OOPS-Algol.

An extension to PS-Algol to support object-oriented programming.

A thesis presented in partial fulfilment of the requirements for the degree of Master of Science at Massey University.

William Dennis Ryder.

1989.

Acknowledgements.

This thesis would not have been possible without the patience and advice of Tom Docker and Chris Phillips. Thanks guys.

I am indebted to Commercial Software Limited for their financial support and the use of their resources to complete this thesis.

Abstract

Object-oriented programming is becoming a widely accepted paradigm to promote software reuse and data abstraction. Many languages are having object oriented capabilities added to them.

PS-Algol is a language which supports procedures as first class data, and supports orthogonality of persistence. OOPS-Algol extends the PS-Algol language to support object-oriented programming.

OOPS-Algol is different from most other object-oriented languages in that it explicitly separates the implementation of a class's protocol from the description of that protocol. The class hierarchy is used solely for defining the conceptual relationships between classes. The inheritance hierarchy is used to promote code sharing, without being constrained by the class hierarchy. This capability furthers progress towards the goal of separating the conceptual design of a system from its implementation.

CONTENTS

1. Introduction

In 1986 the Department of Computer Science at Massey acquired PS-Algol [Atkinson, Bailey, et al 83] as part of a cooperative research arrangement with the University of St. Andrews. We intended to use the language to implement a system for executing the data flow diagrams (DFD) of Structured Systems Analysis as exemplified by De Marco [DeMarco 78], and Gane and Sarson [Gane & Sarson 79].

At the time of this experiment with PS-Algol, object-oriented programming was gaining considerable momentum in the computing community. After investigation of this relatively immature paradigm we established that it appeared to offer advantages that traditional system development paradigms did not offer. The advantages we considered most important were the use of data abstraction (an object is only accessible through its operations) and the ability to define objects incrementally using the inheritance hierarchy.

Given the advantages we saw in object-oriented programming and the power of PS-Algol we began to implement the window system necessary for our DFD system using

object based techniques in PS-Algol. This work was begun on a Macintosh and was continued on Sun workstations.

Although PS-Algol provides good facilities for data abstraction it provides none to support inheritance. It became obvious that without automated support of inheritance the development effort required was too great and we re-evaluated our techniques. It was this re-evaluation that led to the development of OOPS-Algol (Object-Oriented PS-Algol) which this thesis describes.

This chapter provides an overview of PS-Algol, our initial attempt at implementing objects, and the top-level description of OOPS-Algol.

1.1 PS-Algol

We were initially attracted to PS-Algol by its support of 'orthogonality of persistence'; procedures as first class data objects; and graphics objects as built-in data types.

The persistence of a data object is the length of time the object exists. PS-Algol allows any data object, regardless of type, to have the same rights to

long and short term persistence, hence persistence is an orthogonal property of data. This property is important to object based systems as it is necessary to preserve the state of the system between invocations. PS-Algol makes this operation trivial compared to the 'hoop-jumping' required with most other traditional languages.

In PS-Algol procedures have the same rights as any other data object in the language. A procedure can be the result of an expression or another procedure, an element of a structure or an array, assigned to a variable, *et cetera*. Hence a procedure is a first class citizen of the language. This property is important in implementing object-oriented systems as we have to be able to store the procedures to be executed when an object receives a message. The implementation task is clearly much simpler when all of this can be done within the language, without resort to external agents such as linkers or file systems.

The power of graphical interfaces for certain types of application is well known. PS-Algol gives graphics objects (bitmaps and line drawings) the same rights as any other type in the language. This simplifies the implementation of systems requiring graphics

considerably, removing the necessity of using subroutine packages as is common in other systems to support graphics.

1.2 Objects in PS-Algol - a First Attempt

Having decided to use PS-Algol and object-oriented programming we developed a simple technique for representing objects. We used the persistent store to hold procedures that created instances of objects on request. The objects returned were structures whose fields contained the procedures to be executed when the object received a message. The data local to the object were not explicitly represented in the structure as fields but were variables visible to the procedures in the structure by virtue of PS-Algol's scope rules. Our technique is explained in more detail in Appendix 3.

The main advantages of our simple technique were the speed of execution and simplicity. The selection of the appropriate procedure for a message was performed by the compiler which removed the runtime message selection used in most object based systems. It was simple because we did not have to write any message switching software and we did not support inheritance.

The absence of inheritance was partly a result of our simplistic approach and of PS-Algol's type-checking system. As one of PS-Algol's objectives is data protection, it uses runtime checking of structure accesses. This prevents a structure from impersonating another structure. Inheritance requires that an object in the inheritance hierarchy can be used as an object of a type higher in the hierarchy (along the same path). This was not possible in our simple system as we would require different structures to be treated as the same type in some cases. We modelled inheritance by making the object we wanted to enhance a component of the new more complex object. However, this was cumbersome and time consuming. It was this problem that motivated the development of OOPS-Algol.

1.3 **OOPS-Algol - An Improved Object System**

The experience gained with our simple object system and examination of the capabilities of other object-oriented systems led us to design a system with these objectives:

1. We should be able to upgrade methods without adversely impacting existing objects.

2. The implementation and conceptual hierarchies of objects should be separated.

3. The system should be strongly typed.

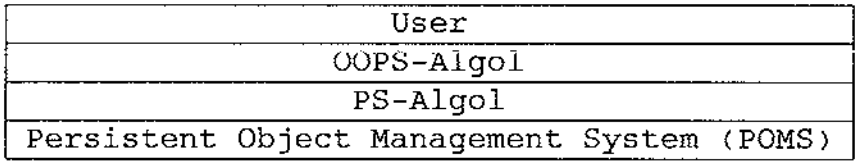
4. Subtyping should be supported in our type checking system.

5. We should be able to have alternate representations for the same object class.

There were two possible ways of implementing this system. Given that we had the source to PS-Algol, we could have enhanced the PS-Algol virtual machine and compiler to support OOPS-Algol. The alternative was to adopt the approach of other retrofitted object systems to existing languages (as in Objective-C [Cox 86], C++ [Stroustrup 86]) and use a preprocessor to add an extra layer of functionality.

We adopted the latter approach because we were not familiar with the internal operation of PS-Algol and we preferred to add our system as a layer above PS-Algol, keeping the systems separate. This approach reduces the perceived complexity of our implementation, which is important as it is an experimental system which needs to be able to be changed easily.

We developed OOPS-Algol on a Sun workstation and a NCR Tower 32/600 system. The following diagram shows the overall structure of the system.



The user writes in OOPS-Algol which is PS-Algol with extra constructs for defining and communicating with objects. OOPS-Algol converts these statements into PS-Algol. PS-Algol acts as the interface with POMS which holds all objects in the system, regardless of their longevity.

1.4 Thesis Structure

We begin in Chapter 2 by introducing the concepts of object oriented programming. This is done mainly to define object oriented programming as we see it and to explain the differences between delegation and inheritance.

In Chapter 3 we survey some current object-oriented languages. The chapter provides some examples of systems that have had objects retrofitted to existing languages in order to provide some comparison with the implementation of OOPS-Algol.

Chapter 4 surveys how subtyping is currently used in other class based systems, and relates this to OOPS-Algol.

Chapter 5 describes the user view of OOPS-Algol without going into excessive detail. The chapter also describes the main syntactic features of the language.

Chapter 6 describes OOPS-Algol's type system and how it relates to the class hierarchy.

Chapter 7 completes our discussion of OOPS-Algol with a description of how exemplars are defined, and how they relate to the class hierarchy. We present an extended example in this section to show how we can separate the type hierarchy from the implementation hierarchy.

The final chapter provides a post-mortem of this experiment, and discusses possible future directions.

The appendices include details that we did not consider appropriate to place in the body of the thesis. Appendix 1 contains the syntax of OOPS-Algol. Appendix 2 contains a description of how we represent objects in OOPS-Algol. We did not consider the latter to be suitable for the body of the dissertation as it is a technical implementation description, and is not relevant to our discussion of OOPS-Algol itself. Appendix 3 contains a description of how to compile and run OOPS-Algol programs, and describes the environment under which they run. Finally, Appendix 4 contains a brief description of our original attempt at implementing objects.