Syntactic Control of Interference[1]

John C. Reynolds
School of Computer and Information Science
Syracuse University

Abstract

In programming languages which permit both assignment and procedures, distinct identifiers can represent data structures which share storage or procedures with interfering side effects. In addition to being a direct source of programming errors, this phenomenon, which we call interference, can impact type structure and parallelism. We show how to eliminate these difficulties by imposing syntactic restrictions, without prohibiting the kind of constructive interference which occurs with higher-order procedures or Simula classes. The basic idea is to prohibit interference between identifiers, but to permit interference among components of collections named by single identifiers.

1. The Problem

It has long been known that a variety of anomalies can arise when a programming language combines assignment with a sufficiently powerful procedure mechanism. The simplest and best-understood case is aliasing or sharing between variables, but there are also subtler phenomena of the kind known vaguely as "interfering side effects."

In this paper we will show that these anomalies are instances of a general phenomenon which we call *interference*. We will argue that it is vital to constrain a language so that interference is syntactically detectable, and we will suggest principles for this constraint.

Between simple variables, the only form of interference is *aliasing* or *sharing*. Consider, for example, the factorial-computing program:

$$
\begin{array}{l}
\textbf{procedure } \mathit{fact}(n, f);\ \textbf{integer } n, f; \\
\quad \textbf{begin integer } k; \\
\quad k := 0;\ f := 1; \\
\quad \textbf{while } k \neq n \textbf{ do begin } k := k + 1;\ f := k \times f \textbf{ end} \\
\quad \textbf{end}\ .
\end{array}
$$

[1]Work supported by National Science Foundation Grant MCS 75-22002, and by the Science Research Council of Great Britain. First appeared in *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, January 1978. ACM, New York. To be reprinted in *Algol-like Languages*, ed. P. W. O'Hearn and R. D. Tennent, Birkhäuser, 1996.

Suppose n and f are called by name as in Algol, or by reference as in Fortran, and consider the effect of a call such as $fact(z, z)$, in which both actual parameters are the same. Then the formal parameters n and f will be aliases, i.e., they will interfere in the sense that assigning to either one will affect the value of the other. As a consequence, the assignment $f := 1$ will obliterate the value of n so that $fact(z, z)$ will not behave correctly.

In this case the problem can be solved by changing n to a local variable which is initialized to the value of the input parameter; this is tantamount to calling n by value. But while this solution is adequate for simple variables, it can become impractical for arrays. For example, the procedure

procedure $transpose(X, Y)$; **real array** X, Y;
  for $i := 1$ **until** 50 **do for** $j := 1$ **until** 50 **do**
    $Y(i, j) := X(j, i)$

will malfunction for a call such as $transpose(Z, Z)$ which causes X and Y to be aliases. But changing X to a local variable only solves this problem at the expense of gross inefficiency in both time and space. Certainly, this inefficiency should not be imposed upon calls which do not produce interference. On the other hand, in-place transposition is best done by a completely different algorithm. This suggests that it is reasonable to permit procedures such as *transpose*, but to prohibit calls of such procedures with interfering parameters.

Although these difficulties date back to Algol and Fortran, more recent languages have introduced new features which exacerbate the problem of interference. One such feature is the union of data types. Suppose x is a variable whose value can range over the union of the disjoint data types **integer** and **character**. Then the language must provide some construct for branching on whether the current value of x is an integer or a character, and thereafter treating x as one type or the other. For example, one might write

unioncase x **of** (**integer**: S; **character**: S') ,

where x may be used as an identifier of type **integer** in S and as an identifier of type **character** in S'. However, consider

unioncase x **of** (**integer**: ($y :=$"A"; $n := x + 1$); **character**: **noaction**) .

It is evident that aliasing between x and y can cause a type error in the expression $x + 1$. Thus, in the presence of a union mechanism, interference can destroy type security. This problem occurs with variant records in Pascal [1], and is only avoided in Algol 68 [2] at the expense of copying union values.

The introduction of parallelism also causes serious difficulties. Hoare [3,4] and Brinch Hansen [5] have argued convincingly that intelligible programming requires all interactions between parallel processes to be mediated by some mechanism such as a critical region or monitor. As a consequence, in the absence of any critical regions or monitor calls, the parallel execution of two statements, written $S_1 \parallel S_2$, can only be permitted when S_1 and S_2 do not interfere with one another. For example,

$$x := x + 1 \parallel y := y \times 2$$

would not be permissible when x and y were aliases.

In this paper, we will not consider interacting parallel processes, but we will permit the parallel construct $S_1 \parallel S_2$ when it is syntactically evident that S_1 and S_2 do not interfere. Although this kind of determinate parallelism is inadequate for practical concurrent programming, it is sufficient to make the consequences of interference especially vivid. For example, when x and y are aliases, the above statement becomes equivalent to

$$z := z + 1 \parallel z := z \times 2$$

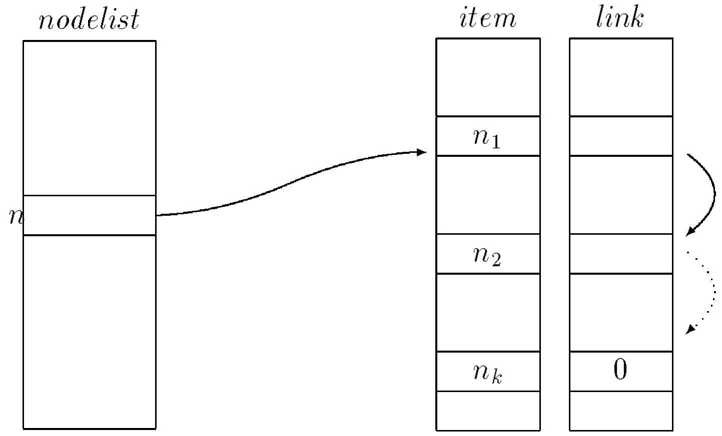
whose meaning, if any, is indeterminate, machine-dependent, and useless.

These examples demonstrate the desirability of constraining a language so that variable aliasing is syntactically detectable. Indeed, several authors have suggested constraints which would eliminate aliasing completely [6,7].

However, aliasing is only the simplest case of the more general phenomenon of interference, which can occur between a variety of program phrases. We have already spoken of two statements interfering when one can perform any action which affects the other. Similarly, two procedures interfere when one can perform a global action which has a global effect upon the other.

Interference raises the same problems as variable aliasing. For example, $P(3) \parallel Q(4)$ is only meaningful if the procedures P and Q do not interfere. Thus the case for syntactic detection extends from aliasing to interference in general. However, the complete prohibition of interference would be untenably restrictive since, unlike variables, interfering expressions, statements, and procedures can have usefully different meanings.

Both the usefulness and the dangers of interference between procedures arise when procedures are used to encapsulate data representations. As an example, consider a finite directed graph whose nodes are labelled by small integers. Such a graph might be represented by giving, for each node n, a linked list of its immediate successors $n_1, \ldots, n_k$:



This representation is used by the procedure

> **procedure** *itersucc*(n, p); **integer** n; **procedure** p;
>   **begin integer** k; $k :=$ *nodelist*(n);
>   **while** $k \neq 0$ **do begin** $p($*item*$(k))$; $k :=$ *link*(k) **end**
>   **end**

which causes the procedure p to be applied to each immediate successor of the node n.

If the graph is ever to change, then something—probably a procedure such as "addedge" or "deleteedge"—must interfere with *itersucc* by assigning to the global arrays *nodelist*, *item*, and *link*. On the other hand, the correct operation of *itersucc* requires that the procedure parameter p must not assign to these arrays, i.e., that p must not interfere with *itersucc*. Indeed, if *itersucc* involved parallelism, e.g., if the body of the **while** statement were

> **begin integer** m; $m :=$ *item*(k);
>   **begin** $p(m) \;\|\; k :=$ *link*(k) **end**
> **end** ,

then noninterference between p and *itersucc* would be required for meaningfulness rather than just correctness.

Of course, the need for interfering procedures would vanish if the graph representation were a parameter to the procedures which use it. But this would preclude an important style of programming—epitomized by Simula 67 [8]—in which data abstraction is realized by using collections of procedures which interfere via hidden global variables.

In summary, these examples motivate the basic goal of this paper: to design a programming language in which interference is possible yet syntactically detectable. To the author's knowledge, the only current language which tries to meet this goal is Euclid [7]. The approach used in Euclid is quite different than that given here, and apparently precludes procedural parameters and call-by-name.

2. The Basic Approach

Before proceeding further, we must delineate the idea of interference more precisely. By a *phrase* we mean a variable, expression, statement, or procedure denotation. In the first three cases, we speak of *exercising* the phrase P, meaning: either assigning or evaluating P if it is a variable, evaluating P if it is an expression, or executing P if it is a statement.

For phrases P and Q, we write $P \# Q$ to indicate that it is syntactically detectable that P and Q do not interfere. More precisely, $\#$ is a syntactically decidable symmetric relation between phrases such that:

1. If neither P nor Q denotes a procedure, then $P \# Q$ implies that, for all ways of exercising P and Q, the exercise of P will have no effect on the exercise of Q (and vice-versa). Thus the meaning of exercising P and Q in parallel is well-defined and determinate.

2. If P denotes a procedure, $A_1, \ldots, A_n$ are syntactically appropriate actual parameters, $P \# Q$, and $A_1 \# Q, \ldots, A_n \# Q$, then $P(A_1, \ldots, A_n) \# Q$. (Thus $P \# Q$ captures the idea that P cannot interfere with Q via global variables.)

It should be emphasized that these rules have a fail-safe character: $P \# Q$ implies that P and Q cannot interfere, but not the converse. Indeed, the rules are vacuously satisfied by defining $\#$ to be universally false, and there is a probably endless sequence of satisfactory definitions which come ever closer to the semantic relation of noninterference at the expense of increasing complexity. Where to stop is ultimately a question of taste: $P \# Q$ should mean that P and Q *obviously* do not interfere.

Our own approach is based upon three principles:

(I) If $I \# J$ for all identifiers I occurring free in P and J occurring free in Q, then $P \# Q$.

In effect, all "channels" of interference must be named by identifiers. For the language discussed in this paper, this principle is trivial, since the only such channels are variables. In a richer language, the principle would imply, for example, that all I/O devices must be named by identifiers.

(II) If I and J are distinct identifiers, then $I \# J$.

This is the most controversial of our principles, since it enforces a particular convention for distinguishing between interfering and noninterfering phrases. Interfering procedures (and other entities) are still permissible, but they must occur within a collection which is named by a single identifier. (An example of such a collection is a typical element in a Simula [8] class. Indeed, the idea of using such collections was suggested by the Simula class mechanism, although we will permit collections which do not belong to any class.)

(III) Certain types of phrases, such as expressions, and procedures which do not assign to global variables, are said to be *passive*. When P and Q are both passive, $P \# Q$.

Passive phrases perform no assignments or other actions which could cause interference. Thus they cannot interfere with one another or even with themselves, although an active phrase and a passive phrase can interfere.

3. An Illustrative Language

To illustrate the above principles we will first introduce an Algol-based language which, although it satisfies Principle I, permits uncontrolled interference. We will then impose Principle II to make interference syntactically detectable. Finally, we will explore the consequences of Principle III.

Unlike Algol, the illustrative language is completely typed, so that reduction (i.e., application of the copy rule) cannot introduce syntax errors. It provides lambda expressions and fixed-point operators for all program types, and a named Cartesian product, which is needed for the collections discussed under Principle II. Procedure declarations, multiple-parameter procedures, and classes are treated as syntactic sugar, i.e., as abbreviations which are defined in terms of more basic linguistic constructs.

Arrays, call-by-value, jumps and labels, unions of types, references, input-output, and critical regions are not considered.

We distinguish between data types, which are the types of values of simple variables, and program types, which are the types which can be declared for identifiers and specified for parameters. The only data types are integer, real, and Boolean, as in Algol, but there are an infinite number of program types. Specifically, the set of program types is the smallest set such that:

(T1) if δ is a data type, then δ var (meaning variable) and δ exp (meaning expression) are program types.

(T2) sta (meaning statement) is a program type.

(T3) If ω and ω' are program types, then $\omega \longrightarrow \omega'$ is a program type.

(T4) If $\overline{\omega}$ is a function from a finite set of identifiers into program types, then $\Pi(\overline{\omega})$ is a program type.

A formal parameter specified to have type δ var can be used on either side of assignment statements, while a formal parameter specified to have type δ exp can only be used as an expression. The program type $\omega \longrightarrow \omega'$ describes procedures whose single parameter has type ω and whose call has type ω'. For example, the Algol procedures

procedure $p1(n)$; **integer** n; $n := 3$;

real procedure $p2(x)$; **real** x; $p2 := x \times x$;

would have types integer var $\longrightarrow$ sta and real exp $\longrightarrow$ real exp, respectively.

The program type $\Pi(\overline{\omega})$ is a Cartesian product in which components are indexed by identifiers rather than by consecutive integers. Specifically, $\Pi(\overline{\omega})$ describes collections in which each i in the domain of $\overline{\omega}$ indexes a component of type $\overline{\omega}(i)$. The function $\overline{\omega}$ will always be written as a list of pairs of the form argument : value. Thus, for example, $\Pi(inc$: sta, val: integer exp) describes collections in which inc indexes a statement and val indexes an integer expression. A typical phrase of this type might be $\langle inc\colon n := n+1;\, val\colon n \times n\rangle$.

To simplify the description of syntax we will ignore aspects of concrete representation such as parenthesization, and we will adopt the fiction that each identifier has a fixed program type (except when used as a component index), when in fact the program type of an identifier will be specified in the format $I : \omega$ when the identifier is bound.

We write $<\omega\ \text{id}>$ and $<\omega>$ to denote the sets of identifiers and phrases with program type ω. Then the syntax of the illustrative language is given by the following production schemata, in which δ ranges over all data types, ω, ω', $\omega_1,\ldots,\omega_n$ range over program types, and $i_1,\ldots,i_n$ range over identifiers:

$<\delta\ \text{exp}> \ ::= \ <\delta\ \text{var}>$

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

$<\text{Boolean exp}> \ ::= \ \textbf{true} \mid <\text{integer exp}> = <\text{integer exp}>$
$\quad \mid <\text{Boolean exp}> \,\&\, <\text{Boolean exp}>$
(and similarly for other constants and operations on data types)

$<\text{sta}> \ ::= \ <\delta\ \text{var}> := <\delta\ \text{exp}>$

$<\text{sta}> \ ::= \ \textbf{noaction} \mid <\text{sta}>\, ;\, <\text{sta}>$
$\quad \mid \textbf{while}\ <\text{Boolean exp}>\ \textbf{do}\ <\text{sta}>$

$<\text{sta}> \ ::= \ \textbf{new}\ <\delta\ \text{var id}>\ \textbf{in}\ <\text{sta}>$

$<\omega> \ ::= \ <\omega\ \text{id}>$

$<\omega \longrightarrow \omega'> \ ::= \ \lambda<\omega\ \text{id}>.\, <\omega'>$

$<\omega'> \ ::= \ <\omega \longrightarrow \omega'>\ (<\omega>)$

$<\Pi(i_1\colon\omega_1,\ldots,i_n\colon\omega_n)> \ ::= \ \langle i_1\colon <\omega_1>,\ldots,i_n\colon <\omega_n>\rangle$

$<\omega_k> \ ::= \ <\Pi(i_1\colon\omega_1,\ldots,i_n\colon\omega_n)>.\, i_k$

$<\omega> \ ::= \ \textbf{if}\ <\text{Boolean exp}>\ \textbf{then}\ <\omega>\ \textbf{else}\ <\omega>$

$<\omega> \ ::= \ \mathbf{Y}(<\omega \longrightarrow \omega>)$

Although a formal semantic specification is beyond the scope of this paper, the meaning of our language can be explicated by various reduction rules. For lambda expressions, we have the usual rule of beta-reduction:

$$(\lambda I.\, P)(Q) \quad\Longrightarrow\quad P|_{I\to Q}$$

where the right side denotes the result of substituting Q for the free occurrences of I in P, after changing bound identifiers in P to avoid conflicts with free identifiers in Q. Note that this rule implies call by name: if P does not contain a free occurrence of I then $(\lambda I.\, P)(Q)$ reduces to P even if Q is nonterminating or causes side effects. For collection expressions, we have

$$\langle i_1 \colon P_1, \ldots, i_n \colon P_n \rangle .\, i_k \quad\Longrightarrow\quad P_k \ .$$

For example,

$$\langle \mathit{inc} \colon n := n+1, \mathit{val} \colon n \times n \rangle .\, \mathit{inc} \quad\Longrightarrow\quad n := n+1 \ .$$

Again, there is a flavor of call-by-name, since the above reduction would still hold if $n \times n$ were replaced by a nonterminating expression. The fixed-point operator $\mathbf{Y}$ can also be elucidated by a reduction rule:

$$\mathbf{Y}(f) \quad\Longrightarrow\quad f(\mathbf{Y}(f)) \ .$$

In addition to lambda expressions, the only other binding mechanism in our language is the declaration of **new** variables. The statement

$$\mathbf{new}\ I : \begin{bmatrix} \text{integer} \\ \text{real} \\ \text{Boolean} \end{bmatrix}\ \mathbf{in}\ S$$

has the same meaning as the Algol statement

$$\mathbf{begin}\ \begin{bmatrix} \textbf{integer} \\ \textbf{real} \\ \textbf{Boolean} \end{bmatrix} I;\ S\ \mathbf{end}.$$

By themselves, lambda expressions and **new** variable declarations are an austere vocabulary for variable binding. But they are sufficient to permit other binding mechanisms to be defined as abbreviations. This approach is vital for the language constraints which will be given below, since it insures that all binding mechanisms will be affected uniformly.

Multiple-parameter procedures are treated following Curry [9]:

$$P(A_1, \ldots, A_n) \ \equiv\ P(A_1) \cdots (A_n) \qquad\qquad \lambda(I_1, \ldots,\ I_n).\, B \ \equiv\ \lambda I_1.\, \cdots \lambda I_n.\, B$$

and definitional forms, including procedure declarations, are treated following Landin [10]:

$$\mathbf{let}\ I = Q\ \mathbf{in}\ P \ \equiv\ (\lambda I.\, P)(Q) \qquad\qquad \mathbf{let\ rec}\ I = Q\ \mathbf{in}\ P \ \equiv (\lambda I.\, P)(\mathbf{Y}(\lambda I.\, Q)).$$

(However, unlike Landin, we are using call-by-name.) We will omit type specifications from **let** and **let rec** expressions when the type of I is apparent from Q.

As shown in the Appendix, classes (in a slightly more limited sense than in Simula) can also be defined as abbreviations.

As an example, the declaration of the procedure *fact* shown at the beginning of this paper, along with a statement S in the scope of this declaration, would be written as:

$$\begin{array}{l} \textbf{let}\ \mathit{fact} = \lambda(n\colon \text{integer exp}, f\colon \text{integer var}). \\ \quad \textbf{new}\ k\colon \text{integer}\ \textbf{in} \\ \qquad (k := 0;\ f := 1;\ \textbf{while}\ k \neq n\ \textbf{do}\ (k := k+1;\ f := k \times f)) \\ \textbf{in}\ S\ . \end{array}$$

After eliminating abbreviations, this becomes

$$\begin{array}{l} (\lambda \mathit{fact}\colon \text{integer exp} \longrightarrow (\text{integer var} \longrightarrow \text{sta}).\ S) \\ \quad (\lambda n\colon \text{integer exp}.\ \lambda f\colon \text{integer var}. \\ \qquad \textbf{new}\ k\colon \text{integer}\ \textbf{in} \\ \qquad\quad (k := 0;\ f := 1;\ \textbf{while}\ k \neq n\ \textbf{do}\ (k := k+1;\ f := k \times f)))\ . \end{array}$$

4. Controlling Interference

The illustrative language already satisfies Principle I. If we can constrain it to satisfy Principle II as well, then $P \# Q$ will hold when P and Q have no free identifiers in common. By assuming the most pessimistic definition of # compatible with this result (and postponing the consequences of Principle III until the next section), we get

$$P \# Q \ \text{ iff } \ F(P) \cap F(Q) = \{\},$$

where $F(P)$ denotes the set of identifiers which occur free in P.

To establish Principle II, we must consider each way of binding an identifier. A **new** variable declaration causes no problems, since new variables are guaranteed to be independent of all previously declared entities. But a lambda expression can cause trouble, since its formal parameter will interfere with its global identifiers if it is ever applied to an actual parameter which interferes with the global identifiers, or equivalently, with the procedure itself. To avoid this interference, we will restrict the call $P(A)$ of a procedure by imposing the requirement $P \# A$.

The following informal argument shows why this restriction works. Consider a beta-reduction $(\lambda I.\, P)(Q) = P|_{I \longrightarrow Q}$. Within P there may be a pair of identifiers which are syntactically required to satisfy the #-relationship, and therefore must be distinct. If so, it is essential that the substitution $I \longrightarrow Q$ preserve the #-relationship. No problem occurs if neither identifier is the formal parameter I. On the other hand, if one identifier is I, then the other distinct identifier must be global. Thus the #-relation will be preserved if $K \# Q$ holds for all global identifiers K, i.e., for all identifiers occurring free in $\lambda I.\, P$. This is equivalent to $(\lambda I.\, P) \# Q$.

More formally, one can show that, with the restriction on procedure calls:

$$<\omega'> \;::=\; <\omega \longrightarrow \omega'>(<\omega>) \quad \text{when } <\omega \longrightarrow \omega'> \# <\omega>\;,$$

syntactic correctness is preserved by beta reduction (and also by reduction of collection expressions), and continues to be preserved when other productions restricted by # are added, e.g.,

$$<\text{sta}> \;::=\; <\text{sta}_1> \,\|\, <\text{sta}_2> \quad \text{when } <\text{sta}_1> \# <\text{sta}_2>\;.$$

The restriction $P \# A$ on $P(A)$ also affects the language constructs which are defined as abbreviations. For **let** $I = Q$ **in** $P \;\equiv\; (\lambda I.\, P)(Q)$, and for **let rec** $I = Q$ **in** $P \;\equiv\; (\lambda I.\, P)(\mathbf{Y}(\lambda I.\, Q))$, we see that, except for I, no free identifier of Q can occur free in P. Thus, although one can declare a procedure or a collection of procedures which use global identifiers (the free identifiers of Q), these globals are *masked* from occurring in the scope P of the declaration, where they would interfere with the identifier I.

For multi-parameter procedures, $P(A_1, \ldots, A_n) \equiv P(A_1) \cdots (A_n)$ implies the restrictions $P \# A_1$, $P(A_1) \# A_2, \ldots,$ $P(A_1) \cdots (A_{n-1}) \# A_n$, which are equivalent to requiring $P \# A_i$ for each parameter and $A_i \# A_j$ for each pair of distinct parameters.

For example, consider the following procedure for a "repeat" statement:

$$\textbf{let } \mathit{repeat} = \lambda(s\colon \text{sta}, b\colon \text{Boolean exp}).\, (s;\ \textbf{while } \neg b \textbf{ do } s) \textbf{ in } \cdots$$

In any useful call $\mathit{repeat}(A_1, A_2)$, the statement A_1 will interfere with the Boolean expression A_2. Although this is permitted in the unconstrained illustrative language, as in Algol, it is prohibited by the restriction $A_1 \# A_2$. Instead, one must group the interfering parameters into a collection:

$$\textbf{let } \mathit{repeat} = \lambda x\colon \Pi(s\colon \text{sta}, b\colon \text{Boolean exp}).\, (x.\, s;\ \textbf{while } \neg x.\, b \textbf{ do } x.\, s) \textbf{ in } \cdots$$

and use calls of the form $\mathit{repeat}(\langle s\colon A_1, b\colon A_2\rangle)$.

This example is characteristic of Principle II. Although interfering parameters are permitted, they require a somewhat cumbersome notation. In compensation, it is immediately clear to the reader of a procedure body when interference between parameters is possible.

5. Passive Phrases

In making interference syntactically detectable, we have been unnecessarily restrictive. For example, we have forbidden parallel constructs such as

$$x := n \parallel y := n$$

or

$$\mathbf{let}\ \mathit{twice} = \lambda s\colon \mathrm{sta}.\,(s\ ;\ s)\ \mathbf{in}\ (\mathit{twice}(x := x + 1) \parallel \mathit{twice}(y := y \times 2))\ .$$

Moreover, the right side of the reduction rule $\mathbf{Y}(f) \implies f(\mathbf{Y}(f))$ violates the requirement $f \# \mathbf{Y}(f)$, giving a clear sign that there is a problem with recursion.

In the first two cases, we have failed to take into account that the expression n and the procedure *twice* are *passive*: they do no assignment (to global variables in the case of procedures), and therefore do not interfere with themselves. Similarly, when f is passive, $f \# \mathbf{Y}(f)$ holds, and the reduction rule for $\mathbf{Y}(f)$ becomes valid. This legitimizes the recursive definition of procedures which do not assign to global variables.

(Recursive procedures which assign to global variables are a more difficult problem. Within the body of such a procedure, the global variables and the procedure itself are interfering entities, and must therefore be represented by components of a collection named by a single identifier. This situation probably doesn't pose any fundamental difficulties, but we have not pursued it.)

The following treatment of passivity is more tentative than the previous development. Expressions in our language are always passive, since they never cause assignment to free variables. Procedures may be active or passive, independently of their argument and result types. Thus we must distinguish the program type $\omega \longrightarrow_P \omega'$ describing passive procedures from the program type $\omega \longrightarrow \omega'$ describing (possibly) active procedures.

More formally, we augment the definition of program types with

(T5) If ω and ω' are program types, then $\omega \longrightarrow_P \omega'$ is a program type.

and we define passive program types to be the smallest set of program types such that

(P1) δ exp is passive.

(P2) $\omega \longrightarrow_P \omega'$ is passive.

(P3) If $\overline{\omega}(i)$ is passive for all i in the domain of $\overline{\omega}$, then $\Pi(\overline{\omega})$ is passive.

Next, for any phrase r, we define $A(r)$ to be the set of identifiers which have at least one free occurrence in r which is outside of any subphrase of passive type. Note that, since identifier occurrences are themselves subphrases, $A(r)$ never contains identifiers of passive type, and since r is a subphrase of itself, $A(r)$ is empty when r has passive type.

Then we relax the definition of $P \# Q$ to permit P and Q to contain free occurrences of the same identifier, providing every such occurrence is within a passive subphrase. We define:

$$P \# Q \ \equiv \ A(P) \cap F(Q) = \{\} \ \ \& \ \ F(P) \cap A(Q) = \{\} \ .$$

Finally, we modify the abstract syntax. We define a passive procedure to be one in which no global identifier has an active occurrence:

$$<\omega \ \longrightarrow_P \omega'> \ \ ::= \ \ \lambda <\omega \ \mathrm{id}> . <\omega'> \ \ \text{when} \ \ A(<\omega'>) - \{<\omega \ \mathrm{id}>\} = \{\}.$$

Passive procedures can occur in any context which permits active procedures

$$<\omega \ \longrightarrow \omega'> \ \ ::= \ \ <\omega \ \longrightarrow_P \omega'> \ ,$$

but only passive procedures can be operands of the fixed-point operator:

$$<\omega> \ \ ::= \ \ \mathbf{Y}(<\omega \ \longrightarrow_P \omega>) \ .$$

6. Some Unresolved Questions

Our abstract syntax is ambiguous, in the sense that specifying the type of a phrase does not always specify a unique type for each subphrase. For example, in the original illustrative language, the subphrase **if** p **then** x **else** y might be either a variable or an expression in contexts such as

$$z := \textbf{if } p \textbf{ then } x \textbf{ else } y$$

$$\langle a\colon \textbf{if } p \textbf{ then } x \textbf{ else } y, b\colon 3\rangle . \, b$$

Similarly, the introduction of passive procedures permits the subphrase $\lambda s\colon \mathrm{sta}.\,(s \ ; \ s)$ to have either type sta $\longrightarrow$ sta or sta $\longrightarrow_P$ sta in the context

$$(\lambda s\colon \mathrm{sta}.\,(s \ ; \ s))(x := x + 1) \ .$$

Although these ambiguities could probably be eliminated, our intuition is to retain them, while insisting that they must not lead to ambiguous meanings. Indeed, it may be fruitful to extend this attitude to a wider variety of implicit conversions.

In normal usage, a procedure call will be active if and only if either the procedure itself or its parameter are active. Although other cases are syntactically permissible they seem to have only trivial instances. Thus it might be desirable to limit the program types of procedures to the cases:

$$\theta \longrightarrow_P \theta' \qquad \alpha \longrightarrow_P \alpha' \qquad \theta \longrightarrow \alpha \qquad \alpha \longrightarrow \alpha'$$

where θ and θ' are passive types and α and α' are nonpassive types.

The most serious problem with our treatment of passivity is our inability to retain the basic property that beta-reduction preserves syntactic correctness. Consider, for example, the reduction

$$\begin{aligned}
&(\lambda p\colon \text{mixed}.\ (x := p.a \parallel y := p.\ a))(\langle a\colon n+1, b\colon n := 0\rangle) \\
&\quad \Longrightarrow \quad x := \langle a\colon n+1, b\colon n := 0\rangle.\ a \parallel y := \langle a\colon n+1, b\colon n := 0\rangle.\ a \\
&\quad \Longrightarrow \quad x := n+1 \parallel y := n+1
\end{aligned}$$

where "mixed" stands for the program type $\Pi(a\colon \text{integer exp}, b\colon \text{sta})$. Although the first and last lines are perfectly reasonable, the intermediate line is rather dubious, since it contains assignments to the same variable n within two statements to be executed in parallel. Nevertheless, our definition of # still permits the intermediate line, on the grounds that assignments within passive phrases cannot be executed.

However, if we accept

$$x := \langle a\colon n+1, b\colon n := 0\rangle.\ a \ \ \#\ \ y := \langle a\colon n+1, b\colon n := 0\rangle.\ a,$$

then it is hard to deny

$$(\lambda s\colon \text{sta}.\ x := \langle a\colon n+1, b\colon (n := 0 \parallel s)\rangle.\ a) \ \ \#\ \ y := \langle a\colon n+1, b\colon n := 0\rangle.\ a\ .$$

But this permits the reduction

$$\begin{aligned}
&(\lambda s\colon \text{sta}.\ x := \langle a\colon n+1, b\colon (n := 0 \parallel s)\rangle.\ a)(y := \langle a\colon n+1, b\colon n := 0\rangle.\ a) \\
&\quad \Longrightarrow \quad x := \langle a\colon n+1, b\colon \underline{(n := 0 \parallel y := \langle a\colon n+1, b\colon n := 0\rangle.\ a)}\,\rangle.\ a \\
&\quad \Longrightarrow \quad x := n+1
\end{aligned}$$

Here the intermediate step, in which the underlined statement is clearly illegal, is prohibited by our syntax.

This kind of problem is compounded by the possibility of collection-returning procedures. For instance, in the above examples, one might have $silly(n+1, n := 0)$, where $silly$ has type integer exp $\longrightarrow$ (sta $\longrightarrow$ mixed), in place of the collection $\langle a\colon n+1, b\colon n := 0\rangle$.

A possible though unaesthetic solution to these problems might be to permit illegal phrases in contexts where passivity guarantees nonexecution. A more hopeful possibility would be to alter the definition of substitution to avoid the creation of illegal phrases in such contexts.

7. Directions for Further Work

Beyond dealing with the above questions, it is obviously essential to extend these ideas to other language mechanisms, particularly arrays.

In addition, the interaction between these ideas and the axiomatization of program correctness needs to be explored. We suspect that many rules of inference might be simplified by using a logic which imposes #-preservation upon substitutions.

A somewhat tangential aspect of this work is the distinction between data and program types, which obviously has implications for user-defined types. (Note the absence of this distinction in Algol 68 [2].) In less Algol-like languages, data types might have as much structure as program types, and user definitions might be needed for both "types" of type. Indeed, there may be grounds for introducing more than two "types" of type.

Finally, these ideas may have implications for the optimization of call-by-name, perhaps to an extent which will overcome the aura of hopeless inefficiency which surrounds this concept. For example, when an expression is a single parameter to a procedure, as opposed to a component of a collection which is a parameter, then its repeated evaluation within the procedure must yield the same value (although nontermination is still possible). This suggests a possible application of the idea of "lazy evaluation" [11, 12].

Appendix: Classes as Syntactic Sugar

In a previous paper, we have argued that classes are a less powerful data abstraction mechanism than either higher-order procedures or user-defined types [14]. The greater generality of higher-order procedures permits the definition of classes (in the reference-free sense of Hoare [13] rather than Simula itself) as abbreviations in our illustrative language. In fact, the basic idea works in Algol, although the absence there of lambda expressions and named collections of procedures makes its application cumbersome.

We consider a class declaration with scope S of the form:

$$\textbf{class}\ C(\text{DECL};\ \text{INIT};\ I_1 : P_1, \ldots, I_n : P_n)\ \textbf{in}\ S \tag{1}$$

which defines C to be a class with component names $I_1, \ldots, I_n$. Here DECL is a list of declarations of variables and procedures which will be private to a class element, INIT is an initialization statement to be executed when each class element is created, and each P_k is the procedure named by I_k, in which the private variables may occur as globals.

Within the scope S, one may declare X to be a new element of class C by writing the statement

$$\textbf{newelement}\ X{:}\, C\ \textbf{in}\ S' \ . \tag{2}$$

Then within the statement S' one may write $X.\, I_k$ to denote the component P_k of the class element X.

To express these notations in terms of procedures, suppose $P_1, \ldots, P_n$ have types $\omega_1, \ldots,$ ω_n, respectively. Then we define (1) to be an abbreviation for:

$$\begin{array}{l} \textbf{let}\ C = \lambda b{:}\, \Pi(I_1{:}\, \omega_1, \ldots, I_n{:}\, \omega_n) \ \longrightarrow \mathrm{sta}\,. \\ \qquad (\mathrm{DECL};\ \mathrm{INIT};\ b(\langle I_1{:}\, P_1, \ldots, I_n{:}\, P_n\rangle)) \\ \textbf{in}\ S\ , \end{array}$$

where b is an identifier not occurring in the original class declaration, and where DECL must be expressed in terms of **new** and **let** declarations. Then we define (2) to be an abbreviation for

$$C(\lambda X{:}\, \Pi(I_1{:}\, \omega_1, \ldots, I_n{:}\, \omega_n).\, S')\ .$$

As an example, where for simplicity P_1 and P_2 are parameterless procedures:

$$\begin{array}{l} \textbf{class}\ \mathit{counter}(\textbf{integer}\ n;\ n := 0;\ \mathit{inc}{:}\, n := n+1, \mathit{val}{:}\, n) \\ \textbf{in}\ \cdots \textbf{newelement}\ k{:}\, \mathit{counter}\ \textbf{in}\ \cdots (k.\, \mathit{inc};\ x := k.\, \mathit{val}) \end{array}$$

is an abbreviation for

$$\begin{array}{l} \textbf{let}\ \mathit{counter} = \lambda b{:}\, \Pi(\mathit{inc}{:}\, \mathrm{sta}, \mathit{val}{:}\, \mathrm{integer\ exp}) \ \longrightarrow \mathrm{sta}. \\ \qquad \textbf{new}\ n{:}\, \mathrm{integer}\ \textbf{in}\ (n := 0;\ b(\langle \mathit{inc}{:}\, n := n+1, \mathit{val}{:}\, n\rangle)) \\ \textbf{in}\ \cdots \mathit{counter}(\lambda k{:}\, \Pi(\mathit{inc}{:}\, \mathrm{sta}, \mathit{val}{:}\, \mathrm{integer\ exp}).\ \cdots\ (k.\, \mathit{inc};\ x := k.\, \mathit{val}))\ , \end{array}$$

which eventually reduces to

$$\textbf{new}\ n{:}\, \mathrm{integer}\ \textbf{in}\ (n := 0;\ \cdots (n := n+1;\ x := n))\ .$$

In the process of reduction, identifiers will be renamed to protect the privacy of n.

The only effect of our interference-controlling constraints is that C must be a passive procedure, i.e., INIT and $P_1, \ldots, P_n$ cannot assign to any variables which are more global than those declared by DECL. This ensures that distinct class elements will not interfere with one another. Otherwise, if C is not passive, then S' in the definition of (2) cannot contain calls of C, so that multiple class elements cannot coexist.

Acknowledgements

Most of this research was done during a delightful and stimulating sabbatical at the University of Edinburgh. Special thanks are due to Rod Burstall and Robin Milner for their encouragement and helpful suggestions, and to the members of IFIP working group 2.3, especially Tony Hoare, for establishing the viewpoint about programming which underlies this work.

References

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

[2] van Wijngaarden, A. (ed.), Mailloux, B. J., Peck, J. E. L., and Koster, C. H. A. *Report on the Algorithmic Language Algol 68*. MR 101, Mathematisch Centrum, Amsterdam, February 1969.

[3] Hoare, C. A. R. Towards a theory of parallel programming. In *Operating Systems Techniques* (eds. C. A. R. Hoare and R. N. Perrott), Academic Press, New York, 1972, pp. 61–71.

[4] Hoare, C. A. R. Monitors: an operating system structuring concept. *Comm. ACM* **17** (October 1974), pp. 549–557.

[5] Brinch Hansen, P. Structured multiprogramming. *Comm. ACM* **15** (July 1972), pp. 574–577.

[6] Hoare, C. A. R. Procedures and parameters: an axiomatic approach. In *Symposium on the Semantics of Algorithmic Languages* (ed. E. Engeler). Springer, Berlin-Heidelberg-New York, 1971, pp. 102–116.

[7] Popek, G. J., Horning, J. J., Lampson, B. W., Mitchell, J. G., and London, R. L. Notes on the design of Euclid. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, *SIGPLAN Notices* **12**, no. 3 (March 1977), pp. 11–18.

[8] Dahl, O.-J. and Hoare, C. A. R. Hierarchical program structures. In *Structured Programming* (O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare), Academic Press, New York 1972, pp. 175–200.

[9] Curry, H. B., and Feys, R. *Combinatory Logic*, vol. I. North-Holland, Amsterdam 1958.

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

[11] Henderson, P., and Morris, Jr., J. H. A lazy evaluator. In *Conference Record of the Third Annual ACM Symposium on Principles of Programming Languages*, Atlanta, Georgia, January 1976. ACM, New York, 1976, pp. 95–103.

[12] Friedman, D. P., and Wise, D. S. cons should not evaluate its arguments. In *Third Int'l Colloquium on Automata, Languages, and Programming* (eds. S. Michaelson and R. Milner), Edinburgh, Scotland, July 1976. Edinburgh University Press, Edinburgh, 1976, pp. 257–284.

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

[14] Reynolds, J. C. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Directions in Algorithmic Languages 1975* (ed. S. A. Schuman). INRIA, Rocquencourt, France, 1975, pp. 157–168.