

An Alias-Free, Object-Oriented Language Concept

Olaf Owe
Department of Informatics
University of Oslo, Norway

March 1988
(Revised June 1990)

Abstract

This paper presents an object-oriented language which conceptually eliminates the use of pointer-variables, and allows programming of objects by means of ordinary program variables. At the same time it allows efficient implementation by means of pointers. This is done by restricting and extending a language with pointer-variables in such a way that aliasing cannot occur, and such that the restrictions are static and simple.

Contents

1 Introduction

In traditional object-oriented programming languages, including Ada, Modula-2, Pascal, Simula [5, 11, 10, 3], objects may be referenced by pointer-variables. This gives flexible and efficient programming of objects. On the other hand, pointer-variables are conceptually more difficult than ordinary program variables, due to the fact that they give rise to aliasing in situations where ordinary variables do not. As a consequence, it is difficult to master object-oriented programming, to understand such programs, and to reason about them by formal or even informal techniques [7, 9].

We here present an alias-free object-oriented language. In this language the variables which denote objects therefore follow the same semantics as ordinary variables, even though they are implemented as pointers. This means that the conceptual complexity ordinarily inherent in pointer-variables is avoided, and that the language has only one variable concept. We may safely reflect this in the syntax of the language by using the same syntax for variables denoting objects as that for ordinary variables. For instance, the assignment $x := e$ may

semantically be understood as copying the value of the expression e into the variable x , but is efficiently implemented by pointer-swinging when this value is an object. In order to avoid aliasing, e may not be an object-variable.

In order to make non-trivial assignments on object-variables, we allow assignments of form

$$x := y := e \quad \text{and} \quad x := y := z := e$$

where e may not be an object-variable (but for instance a NEW-construct or NONE), and where the assignments are done from left to right (after first computing the addresses of the variables x , y (and z) and evaluating e). Examples of such statements are

$l := l.next := NONE$	removes the first element of a list l ,
$first := l := l.next := NONE$	removes the first element of a list l
	and puts it in $first$.

In order to traverse a recursive data structure, we must use recursion rather than iteration.

Rather than building up a language from scratch, we choose to present our ideas by modifying an existing language. For this purpose we choose Simula because it is well-known and has a clear syntax for class-objects and pointer-variables [3, 4]. This will make our discussion less language dependent. The critical constructs are assignments and parameter passing; in particular, call by name must be removed from Simula. Simula allows side-effects; and like aliasing, side-effects make reasoning about programs difficult. However, the goal of this work is to eliminate pointer-variables. In order to make our language restrictions as simple as possible, we will not avoid side-effects in expressions. The order of evaluation in an expression is therefore significant, as in Simula.

In the Euclid language, aliasing caused by operations on pointer-variables is eliminated by introducing pointer collections [6]. However, the restrictions are not static, and therefore verification or dynamic checking is required. Our approach eliminates aliasing without use of pointer collections. Our language is more restrictive than Euclid.

An alias-free language is presented in [9]. However, arrays and pointer-variables are not considered, and these cause the main difficulties considered here. Language restrictions which eliminate aliasing caused by parameter passing, are well-known [8, 9]. We use an alias-free parameter passing method based on “destructive” parameter transmission. This method is free from any restrictions; for instance, the same actual parameter may appear more than once in a call, and may even be a global variable to the procedure.

In the ABEL language [1, 2] both side-effects in expressions and aliasing are avoided, except a controlled form of aliasing introduced by the case-statement. This is achieved by syntactic restrictions (concerning distinctness of variables and expressions) which are significantly more complex than the ones presented

here, especially those related to the case-statement. Our “non-basic functions” provide comparable flexibility without distinctness restrictions. Also ABEL has chosen a different approach to parameter passing than ours.

To demonstrate that our suggested language concept is powerful enough to be useful, we will present a search tree example including procedures for lookup, addition, and deletion of elements.

2 The Language Invariant

According to [7], two distinct variables expressible in the same scope are called *aliases* when they denote the same data object. In our language we would like to ensure that distinct variables (expressible in the same scope) always denote distinct objects, except objects which cannot be updated or changed. We will achieve this by allowing no other language constructs than those maintaining this property, the so-called (language) *Invariant*, which will be formalized below. In order to maintain the Invariant we will carefully discuss what kinds of assignment statements, side-effects, and parameter passing we may accept.

2.1 Notation and Conventions

In the examples below, the attributes *next*, *left*, *right* are defined in the following classes:

```
CLASS List(val); Integer val;
BEGIN REF (List) next; END List;

CLASS Tree(val); Integer val;
BEGIN REF (Tree) left, right; END Tree;
```

We will use the following meta notation:

- C denotes classes,
- x and y denote (simple, indexed, or remote) variables,
- v and w denote simple variables (expressed by an identifier),
- t denotes basic expressions (excluding variables) as defined below,
- e denotes expressions (including variables),
- f denotes a user-defined function returning an object.

In Simula, a remote variable has the form $e.v$ or $e.v[e']$, where the object-expressions e and e' may have side-effects. As in Simula, we adopt inside-out, left-to-right evaluation of subexpressions, for instance e is evaluated before e' in the latter expression. In a language with arrays, distinctness of variables cannot in general be detected statically, since $a[j]$ and $a[i]$ are distinct if and only if $i = j$. In Simula, the two remote variables $l.val$ and $first(l).val$ are not distinct if $first(l)$ returns l .

2.2 Distinctness

In order to talk more precisely about distinctness and aliasing, we introduce two semantic functions **var** and **obj**. For a given state s , we define $\mathbf{var}_s$ of an object-variable (pointer) x to be the address of the pointer variable x in state s ; and we define $\mathbf{obj}_s$ of an object-expression e to be the address of the object resulting from an evaluation of e in state s . (When no confusion results we sometimes omit the index s). Obviously, in a state s where $\mathbf{var}_s x$ equals $\mathbf{var}_s y$, we have that $\mathbf{obj}_s x$ equals $\mathbf{obj}_s y$, (we may think of $\mathbf{obj}_s x$ as the “contents” of $\mathbf{var}_s x$). We therefore define $\mathbf{obj}_s$ of $\mathbf{var}_s x$ to be $\mathbf{obj}_s x$. We may regard the assignment

$$x := e$$

as an abbreviation of

$$\mathbf{var} x := \mathbf{obj} e$$

When $\mathbf{var}_s x$ is different from $\mathbf{var}_s y$, we say that x and y are *distinct variables*, otherwise *identical*. For instance, the two variables l and $l.next$ are distinct, but $l.next$ and $first(l).next$ are identical if $first(l)$ returns l . We say that x and y are aliases in a state s if $\mathbf{var}_s x$ does not equal $\mathbf{var}_s y$ but $\mathbf{obj}_s x$ is the same as $\mathbf{obj}_s y$. We may now formulate the Invariant as follows: In a given state s , and for each two variables x and y (possibly remote) we have that

$$\mathbf{obj}_s x = \mathbf{obj}_s y \text{ implies } \mathbf{var}_s x = \mathbf{var}_s y$$

except when x is an object with constant value (i.e., one that cannot be changed in any way). We may ignore the exception about constant objects if we imagine that each occurrence of a constant object (such as NONE) makes a new copy of the constant object. For convenience we shall assume that NONE is never shared.

It is sometimes convenient to talk about the *reference count* of a given object, indicating how many distinct variables are denoting this object. We define the reference count of an object o in a state s to be the number of distinct variables x expressible in s such that $\mathbf{obj}_s x$ is o . We may now restate the Invariant as follows:

For each (updatable) object its reference count is at most 1.

An expression e is said to be *contained in* another ee if e is not constant and either $\mathbf{obj} e$ equals $\mathbf{obj} ee$ or e is contained in a (variable) attribute of ee . It follows from the invariant that if an expression is contained in two others then one of the two latter must be contained in the other. Therefore we say that two expressions are *disjoint* if none of them is contained in the other. For purposes of program reasoning, it is possible to define static restrictions which imply disjointness, as follows: Two expressions are disjoint if one is basic (as defined below) or the two expressions do not refer to common variables, including global variables in non-basic function calls.

2.3 Basic Expressions

An object expression with reference count 0 is called a *basic expression*. Examples of basic expression are NONE and NEW C. We will later show how to define functions with basic function values, and allow parameterized NEW-constructs. The evaluation of a basic expression t obviously maintains the Invariant.

We do not allow pointer-equality ($==$ in Simula), since it would always return false (except for constant objects). Object-equality ($=$) could be allowed, but this might lead to inefficient code. For our purposes it suffices to allow test for NONE (using $=$).

3 Assignments

Consider multiple assignments of form

$$x_1 : - x_2 : - \dots : - x_n : - t$$

(where t is a basic expression and $n > 0$) executed from left to right in two passes,¹ where the first pass evaluates **var** x_i (say x'_i) and **obj** t (say t'), and the second pass performs the assignments, i.e.

$$x'_1 := \mathbf{obj} \ x'_2; x'_2 := \dots := \mathbf{obj} \ x'_n; x'_n := t'$$

If $n > 3$ this assignment may violate the Invariant, for instance

$$x : - y : - z : - y : - \text{NONE}$$

may cause x and z to be aliases.

However, if $n \leq 3$ the Invariant is maintained: Consider

$$x_1 : - x_2 : - x_3 : - t$$

The first pass can not break the Invariant since side-effects must maintain the Invariant, and since the reference count of t' is 0 (and since $x1', x2', x3'$ are not program variables). The second pass (which is free from side-effects) has the following cases:

- $x1', x2', x3'$ are distinct: the aliases caused are cancelled.
- $x1'$ and $x2'$ are identical: the assignment $x1' : - x2'$; may be ignored
- $x1'$ and $x3'$ are identical: the assignment $x1' : - x2'$; may be ignored
- $x2'$ and $x3'$ are identical: the assignment $x2' : - x3'$; may be ignored

¹We have here deviated from the Simula semantics, which has right to left evaluation in the second pass, since this does not maintain the Invariant.

- $x1', x2'$ and $x3'$ are identical: the two first assignments may be ignored

In the four last cases, we have essentially double or single assignments — which obviously maintain the Invariant.

We therefore allow multiple assignments with $n \leq 3$. Examples are:

$l.next.next : - NONE$	<i>removes all elements in a list except the two first</i>
$l : - l.next.next : - NONE$	<i>removes the two first elements of a list l</i>
$s : - l : - l.next : - NONE$	<i>removes the first element of l and puts it in s</i>
$last(l).next : - NEWList(1)$	

3.1 Cyclic Assignments

Consider cyclic assignments, with the syntax

$$x_1 : - x_2 : - \dots : - x_n : -$$

($n > 1$). The execution defined above is modified by replacing $x'_n := t'$ by $x'_n := \mathbf{obj} \ x'_1$. This results in a cyclic shift towards the left on the variables $x_1, x_2, \dots, x_n$.

When $n > 2$ static restriction (to distinctness) are needed. We therefore allow only cyclic assignments of form $x : - y : -$ which swaps x and y . Notice that it has no effect when x and y are identical.

3.2 Assignments on Non-Object Variables

In order to obtain the same semantics for all kinds of multiple assignments, we must modify the Simula multiple assignment for non-object variables,

$$x_1 := x_2 := \dots := x_n := e$$

such that the second pass is left to right and such that e may be omitted, as above. (When n is large, distinctness is not required; and e may be any expression.)

3.3 Implementation of Deallocation

We will here look at how objects become inaccessible and how they can be deallocated.

Consider a multiple assignment of the form

$$x_1 : - x_2 : - \dots : - x_n : - t$$

($n \geq 1$). Here the object (if any) denoted by x'_1 , before the second pass, becomes inaccessible, and also any objects contained in it. This statement may therefore be interpreted as

$$deallocate : - x_1 : - x_2 : - \dots : - x_n : - t$$

where the system variable *deallocate* deallocates all objects assigned to it, including those contained in them. For instance, the assignment *deallocate* : - *t.left* deallocates all objects in the left sub-tree of *t* (if any).

Another form of deallocation occurs if any x_i ($0 < i < n$) is strictly contained in x_{i+1} ; Then the object (if any) denoted by x'_{i+1} (just before the assignment to it) becomes inaccessible. With $n \leq 3$ this is only a problem in triple assignments for $i = 2$. Examples of this situation are:

$$\begin{aligned} \dots : - l.next : - l : - \dots \\ w : - t.left.right : - t : - NONE \\ rest : - l.find(x).next : - l.find(y).next : - NONE \end{aligned}$$

In the second example, all objects in the tree *t*, except those in the subtree *t.left.right*, become inaccessible.

The correct deallocation results if *deallocate* : - x'_{i+1} ; is inserted just after $x'_i : - x'_{i+1}$; (in the second pass). However, since this assignment is semantically equivalent to the two assignments

$$\begin{aligned} x_1 : - \dots : - x_i : - NONE; \\ x_{i+1} : - \dots : - t; \end{aligned}$$

which more clearly states the effect, it is reasonable to introduce the language restriction that no x_{i+1} may be contained in x_i — by sufficient static tests; for instance ensuring that a remote x_i ($0 < i < n$) may only be followed by a disjoint variable or one textually contained in it. All examples outside this section satisfy this restriction; in particular, only the last left hand side in any assignment is remote.

In contrast to Simula, our modified language allows complete static control of deallocation of objects.

4 Functions and Procedures

We here show an alias-free parameter passing method without any static (or dynamic) restrictions. It has the same semantics as value-result passing, and Hoare like program reasoning may be done as usual, provided the actual parameters are disjoint and disjoint from any global variables (actual IN-parameters need not be disjoint). Otherwise, the effect may sometimes be other than expected (whatever that might be).

We introduce the keywords IN and INOUT for parameter specification (and remove the Simula keywords VALUE and NAME), such that IN gives read-only access, and INOUT gives read and write access. As in Ada, we let IN be default. Only variables may occur as actual INOUT-parameters. A formal IN-parameter, or any of its attributes, may not occur to the left of : - (or :=) and may not occur as an actual INOUT-parameter.

For a procedure local to a class C the implicit **THIS**-variable is regarded as an **IN**-parameter (like in Simula). And we handle the **THIS**-variable of an inspect statement similarly.

4.1 Parameter Passing

Consider a procedure p , say:

```
PROCEDURE  $p(x, y)$ ; REF ( $C$ )  $x, y$ ;
<body of  $p$ >;
```

In Simula, the effect of the call $p(a, b)$, including parameter passing (by reference), can be semantically defined as follows:

```
BEGIN REF ( $C$ )  $x, y$ ;
 $x$  : -  $a$ ;
 $y$  : -  $b$ ;
<body of  $p$ >;
END;
```

assuming that the formal variables are distinct from any variable occurring in the actuals (if not, we could rename the formals). Clearly, this parameter passing is not alias-free.

Instead, we shall use “destructive” parameter passing, defined as follows:

```
BEGIN REF ( $C$ )  $x, y$ ;
 $x$  : -  $a$  : - NONE;   where : - NONE is omitted when  $a$  is basic
 $y$  : -  $b$  : - NONE;   where : - NONE is omitted when  $b$  is basic
<body of  $p$ >;
IF  $a'$  = NONE THEN  $a'$  : -  $x$  : - NONE; omitted when  $a$  is basic
IF  $b'$  = NONE THEN  $b'$  : -  $y$  : - NONE; omitted when  $b$  is basic
END;
```

where the primes indicate that the addresses of the actuals are evaluated only once, and where the if-tests protect results passed earlier (in case of identical actual parameters) and updates done on actual parameters used globally. This parameter passing method obviously maintains the Invariant since it is done by legal assignments, and can be implemented efficiently since no object is copied. It is semantically consistent with ordinary call by value-result provided the (non-basic) actual parameters are disjoint and disjoint from the global variables.

Example:

```
PROCEDURE swap( $x, y$ ); REF ( $C$ )  $x, y$ ;
 $x$  : -  $y$  : - ;
```

The call $swap(a[i], a[j])$ will swap its arguments and has no effect when $i = j$. Also any call on a procedure with empty body would have no effect.

The parameter passing method presented here offers both read and write access to the parameters (INOUT-parameters). It would also be possible to offer read-only access when desired (IN-parameters). In order to achieve this, we introduce the keywords IN and INOUT for parameter specification (and remove the Simula keywords VALUE and NAME), letting IN be default. Only variables (and formal INOUT-parameters) may occur as actual INOUT-parameters. Both kinds of parameters are passed to and from the procedure as described above, but formal IN-parameters may not be modified. A sufficient static condition is that a formal IN-parameter (or any of its attributes) may not occur to the left of an assignment operator and may not occur as an actual INOUT-parameter.

We require that IN-parameters precede any INOUT-parameters. The destruction of actual IN-parameters can then be done after all the IN-parameters are passed to the procedure, allowing several formal IN-parameters to denote the same object. This aliasing does not violate the Invariant since the object is constant. (In particular, this object cannot be accessed through an INOUT-parameter or a global variable.) It follows that actual IN-parameters denote the same object before and after the procedure call, assuming normal termination — and, due to the protecting if-tests, provided that they are not updated as global variables. (The latter condition disappears if IN-parameters are passed back without testing.)

In order to obtain the same parameter passing semantics for all kinds of variables, non-object-parameters (of type Integer, Boolean, Character, etc) could be passed similarly, using default values of appropriate types rather than NONE. Simula defines default values for each type (0 for numbers, *false* for boolean, *NONE* for classes, etc.²). Alternatively, one could pass non-object parameters by value-result; and by disjointness requirements on the actual object-parameters one may ensure that all parameters has the same semantics.

4.2 Objects Returned from Procedures and Functions

Consider an object-valued procedure, say

REF (C) PROCEDURE f(...);

In order to define a function-value (other than NONE), we may (as in Simula) assign to *f*, using the convention that a function name occurring to the left of : — is understood as a local variable. It follows that an object function value produced this way must be basic.

The following example of a function removes and returns the first element of a list *l*:

*REF (List) PROCEDURE pop(l); INOUT l; REF(List) l;
IF l ≠ NONE THEN pop : — l : — l.next : — NONE;*

²If arrays are passed as objects, the Simula default value for arrays should be reconsidered.

and may be used in the assignment $top : - pop(l)$.

With this method, it is not possible to return a substructure of a parameter x without removing it from x . We therefore introduce so-called *non-basic functions*. A non-basic function is a procedure which, rather than assigning to the function name, may return function values by the return statement

RETURN e

where e is any object expression (including variables and non-basic function calls). Obviously, the returned object (**obj** e) has a reference count of at most 1. Therefore a non-basic function call may not be used where a basic expression is required, but may occur elsewhere, for instance as an IN-parameter, in front of a dot, or after INSPECT and RETURN.

The parameter passing above must then allow a non-basic function call a as an actual IN-parameter. In this case the assignments prescribed to a must be to done on the pointer to **obj** a (by the Invariant it is unique). For efficient implementation one may let the return-statement return **var** e (rather than **obj** e), when e is non-basic, otherwise **var** of a dummy pointer to **obj** e .

Notice that an application of a non-basic function may return a variable expressible in the same state, but this does not cause an alias. Example of a non-basic lookup-function local to class List:

```
REF(List) FUNCTION Find(x); Integer x;
RETURN IF x=val THEN THIS List ELSE
      IF next=NONE THEN NONE ELSE next.Find(x);
```

Note that this function does not remove the found object from the list. We use the keyword FUNCTION to distinguish non-basic functions from basic ones (written as PROCEDURE).

4.3 Inspect Statements

Consider an inspect statement, say

INSPECT e DO S

Simula offers read-only access to the inspected object through *THIS C* (where e is of class C). We avoid aliasing by treating *THIS C* as a formal IN-parameter, and e as the corresponding actual. It follows that when e is a variable, one may assign to it inside S (as in the *del*- and *remin*-procedures below) without affecting the value of *THIS*; but affecting the final value of e . As in Simula we let *THIS C.e* be abbreviated to e .

It is possible to inspect a substructure, for instance with the above non-basic Find-function we may search for a given value, and then observe and update the found sublist, as follows:

INSPECT l.Find(...) DO < interact with the found sublist >

4.4 Parameterized NEW-Constructs

For classes parameterized by objects, the NEW-construct may cause an alias in Simula. For instance, given

```
CLASS L(next); REF (L) next ;
BEGIN Integer x; END L
```

the assignment $w : - \text{NEW } L(v)$ makes $w.\text{next}$ an alias for v . In order to maintain the Invariant we chose to redefine $\text{NEW } L(v)$ as

```
next : - v : - NONE
```

which has the side-effect of destroying v . Thus class-parameters act as INOUT-parameters. The advantage is that parameterized NEW-constructs are basic.

5 Unified Syntax for Variables

We have above developed restrictions on the use of pointer-variables such that aliasing may never occur, and such that parameter passing and object-assignments are semantically consistent with those for ordinary variables. Pointer-variables may therefore be conceptually regarded as ordinary variables (with special restrictions), and we may without confusion use the same syntax for all variables. In particular, we will rewrite $: -$ as $:=$ and we will rewrite $\text{REF}(C)$ simply as C . This means that declarations of object-variables and object-returning functions, for instance

```
C x
C PROCEDURE f(...);
```

follow the same syntax as ordinary variables and functions, for instance

```
Integer x
Integer PROCEDURE f(...);
```

But of course, for C-variables, the static restrictions given above must be checked. The pop function above can now be rewritten as follows:

```
List PROCEDURE pop(l); INOUT l; List l;
IF l ≠ NONE THEN pop := l := l.next := NONE;
```

5.1 The Search Tree Example

This section presents the concept of search trees as an abstract data type. A search tree is a tree with the invariant that the infix sequence of integer values is sorted.

The above declaration of class Tree can be generalized if the search tree should contain more complex information, say a subclass of class Item, defined as follows

```

CLASS Item(val); Integer val;
  HIDDEN PROTECTED val;
  BEGIN
    Integer PROCEDURE key; key:=val;
  END Item;

```

```

CLASS Tree(v); Item v;
  BEGIN Tree l,r;
  END Tree;

```

(where HIDDEN hides *val* from outside the class, and PROTECTED in a subclass.)

We show procedures for looking up a value (find), adding an item (add), returning and removing the smallest value (remin), and deleting a value (del). The procedures maintain the search tree invariant.

```

Item FUNCTION find(x,t); Integer x; Tree t;
INSPECT t DO RETURN IF x<v.key THEN l.has(x) ELSE
                      IF x>v.key THEN r.has(x) ELSE v;

```

```

PROCEDURE add(x,t); INOUT t; Item x; Tree t;
IF x≠NONE THEN INSPECT t DO IF x.key<v.key THEN l.add(x) ELSE
                             IF x.key>v.key THEN r.add(x)
                             OTHERWISE t:= NEW Tree(x);

```

```

Item PROCEDURE remin(t); INOUT t; Tree t;
INSPECT t DO IF l=NONE
              THEN BEGIN remin:=v:=NONE; t:=r:=NONE END
              ELSE remin:=l.remin;

```

```

PROCEDURE del(x,t); INOUT t; Integer x; Tree t;
INSPECT t DO IF x<v.key THEN l.del(x) ELSE
              IF x>v.key THEN r.del(x) ELSE
              IF r=NONE THEN t := l := NONE
              ELSE v:=r.remin;

```

where the last assignment is legal since *remin* is basic. In order to protect the search tree invariant, the above functions and class *Tree* could be hidden inside a class *SearchTree* as follows:

```

CLASS SearchTree;
  HIDDEN Tree, find, add, del, remin, root;
  BEGIN
    < Declaration of class Tree and the procedures find, add, remin, del as above. >
  Tree root;

```

```

Item FUNCTION Find(x); Integer x;
    RETURN find(x,root);

Boolean PROCEDURE Has(x); Integer x;
    Has := Find(x)≠NONE;

PROCEDURE Add(x); Item x;
    add(x,root);

PROCEDURE Del(x); Integer x;
    del(x,root);

...
END SearchTree;

```

where the visible functions are capitalized, and the lower-case ones are hidden.

6 Conclusions

By restricting assignments to and parameter passing of object-variables, we have found an alias-free language concept which conceptually, and syntactically, eliminates the need for pointer-variables, but at the same time allows efficient implementation of object-variables by means of pointers, with statically controlled garbage collection.

The expressive power of our restricted language seems to be the same as that of Simula. However, it is less efficient for applications where an alias is needed for efficiency reasons; for instance, it is not possible to do efficient insertion at the end of a list because a pointer to the last object cannot be maintained in the data-structure (since it is an alias). Similarly, one can not program two-way lists or any kind of cyclic structures.

It would be possible to extend our language with pointer-variables as in Simula (and even with the syntax of Simula) in order to combine the safety of alias-free variables with the flexibility of pointer-variables (when needed), for instance as hidden variables in classes distinguished as “unsafe”. Alternatively, one may define a set of semantics-preserving and efficiency-improving transformations from non-cyclic to cyclic structures.

Our language concept poses some static restrictions and uses a certain parameter passing method. But it is independent of other language constructs, such as subclass and virtual mechanisms. We have also shown how to combine the concept of abstract data types with that of object orientation. We have shown our ideas using Simula, but they could easily be applied to other languages.

Acknowledgements

The author is thankful for comments and feedback from the computer science group at Institute of Informatics. In particular Ole-Johan Dahl has provided inspiration and fruitful discussions.

References

- [1] Ole-Johan Dahl, Dag Langmyhr, and Olaf Owe, “Preliminary Report on the Specification and Programming Language ABEL”, Research Report no. 106, Institute of Informatics, University of Oslo, Norway, 1987.
- [2] Ole-Johan Dahl, “Object Oriented Specifications”, in Research Directions in Object-Oriented Programming. Edited by Bruce Shriver and Peter Wegner, Computer Systems Series, MIT Press, 1987.
- [3] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard, “Simula 67 Common Base Language”, NCC pub. S-22, Norwegian Computing Center, 1971.
- [4] Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard, “Simula BEGIN”, Chartwell-Bratt Ltd, 1973.
- [5] U.S. Department of Defense, “Reference Manual for the Ada Programming Language”, U.S. Department of Defense, 1983.
- [6] Popek et al., “Notes on the Design of Euclid”, Proceedings ACM Conference on Language Design for Reliable Software, 1977.
- [7] Carlo Ghezzi and Mehdi Jazayeri, “Programming Language Concepts”, John Wiley & Sons, 1982.
- [8] C.A.R. Hoare, “Procedures and Parameters: An Axiomatic Approach” In Symposium on the Semantics of Algorithmic Languages. (ed. E.Engeler) Springer Verlag, 1971.
- [9] J.C. Reynolds, “Syntactic Control of Interference.” Proceedings of Fifth Annual ACM Symp. on Principles of Programming Languages. Tucson, Az: Jan. 23-25, 1978.
- [10] N. Wirth, “The Programming Language PASCAL” Acta Informatica 1 (pages 35-63) 1971.
- [11] N. Wirth, “Programming in MODULA-2 Language Pascal” Springer Verlag, 1982.