbf{comm}]\!]$$

$$[\![\iota\colon \theta_1 \,\&\, (\theta_2 \to \theta_3)]\!] \approx [\![\iota\colon \theta_1]\!] \times [\![\theta_2 \to \theta_3]\!] \approx [\![\theta_1]\!] \times [\![\theta_2 \to \theta_3]\!]$$

and, when $\iota_1 \neq \iota_2$,

$$[\![\iota_1\colon \theta_1 \,\&\, \iota_2\colon \theta_2]\!] \approx [\![\iota_1\colon \theta_1]\!] \times [\![\iota_2\colon \theta_2]\!] \approx [\![\theta_1]\!] \times [\![\theta_2]\!] .$$

- If $\theta_1 \leq \theta_2$, so that $\theta_1 \sqcup \theta_2 = \theta_2$, then

$$[\![\theta_1 \,\&\, \theta_2]\!] \approx \left\{ \langle x_1, x_2 \rangle \;\middle|\; x_1 \in [\![\theta_1]\!] \text{ and } x_2 \in [\![\theta_2]\!] \text{ and } [\![\theta_1 \leq \theta_2]\!] x_1 = x_2 \right\} \approx [\![\theta_1]\!] .$$

For example,

$$[\![\mathbf{int} \,\&\, \mathbf{real}]\!] \approx [\![\mathbf{int}]\!]$$

$$[\![\mathbf{compl} \,\&\, \mathbf{comm}]\!] \approx [\![\mathbf{compl}]\!] .$$

- If $[\![\theta_1]\!]$ and $[\![\theta_2]\!]$ are subsets of $[\![\theta_1 \sqcup \theta_2]\!]$, and $[\![\theta_1 \leq \theta_1 \sqcup \theta_2]\!]$ and $[\![\theta_2 \leq \theta_1 \sqcup \theta_2]\!]$ are identity injections, then

$$[\![\theta_1 \,\&\, \theta_2]\!] \approx \left\{ \langle x_1, x_2 \rangle \;\middle|\; x_1 \in [\![\theta_1]\!] \text{ and } x_2 \in [\![\theta_2]\!] \text{ and } x_1 = x_2 \right\} \approx [\![\theta_1]\!] \cap [\![\theta_2]\!] .$$

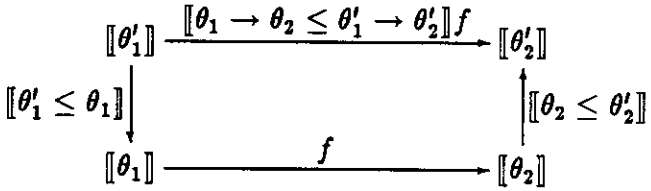
For example, since $\mathbf{int} \sqcup \mathbf{char} = \mathbf{value}$ and the corresponding conversions are identity injections,

$$[\![\mathbf{int} \,\&\, \mathbf{char}]\!] \approx [\![\mathbf{int}]\!] \cap [\![\mathbf{char}]\!] .$$

This is the purpose of introducing the type **value**. Had we not done so, we would have $\mathbf{int} \sqcup \mathbf{char} = \mathbf{ns}$, which would give $[\![\mathbf{int} \,\&\, \mathbf{char}]\!] \approx [\![\mathbf{int}]\!] \times [\![\mathbf{char}]\!]$.

Strictly speaking, this argument applies to data types rather than phrase types, but essentially the same situation holds for phrase types. In Forsythe, the sets denoted by the data types **real**, **bool**, and **char** are disjoint, and the set denoted by **int** is a subset of that denoted by **real**, so that conjunctions such as **int** & **char** denote the empty set. However, this is a detail of the language design, while the preceding argument is more general.

- Finally, we consider the conjunction of procedural types. First, we must define the implicit conversions between such types. If $\theta_1' \leq \theta_1$ and $\theta_2 \leq \theta_2'$ then the conversion of $f \in [\![\theta_1 \to \theta_2]\!]$ to $[\![\theta_1' \to \theta_2']\!]$ is obtained by composing f with appropriate conversions of its arguments and results:



or as an equation,

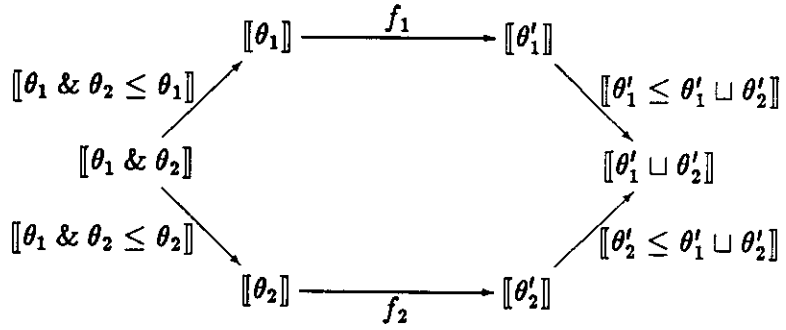
$$[\![\theta_1 \to \theta_2 \leq \theta_1' \to \theta_2']\!] f = [\![\theta_1' \leq \theta_1]\!] \,;\, f \,;\, [\![\theta_2 \leq \theta_2']\!] \,,$$

where ; denotes composition in diagrammatic order.

From the definition of conjunction, by substituting the equation for the least upper bound of two procedural types, and the above equation for the implicit conversion of procedural types, we obtain

$$\begin{aligned}
&[\![(\theta_1 \to \theta_1') \,\&\, (\theta_2 \to \theta_2')]\!] \approx \\
&\Big\{ \langle f_1, f_2 \rangle \;\Big|\; f_1 \in [\![\theta_1 \to \theta_1']\!] \text{ and } f_2 \in [\![\theta_2 \to \theta_2']\!] \\
&\qquad \text{and } [\![\theta_1 \to \theta_1' \leq (\theta_1 \,\&\, \theta_2) \to (\theta_1' \sqcup \theta_2')]\!] f_1 = [\![\theta_2 \to \theta_2' \leq (\theta_1 \,\&\, \theta_2) \to (\theta_1' \sqcup \theta_2')]\!] f_2 \Big\} \\
&= \Big\{ \langle f_1, f_2 \rangle \;\Big|\; f_1 \in [\![\theta_1 \to \theta_1']\!] \text{ and } f_2 \in [\![\theta_2 \to \theta_2']\!] \\
&\qquad\qquad \text{and } [\![\theta_1 \,\&\, \theta_2 \leq \theta_1]\!] \,;\, f_1 \,;\, [\![\theta_1' \leq \theta_1' \sqcup \theta_2']\!] = [\![\theta_1 \,\&\, \theta_2 \leq \theta_2]\!] \,;\, f_2 \,;\, [\![\theta_2' \leq \theta_1' \sqcup \theta_2']\!] \Big\} \,.
\end{aligned}$$

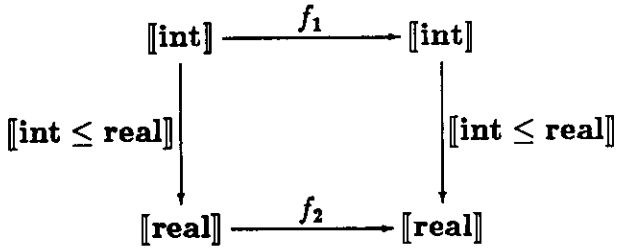
Here the constraint on f_1 and f_2 is the commutativity of a hexagon:



This constraint implies that the "versions" of a procedure with conjunctive type must respect implicit conversions. For example, since $\mathbf{int} \,\&\, \mathbf{real} = \mathbf{int}$ and $\mathbf{int} \sqcup \mathbf{real} = \mathbf{real}$ (taking $=$ rather than $\simeq$ here simplifies the argument),

$$\begin{aligned}
&[\![(\mathbf{int} \to \mathbf{int}) \,\&\, (\mathbf{real} \to \mathbf{real})]\!] \approx \\
&\Big\{ \langle f_1, f_2 \rangle \;\Big|\; f_1 \in [\![\mathbf{int} \to \mathbf{int}]\!] \text{ and } f_2 \in [\![\mathbf{real} \to \mathbf{real}]\!] \\
&\qquad\qquad \text{and } f_1 \,;\, [\![\mathbf{int} \leq \mathbf{real}]\!] = [\![\mathbf{int} \leq \mathbf{real}]\!] \,;\, f_2 \Big\} \,.
\end{aligned}$$

Here the hexagon collapses into a rectangle, so that f_1 and f_2 must satisfy



On the other hand,

$$[\![(\mathbf{int} \to \mathbf{int}) \,\&\, (\mathbf{char} \to \mathbf{char})]\!] \approx [\![\mathbf{int} \to \mathbf{int}]\!] \times [\![\mathbf{char} \to \mathbf{char}]\!]\,,$$

since in this case the hexagonal constraint on f_1 and f_2 is vacuously true because $[\![\mathbf{int} \,\&\, \mathbf{char}]\!]$ is the empty set.

4. Phrases and their Typings

We now introduce the phrases of Forsythe and give rules for determining their types. Specifically, we will give inference rules for formulas called typings.

A *type assignment* is a function from a finite set of identifiers to types. If π is a type assignment, then $[\pi \mid \iota{:}\theta]$ denotes the typing whose domain is $\mathrm{dom}\,\pi \cup \{\iota\}$, such that $[\pi \mid \iota{:}\theta]\iota = \theta$ and $[\pi \mid \iota{:}\theta]\iota' = \pi\iota'$ when $\iota' \neq \iota$. We write $[\pi \mid \iota_1{:}\theta_1 \mid \ldots \mid \iota_n{:}\theta_n]$ to abbreviate $[\ldots[\pi \mid \iota_1{:}\theta_1]\ldots \mid \iota_n{:}\theta_n]$.

If π is a type assignment, p is a phrase, and θ is a type, then the formula $\pi \vdash p : \theta$, called a *typing*, asserts that the phrase p has the type θ when its free identifiers are assigned types by π.

An *inference rule* consists of zero or more typings called *premisses* followed (after a horizontal line) by one or more typings called *conclusions*. The rule may contain metavariables denoting type assignments, phrases, identifiers, or types; an instance of the rule is obtained by replacing these metavariables by particular type assignments, phrases, identifiers, or types. (Some rules will have restrictions on the permissible replacements.) The meaning of a rule is that, for any instance, if all the premisses are valid typings then all of the conclusions are valid typings.

First, we have rules describing the behavior of subtypes, the nonsense type, and conjunction of types:

- Subtypes

$$\frac{\pi \vdash p : \theta}{\pi \vdash p : \theta'} \quad \text{when } \theta \leq \theta'$$

- Nonsense

$$\frac{}{\pi \vdash p : \mathbf{ns}}$$

- Conjunction

$$\frac{\pi \vdash p : \theta_1 \qquad \pi \vdash p : \theta_2}{\pi \vdash p : \theta_1 \,\&\, \theta_2}$$

Then there are rules for typing identifiers, applications (procedure calls), and conditional phrases:

- Identifiers

$$\frac{}{\pi \vdash \iota : \pi(\iota)} \quad \text{when } \iota \in \operatorname{dom} \pi$$

- Applications

$$\frac{\pi \vdash p_1 : \theta \to \theta' \qquad \pi \vdash p_2 : \theta}{\pi \vdash p_1\, p_2 : \theta'}$$

- Conditionals

$$\frac{\pi \vdash p_1 : \mathbf{bool} \qquad \pi \vdash p_2 : \theta \qquad \pi \vdash p_3 : \theta}{\pi \vdash \mathbf{if}\, p_1\, \mathbf{then}\, p_2\, \mathbf{else}\, p_3 : \theta}$$

Notice that the conditional construction is applicable to arbitrary types.

Next, we remedy a serious defect of Algol, by introducing abstractions (lambda expressions) to denote procedures:

- Abstractions

$$\frac{[\pi \mid \iota{:}\,\theta_i] \vdash p : \theta'}{\pi \vdash (\lambda \iota{:}\,\theta_1, \ldots, \theta_n.\ p) : \theta_i \to \theta'}$$

In this rule, notice that θ_i must be one of a list of types appearing explicitly in the abstraction. For example, under any type assignment, the abstraction

$$\lambda x{:}\,\mathbf{int}.\ x \times x + 2$$

has type **int** → **int**, while the abstraction

$$\lambda x\text{: } \mathbf{int}, \mathbf{real}.\ x \times x + 2$$

has both type **int** → **int** and type **real** → **real**, so that, by the rule for conjunction, it also has type (**int** → **int**) & (**real** → **real**). Also note the role of the colon, which is always used in Forsythe to specify the types of identifiers.

Then there are constructions for denoting objects and selecting their fields:

- Object Construction

$$\frac{\pi \vdash p : \theta}{\pi \vdash (\iota \equiv p) : (\iota\text{: } \theta)}$$

- Field Selection

$$\frac{\pi \vdash p : (\iota\text{: } \theta)}{\pi \vdash p.\iota : \theta}$$

The first of these forms only denotes objects with a single field; objects with several fields can be denoted by the merge construction, which will be described later. Note the role of the connective ≡, which is always used to connect identifiers with their meanings.

Next comes a long list of rules describing various types of constants and expressions:

- Constants

$$\frac{}{\pi \vdash \langle \text{int const} \rangle : \mathbf{int}}$$

$$\frac{}{\pi \vdash \langle \text{real const} \rangle : \mathbf{real}}$$

$$\frac{}{\pi \vdash \langle \text{char const} \rangle : \mathbf{char}}$$

$$\frac{}{\pi \vdash \langle \text{string} \rangle : \mathbf{charseq}}$$

- Arithmetic Expressions

$$\frac{\pi \vdash p : \textbf{int}}{\begin{array}{l}\pi \vdash +p : \textbf{int} \\ \pi \vdash -p : \textbf{int}\end{array}} \qquad \frac{\pi \vdash p : \textbf{real}}{\begin{array}{l}\pi \vdash +p : \textbf{real} \\ \pi \vdash -p : \textbf{real}\end{array}}$$

$$\frac{\begin{array}{l}\pi \vdash p_1 : \textbf{int} \\ \pi \vdash p_2 : \textbf{int}\end{array}}{\begin{array}{l}\pi \vdash p_1 + p_2 : \textbf{int} \\ \pi \vdash p_1 - p_2 : \textbf{int} \\ \pi \vdash p_1 \times p_2 : \textbf{int}\end{array}} \qquad \frac{\begin{array}{l}\pi \vdash p_1 : \textbf{real} \\ \pi \vdash p_2 : \textbf{real}\end{array}}{\begin{array}{l}\pi \vdash p_1 + p_2 : \textbf{real} \\ \pi \vdash p_1 - p_2 : \textbf{real} \\ \pi \vdash p_1 \times p_2 : \textbf{real}\end{array}}$$

$$\frac{\begin{array}{l}\pi \vdash p_1 : \textbf{int} \\ \pi \vdash p_2 : \textbf{int}\end{array}}{\begin{array}{l}\pi \vdash p_1 \div p_2 : \textbf{int} \\ \pi \vdash p_1 \textbf{ rem } p_2 : \textbf{int} \\ \pi \vdash p_1 ** p_2 : \textbf{int}\end{array}}$$

$$\frac{\begin{array}{l}\pi \vdash p_1 : \textbf{real} \\ \pi \vdash p_2 : \textbf{real}\end{array}}{\pi \vdash p_1/p_2 : \textbf{real}}$$

$$\frac{\begin{array}{l}\pi \vdash p_1 : \textbf{real} \\ \pi \vdash p_2 : \textbf{int}\end{array}}{\pi \vdash p_1 \uparrow p_2 : \textbf{real}}$$

- Relations

$$\frac{\begin{array}{l}\pi \vdash p_1 : \textbf{real} \\ \pi \vdash p_2 : \textbf{real}\end{array}}{\begin{array}{l}\pi \vdash p_1 = p_2 : \textbf{bool} \\ \pi \vdash p_1 \neq p_2 : \textbf{bool} \\ \pi \vdash p_1 < p_2 : \textbf{bool} \\ \pi \vdash p_1 \leq p_2 : \textbf{bool} \\ \pi \vdash p_1 > p_2 : \textbf{bool} \\ \pi \vdash p_1 \geq p_2 : \textbf{bool}\end{array}} \qquad \frac{\begin{array}{l}\pi \vdash p_1 : \textbf{char} \\ \pi \vdash p_2 : \textbf{char}\end{array}}{\begin{array}{l}\pi \vdash p_1 = p_2 : \textbf{bool} \\ \pi \vdash p_1 \neq p_2 : \textbf{bool} \\ \pi \vdash p_1 < p_2 : \textbf{bool} \\ \pi \vdash p_1 \leq p_2 : \textbf{bool} \\ \pi \vdash p_1 > p_2 : \textbf{bool} \\ \pi \vdash p_1 \geq p_2 : \textbf{bool}\end{array}}$$

$$\frac{\begin{array}{l}\pi \vdash p_1 : \textbf{bool} \\ \pi \vdash p_2 : \textbf{bool}\end{array}}{\begin{array}{l}\pi \vdash p_1 = p_2 : \textbf{bool} \\ \pi \vdash p_1 \neq p_2 : \textbf{bool}\end{array}}$$

- Boolean Expressions

$$\frac{\pi \vdash p : \mathbf{bool}}{\pi \vdash \sim p : \mathbf{bool}}$$

$$\frac{\pi \vdash p_1 : \mathbf{bool} \quad \pi \vdash p_2 : \mathbf{bool}}{\begin{array}{l}\pi \vdash p_1 \wedge p_2 : \mathbf{bool} \\ \pi \vdash p_1 \vee p_2 : \mathbf{bool} \\ \pi \vdash p_1 \Rightarrow p_2 : \mathbf{bool} \\ \pi \vdash p_1 \Leftrightarrow p_2 : \mathbf{bool}\end{array}}$$

Here the only real novelty is the provision of two operators for exponentiation: $\uparrow$ accepts two integers and yields a real, while $**$ accepts two integers and yields an integer (giving an error stop if its second operand is negative). The boolean operators $\Rightarrow$ and $\Leftrightarrow$ denote implication and equivalence (if-and-only-if) respectively.

As in Algol, the semicolon denotes sequential composition of commands. But now it can also be used to compose a command with a completion, giving a completion:

- Sequential Composition

$$\frac{\pi \vdash p_1 : \mathbf{comm} \quad \pi \vdash p_2 : \mathbf{comm}}{\pi \vdash p_1 \,;\, p_2 : \mathbf{comm}} \qquad \frac{\pi \vdash p_1 : \mathbf{comm} \quad \pi \vdash p_2 : \mathbf{compl}}{\pi \vdash p_1 \,;\, p_2 : \mathbf{compl}}$$

Two iterative constructions are provided: the traditional **while** command, and a **loop** construction which iterates its operand ad infinitum (i.e. until the operand jumps out of the loop by executing a completion):

- **while** Commands

$$\frac{\pi \vdash p_1 : \mathbf{bool} \quad \pi \vdash p_2 : \mathbf{comm}}{\pi \vdash \mathbf{while}\, p_1\, \mathbf{do}\, p_2 : \mathbf{comm}}$$

- **loop** Completions

$$\frac{\pi \vdash p : \mathbf{comm}}{\pi \vdash \mathbf{loop}\, p : \mathbf{compl}}$$

In place of the procedure declarations of Algol, Forsythe provides the more general let-definition construct invented by Peter Landin:

- Definitions

$$\frac{\begin{array}{l}\pi \vdash p_1 : \theta_1 \\ \quad\vdots \\ \pi \vdash p_n : \theta_n \\ [\,\pi \mid \iota_1{:}\,\theta_1 \mid \ldots \mid \iota_n{:}\,\theta_n\,] \vdash p : \theta\end{array}}{\pi \vdash (\mathbf{let}\ \iota_1 \equiv p_1, \ldots, \iota_n \equiv p_n\ \mathbf{in}\ p) : \theta}$$

For example, in place of

$$\mathbf{begin\ procedure}\ p(x);\ \theta\ x;\ B_{\text{proc}};\ B\ \mathbf{end}\,,$$

one writes

$$\mathbf{let}\ p \equiv \lambda x{:}\,\theta.\ B_{\text{proc}}\ \mathbf{in}\ B\,.$$

Such definitions are not limited to procedures. One can write

$$\mathbf{let}\ x \equiv 3\ \mathbf{in}\ B\,,$$

which will have exactly the same meaning as the phrase obtained from B by substituting 3 for x. Note, however, that this is not a variable declaration; x has the type **int** (the type of 3) and cannot be assigned to within B. Moreover, if y is an integer variable then

$$\mathbf{let}\ x \equiv y\ \mathbf{in}\ B$$

has the same meaning as the phrase obtained from B by substituting y for x, i.e. y is defined to be an alias of x.

Recursion is provided in two ways: a simple fixed-point operator **rec**, and an elaborate form of recursive definition:

- Fixed Points

$$\frac{\pi \vdash p : \theta \to \theta}{\pi \vdash \mathbf{rec}\ p : \theta}$$

- Recursive Definitions

$$\frac{\begin{array}{ll}[\,\pi \mid \iota_1{:}\,\theta_1 \mid \ldots \mid \iota_m{:}\,\theta_m\,] \vdash p_1 : \theta'_1 & \\ \quad\vdots & \text{when } 0 \le m \le n \text{ and} \\ [\,\pi \mid \iota_1{:}\,\theta_1 \mid \ldots \mid \iota_m{:}\,\theta_m\,] \vdash p_n : \theta'_n & \theta'_1 \le \theta_1, \ldots, \theta'_m \le \theta_m \\ [\,\pi \mid \iota_1{:}\,\theta'_1 \mid \ldots \mid \iota_n{:}\,\theta'_n\,] \vdash p : \theta & \end{array}}{\pi \vdash (\mathbf{letrec}\ \iota_1{:}\,\theta_1, \ldots, \iota_m{:}\,\theta_m\ \mathbf{where}\ \iota_1 \equiv p_1, \ldots, \iota_n \equiv p_n\ \mathbf{in}\ p) : \theta}$$

In the last form, one can mix recursive and nonrecursive definitions. However, the types of the recursively defined identifiers must be listed before the definitions themselves, so that the reader (and compiler) knows these types when reading the definitions. (Actually, $\theta_1, \ldots, \theta_m$ must be the types of the occurrences of $\iota_1, \ldots, \iota_m$ in the definition bodies $p_1, \ldots, p_n$; the occurrences in p must have the types of the p_i's, which may be subtypes of the θ_i's.)

Note that, in both nonrecursive and recursive definitions, the identifiers $\iota_1, \ldots, \iota_n$ must be distinct, and the ordering of the lists separated by commas has no effect on type or meaning.

Next we consider a construction for conjoining or "merging" meanings. Suppose p_1 has type θ_1, p_2 has type θ_2, and $\theta_1 \sqcup \theta_2 \simeq \mathbf{ns}$, so that $[\![\theta_1 \,\&\, \theta_2]\!] \approx [\![\theta_1]\!] \times [\![\theta_2]\!]$. One might hope to write p_1, p_2 to denote a meaning of type $\theta_1 \,\&\, \theta_2$.

Unfortunately, this conflicts with the behavior of subtypes, since p_1 and p_2 might have types θ_1' and θ_2' such that $\theta_1' \leq \theta_1$ and $\theta_2' \leq \theta_2$ but $\theta_1' \sqcup \theta_2' \not\simeq \mathbf{ns}$. For example, although $(a \equiv 3, b \equiv 4)$ and $b \equiv 5$ have types $a\colon \mathbf{int}$ and $b\colon \mathbf{int}$, whose least upper bound is $\mathbf{ns}$, the phrase

$$(a \equiv 3, b \equiv 4), b \equiv 5$$

would be ambiguous.

Our solution to this problem is to permit p_1, p_2 only when p_2 is an abstraction or an object construction, whose meaning then overwrites all components of the meaning of p_1 that have procedural types, or object types with the same field name. The inference rules are:

- Merging

$$\frac{[\,\pi \mid \iota\colon \theta_i\,] \vdash p_2 : \theta'}{\pi \vdash (p_1, \lambda\iota\colon \theta_1, \ldots, \theta_n.\ p_2) : \theta_i \to \theta'}$$

$$\frac{\pi \vdash p_1 : \rho}{\pi \vdash (p_1, \lambda\iota\colon \theta_1, \ldots, \theta_n.\ p_2) : \rho} \quad \text{when } \rho \text{ is a primitive type}$$

$$\frac{\pi \vdash p_1 : (\iota_1\colon \theta)}{\pi \vdash (p_1, \lambda\iota\colon \theta_1, \ldots, \theta_n.\ p_2) : (\iota_1\colon \theta)}$$

$$\frac{\pi \vdash p_2 : \theta}{\pi \vdash (p_1, \iota \equiv p_2) : (\iota{:}\theta)}$$

$$\frac{\pi \vdash p_1 : \rho}{\pi \vdash (p_1, \iota \equiv p_2) : \rho} \quad \text{when } \rho \text{ is a primitive type}$$

$$\frac{\pi \vdash p_1 : \theta \to \theta'}{\pi \vdash (p_1, \iota \equiv p_2) : \theta \to \theta'}$$

$$\frac{\pi \vdash p_1 : (\iota_1{:}\theta)}{\pi \vdash (p_1, \iota \equiv p_2) : (\iota_1{:}\theta)} \quad \text{when } \iota \neq \iota_1$$

Next, we introduce a construction for defining a sequence by giving a list of its elements:

- Sequences

$$\frac{\begin{array}{l}\pi \vdash p_0 : \theta \\ \quad \vdots \\ \pi \vdash p_{n-1} : \theta\end{array}}{\pi \vdash \mathbf{seq}(p_0, \ldots, p_{n-1}) : (\mathbf{int} \to \theta)\ \&\ \mathit{len}{:}\,\mathbf{int}} \quad \text{when } n \geq 1$$

The effect of this construction is that, if e is an integer expression with value k such that $0 \leq k < n$, then

$$\mathbf{seq}(p_0, \ldots, p_{n-1})\, e$$

has the same meaning as p_k (and thus can be used in the role of a **case** construction). Moreover,

$$\mathbf{seq}(p_0, \ldots, p_{n-1}).\mathit{len}$$

is an integer expression with value n.

Finally, we introduce type definitions that permit the user to let identifiers stand for types. The types occurring in phrases are generalized to type expressions that can contain type identifiers, which are give meaning by the inference rule

- Type Definitions

$$\frac{\pi \vdash (p/\iota_1, \ldots, \iota_n \to \theta_1, \ldots, \theta_n) : \theta}{\pi \vdash (\mathbf{lettype}\, \iota_1 \equiv \theta_1, \ldots, \iota_n \equiv \theta_n\, \mathbf{in}\, p) : \theta}$$

where $(p/\iota_1, \ldots, \iota_n \to \theta_1, \ldots, \theta_n)$ denotes the result of simultaneously substituting $\theta_1, \ldots, \theta_n$ for the free occurrences (as type identifiers) of $\iota_1, \ldots, \iota_n$ in type expressions within p. Notice that this is a transparent, rather than opaque, form of type definition.

The inference rules we have given define an "explicitly typed" language, in which the type of an identifier must be stated whenever it is bound by an abstraction or defined recursively. It is known that (in contrast to ML) the conjunctive type discipline is too rich to permit compile-time type checking in the absence of all type information. However, we hope to provide partial type inference, so that some type information can be elided.

It is not clear how far we can go in this direction. However, there is one context in which type information can obviously be omitted: When an abstraction occurs as the parameter of an application, the type of the procedure being applied determines the type of the parameter, which in turn determines the type of the identifiers bound by the abstraction. This is formalized by the following inference rule:

- Type Elision

$$\frac{\begin{array}{l}\pi \vdash p_1 : \big((\theta_{11} \to \cdots \to \theta_{1n} \to \theta_1)\ \&\ \cdots\ \&\ (\theta_{k1} \to \cdots \to \theta_{kn} \to \theta_k)\big) \to \theta \\ [\,\pi \mid \iota_1{:}\theta_{11} \mid \ldots \mid \iota_n{:}\theta_{1n}\,] \vdash p_2 : \theta_1 \\ \quad\vdots \\ [\,\pi \mid \iota_1{:}\theta_{k1} \mid \ldots \mid \iota_n{:}\theta_{kn}\,] \vdash p_2 : \theta_k\end{array}}{\pi \vdash p_1(\lambda\iota_1.\ \ldots\ \lambda\iota_n.\ p_2) : \theta}$$

5. Predefined Identifiers

In place of various constants, Forsythe provides predefined identifiers, which may be redefined by the user, but which take on standard types and meanings outside of these bindings. In describing these identifiers, we simply state the type of their unbound occurrences, e.g. we write *true*: **bool** as an abbreviation for the inference rule

$$\frac{}{\pi \vdash \mathit{true} : \mathbf{bool}} \quad \text{when } \mathit{true} \notin \operatorname{dom}\pi\ .$$

In the first place, there are the usual boolean constants, a **skip** command that leaves the state unchanged, and a standard phrase of type **ns**:

true: **bool**
false: **bool**
skip: **comm**
null: **ns**

(Of course, there are many other nonsense phrases — phrases whose only types are equivalent to **ns** — that are all too easy to write, but *null* is the only such phrase that will not activate a warning message from the compiler. The point is that there are contexts in which *null* is sensible, for example as the denotation of an object with no fields.)

The remaining predeclared identifiers denote built-in procedures. Four of these procedures serve to declare variables:

$$
\begin{aligned}
&\mathit{new}\delta\mathit{var}\colon \delta \to \\
&\quad \big((\delta\mathbf{var} \to \mathbf{comm}) \to \mathbf{comm} \\
&\quad \&(\delta\mathbf{var} \to \mathbf{compl}) \to \mathbf{compl} \\
&\quad \&(\delta\mathbf{var} \to \mathbf{int}) \to \mathbf{int} \\
&\quad \&(\delta\mathbf{var} \to \mathbf{real}) \to \mathbf{real} \\
&\quad \&(\delta\mathbf{var} \to \mathbf{bool}) \to \mathbf{bool} \\
&\quad \&(\delta\mathbf{var} \to \mathbf{char}) \to \mathbf{char}\big)
\end{aligned}
$$

The application $\mathit{new}\delta\mathit{var}\ \mathit{init}\ p$ causes a new δ variable to be added to the state of the computation; this variable is initialized to the value *init*, and then the procedure p is applied to the variable. Thus

$$\mathit{newintvar}\ \mathit{init}\ \lambda x.\ B$$

is equivalent to the Algol block

$$\mathbf{begin\ integer}\ x;\ x := \mathit{init};\ B\ \mathbf{end}\,.$$

The multiplicity of types of the $\mathit{new}\delta\mathit{var}$ procedures permits variables to be declared in completions and expressions as well as commands.

Four analogous procedures are provided for declaring variable sequences:

$$
\begin{aligned}
&\mathit{new}\delta\mathit{varseq}\colon \mathbf{int} \to (\mathbf{int} \to \delta) \to \\
&\quad \big((\delta\mathbf{varseq} \to \mathbf{comm}) \to \mathbf{comm} \\
&\quad \&(\delta\mathbf{varseq} \to \mathbf{compl}) \to \mathbf{compl} \\
&\quad \&(\delta\mathbf{varseq} \to \mathbf{int}) \to \mathbf{int} \\
&\quad \&(\delta\mathbf{varseq} \to \mathbf{real}) \to \mathbf{real} \\
&\quad \&(\delta\mathbf{varseq} \to \mathbf{bool}) \to \mathbf{bool} \\
&\quad \&(\delta\mathbf{varseq} \to \mathbf{char}) \to \mathbf{char}\big)
\end{aligned}
$$

The application $\mathit{new}\delta\mathit{varseq}\ l\ \mathit{init}\ p$ causes a new δ variable sequence of length l to be added to the state of the computation; the elements of this sequence are initialized to values obtained by applying the procedure *init*, and then the procedure p is applied to the sequence. Thus

$$\mathit{newintvarseq}\ l\ \mathit{init}\ \lambda x.\ B$$

is equivalent to the Algol block

begin integer array $x(0 : l - 1)$;
    begin integer i;
    for $i := 0$ **to** $l - 1$ **do** $x(i) := init(i)$
    end;
B
end .

In essence, this approach to the declaration of variables and sequences is a syntactic desugaring of the conventional form of declarations into the application of a procedure; procedures such as *newintvar init* or *newintvarseq l init* that are intended to be used this way are called *declarators*. The advantage of this view is that the user can define his own declarators or declarator-producing procedures. For example (as we will illustrate later), the user can define his own declarators for any kind of array for which he can program the index-mapping function.

The procedure

escape: **(compl → comm) → comm**

applies its parameter to a completion whose execution causes immediate termination of the application of *escape*. Thus

escape $\lambda e.\ C$

is equivalent to the Algol block

begin C'; e: **end** ,

where C' is obtained from C by substituting **goto** e for e.

The procedure

error: **charseq → (int & bool & char & compl)**

is such that *error* s is an expression or completion whose execution terminates the computation after printing s as an error message.

The above is not a complete compendium of predeclared identifiers; others will be provided for numerical functions and input-output procedures.

It should also be noted that types such as **int**, **comm**, and **intvar** are predefined type identifiers, whose meaning can be redefined by using the **lettype** definition.

6. Syntactic Sugar

The following abbreviations are provided to avoid repeating type information when several identifiers range over the same type. In types,

$$\iota_1, \ldots, \iota_n{:}\theta \quad \text{abbreviates} \quad \iota_1{:}\theta \,\&\, \cdots \,\&\, \iota_n{:}\theta\,.$$

In lambda expressions

$$\lambda\iota_1, \ldots, \iota_n{:}\theta_1, \ldots, \theta_k.\ p \quad \text{abbreviates}$$

$$\lambda\iota_1{:}\theta_1, \ldots, \theta_k. \ \ldots\ \lambda\iota_n{:}\theta_1, \ldots, \theta_k.\ p$$

and

$$\lambda\iota_1, \ldots, \iota_n.\ p \quad \text{abbreviates} \quad \lambda\iota_1. \ \ldots\ \lambda\iota_n.\ p\,.$$

In **letrec** definitions

$$\iota_1, \ldots, \iota_n{:}\theta \quad \text{abbreviates} \quad \iota_1{:}\theta, \ \ldots\ , \iota_n{:}\theta\,.$$

Also, to permit a more Algol-like appearance,

$$p_1 := p_2 \quad \text{abbreviates} \quad p_1 p_2\,.$$

7. Reduction Rules

An operational way of describing Forsythe is to say that a program is a phrase of type **comm**, in an enriched typed lambda calculus, that is executed by first reducing the phrase to normal form (more precisely, to a possibly infinite or partial head-normal form) and then executing the normal form, which will be a program in the simple imperative language. Although we will not pursue this view in these notes, it is useful to list some of the reduction rules, which preserve the meanings of programs and thus provide insight into their semantics.

First there is the lambda-calculus rule of β-reduction:

$$(\lambda\iota.\ p_1)p_2 \quad \Longrightarrow \quad (p_1/\iota \to p_2)$$

where $(p_1/\iota \to p_2)$ denotes the result of substituting p_2 for the free occurrences of ι (except as a type identifier or field name) in p_1.

Then there is a rule for selecting fields:

$$(\iota \equiv p).\iota \quad \Longrightarrow \quad p\,,$$

two rules for conditionals:

$$(\textbf{if } p_1 \textbf{ then } p_2 \textbf{ else } p_3)p_4 \quad \Longrightarrow \quad \textbf{if } p_1 \textbf{ then } p_2\, p_4 \textbf{ else } p_3\, p_4$$

$$(\textbf{if } p_1 \textbf{ then } p_2 \textbf{ else } p_3).\iota \quad \Longrightarrow \quad \textbf{if } p_1 \textbf{ then } p_2.\iota \textbf{ else } p_3.\iota\,,$$

a rule for nonrecursive definitions:

$$\textbf{let } \iota_1 \equiv p_1,\ \ldots\ , \iota_n \equiv p_n \textbf{ in } p \quad \Longrightarrow \quad (p/\iota_1\ \ldots\ , \iota_n \rightarrow p_1,\ \ldots\ , p_n)\,,$$

and a rule for the fixed-point operator:

$$\textbf{rec } p \quad \Longrightarrow \quad p(\textbf{rec } p)\,.$$

For recursive definitions, one can give a simple rule that excludes simultaneous recursion:

$$\textbf{letrec } \iota_1\!:\theta_1 \textbf{ where } \iota_1 \equiv p_1 \textbf{ in } p \quad \Longrightarrow \quad \textbf{let } \iota_1 \equiv \textbf{rec}(\lambda\iota_1\!:\theta_1.\ p_1) \textbf{ in } p\,.$$

However, the general rule including simultaneous recursion is much more complex:

$$\begin{aligned}&\textbf{letrec } \iota_1\!:\theta_1,\ \ldots\ , \iota_m\!:\theta_m \textbf{ where } \iota_1 \equiv p_1,\ \ldots\ , \iota_n \equiv p_n \textbf{ in } p \quad \Longrightarrow\\ &\quad \textbf{let } \iota \equiv \textbf{rec}\big(\lambda\iota\!:(\iota_1\!:\theta_1\ \&\ \cdots\ \&\ \iota_m\!:\theta_m).\\ &\qquad \textbf{let } \iota_1 \equiv \iota.\iota_1,\ \ldots\ , \iota_m \equiv \iota.\iota_m \textbf{ in}(\iota_1 \equiv p_1,\ \ldots\ , \iota_n \equiv p_n)\big)\\ &\quad \textbf{in let } \iota_1 \equiv \iota.\iota_1,\ \ldots\ , \iota_n \equiv \iota.\iota_n \textbf{ in } p\,,\end{aligned}$$

where ι is any identifier not occurring in the **letrec** definition. (A useful exercise is to show that any valid typing of the left side of this rule is also a valid typing of the right side.)

In addition, there are a number of rules dealing with the merging operation:

$$\begin{aligned}(p_1, \lambda\iota.\ p_2)p_3 \quad &\Longrightarrow \quad (\lambda\iota.\ p_2)p_3\\ (p_1, \iota \equiv p_2)p_3 \quad &\Longrightarrow \quad p_1\, p_3\\ (p_1, \lambda\iota.\ p_2).\iota' \quad &\Longrightarrow \quad p_1.\iota'\\ (p_1, \iota \equiv p_2).\iota \quad &\Longrightarrow \quad (\iota \equiv p_2).\iota\\ (p_1, \iota \equiv p_2).\iota' \quad &\Longrightarrow \quad p_1.\iota' \quad \text{when } \iota \neq \iota'\,.\end{aligned}$$

(It should be noted that these rules are not complete; in particular, we have not provided rules for reducing merges in contexts that require primitive types.)

The reduction rules make it clear that call by name pervades Forsythe. For example, if p_c is any phrase that does not contain free occurrences of ι, and p_1 and p_2 are any phrases, then

$$\begin{aligned}(\lambda\iota.\ p_c)p_1 \quad &\Longrightarrow \quad p_c\\ \textbf{let } \iota \equiv p_1 \textbf{ in } p_c \quad &\Longrightarrow \quad p_c\\ (\iota_1 \equiv p_1, \iota_2 \equiv p_2).\iota_1 \quad &\Longrightarrow \quad p_1\\ (\iota_1 \equiv p_1, \iota_2 \equiv p_2).\iota_2 \quad &\Longrightarrow \quad p_2\end{aligned}$$

hold even when p_1 or p_2 denote nonterminating computations.

8. Examples of Procedures

In this and the next three sections, we provide a variety of examples of Forsythe programs. Many of these examples are translations of Algol W programs given in [7], which the reader may wish to compare with the present versions.

To define a proper procedure that sets its second parameter to the factorial of its first parameter, we define *fact* to be the obvious program, abstracted on an integer expression n and an integer variable f:

$$\begin{array}{l}\textbf{let } \mathit{fact} \equiv \lambda n\text{:}\,\textbf{int}.\ \lambda f\text{:}\,\textbf{intvar}.\\ \quad \mathit{newintvar}\ 0\ \lambda k.\\ \quad\quad \big(f := 1\,;\,\textbf{while } k \neq n \textbf{ do } (k := k+1\,;\,f := k \times f)\big)\end{array}$$

However, this procedure has the usual shortcoming of call by name: it will repeatedly evaluate the expression n. To remedy this defect, we replace n by a local variable (also called n) that is initialized to the input parameter n. Notice that this is equivalent to the definition of call by value in Algol 60.

$$\begin{array}{l}\textbf{let } \mathit{fact} \equiv \lambda n\text{:}\,\textbf{int}.\ \lambda f\text{:}\,\textbf{intvar}.\\ \quad \mathit{newintvar}\ n\ \lambda n.\\ \quad\quad \mathit{newintvar}\ 0\ \lambda k.\\ \quad\quad\quad \big(f := 1\,;\,\textbf{while } k \neq n \textbf{ do } (k := k+1\,;\,f := k \times f)\big)\end{array}$$

We can also modify this procedure to obtain the effect of calling f by result. We replace f by a local variable, and then assign the final value of this local variable to the parameter f, which now has type **intacc**, since it is never evaluated by the procedure.

$$\begin{array}{l}\textbf{let } \mathit{fact} \equiv \lambda n\text{:}\,\textbf{int}.\ \lambda f\text{:}\,\textbf{intacc}.\\ \quad \mathit{newintvar}\ n\ \lambda n.\ \mathit{newintvar}\ 1\ \lambda \mathit{localf}.\\ \quad\quad \big(\mathit{newintvar}\ 0\ \lambda k.\\ \quad\quad\quad \textbf{while } k \neq n \textbf{ do } (k := k+1\,;\,\mathit{localf} := k \times \mathit{localf})\,;\\ \quad\quad f := \mathit{localf}\big)\end{array}$$

This transformation is sufficiently complex that it is worthwhile to encapsulate it as a procedure. We define

$$\begin{array}{l}\textbf{let } \mathit{newintvarres} \equiv\\ \quad \lambda \mathit{init}\text{:}\,\textbf{int}.\ \lambda \mathit{fin}\text{:}\,\textbf{intacc}, \textbf{int} \to \textbf{compl}.\ \lambda b\text{:}\,\textbf{intvar} \to \textbf{comm}.\\ \quad\quad \mathit{newintvar}\ \mathit{init}\ \lambda \mathit{local}.\ (b\ \mathit{local}\,;\,\mathit{fin} := \mathit{local})\end{array}$$

(Here *fin* is permitted to have the type **int** → **compl** as well as **intacc** ≃ **int** → **comm**. This makes sense because the final assignment *fin* := *local*, really means *fin local*, which can be a completion.) Then to call f by result we define

$$\begin{array}{l}\textbf{let } \mathit{fact} \equiv \lambda n\colon \textbf{int}.\ \lambda f\colon \textbf{intacc}.\\ \quad \mathit{newintvar}\ n\ \lambda n.\ \mathit{newintvarres}\ 1\ f\ \lambda f.\\ \quad\quad \mathit{newintvar}\ 0\ \lambda k.\\ \quad\quad\quad \textbf{while } k \neq n \textbf{ do } (k := k+1\ ;\ f := k \times f)\end{array}$$

We can also define the traditional recursive function procedure for computing the factorial. Here again we call n by value, illustrating the use of *newintvar* within an expression.

$$\begin{array}{l}\textbf{letrec } \mathit{fact}\colon \textbf{int} \to \textbf{int}\\ \textbf{where } \mathit{fact} \equiv \lambda n\colon \textbf{int}.\\ \quad \mathit{newintvar}\ n\ \lambda n.\\ \quad\quad \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times \mathit{fact}(n-1)\end{array}$$

Next, we give some examples of procedures that take advantage of call by name. In a boolean function procedure for implication, call by name gives "short-circuit" evaluation,

$$\textbf{let } \mathit{implies} \equiv \lambda p\colon \textbf{bool}.\ \lambda q\colon \textbf{bool}.\ \textbf{if } p \textbf{ then } q \textbf{ else } \mathit{true}$$

i.e. q will not be evaluated when p is false. In a proper procedure akin to the Pascal **repeat** command,

$$\textbf{let } \mathit{repeat} \equiv \lambda c\colon \textbf{comm}.\ \lambda b\colon \textbf{bool}.\ (c\ ;\ \textbf{while } \sim b \textbf{ do } c)$$

b must be called by name to permit its repeated evaluation. Repeated evaluation is also crucial to "Jensen's device", an example of call by name in the original Algol 60 Report:

$$\begin{array}{l}\textbf{let } \mathit{sum} \equiv \lambda i\colon \textbf{intvar}.\ \lambda e\colon \textbf{int}.\\ \quad \textbf{begin } s := 0\ ;\ i := a-1;\\ \quad \textbf{while } i < b \textbf{ do } (i := i+1\ ;\ s := s+e)\\ \quad \textbf{end}\\ \textbf{in } \mathit{sum}\ i\ \big(X(i) \times X(i)\big)\end{array}$$

Finally, we give two higher-order procedures akin to the **for** command:

$$\begin{array}{l}\textbf{let } \mathit{for} \equiv \lambda l, u\text{:}\,\textbf{int}.\ \lambda b\text{:}\,\textbf{int} \to \textbf{comm}.\\ \quad \mathit{newintvar}(l-1)\ \lambda k.\ \mathit{newintvar}\ u\ \lambda u.\\ \quad\quad \textbf{while } k < u \textbf{ do } (k := k+1\,;\, b\ k),\\ \mathit{fordown} \equiv \lambda l, u\text{:}\,\textbf{int}.\ \lambda b\text{:}\,\textbf{int} \to \textbf{comm}.\\ \quad \mathit{newintvar}(u+1)\ \lambda k.\ \mathit{newintvar}\ l\ \lambda l.\\ \quad\quad \textbf{while } k > l \textbf{ do } (k := k-1\,;\, b\ k)\\ \textbf{in } \mathit{for}\ 0\ 9\ \lambda i.\ s := s + X(i) \times X(i)\end{array}$$

Notice that, in these procedures, since the parameter of b has type **int**, the application $b\ k$ cannot change the value of k. Moreover, although this application can change the values of the parameters l and u, the interval iterated over is always determined by the initial values of these parameters.

9. Escapes and Completions

The procedure *escape* declares a completion whose execution causes an exit from the call of *escape*. A simple example of its use is the following procedure for searching an integer function X (which might be an integer sequence or array) over the interval l to u for a value that is equal to y. If such a value is found, the procedure sets *present* to *true* and j to the argument for which $X(j) = y$; otherwise it sets *present* to *false*.

$$\begin{array}{l}\textbf{let } \mathit{linsearch} \equiv \lambda X\text{:}\,\textbf{int} \to \textbf{int}.\ \lambda l, u, y\text{:}\,\textbf{int}.\ \lambda \mathit{present}\text{:}\,\textbf{boolacc}.\ \lambda j\text{:}\,\textbf{intacc}.\\ \quad \mathit{escape}\ \lambda \mathit{out}.\\ \quad\quad \big(\mathit{for}\ l\ u\ \lambda k.\ \textbf{if } X(k) = y \textbf{ then } (\mathit{present} := \mathit{true}\,;\, j := k\,;\, \mathit{out}) \textbf{ else } \mathit{skip};\\ \quad\quad \mathit{present} := \mathit{false}\big)\end{array}$$

An alternative version of this procedure branches to one of two parameters depending upon whether the search succeeds. If the search fails, it goes to the completion *failure*; if the search succeeds, it goes to the completion procedure *success*, passing it the integer k such that $X(k) = y$.

$$\begin{array}{l}\textbf{let } \mathit{linsearch} \equiv \lambda X\text{:}\,\textbf{int} \to \textbf{int}.\ \lambda l, u, y\text{:}\,\textbf{int}.\\ \quad \lambda \mathit{success}\text{:}\,\textbf{int} \to \textbf{compl}.\ \lambda \mathit{failure}\text{:}\,\textbf{compl}.\\ \quad\quad \big(\mathit{for}\ l\ u\ \lambda k.\ \textbf{if } X(k) = y \textbf{ then } \mathit{success}\ k \textbf{ else } \mathit{skip}\,;\, \mathit{failure}\big)\end{array}$$

In addition to using *escape*, one can define completions recursively, to obtain the equivalent of conventional labels. For example, the following procedure sets y to x^n (in time

$\log n$), without doing unnecessary tests:

```
let power ≡ λx, n: int. λy: intacc.
    newintvar n λk. newintvarres 1 y λy. newintvar x λz.
        escape λzr {k = 0}.
            letrec tr, nz, ev, od, nzev: compl
            where
                tr {true} ≡ if k = 0 then zr else nz,
                nz {k ≠ 0} ≡ if odd k then od else nzev,
                ev {even k} ≡ if k = 0 then zr else nzev,
                od {odd k} ≡ k := k − 1 ; y := y × z ; ev,
                nzev {k ≠ 0 ∧ even k} ≡ k := k ÷ 2 ; z := z × z ; nz
            in tr
```

The invariant of this program, i.e. the assertion that holds whenever any completion is executed, is

$$y \times z^k = x^n \wedge k \geq 0\,.$$

The additional assertions given as comments at the binding of each completion hold when the corresponding completion is executed.

Notice that, in contrast to labels, one can never execute a completion by "passing through" to its definition. Indeed, the meaning of the above program is independent of the order of the definitions of completions.

10. Sequences and Arrays

As we remarked earlier, using the built-in procedures for declaring variable sequences, the programmer can define his own procedures for declaring more complex kinds of arrays. For example, suppose we want Algol-like one-dimensional integer arrays with arbitrary lower and upper bounds (denoted by the field names *ll* and *ul*). First we define abbreviations for the relevant types:

```
lettype intarray ≡ (int → int & ll, ul: int),
    intaccarray ≡ (int → intacc & ll, ul: int),
    intvararray ≡ (int → intvar & ll, ul: int)
```

Then the following procedure serves to declare integer variable arrays:

$$\begin{array}{l}\textbf{let } \mathit{newintvararray} \equiv \lambda l, u\colon \textbf{int}.\ \lambda \mathit{init}\colon \textbf{int} \to \textbf{int}.\\ \quad \lambda b\colon \textbf{intvararray} \to \textbf{comm}, \textbf{intvararray} \to \textbf{compl}, \textbf{intvararray} \to \textbf{int},\\ \qquad \textbf{intvararray} \to \textbf{real}, \textbf{intvararray} \to \textbf{bool}, \textbf{intvararray} \to \textbf{char}.\\ \quad \mathit{newintvar}\ l\ \lambda l.\ \mathit{newintvar}\ u\ \lambda u.\\ \qquad \mathit{newintvarseq}(u-l+1)\big(\lambda k\colon \textbf{int}.\ \mathit{init}(k+l)\big)\lambda X.\\ \qquad\quad b\big(\lambda k\colon \textbf{int}.\ X(k-l), ll \equiv l, ul \equiv u\big)\end{array}$$

We can also define a procedure *slice* that, given an array and two integers, yields a subsegment of the array with new bounds. The simplest definition is

$$\textbf{let } \mathit{slice} \equiv \lambda X\colon \textbf{int} \to \textbf{int}, \textbf{int} \to \textbf{intacc}.\ \lambda l, u\colon \textbf{int}.\ (X, ll \equiv l, ul \equiv u)$$

Notice that the type list for X makes *slice* applicable to parameters of type **intarray** and **intaccarray** (and even to sequences and certain procedures), as well as **intvararray**. A safer alternative, which checks applications of the array against the new bounds, is

$$\begin{array}{l}\textbf{let } \mathit{errorvar} \equiv \lambda s\colon \textbf{charseq}.\ (\mathit{error}\ s, \lambda e\colon \textbf{ns}.\ \mathit{error}\ s)\ \textbf{in}\\ \textbf{let } \mathit{slicecheck} \equiv \lambda X\colon \textbf{int} \to \textbf{int}, \textbf{int} \to \textbf{intacc}.\ \lambda l, u\colon \textbf{int}.\\ \quad (\lambda k\colon \textbf{int}.\ \textbf{if } l \le k \wedge k \le u \textbf{ then } X\ k \textbf{ else}\\ \qquad \mathit{errorvar}\ \text{'SUBSCRIPT ERROR'},\\ \quad ll \equiv l, ul \equiv u)\end{array}$$

Here *errorvar* is a generalization of *error* whose calls can be used as an acceptor as well as an expression. (Note the use of **ns** as the parameter type of a procedure that ignores its parameter.)

To illustrate the use of these procedures, we develop a program for sorting by finding maxima. First we define a procedure that sets j to the subscript of a maximum of an array X:

$$\begin{array}{l}\textbf{let } \mathit{max} \equiv \lambda X\colon \textbf{intarray}.\ \lambda j\colon \textbf{intacc}.\\ \quad \mathit{newintvar}\ X.ll\ \lambda a.\ \mathit{newintvar}\ X.ul\ \lambda b.\\ \qquad \mathit{newintvarres}\ a\ j\ \lambda j.\\ \qquad\quad \textbf{while } a < b \textbf{ do } (a := a+1\ ;\ \textbf{if } X\ a > X\ j \textbf{ then } j := a \textbf{ else } \mathit{skip})\end{array}$$

(Here giving X the type **intarray** rather than **intvararray** indicates that *max* will examine X but not assign to it.) Next comes a procedure for exchanging a pair of array elements:

$$\begin{array}{l}\textbf{let } \mathit{exchange} \equiv \lambda X\colon \textbf{int} \to \textbf{intvar}.\ \lambda i, j\colon \textbf{int}.\\ \quad \mathit{newintvar}\ i\ \lambda i.\ \mathit{newintvar}\ j\ \lambda j.\\ \qquad \mathit{newintvar}(X\ i)\ \lambda t.\ (X\ i := X\ j\ ;\ X\ j := t)\end{array}$$

(Giving X the type **int** $\rightarrow$ **intvar** indicates that *exchange* does not evaluate bounds; e.g. it would also be applicable to an integer variable sequence.) Then the sort procedure can be defined by

let *maxsort* $\equiv \lambda X$: **intvararray**.
  newintvar X.ll λa. *newintvar X.ul* λb.
    while $a \leq b$ **do** *newintvar* 0 λj.
      $\big($*max*(*slice X a b*) j ; *exchange X j b* ; $b := b - 1\big)$

The above procedure contains a spurious initialization of the variable j. The purpose of this variable is to accept the result of *max*, but *newintvar* requires us to initialize it to some value before calling *max*. However, this unpleasantness can be avoided by taking advantage of the fact that assignments are really applications. By substituting the definition of *newintvarres* into the definition of *max* and reducing, we find that the definition of *max* is equivalent to

let *max* $\equiv \lambda X$: **intarray**. λj: **intacc**.
  newintvar X.ll λa. *newintvar X.ul* λb.
    newintvar a λ*local*.
      $\big($**while** $a < b$ **do**
        $(a := a + 1$; **if** $X\ a > X$ *local* **then** *local* $:= a$ **else** *skip*$)$;
      $j :=$ *local*$\big)$

where $j :=$ *local* is syntactic sugar for j *local*. Thus the second parameter of *max* can be any procedure of type **int** $\rightarrow$ **comm**, i.e. any proper procedure accepting an integer; the effect of *max* will be to apply this procedure to the subscript of the maximum of X.

To avoid the spurious initialization, we make this parameter a procedure that carries out the appropriate exchange, dispensing with the variable j entirely:

let *maxsort* $\equiv \lambda X$: **intvararray**.
  newintvar X.ll λa. *newintvar X.ul* λb.
    while $a \leq b$ **do**
      $\big($*max*(*slice X a b*) λj. *exchange X j b* ; $b := b - 1\big)$

A similar use of "generalized call by result" occurs in the following definition of quicksort:

```
let partition ≡ λX:intvararray. λr:int. λp:intacc.
    newintvar X.ll λc. newintvar X.ul λd. newintvar r λr.
        (while c ≤ d do
            if X c ≤ r then c := c + 1 else
            if X d > r then d := d − 1 else
                (exchange X c d ; c := c + 1 ; d := d − 1);
        p c)
in
letrec quicksort: intvararray → comm
where quicksort ≡ λX:intvararray.
    newintvar X.ll λa. newintvar X.ul λb.
        if a < b then
            if X a > X b then exchange X a b else skip;
            partition(slice X (a + 1) (b − 1)) ((X a + X b) ÷ 2) λc.
                (quicksort(slice X a (c − 1)) ; quicksort(slice X c b))
        else skip
```

11. Data Abstraction with Objects

Perhaps the most important way in which Forsythe is more general than Algol is in its provision of objects, which are a powerful tool for data abstraction. One can write abstract programs in which various kinds of data are realized by types of objects, and then encapsulate the representation of the data, and the expression of primitive operations in terms of this representation, in declarators for the objects.

To illustrate this style of programming, we will develop a program for computing reachability in a finite directed graph. Specifically, we will define a procedure *reachable* that, given a node x and a graph g, will compute the set of nodes that can be reached from x.

Throughout most of this development we will assume that "**node**" is a new data type; eventually we will see how this assumption can be eliminated. Then we can define a "**set**" to be an object denoting a finite set of nodes, whose fields are procedures for manipulating the denoted set:

lettype set ≡

$\big($*member*: **node** → **bool**

& *insertnew*: **node** → **comm**

& *iter*: (**node** → **comm**) → **comm**

& *pick*: **comm** → (**node** → **comm**) → **comm**$\big)$

The intention is that, if s is a set, x is a node, d is a procedure of type **node** → **comm**, and e is a command, then:

- $s.\,member\ x$ gives *true* if and only if $x \in s$.
- $s.\,insertnew\ x$ inserts x into s, providing x is not already in s.
- $s.\,iter\ d$ applies d to each member of s.
- If s is empty then $s.\,pick\ e\,d$ executes e; otherwise $s.\,pick\ e\,d$ removes an arbitrary member from s and applies d to the removed member.

In terms of **set**, we can give an initial "naive" version of the reachability procedure. The procedure maintains a set t of all nodes that have been found to be reachable from x, and a set u of those members of t whose immediate successors have yet to be added to t. (An immediate successor of a node y is a node that can be reached from y in one step.) Thus its invariant is

$$x \in t \wedge u \subseteq t \wedge (\forall y \in t)\ y \text{ is reachable from } x \wedge (\forall y \in t - u)\ g\,y \subseteq t\,,$$

where g is a function of type **node** → **set** such that $g\,y$ is the set of immediate successors of y. This invariant implies that, when u is empty, t is the set of all nodes reachable from x.

In writing *reachable*, we assume that the parameter g is the immediate-successor function of the graph, and that the result is to be communicated by applying a procedural parameter p to the final value of t:

```
let reachable ≡ λx: node. λg: node → set. λp: set → comm, set → compl.
    newset λt. newset λu.
        (t.insertnew x ; u.insertnew x ;
        escape λout.
            loop u.pick out λy: node.
                (g y).iter λz: node.
                    if ~ t.member z then
                        (t.insertnew z ; u.insertnew z)
                    else skip ;
        p t)
```

Here *newset* is a declarator that creates an object of type **set**, initialized to the empty set. Thus

$$newset : \big((\mathbf{set} \to \mathbf{comm}) \to \mathbf{comm}\ \&\ (\mathbf{set} \to \mathbf{compl}) \to \mathbf{compl}\big) .$$

Next, we refine the reachability procedure to provide greater flexibility for the representation of sets. In place of the object type **set**, we introduce different object types for the different sets used in the program:

- **setg** for the sets produced by applying g,
- **sett** for the set t,
- **setu** for the set u.

The basic idea is to limit the fields of each of these object types to those procedures that are actually needed by our program. However, even greater flexibility is gained by taking advantage of the fact that the sets t and u are declared at the same time, and that u is always a subset of t. For this purpose, we introduce a "double declarator",

$$newdoubleset : \big((\mathbf{sett} \to \mathbf{setu} \to \mathbf{comm}) \to \mathbf{comm}$$

$$\&(\mathbf{sett} \to \mathbf{setu} \to \mathbf{compl}) \to \mathbf{compl}\big) .$$

such that *newdoubleset* λt: **sett.** λu: **setu.** C executes C after binding both t and u to new (initially empty) sets. Morever, to enforce the invariant $u \subseteq t$, we will eliminate the operation *t.insertnew* and redefine *u.insertnew* to insert its argument (which must not already belong to t) into both u and t.

Thus we have

```
lettype setg ≡ (iter: (node → comm) → comm),
     sett ≡ (member: node → bool
          & iter: (node → comm) → comm),
     setu ≡ (insertnew: node → comm
          & pick: comm → (node → comm) → comm)
in
let reachable ≡ λx: node. λg: node → setg. λp: sett → comm, sett → compl.
     newdoubleset λt: sett. λu: setu.
          (u.insertnew x ;
          escape λout.
               loop u.pick out λy: node.
                    (g y).iter λz: node.
                         if ~t.member z then u.insertnew z else skip ;
          p t)
```

Notice that we have retained the *iter* field for objects of type **sett**, even though this procedure is never used in our program. The reason is that the result of *reachable* is an object of type **sett**, for which the user of *reachable* may need an iteration procedure.

Now we define the representation of t and u by programming *newdoubleset*. Within this declarator, we represent t by a characteristic vector c, which is a boolean variable array that is indexed by nodes, i.e. a procedure of type **node** → **boolvar**, such that

$$t = \{\, y \mid y\text{: node} \wedge c\, y = \mathit{true} \,\} .$$

We also represent both t and u by a node variable sequence w that (with the help of two integer variables a and b) enumerates the members of these sets without duplication. Specifically,

$$t = \{\, w\, k \mid 0 \le k < b \,\} ,$$
$$u = \{\, w\, k \mid a \le k < b \,\} .$$

Thus we have

```
let newdoubleset ≡ λp: sett → setu → comm, sett → setu → compl.
    newboolvarnodearray(λn: node. false) λc: node → boolvar.
    newnodevarseq N (λk: int. dummynode) λw: int → nodevar.
    newintvar 0 λa. newintvar 0 λb.
        p(member ≡ c,
            iter ≡ λd: node → comm. for 0 (b − 1) λk. d(w k))
          (insertnew ≡ λn: node. (c n := true ; w b := n ; b := b + 1),
            pick ≡ λe: comm. λd: node → comm.
                if a ≥ b then e else (a := a + 1 ; d(w a)))
```

Here N is an integer expression giving an upper bound on the number of nodes, and *dummynode* is an arbitrary entity of type **node** used to give a spurious initialization to w.

Next, we consider the representation of graphs. As far as *reachable* is concerned, a graph is simply its immediate-successor function, of type **node → setg**. But the part of the program that creates graphs must have some primitive procedure for graph construction. Thus we make **graph** an object type with a field named *addedge*, denoting a procedure that, given its source and destination nodes, adds an edge to the graph:

```
lettype graph ≡ (node → setg & addedge: node → node → comm)
```

Notice that the immediate-successor function is a "nameless" field of a graph, so that a graph can be passed directly to *reachable*.

We choose to represent a graph by an integer variable array *succlist*, indexed by nodes, such that *succlist* n is a list of the immediate successors of n. The lists are represented by a node variable sequence *car* and an integer variable sequence *cdr*. The integer variable k gives the number of active list cells. The empty list is represented by -1.

Thus the declarator for graphs is:

```
let newgraph ≡ λp: graph → comm, graph → compl.
    newintvarnodearray(λn: node. − 1) λsucclist.
    newnodevarseq E (λk: int. dummynode) λcar.
    newintvarseq E (λk: int. − 1) λcdr.
    newintvar 0 λk.
    p(λn: node. (iter ≡ λd: node → comm.
            newintvar(succlist n) λl.
                while l ≠ −1 do (d(car l) ; l := cdr l)),
        addedge ≡ λm, n: node.
            (car k := n ; cdr k := succlist m ; succlist m := k ; k := k + 1))
```

Here E is an upper bound on the number of edges in the graph.

Next, we consider extending our program so that, in addition to determining the set of nodes that can be reached from x, it computes paths from x to each of these nodes. We will alter *reachable* so that it gives its parameter p an additional argument r of type **paths**, where an object of type **paths** provides two procedures for iterating over paths in forward and backward directions:

lettype paths $\equiv$ $\big($*forward*, *backward*: **node** $\rightarrow$ (**node** $\rightarrow$ **comm**) $\rightarrow$ **comm**$\big)$

If r is an object of type **paths**, y is a node reachable from x, and d is a procedure of type **node** $\rightarrow$ **comm**, then $r.forward\ y\ d$ or $r.backward\ y\ d$ will apply d to each node on the path from x to y.

Within *reachable*, each time an immediate successor z of y is inserted in t and u, the path to z formed by adding z to the already known path to y will be recorded in r. Thus, within *reachable*, r will have the type

lettype pathsvar $\equiv$ (**paths** & *record*: **node** $\rightarrow$ **node** $\rightarrow$ **comm**)

where $r.record$ is a procedure such that $r.record\ z\ y$ records the path to z formed by adding z to the path to y. (Notice the terminology here: although an object of type **pathsvar** cannot be assigned to in the conventional sense, it still consists of an object of type **paths** conjoined with an operation that changes the state of the object.)

The new version of *reachable* is:

```
let reachable ≡ λx: node. λg: node → setg.
    λp: sett → paths → comm, sett → paths → compl.
    newdoubleset λt: sett. λu: setu. newpathsvar λr: pathsvar.
        (u.insertnew x ;
        escape λout.
            loop u.pick out λy: node.
                (g y).iter λz: node.
                    if ~t.member z then
                        u.insertnew z ; r.record z y
                    else skip ;
        p t r)
```

The representation of paths is defined within the declarator for **pathsvar**. A node variable array *link*, indexed by nodes, is used to record the calls of *record*, so that $link\ z = y$ holds after a call of $r.record\ z\ y$. Then *forward* scans *link* recursively, while *backward* scans *link* iteratively:

```
let newpathsvar ≡ λp: pathsvar → comm, pathsvar → compl.
    newnodevarnodearray(λn: node. dummynode) λlink: node → nodevar.
    p(record ≡ λz, y: node. link z := y,
        forward ≡ λn: node. λd: node → comm.
            letrec scan: node → comm
            where scan ≡ λn: node. newnodevar n λn.
                if eqnode n x then d x else (scan(link n) ; d n)
            in scan n,
        backward ≡ λn: node. λd: node → comm.
            newnodevar n λn.
            (while ~ eqnode n x do (d n ; n := link n) ; d x))
```

Here *eqnode* is a primitive operation for comparing nodes.

Finally, we must define the data type **node**. Forsythe lacks facilities for defining new data types, but the effect of a new data type can be obtained by defining the relevant phrase types, primitive operations, and declarators. This is easy if we use a trivial representation, where a node is represented by an integer n such that $0 \leq n < N$:

```
lettype node ≡ int,
    nodeacc ≡ intacc,
    nodevar ≡ intvar
in
let dummynode ≡ −1,
    eqnode ≡ λm, n: node. m = n,
    newnodevar ≡ newintvar,
    newnodevarseq ≡ newintvarseq,
    newboolvarnodearray ≡ newboolvarseq N,
    newintvarnodearray ≡ newintvarseq N,
    newnodevarnodearray ≡ newintvarseq N
```

Unfortunately, this way of defining **node** is limited by the fact that **lettype** definitions are transparent rather than opaque. Thus typechecking would not detect an erroneous operation that treated nodes as integers.

To avoid this difficulty, one would like to have opaque type definitions in Forsythe. However, even in the absence of opaque definitions, one can still achieve a degree of data abstraction by defining nodes to be one-field objects containing integers, rather than "raw" integers. This approach assures that the integrity of the abstraction **node** will

not be violated by the reachability program providing this program does not contain any occurrence of the field name used in the definition of **node**.

This approach is embodied in the following definitions, in which *nn* is used as the "secret" field name:

lettype node $\equiv$ (*nn*: **int**) **in**
lettype nodeacc $\equiv$ **node** $\rightarrow$ **comm in**
lettype nodevar $\equiv$ (**node** & **nodeacc**) **in**
lettype nodevarseq $\equiv$ (**int** $\rightarrow$ **nodevar** & *len*: **int**),
boolvarnodearray $\equiv$ **node** $\rightarrow$ **boolvar**,
intvarnodearray $\equiv$ **node** $\rightarrow$ **intvar**,
nodevarnodearray $\equiv$ **node** $\rightarrow$ **nodevar**
in
let *dummynode* $\equiv (nn \equiv -1)$,
eqnode $\equiv \lambda m, n$: **node**. $m.nn = n.nn$,
newnodevar $\equiv \lambda init$: **node**.
λb: **nodevar** $\rightarrow$ **comm**, **nodevar** $\rightarrow$ **compl**.
newintvar $(init.nn)\ \lambda x$.
$b(nn \equiv x, \lambda m$: **node**. $x := m.nn)$,
newnodevarseq $\equiv \lambda l$: **int**. $\lambda init$: **int** $\rightarrow$ **node**.
λb: **nodevarseq** $\rightarrow$ **comm**, **nodevarseq** $\rightarrow$ **compl**.
newintvarseq $l\ \big(\lambda k.\ (init\ k).nn\big)\ \lambda x$.
$b\big(\lambda k$: **int**. $(nn \equiv x\ k, \lambda m$: **node**. $x\ k := m.nn), len \equiv x.len\big)$
in
let *newboolvarnodearray* $\equiv \lambda init$: **node** $\rightarrow$ **bool**.
λb: **boolvarnodearray** $\rightarrow$ **comm**, **boolvarnodearray** $\rightarrow$ **compl**.
newboolvarseq $N\ \big(\lambda k.\ init(nn \equiv k)\big)\ \lambda x$.
$b\big(\lambda n$: **node**. $x(n.nn)\big)$,
newintvarnodearray $\equiv \lambda init$: **node** $\rightarrow$ **int**.
λb: **intvarnodearray** $\rightarrow$ **comm**, **intvarnodearray** $\rightarrow$ **compl**.
newintvarseq $N\ \big(\lambda k.\ init(nn \equiv k)\big)\ \lambda x$.
$b\big(\lambda n$: **node**. $x(n.nn)\big)$,
newnodevarnodearray $\equiv \lambda init$: **node** $\rightarrow$ **node**.
λb: **nodevarnodearray** $\rightarrow$ **comm**, **nodevarnodearray** $\rightarrow$ **compl**.
newnodevarseq $N\ \big(\lambda k.\ init(nn \equiv k)\big)\ \lambda x$.
$b\big(\lambda n$: **node**. $x(n.nn)\big)$

12. Possible Extensions of Forsythe

There are a number of directions in which it would be desirable to extend Forsythe, providing such extensions do not impact the uniformity of the language. We are currently investigating, or hope to investigate, the following possibilities:

- Sums or disjunctions of phrase types. Unfortunately, sums of phrase types interact with the conditional construction in a counterintuitive manner. Suppose, for example, that **comm** + **int** is the binary sum of **comm** and **int**, with injection operations in_1 of type **comm** $\rightarrow$ (**comm** + **int**) and in_2 of type **int** $\rightarrow$ (**comm** + **int**), and a case operation $\oplus$ such that, for any phrase type θ, if p_1 has type **comm** $\rightarrow \theta$ and p_2 has type **int** $\rightarrow \theta$ then $p_1 \oplus p_2$ has type (**comm** + **int**) $\rightarrow \theta$, with the reduction rule

  $$(p_1 \oplus p_2)(in_i\ x) \quad \Longrightarrow \quad p_i\ x\ .$$

  If application of the conditional construction to sum types is to make sense, then the reduction

  $$(p_1 \oplus p_2)(\textbf{if}\ b\ \textbf{then}\ in_1\ c\ \textbf{else}\ in_2\ e) \quad \Longrightarrow \quad \textbf{if}\ b\ \textbf{then}\ p_1\ c\ \textbf{else}\ p_2\ e$$

  must hold. Then, if the language is to exhibit reasonably uniform behavior, the similar reduction

  $$(p_1 \oplus p_2)(\textbf{if}\ b\ \textbf{then}\ in_1\ c\ \textbf{else}\ in_1\ c') \quad \Longrightarrow \quad \textbf{if}\ b\ \textbf{then}\ p_1\ c\ \textbf{else}\ p_1\ c'$$

  must also hold. But then, the reduction

  $$in_1(\textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ c') \quad \Longrightarrow \quad \textbf{if}\ b\ \textbf{then}\ in_1\ c\ \textbf{else}\ in_1\ c'$$

  cannot hold, for otherwise we would have both

  $$(p_1 \oplus p_2)\big(in_1(\textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ c')\big) \quad \Longrightarrow \quad p_1(\textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ c')$$

  and

  $$(p_1 \oplus p_2)\big(in_1(\textbf{if}\ b\ \textbf{then}\ c\ \textbf{else}\ c')\big) \quad \Longrightarrow \quad (p_1 \oplus p_2)(\textbf{if}\ b\ \textbf{then}\ in_1\ c\ \textbf{else}\ in_1\ c')$$
  $$\Longrightarrow \quad \textbf{if}\ b\ \textbf{then}\ p_1\ c\ \textbf{else}\ p_1\ c'\ ,$$

  which reduce the same phrase to two phrases that will have different meanings if the procedure p_1 changes the value of b before executing its parameter. In particular, the falsity of the reduction rule for injections and conditionals implies that injections cannot be treated as implicit conversions.

- Polymorphic or universally quantified phrase types [10], possibly with bounded quantification in the sense of [2]. In addition to providing polymorphic procedures, this extension would also provide opaque type definitions. It is tempting to regard universally quantified types as infinite conjunctions, so that, for example, the quantified type $\Delta t.\ t \rightarrow t$ would be a subtype of every type of the form $\theta \rightarrow \theta$. However, this approach seems to make type checking infeasible.

- Recursively defined phrase types.
- Enriched data types. Although the data types of Algol (and so far of Forsythe) are limited to primitive, unstructured types, there would be no inconsistency in providing a much richer variety of data types. The real question is which of the many possible enrichments would provide additional expressive power without degrading efficiency of execution.
- Coroutines.
- Alternative treatment of arrays. Array facilities along the lines of those described in [4] would serve to avoid spurious array initializations, such as the initialization of w in *newdoubleset* in the section on Data Abstraction with Objects. But it is not clear how this approach can be extended to encompass multidimensional arrays.

On the other hand, there is also a direction in which it might be fruitful to restrict Forsythe: to impose syntactic restrictions so that one can determine syntactically that certain phrases do not interfere with one another. (Two phrases *interfere* if their concurrent execution is indeterminate. For example, aliased variables interfere, as do procedures that assign to the same global variables.) Such a restriction would open the door for the concurrent, yet determinate, execution of noninterfering commands, as well as for a form of block expression (in the sense of Algol W) that is restricted to avoid side effects.

A decade ago, I wrote a paper [9] proposing a scheme for restricting Algol-like languages for this purpose. At the time, certain syntactic anomalies (described in the final section of [9]) discouraged me from pursuing the matter further. But it is now clear that these anomalies are avoided by the use of a conjunctive type discipline, which allows nonsensical phrases to occur within sensible ones in contexts in which the nonsensical phrases are ignored. Moreover, it appears that this approach does not raise insuperable type-checking complications.

However, the syntactic disciple described in [9] is too restrictive. For example, one cannot regard $a := e$ as $a\ e$ when a and e interfere; nor can one write *newintvar init b* when *init* and b interfere. We are currently searching for a more general scheme that would avoid these difficulties.

APPENDICES

A Concrete Syntax

We will specify the concrete syntax of Forsythe by giving a context-free grammar that defines the set of "parsable" programs as strings of tokens. (We omit defining tokens, which are fairly similar to the basic symbols of Algol W [7].) Notice that a parsable program is not necessarily typable; typing is specified by the inference rules given previously and is independent of the concrete syntax.

The one novelty of this syntax is its treatment of "heavy prefixes" such as the conditional phrase. Such phrases are permitted to follow operators even when those operators have high precedence. For example one can write

$$A \times \textbf{if } B \textbf{ then } C \textbf{ else } D + E$$

instead of

$$A \times (\textbf{if } B \textbf{ then } C \textbf{ else } D + E)\,.$$

To illustrate the treatment of heavy prefixes, consider augmenting a simple language of arithmetic expressions,

⟨factor⟩ ::= ⟨id⟩ | (⟨expression⟩)
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ × ⟨factor⟩
⟨expression⟩ ::= ⟨term⟩ | ⟨term⟩ + ⟨expression⟩

with a conditional expression, treated as a heavy prefix. The resulting grammar is:

⟨factor⟩ ::= ⟨id⟩ | (⟨general expression⟩)
⟨heavy factor⟩ ::= **if**⟨general expression⟩ **then**⟨general expression⟩ **else**
⟨general expression⟩
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ × ⟨factor⟩
⟨heavy term⟩ ::= ⟨heavy factor⟩ | ⟨term⟩ × ⟨heavy factor⟩
⟨expression⟩ ::= ⟨term⟩ | ⟨term⟩ + ⟨expression⟩
⟨heavy expression⟩ ::= ⟨heavy term⟩ | ⟨term⟩ + ⟨heavy expression⟩
⟨general expression⟩ ::= ⟨expression⟩ | ⟨heavy expression⟩

In this simple example, a phrase beginning with a heavy prefix will extend to the next right parenthesis (or to the end of the text). In the actual syntax of Forsythe, such phrases extend to the next semicolon, comma, right parenthesis, right bracket, or *end*.

The grammar of Forsythe is given by the following productions, in which we use ⟨id⟩ to denote identifiers, ⟨type *n*⟩ to denote type expressions, ⟨p *n*⟩ to denote phrases, and ⟨hp *n*⟩ to denote heavy phrases. The integer *n* indicates the precedence level (with small *n* for high precedence).

⟨type id⟩ ::= ⟨id⟩
⟨id list⟩ ::= ⟨id⟩ | ⟨id list⟩, ⟨id⟩

⟨type 0⟩ ::= ⟨type id⟩ | (⟨type 3⟩) | [⟨type 3⟩] | **begin**⟨type 3⟩ **end**
⟨type 1⟩ ::= ⟨type 0⟩ | ⟨type 0⟩ → ⟨type 1⟩
⟨type 2⟩ ::= ⟨type 1⟩ | ⟨id list⟩ : ⟨type 2⟩
⟨type 3⟩ ::= ⟨type 2⟩ | ⟨type 3⟩ & ⟨type 2⟩

⟨type list⟩ ::= ⟨type 1⟩ | ⟨type list⟩, ⟨type 1⟩
⟨id type list⟩ ::= ⟨id list⟩ : ⟨type 1⟩ | ⟨id type list⟩, ⟨id list⟩ : ⟨type 1⟩
⟨type def list⟩ ::= ⟨type id⟩ ≡ ⟨type 1⟩ | ⟨type def list⟩, ⟨type id⟩ ≡ ⟨type 1⟩

⟨p 0⟩ ::= ⟨id⟩ | ⟨int const⟩ | ⟨real const⟩ | ⟨char const⟩ | ⟨string⟩
    |(⟨p 16⟩) | [⟨p 16⟩] | **begin**⟨p 16⟩ **end**
⟨hp 0⟩ ::= **if**⟨p 16⟩ **then**⟨p 16⟩ **else**⟨p 13⟩
    | **while**⟨p 16⟩ **do**⟨p 13⟩
    | **loop**⟨p 13⟩
    |λ⟨id list⟩ : ⟨type list⟩. ⟨p 13⟩
    |λ⟨id list⟩. ⟨p 13⟩
    | **let**⟨def list⟩ **in**⟨p 13⟩
    | **letrec**⟨id type list⟩ **where**⟨def list⟩ **in**⟨p 13⟩
    | **lettype**⟨type def list⟩ **in**⟨p 13⟩

⟨p 1⟩ ::= ⟨p 0⟩ | ⟨p 1⟩.⟨id⟩
⟨hp 1⟩ ::= ⟨hp 0⟩
⟨p 2⟩ ::= ⟨p 1⟩ | ⟨p 2⟩⟨p 1⟩ | **rec**⟨p 1⟩
    | **seq**(⟨p list⟩) | **seq**[⟨p list⟩] | **seq begin**⟨p list⟩ **end**
⟨hp 2⟩ ::= ⟨hp 1⟩ | ⟨p 2⟩⟨hp 1⟩ | **rec**⟨hp 1⟩

⟨p 3⟩ ::= ⟨p 2⟩ | ⟨p 3⟩⟨exp op⟩⟨p 2⟩
⟨hp 3⟩ ::= ⟨hp 2⟩ | ⟨p 3⟩⟨exp op⟩⟨hp 2⟩
⟨exp op⟩ ::= ↑ | **
⟨p 4⟩ ::= ⟨p 3⟩ | ⟨p 4⟩⟨mult op⟩⟨p 3⟩
⟨hp 4⟩ ::= ⟨hp 3⟩ | ⟨p 4⟩⟨mult op⟩⟨hp 3⟩
⟨mult op⟩ ::= × | / | ÷ | **rem**
⟨p 5⟩ ::= ⟨p 4⟩ | ⟨add op⟩⟨p 4⟩ | ⟨p 5⟩⟨add op⟩⟨p 4⟩
⟨hp 5⟩ ::= ⟨hp 4⟩ | ⟨add op⟩⟨hp 4⟩ | ⟨p 5⟩⟨add op⟩⟨hp 4⟩
⟨add op⟩ ::= + | −

⟨p 6⟩ ::= ⟨p 5⟩ | ⟨p 5⟩⟨rel op⟩⟨p 5⟩
⟨hp 6⟩ ::= ⟨hp 5⟩ | ⟨p 5⟩⟨rel op⟩⟨hp 5⟩
⟨rel op⟩ ::= = | ≠ | ≤ | < | ≥ | >

⟨p 7⟩ ::= ⟨p 6⟩ | ∼⟨p 6⟩
⟨hp 7⟩ ::= ⟨hp 6⟩ | ∼⟨hp 6⟩
⟨p 8⟩ ::= ⟨p 7⟩ | ⟨p 8⟩ ∧ ⟨p 7⟩
⟨hp 8⟩ ::= ⟨hp 7⟩ | ⟨p 8⟩ ∧ ⟨hp 7⟩
⟨p 9⟩ ::= ⟨p 8⟩ | ⟨p 9⟩ ∨ ⟨p 8⟩
⟨hp 9⟩ ::= ⟨hp 8⟩ | ⟨p 9⟩ ∨ ⟨hp 8⟩
⟨p 10⟩ ::= ⟨p 9⟩ | ⟨p 10⟩ ⇒ ⟨p 9⟩
⟨hp 10⟩ ::= ⟨hp 9⟩ | ⟨p 10⟩ ⇒ ⟨hp 9⟩
⟨p 11⟩ ::= ⟨p 10⟩ | ⟨p 11⟩ ⇔ ⟨p 10⟩
⟨hp 11⟩ ::= ⟨hp 10⟩ | ⟨p 11⟩ ⇔ ⟨hp 10⟩
⟨p 12⟩ ::= ⟨p 11⟩ | ⟨p 11⟩ := ⟨p 11⟩
⟨hp 12⟩ ::= ⟨hp 11⟩ | ⟨p 11⟩ := ⟨hp 11⟩
⟨p 13⟩ ::= ⟨p 12⟩ | ⟨hp 12⟩

⟨p 14⟩ ::= ⟨p 13⟩ | ⟨p 14⟩ ; ⟨p 13⟩
⟨p 15⟩ ::= ⟨p 14⟩ | ⟨id⟩ ≡ ⟨p 15⟩
⟨p 16⟩ ::= ⟨p 15⟩ | ⟨p 16⟩, ⟨id⟩ ≡ ⟨p 15⟩
    |⟨p 16⟩, λ⟨id list⟩ : ⟨type list⟩. ⟨p 13⟩
    |⟨p 16⟩, λ⟨id list⟩. ⟨p 13⟩

⟨p list⟩ ::= ⟨p 15⟩ | ⟨p list⟩, ⟨p 15⟩
⟨def list⟩ ::= ⟨id⟩ ≡ ⟨p 15⟩ | ⟨def list⟩, ⟨id⟩ ≡ ⟨p 15⟩

B Canonical Types

The subtype relation can be explicated by viewing types as standing for finite sets of "canonical types". A canonical type is one of the following:

1. A primitive type,
2. $\sigma \to \omega$, where σ is a finite set of canonical types and ω is a canonical type,
3. $\iota{:}\,\omega$, where ω is a canonical type.

The subtype preorder is defined for both canonical types and finite sets of canonical types by the following simultaneous recursive definition: For canonical types, $\omega \leq \omega'$ iff one of the following holds:

1. There are primitive types ρ and ρ' such that $\omega = \rho$, $\omega' = \rho'$, and $\rho \leq_{\text{prim}} \rho'$,
2. There are σ_1, σ_1', ω_2, and ω_2' such that $\omega = \sigma_1 \to \omega_2$, $\omega' = \sigma_1' \to \omega_2'$, $\sigma_1' \leq \sigma_1$, and $\omega_2 \leq \omega_2'$,
3. There are ι, σ_1, and σ_1' such that $\omega = \iota{:}\,\omega_1$, $\omega' = \iota{:}\,\omega_1'$, and $\omega_1 \leq \omega_1'$.

For finite sets of canonical types, $\sigma \leq \sigma'$ iff

$$(\forall \omega' \in \sigma')(\exists \omega \in \sigma)\, \omega \leq \omega' .$$

The function $(-)^*$ maps types into finite sets of canonical types as follows:

$$\rho^* = \{\rho\}$$

$$(\iota{:}\,\theta)^* = \{\, \iota{:}\,\omega \mid \omega \in \theta^* \,\}$$

$$(\theta_1 \to \theta_2)^* = \{\, \theta_1^* \to \omega_2 \mid \omega_2 \in \theta_2^* \,\}$$

$$\mathbf{ns}^* = \{\}$$

$$(\theta_1 \,\&\, \theta_2)^* = \theta_1^* \cup \theta_2^* .$$

This function can be used to give an alternative definition of the subtype relation on types that is equivalent to that given in the previous section:

$$\theta \leq \theta' \text{ iff } \theta^* \leq \theta'^* .$$

References

[1] Cardelli, L. *A Semantics of Multiple Inheritance.* In: **Semantics of Data Types**, edited by G. Kahn, D. B. MacQueen, and G. D. Plotkin. **Lecture Notes in Computer Science**, vol. 173, Springer-Verlag, Berlin, 1984, pp. 51–67.

[2] Cardelli, L. and Wegner, P. *On Understanding Types, Data Abstraction, and Polymorphism.* **ACM Computing Surveys**, vol. 17 (1985), pp. 471–522.

[3] Coppo, M., Dezani-Ciancaglini, M., and Venneri, B. *Functional Characters of Solvable Terms.* **Zeitschrift für Mathematische Logik und Grundlagen der Mathematik**, vol. 27 (1981), pp. 45–58.

[4] Dijkstra, E. W. **A Discipline of Programming.** Prentice-Hall, Englewood Cliffs, 1976, xviii+277 pp.

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

[6] Oles, F. J. *Type Algebras, Functor Categories, and Block Structure.* In: **Algebraic Methods in Semantics**, edited by M. Nivat and J. C. Reynolds. Cambridge University Press, 1985, pp. 543–573.

[7] Reynolds, J. C. **The Craft of Programming.** Prentice-Hall International, London, 1981.

[8] Reynolds, J. C. *The Essence of Algol.* In: **Algorithmic Languages**, edited by J. W. de Bakker and J. C. van Vliet. North-Holland, Amsterdam, 1981, pp. 345–372.

[9] Reynolds, J. C. *Syntactic Control of Interference.* In: **Conference Record of the Fifth ACM Symposium on Principles of Programming Languages**, Tucson. 1978, pp. 39–46.

[10] Reynolds, J. C. *Towards a Theory of Type Structure.* In: **Proceedings, Colloque sur la Programmation. Lecture Notes in Computer Science**, vol. 19, Springer-Verlag, Berlin, 1974, pp. 408–425.