This paper is based on lectures given at the DARPA/ISTO Software Principal Investigators Meeting, Providence, Rhode Island, February 27, 1991, and the University of Illinois, April 4, 1991.

Replacing Complexity with Generality: The Programming Language Forsythe

John C. Reynolds*
Carnegie Mellon University

April 12, 1991

Abstract

By using concepts of modern programming-language theory, such as higher-order procedures and intersection types, it is possible to design a programming language that is extremely general without being complicated. This thesis is illustrated by the Algol-like language Forsythe.

*This research was sponsored in part by National Science Foundation Grant CCR-8922109 and in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

The present-day variety of programming languages spreads across a spectrum from the "commercial" to the "theoretical". On the commercial side, the primary design goal is functionality: The designer is confronted by prescribed applications that must be expressible in the language, and this requirement inevitably takes precedence over such goals as readability, reliability, or ease of use. Inevitably, the result is a complicated language, with design quirks such as strange binding, leaky type systems, failures of abstraction, weak and complicated procedure mechanisms, mismatched implicit conversions and generic operations, aliasing and interference, concurrency without enforced critical regions, or language features that simply cannot be explained clearly — all of which are a rich source of programming errors.

At the theoretical end of the spectrum, in constrast, the primary design goals are simplicity and uniformity. In the past, the achievement of these goals has seriously limited functionality. At present, however, there are reasons to hope that simplicity and uniformity can coexist with enough functionality to satisfy a wide class of users. Recent research has yielded more flexible type systems and better implementation techniques. At the same time, as hardware has become cheaper and more powerful, minor inefficiencies of execution (which are often the only real cost of limited functionality) have become an acceptable price for avoiding programming complexity.

In this paper, we will use the programming language Forsythe to illustrate the extent to which generality can be achieved without sacrificing simplicity and uniformity, by using appropriate theory to unify superficially diverse language features. 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.

Before considering Forsythe in particular, we specify its location in the design space of programming languages. As illustrated in Figure 1, Forsythe lies on one side of each of three fundamental dichotomies in language design. First, it is a typed language. It has long been understood that imposing a type discipline can yield major improvements in compile-time error detection and in the efficiency of run-time data representations. But type systems that are flexible enough to support sophisticated programming techniques are a much more recent development.

Second, Forsythe has imperative features (i.e. assignment and control flow) as well as a powerful procedure mechanism. Like all such languages, it suffers from the problems of aliasing and interference. However, we are convinced that imperative programming is a fundamental paradigm that should not be ignored in programming language design.

Finally, Forsythe uses call by name rather than call by value. For purely functional languages this is merely a distinction between orders of evaluation, but for languages with imperative features it is a fundamental dichotomy in the way that the imperative and functional aspects are linked; one is tempted to speak of Algol-like versus Scheme-like languages.

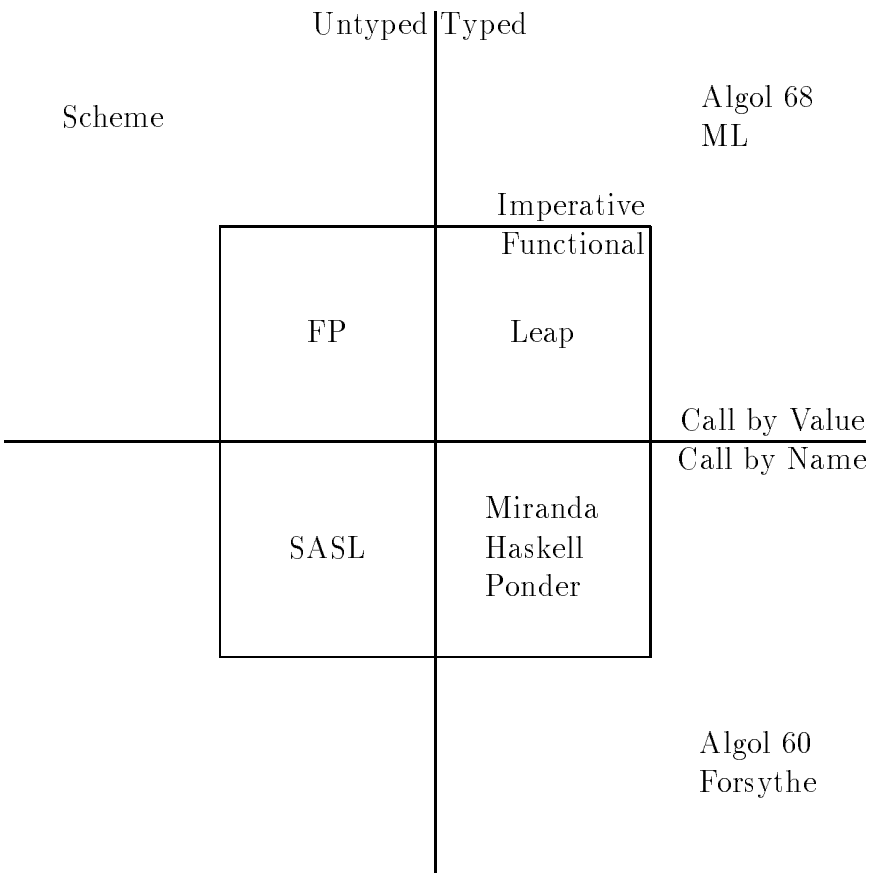


Figure 1: A Classification of Programming Languages

In any event, the following basic view, which is implicit in Algol 60, underlies Forsythe and distinguishes it from such languages as Algol 68, Scheme, and ML: The programming language is a typed lambda calculus with a primitive type **command**, such that terms of this type, when reduced to normal form, are programs in the simple imperative language. Thus a program, which must be a term of type **command**, is executed in two phases. First the program is reduced to normal form. (In Algol jargon, the copy rule is repeatedly applied to eliminate procedure calls.) Then the resulting simple imperative program is executed.

Reduction of lambda expressions (Copy Rule)

↓ normal form

Execution of commands in the simple imperative language

The only complication is that, in the presence of recursion, the reduction phase may go on forever, producing an infinite "head-normal" term. Nevertheless, such an infinite term can still be viewed as a simple imperative program; operationally, one simply implements the two phases as coroutines.

Even in this more general situation, the above diagram still describes an essential restriction on the flow of information: Nothing that happens in the second phase ever affects anything that happens in the first phase. Thus Forsythe inherits the basic property of the lambda calculus that meaning does not depend upon the order or timing of reductions. Indeed, reduction rules can be viewed as equations satisfied by the language.

In contrast, consider the situation in a language such as Scheme or ML that provides assignable function variables. If f is such a variable, then the effect of reducing $f(\cdots)$ will depend upon when the reduction occurs relative to the sequence of assignments to f that are executed in the imperative phase. In this situation, the procedure mechanism is completely stripped of its functional character.

Just as there are two execution phases for Forsythe, so there are two kinds of type. The imperative aspect of the language is characterized by *data types*,

$$\textbf{int}(\text{eger}),\ \textbf{real},\ \textbf{bool}(\text{ean}),\ \textbf{char}(\text{acter})\,,$$

which denote sets of values taken on by variables and expressions. On the other hand, the types of the typed lambda calculus are called *phrase types*, since they denote sets of meanings taken on by various kinds of phrases. (The unqualified word "type" will always refer to phrase types.)

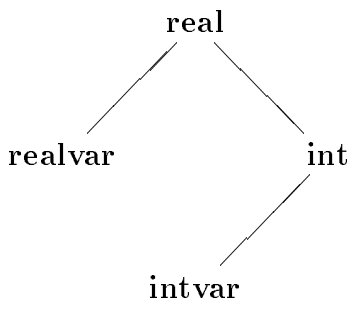
The primitive or ground phrase types correspond to kinds of phrases of the simple imperative language, such as

$$\textbf{int}\ (\text{expression}),\ \textbf{intvar}(\text{iable}),\ \ldots\ ,\ \textbf{comm}(\text{and})\,.$$

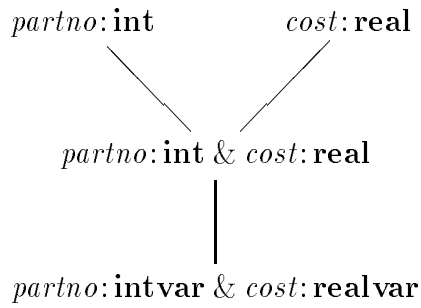
In addition, there are phrase types that denote procedures and objects. For example, $\theta \to \theta'$ is the type of a procedure that accepts an argument of type θ and whose call has type θ', and $\iota_1{:}\theta_1\ \&\ \cdots\ \&\ \iota_n{:}\theta_n$ is the type of an object with fields (or methods) named $\iota_1, \ldots, \iota_n$ of types $\theta_1, \ldots, \theta_n$.

Phrase types are ordered by a subtype relation. We say θ is a *subtype* of θ', and write $\theta \leq \theta'$, when there is an implicit conversion from meanings of type θ to meanings of type θ' or, viewed syntactically, when any phrase of type θ can be used in any context requiring a phrase of type θ'.

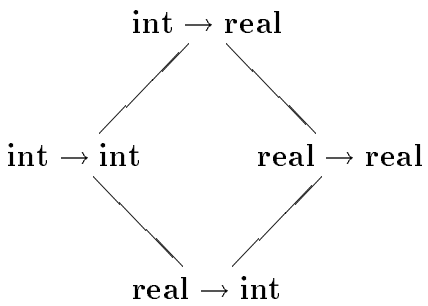
Among primitive types, the subtype relation indicates that variables can be used as expressions, and that integer expressions can be used as real expressions, e.g.



Objects can be implicitly converted by converting their fields, and also by forgetting fields, e.g.



Procedures can be implicitly converted by converting either their arguments or their calls, e.g.



More generally, procedural types satisfy

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

Having laid the necessary groundwork, we now embark on a brief tour of Forsythe with an unusual emphasis: instead of describing the features of the language and why they are useful, we will spend more of our time describing the features that are missing and why they are unnecessary.

An obvious way in which Forsythe is simpler and more uniform than many languages with imperative procedures is its use of lambda notation to denote procedures. Instead of

begin real procedure *poly*(x:**real**) ; $x \times x + 1$; ... **end** ,

in Forsythe we write

let *poly*:**real** $\to$ **real** $\equiv \lambda x.\ x \times x + 1$ **in** ... .

Here the *let* construction shows that the identifier *poly* stands for something of type **real** $\to$ **real**, while the lambda construction shows that the "something" is a particular procedure. (Notice that the formal parameter x is bound on the right of $\equiv$, making it clear that the *let construction* does not make x stand for anything.)

Proper procedures behave similarly; instead of

begin procedure *inc*(n:**integer**) ; $n := n + 1$; ... **end** ,

we write

$$\textbf{let } inc\text{:}\textbf{intvar} \to \textbf{comm} \equiv \lambda n.\ n := n + 1 \textbf{ in } \ldots\ .$$

Here the argument type is **intvar** since the parameter n is used as a integer variable, i.e. on both sides of the assignment $n := n + 1$, and the "result" type is **comm** since calls of *inc* are commands. (Strictly speaking, there is no result.)

Lambda notation is particularly useful for arguments of higher-order procedures. In Algol and most of its descendents, to use an integration procedure **integrate** to compute $\int_0^1 dx\ x^2 + 1$, one would have to write something like

begin real procedure $integrand(x\text{:}\textbf{real})\ ;\ x \times x + 1;$
$z := integrate(0, 1, integrand)$
end .

But in Forsythe, one can describe the function to be integrated directly as an argument to **integrate**:

$$z := integrate\ 0\ 1\ \lambda x.\ x \times x + 1\ ,$$

where *integrate* has type $\textbf{real} \to \textbf{real} \to (\textbf{real} \to \textbf{real}) \to \textbf{real}$. (Procedure application is left-associative, while $\to$ is right-associative). This illustrates that Forsythe does not provide procedures with multiple arguments; instead one uses procedures whose calls are procedures.

There are several benefits of regarding variables (of each data type) as a type. Since the conditional construction is applicable to all types, there are conditional variables that can be written on either side of an assignment command:

$$z := \textbf{if } p \textbf{ then } x \textbf{ else } y\ ;\ \ldots\ ;\ (\textbf{if } p \textbf{ then } x \textbf{ else } y) := z\ .$$

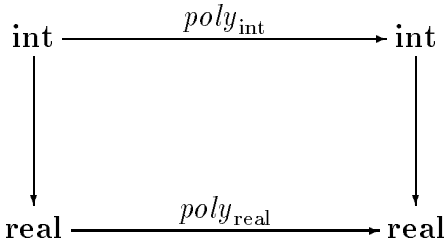
More important, an array is just a procedure whose calls are variables. For example, given a two-dimensional real array X, which would have type $\textbf{int} \to \textbf{int} \to \textbf{realvar}$, one could define its diagonal by a lambda expression, as in

$$\textbf{let } D\text{:}\textbf{int} \to \textbf{realvar} \equiv \lambda n.\ X\ n\ n \textbf{ in } D(k) := D(k) + 1\ .$$

Technically, the most innovative feature of Forsythe is its use of *intersection* types (sometimes called *conjunctive types*), a rich type discipline that was originally developed in the study of the lambda calculus [1]. By giving a phrase the type $\theta_1\ \&\ \theta_2$ one can indicate that it simultaneously has both of the types θ_1 and θ_2. Thus, for example, since the operators $+$ and $\times$ both map integers into integers and reals into reals, one can define procedures with the same property:

$$\textbf{let } poly\text{:}(\textbf{int} \to \textbf{int})\ \&\ (\textbf{real} \to \textbf{real}) \equiv \lambda x.\ x \times x + 1 \textbf{ in } poly(3) + poly(3.5)\ .$$

However, this is not the usual kind of ad hoc generic procedure facility; the semantics of Forsythe guarantees that the integer and real versions of *poly* respect the implicit conversion from integers to reals:



The use of intersection types permits substantial simplifications of the type structure of Forsythe. Variables can be defined to be the intersection of expressions, which produce values and are used on the right of assignment, and *acceptors*, which accept values and are used on the left of assignments:

intvar $\stackrel{\text{def}}{=}$ **int** & **intacc**(eptor) .

Moreover, the types $\iota_1{:}\theta_1 \ \& \ \cdots \ \& \ \iota_n{:}\theta_n$ of multiple-field objects are just intersections of types of single-field objects.

Intersection types can also be used to describe entities that are simultaneously procedures and objects, so that they can be applied to arguments and also subject to field selection. For example, one can have "generalized arrays" that behave like arrays when applied to subscripts, but also yield bounds information by field selection. The simplest case plays a special role in Forsythe: A *θ-sequence* of length n is both a θ-returning procedure that can be applied to the integers between 0 and $n-1$ inclusive, and an object with a field named *len* whose value is n. Such a sequence has the type θ**seq** $\stackrel{\text{def}}{=}$ (**int** $\to \theta$) & *len*:**int**. For example, a **realvarseq**(uence) is a one-dimensional real array whose lower limit is zero.

Another major novelty is the use of higher-order procedures to declare variables and arrays. Instead of

begin integer n **initial** 0 ; B **end** ,

one writes

newintvar 0 $\lambda n.\ B$,

where *newintvar* is a built-in procedure of type **int** $\to$ (**intvar** $\to$ **comm**) $\to$ **comm**. Similarly, to declare a sequence X of real variables of length n, one writes

newrealvarseq n f $\lambda X.\ B$,

where *newrealvarseq* has type **int** $\to$ (**int** $\to$ **real**) $\to$ (**realvarseq** $\to$ **comm**) $\to$ **comm** and the function f specifies an initialization in which each Xi is initialized to fi. Procedures such as *newintvar* and *newrealvarseq* are called *declarators*.

The advantage of this approach is that the programmer can use the procedure mechanism to define his own declarators. In particular, only the simplest kind of arrays (i.e. sequences) need to be provided in Forsythe, since more complex kinds can be defined by users.

For example, one could define the type of triangular arrays, and an appropriate declarator (which initializes the array elements to zero) as follows:

$$
\begin{array}{l}
\textbf{lettype } \mathit{triarray} \equiv (\textbf{int} \to \textbf{int} \to \textbf{realvar}) \,\&\, \mathit{size}\!:\!\textbf{int } \textbf{in} \\
\textbf{let } \mathit{newtriarray}\!:\!\textbf{int} \to (\mathit{triarray} \to \textbf{comm}) \to \textbf{comm} \equiv \\
\quad \lambda n.\ \lambda \mathit{body}.\ \mathit{newintvar}\ n\ \lambda n.\ \mathit{newrealvarseq}\Big((n \times (n+1)) \div 2\Big)(\lambda k.\ 0)\ \lambda X. \\
\qquad \mathit{body}\Big(\lambda i.\ \lambda j. \\
\qquad\quad \textbf{if } 0 \le j \wedge j \le i \wedge i < n \textbf{ then } X\Big((i \times (i+1)) \div 2 + j\Big) \\
\qquad\qquad \textbf{else } \mathit{error}\ \text{`SUBSCRIPT ERROR'}, \\
\qquad\quad \mathit{size} \equiv n\Big) \\
\textbf{in } \ldots .
\end{array}
$$

Here the argument $P, \mathit{size} \equiv n$ of the call of *body* is an operation (called a *merge* operation) that combines the procedure P and an object with a single field named *size*.

Since Forsythe inherits the expressive power of call by name from Algol 60, there is no need to use special forms or procedures of no arguments to cause delayed or repeated evaluation. For example, the straightforward definition

$$\textbf{let } \mathit{and}\!:\!\textbf{bool} \to \textbf{bool} \to \textbf{bool} \equiv \lambda p.\ \lambda q.\ \textbf{if } p \textbf{ then } q \textbf{ else } \mathit{false}$$

yields a procedure that will not evaluate q when p is false. (Actually, Forsythe provides a built-in conjunction operator with this kind of short-circuit evaluation.) Similarly, one can define

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

and then write $\mathit{repeat}\ C\ B$ to achieve the effect of **repeat** C **until** B. Thus Forsythe needs no **repeat** statement.

Simlarly, Forsythe needs no **for** statement, since one can define

$$
\begin{array}{l}
\textbf{let } \mathit{for}\!:\!\textbf{int} \to \textbf{int} \to (\textbf{int} \to \textbf{comm}) \to \textbf{comm} \equiv \\
\quad \lambda l.\ \lambda u.\ \lambda \mathit{body}.\ \mathit{newintvar}(l-1)\ \lambda k.\ \mathit{newintvar}\ u\ \lambda u. \\
\qquad \textbf{while } k < u \textbf{ do } (k := k+1\,;\, b\ k)\,,
\end{array}
$$

and then write $\mathit{for}\ \ 0\ \ 9\ \ \lambda i.\ C$ instead of **for** $i := 0$ **until** 9 **do** C. In this case the procedure call is actually more succinct than the conventional notation. (Of course, the use of higher-order procedures to define control constructs is another inheritance from Algol 60, but succinct calls of such procedures require the lambda notation.)

Forsythe provides a generalization of Hoare's **case** construction called a *sequence* construction. If $P_0, \ldots, P_{n-1}$ have type θ, then $\mathbf{seq}(P_0, \ \ldots \ , P_{n-1})$ is a θ-sequence that, when applied to an integer k, evaluates or executes P_k. Thus, instead of **case** k **of** $(C_0, \ \ldots \ , C_{n-1})$, we write

$$\mathbf{seq}(C_0, \ \ldots \ , C_{n-1})\, k\ .$$

This construction can be used to initialize an array, e.g.

$$\mathit{newrealvarseq}\ 3\ \Big(\mathbf{seq}(1.0, 1.5, 2.0)\Big)\ \lambda X.\ \ldots\ .$$

It can also be used in conjunction with higher-order procedures to provide the effect of procedures with a variable number of arguments. For example, if one defines

$$\begin{array}{l}
\mathbf{let}\ \mathit{times}\colon \mathbf{realseq} \to \mathbf{real} \equiv \lambda s. \\
\quad \mathbf{letrec}\ \mathit{times1}\colon \mathbf{int} \to \mathbf{real} \equiv \lambda k. \\
\quad\quad \mathbf{if}\ k \leq 0\ \mathbf{then}\ 1\ \mathbf{else}\ \mathit{newintvar}(k-1)\ \lambda k. \\
\quad\quad\quad \mathit{newrealvar}(\mathit{times1}\ k)\ \lambda x.\ \mathbf{if}\ x = 0\ \mathbf{then}\ 0\ \mathbf{else}\ x \times s\ k \\
\quad \mathbf{in}\ \mathit{times1}(s.\mathit{len})\ ,
\end{array}$$

then $\mathit{times}(\mathbf{seq}(E_0, \ \ldots \ E_{n-1}))$ will compute the product $E_0 \times \cdots \times E_{n-1}$. (Notice that when this procedure encounters an argument with value zero it will not evaluate further arguments.)

Forsythe also provides a primitive type **compl**(etion) that is a subtype of **comm**(and) and denotes a special class of commands that (like **goto**'s in a conventional language) are syntactically guaranteed never to return control. By specifying this type for procedure parameters, one can clarify the flow of control. For example, in

$$\begin{array}{l}
\mathbf{let}\ \mathit{sqroot}\colon \mathbf{real} \to \mathbf{realacc} \to \mathbf{compl} \to \mathbf{comm} \equiv \\
\quad \lambda x.\ \lambda y.\ \lambda \mathit{error}.\ (\mathbf{if}\ x < 0\ \mathbf{then}\ \mathit{error}\ \mathbf{else}\ \mathit{skip}\ ;\ C)
\end{array}$$

it is clear, since *error* is a completion, that C will never be executed when $x < 0$. A more elaborate example is the procedure

$$\begin{array}{l}
\mathbf{let}\ \mathit{linsearch}\colon \mathbf{int} \to (\mathbf{int} \to \mathbf{compl}) \to \mathbf{compl} \to \mathbf{compl} \equiv \\
\quad \lambda \mathit{key}.\ \lambda \mathit{success}.\ \lambda \mathit{failure}. \\
\quad\quad (\mathit{for}\ 1\ 100\ \lambda k.\ \mathbf{if}\ X\ k = \mathit{key}\ \mathbf{then}\ \mathit{success}\ k\ \mathbf{else}\ \mathit{skip}; \\
\quad\quad \mathit{failure})\ ,
\end{array}$$

which searches the array X for an index k such that $X\ k = \mathit{key}$, and terminates by calling the completion *success* k if the search succeeds or the completion *failure* if the search fails. A typical application of this procedure might be

$$\mathit{escape}\ \lambda \mathit{out}.\ \mathit{linsearch}\ 0\ \Big(\lambda k.\ (z := k\ ;\ \mathit{out})\Big)(z := \mathit{default}\ ;\ \mathit{out})\ .$$

Here *escape* is a built-in declarator that makes *out* denote a completion that terminates the call of *escape*. (Note that, since *out* is a completion, both $z := k\,;\, out$ and $z := default\,;\, out$ are completions.)

In fact, completions provide Forsythe with an expressiveness similar to that of **goto**'s and labels. For example, the following is a highly efficient procedure for setting y to x^n in which recursively defined completions are used in place of labels. (The defining occurrence of each completion is followed by a bracketed assertion that will hold whenever control passes to that completion.)

$$
\begin{array}{l}
\textbf{let } power\text{:}\,\textbf{int} \to \textbf{int} \to \textbf{intacc} \to \textbf{comm} \equiv \lambda x.\ \lambda n.\ \lambda y. \\
\quad newintvar\ n\ \lambda k.\ newintvarres\ 1\ y\ \lambda y.\ newintvar\ x\ \lambda z. \\
\quad escape\ \lambda zr\ \{k = 0\}. \\
\qquad \textbf{letrec } tr, nz, ev, od, nzev\text{:}\,\textbf{compl} \\
\qquad \textbf{where} \\
\qquad\quad tr\ \{true\} \equiv \textbf{if } k = 0 \textbf{ then } zr \textbf{ else } nz, \\
\qquad\quad nz\ \{k \neq 0\} \equiv \textbf{if } odd\ k \textbf{ then } od \textbf{ else } nzev, \\
\qquad\quad ev\ \{even\ k\} \equiv \textbf{if } k = 0 \textbf{ then } zr \textbf{ else } nzev, \\
\qquad\quad od\ \{odd\ k\} \equiv k := k - 1\ ;\ y := y \times z\ ;\ ev, \\
\qquad\quad nzev\ \{k \neq 0 \wedge even\ k\} \equiv k := k \div 2\ ;\ z := z \times z\ ;\ nz \\
\qquad \textbf{in } tr
\end{array}
$$

A final generalization in the design of Forsythe is that assignments are really procedure calls, i.e. $A := E$ is an abbreviation for $A(E)$. Thus **realacc** is an abbreviation for **real** $\to$ **comm** (and similarly for the other data types), and the subtype ordering **realacc** $\leq$ **intacc** is a consequence of the subtype rule for procedural types.

This generalization makes it possible for the user to define his own data types by defining the relevant phrase types. To define complex numbers, for example, one might write

$$
\begin{array}{l}
\textbf{lettype } complex \equiv r\text{:}\,\textbf{real} \,\&\, i\text{:}\,\textbf{real } \textbf{in} \\
\textbf{lettype } complexacc \equiv complex \to \textbf{comm } \textbf{in} \\
\textbf{lettype } complexvar \equiv complex \,\&\, complexacc \textbf{ in} \\
\textbf{let } newcomplexvar\text{:}\, complex \to (complexvar \to comm) \to comm \equiv \\
\quad \lambda init.\ \lambda body.\ newrealvar\ init.r\ \lambda rpart.\ newrealvar\ init.i\ \lambda ipart. \\
\qquad body\big(r \equiv rpart, i \equiv ipart, \\
\qquad\quad \lambda c.\ newrealvar\ c.r\ \lambda t.\ (ipart := c.i, rpart := t)\big)\ .
\end{array}
$$

Here *newcomplexvar* is a declarator that represents a complex variable by two real variables *rpart* and *ipart*. The last line defines an acceptor that sets the representation variables. (Note the use of the temporary t to insure that both parts of the complex argument c are evaluated before either representation variable is set.)

In conclusion, it should be emphasized that this cursory survey of Forsythe is only intended to demonstrate that modern language design can produce generality without complexity. For those whose appetite has been whetted, a systematic introduction to the language is given in [2].

At present, we are working on an implementation of the language, which raises two significant research topics: the design of a type checking algorithm that is efficient in commonly occurring cases (we know that no type checker for the language can be efficient in all cases), and the development of a facility for the judicious inline expansion of procedure calls (which is needed to overcome the inefficiencies of call by name).

When a working implementation has been achieved, we intend to experiment with "syntactic sugar". Whenever a language feature is revealed to be a "mere abbreviation", one can ask whether it is really useful or merely conventional. To learn the answer, we have frequently omitted such features from our initial language design. However, we may well learn that we have gone too far, and must retrench to obtain an easily used language.

Eventually, we intend to investigate possible extensions of Forsythe, including polymorphism, concurrency, and facilities requiring more general storage allocation than the stack discipline. It is difficult to foresee how far one can go in such directions without violating the Algol-like framework. But it is vital to continue searching for simple mechanisms of great generality. Unless such mechanisms are found, the complexity of languages in particular and software in general will continue to grow until, when we have achieved powerful computers the size of a book, they will have manuals the size of ENIAC.

References

[1] 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.

[2] Reynolds, J. C. *Preliminary Design of the Programming Language Forsythe.* Report, no. CMU–CS–88–159, Carnegie Mellon University, Computer Science Department, June 21, 1988.