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.

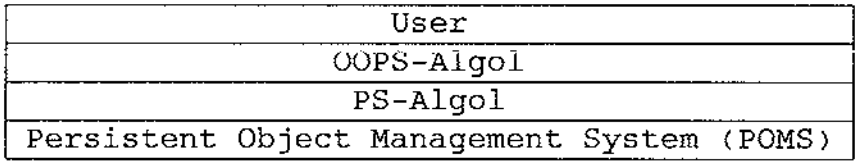
4. Subtyping should be supported in our type checking system.

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.

2. Basic Concepts of Object Oriented Programming.

2.1 What is an Object

Before the concepts of object-oriented programming are introduced, it is worth noting that a number of slightly different definitions exist. We generally follow that of [Wegner 87]:

> "An object has a set of 'operations' and a 'state' that remembers the effect of operations".

This reflects the difference between functions and objects. The result of a function is solely determined by its arguments. In contrast, the result of an operation on an object depends on the results of previous operations on that object and the arguments included with the operation. That is, an object has a history.

The definition highlights the similarity between objects and *abstract data types*. The code sharing offered by object-oriented systems distinguishes objects

from instances of abstract data types. Wegner calls systems that do not have some form of inheritance to implement code sharing 'object *based*' systems, rather than 'object *oriented*' systems.

As an example of how we could represent an application with objects we will consider a simple debtors system. In this system we will have two major object types, *debtor* and *transaction*. If we want to know how much a debtor owes we ask the corresponding debtor object to give us that information. The object will respond with the value.

The user of the debtor object does not need to know whether the debtor remembers the total in a variable that is updated each time a transaction arrives, or if it iterates over all of its transactions to get the current balance. This illustrates the data abstraction offered by objects.

The transaction object will remember at least its value and will return that value if asked. When we want to add a transaction to a debtor we tell the debtor that it should add the new transaction and the debtor will update its own information with no interference from the

'outside'. The user controlling the update needs no knowledge of the internal representation of either the debtor or transaction. The user only needs to know that a debtor knows how to add a transaction.

This shows that when we build a system using objects we are essentially creating building blocks of different shapes, and then fitting them together. The process of intergrating the blocks can be separated from the actual manufacture of the blocks, thus enhancing reusability. The success of this approach has been demonstrated many time in the 'real' world with products like *Lego* and Integrated Circuits.

2.2 **Messages**

An operation on an object is triggered by sending a *message* to that object. The expression *send a message* does not necessarily mean we send a physical message. When we send a message we are selecting an operation to perform on the object. The term *message* is used to underline the fact that an object needs to be considered a separate entity and that communication between objects is carried out according to some protocol.

Depending on the language, sending a message may simply be the selection of a field from a structure and executing the code pointed to by that field (as in C++ [Stroustrup 86]); or it may be a call to a subroutine that selects the appropriate code from tables (as in OOPS-Algol). Messages as such are generally only used in distributed environments, or in some concurrent object systems.

To demonstrate the difference between terminologies here, a small example will be used. Assume we have the ubiquitous stack and we want to push some data on it. In an object oriented system we would have a *stack* object to which we would send messages. We would ask the stack object to push some data onto it (the data would be passed as an argument). This would cause the procedure corresponding to the *push* message to be executed, which would change the state of the object. In a functional system we would say: apply the *push* function to this *stack* with this item of data. In this case the function would return a new stack with some more data on it.

This demonstrates two differences between functional and object based systems:

1. A functional system deletes the old data and creates some new data, an object based system changes the existing 'data'.

2. In a functional system the user selects the appropriate procedure to apply to some data, in an object based system the object (or the system supporting the object) decides which procedure to use.

The collection of messages to which an object can respond to is known as the *protocol* for that object. A possible subset of the protocol for the debtor object we referred to earlier might be:

addTransaction
whatIsYourBalance
whatAreYourTransactions
yourCreditLimitIs

A protocol also specifies the required arguments that are to be sent with the message. For instance *addTransaction* would require a transaction object as an argument, and *whatIsYourBalance* would require no arguments.

2.3 **Class**

A *class* is a template used to create new objects. The class contains at least the protocol for objects which are members of that class. The class generally also contains information about what variables are required to encode the state of the object. We call a particular object which is a member of a class an *instance* of that class.

Here are some important points about classes in class based languages like Smalltalk-80.

1. The class of an object determines which messages the object can respond to.

2. The class actually 'stores' the method.

3. The definitions of the instance variables are included in the class.

4. Instances of classes only contain the values of their instance variables and use the class for the rest of the information.

Classes also serve another purpose. They provide a mechanism for grouping objects. Classes impose structure on the 'sea' of objects.

A classless system is useful when dealing with 'small' systems. As an analogy, consider personal experience. If a person knows a cat, *Monty*, and then sees another cat they will probably think "that cat is like Monty except that it differs in these ways". This model is useful for a small number of animals. When we want to study the whole 'universe' of animals this is inadequate and we need some way of structuring the information. Classes provide this structure, together with inheritance which we discuss in the following section.

Classless systems are also useful when prototyping. In prototyping we normally have no clear perception of the system so we want to delay the imposition of structure on the solution until we fully understand the problem.

The difference between classical (have classes) and classless systems is similar to the difference between typed and untyped systems. Typed systems are useful for

production environments, untyped systems are useful for experimental environments.

2.4 **Inheritance**

In this discussion we view inheritance as only applying to class based systems. We consider the more general 'delegation' later.

Inheritance allows a class to inherit methods from its "superclass" and its methods may be inherited by its "subclasses". When an instance of a class *c* is created the new object can use the operations of class *c* and the operations of *c*'s superclasses.

When a class can have only one superclass directly above it this is termed *single inheritance*. The more general *multiple inheritance* allows a class to have more than one immediate superclass.

The relationship between the classes in the system is called the *class hierarchy*. In a system supporting single inheritance this hierarchy is tree structured. The classes in a multiple inheritance system form a directed acyclic graph.

Multiple inheritance poses a problem: if a class inherits from two classes, and there is a method or instance variable of the same name in both, which one do we choose? The most common method used is to select the method from the class with the higher 'precedence'. Usually this is the first superclass mentioned when declaring a new class. The language CommonLoops does it this way [Bobrow, Kahn et al 86]. The problem with any arbitrary means of deciding which one to use is that it may not be what the user intended. For a formal treatment of multiple inheritance see [Ducournau & Habib 1987] and [Cardelli 84].

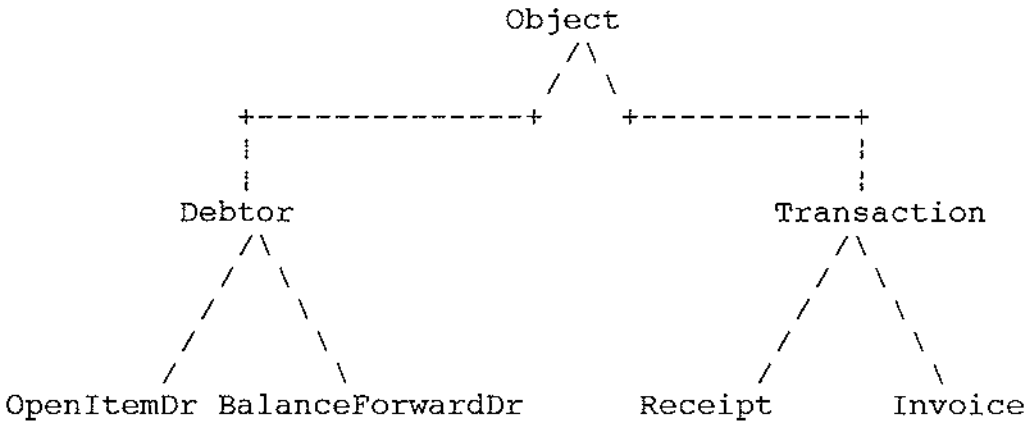


Figure 2.1: The Debtor System Inheritance Hierarchy.

We illustrate inheritance by returning to our debtors system. Figure 2.1 illustrates a possible inheritance hierarchy for this system. We have extended the example to include two types of debtors and two types of transactions. At the top of the hierarchy we have the *root* class called *Object*. The protocol for *Object* would include messages like: *class* - returns the class of the object; *respondsTo:* - informs the caller whether or not the object can respond to a given message.

We have introduced two types of debtor. The commonalities between them are captured in the *debtor* class. To allow for the different operations required for open item debtors and balance forward debtors we create two separate classes. The *OpenItemDr* debtors class will have messages like: *whatAreYourOpenItems*. The BalanceForwardDr debtors will have message such as *performPeriodAge*. The debtor class defines messages such as *whatIsYourAddress*.

The *Transaction* class includes common messages for both *Receipts* and *Invoices*. Some messages might be *howMuch* and *whatDate*. The *invoice* transaction type will have messages for getting the line item details.

We have used the inheritance hierarchy to classify the different classes (for example, *OpenItemDr is a Debtor*), and to group them together. As the inheritance hierarchy allows instances of the *Receipt* class to use operations defined for the *Transaction* class we are saved from re-implementing the common messages.

This illustrates the two uses of the inheritance hierarchy:

1. *Classification* - we classify objects by positioning them in a logical place in the hierarchy.

2. *Implementation* - the hierarchy is used to make implementation decisions based on the amount of code sharing desired between classes.

These two uses of the hierarchy do not necessarily coincide. We illustrate this by using the example presented in [LaLonde, Thomas et al 86].

Suppose we wish to implement a list by using two different representations, one for an empty list and another for a non-empty list. By doing this we can eliminate the need for empty list handling in the instances

of lists that are non-empty. We can do this by using the inheritance hierarchy thus:

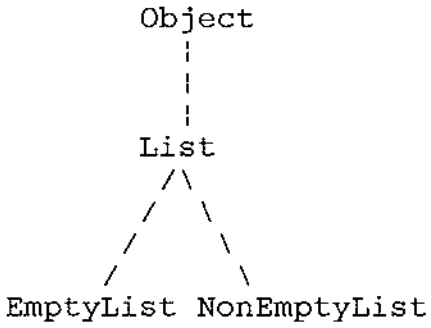


Figure 2.2: The List Inheritance Hierarchy.

Our *EmptyList* does not need any instance variables so we use less space. The *NonEmptyList* does not need to continually check to see if it is empty, giving us a speed increase. This representation shows how our class hierarchy is used to facilitate implementation. Problems arise, however when we wish to create a new class *AnnotatedList*, which is a list with notes attached to each element.

There are a number of ways of setting up an inheritance hierarchy:

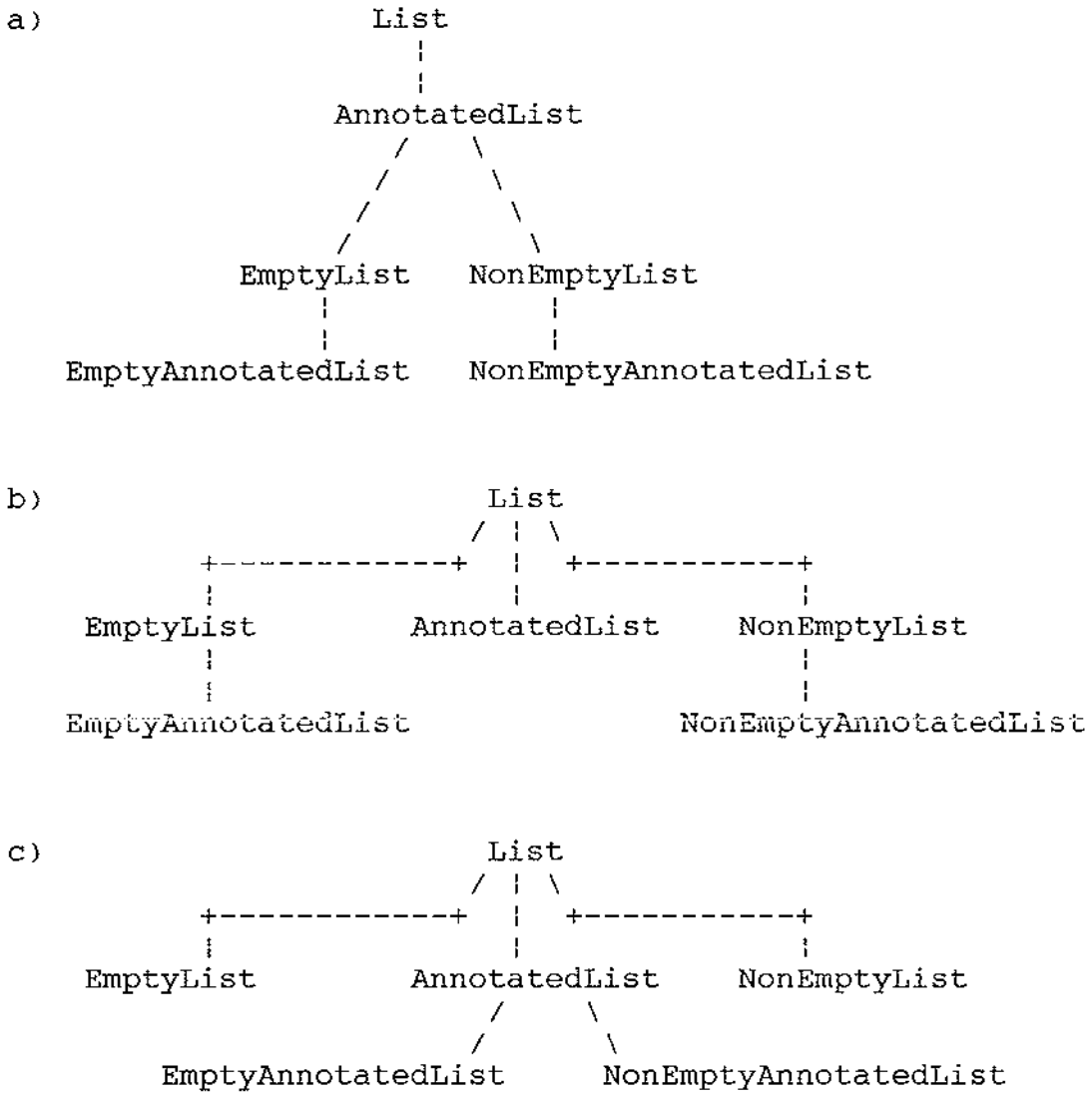


Figure 2.3: Possible Inheritance Hierarchies for Annotated Lists

Each of these representations causes problems:

a) causes lists to be incorrectly viewed as annotated lists.

b) implies *EmptyAnnotatedLists* and *NonEmptyAnnotatedLists* are not AnnotatedLists.

c) is not useful because the standard methods for EmptyLists and NonEmptyLists can not be inherited by the annotated versions without the use of multiple inheritance.

Clearly we need to separate the implementation and conceptual hierarchy. We come back to this problem, and our solution to it in OOPS-Algol, in Chapter 7.

2.5 **Delegation**

The other technique used to represent shared behaviour amongst objects is delegation. [Wegner 87] defines delegation thus:

> "Delegation is a mechanism that allows objects to delegate responsibility for performing an operation or finding a value to

one or more designated 'ancestors'".

Delegation operates independently from a class hierarchy, allowing a free-form approach to resource sharing. There are two useful metaphors for describing delegation. The first is the concept of a *prototype*, the second is the concept of the *extended self*.

2.5.1 Prototypes

The idea of a prototype is used to describe how objects are characterised in classless delegation. In our discussion of classes we mentioned a specific instance of a cat, *Monty*. Classless delegation would use Monty as the *prototypical* cat, and further cats would use Monty as their example to follow (the *exemplar*). A new cat *Rommel*, would *delegate* responsibility for operations that are common to both cats to the Monty object.

It can be argued that a prototype based system better represents the way humans learn than class based systems. When we see something new we generally compare it to something with which we already have experience. Only after we have seen many different examples of the new object type will we make the intuitive leap to being able to describe that object in general.

In the same manner in which prototypes are used in software engineering, we can use a prototype based object-oriented system to understand the problem we are trying to solve.

After we feel the problem is understood we can reimplement the solution in a class based system. We consider a class based system to be more suitable for a long lived application because the class hierarchy provides a convenient road-map, so that people who have to maintain the system can gain an understanding of the system as a whole.

2.5.2 **Extended Self**

The other concept that is useful when describing delegation is the idea that the ancestors of an object form the *extended self* of that object. This means that when an object requests a service of an ancestor, the ancestor will always refer back to the original object whenever a reference to *self* is made. This is in direct contrast to inheritance, which rebinds self as operations are performed further up the inheritance hierarchy.

We illustrate this with an example. Let us assume we have a class for simple display of fonts called *simpleFont*. We will select two messages for consideration:

displayAGlyph Takes a character as an argument and displays the corresponding glyph from the font.

displayLotsOfGlyphs Takes a string and displays it. To avoid duplication of the code used for displaying a glyph, it sends the *displayAGlyph* message to *self* for each character in the string.

At this stage inheritance and delegation will give the same results. *Self* remains bound to an instance of *simpleFont*.

Now we decide to enhance our class by adding a colour capability. This will use most of the code for *simpleFont* but will add the ability to display colour. Because *displayLotsOfGlyphs* uses the *displayAGlyph* message we should be able to just redefine *displayAGlyph* for the new *colourFonts* which will override

displayAGlpyh in *simpleFont* to enable us to display coloured glyphs (ignoring the obvious change in instance variables).

Under inheritance this will cause problems because *displayLotsOfGlpyhs* will make a call to *self* to display each one. This will not work, because *self* is rebound as a message works its way up the inheritance hierarchy. In this case it will be rebound to the simpleFont part of the hierarchy, not the colourFont part. The effect of this is that the message search for *displayAGlyph* will start from *simpleFont* when *displayLotsOfGlyphs* is evaluated. This will cause the loss of colour information so we would be forced to reimplement *displayLotsOfGlyphs* in our new class.

The concept of extended self provides a solution to this problem as we always view anything we inherit as extending the original object. This means that *self* stays bound to the original object, not rebound as the delegation hierarchy is traversed. In this case, the message search for *displayAGlyph* will start at the colourFont and so the object will behave as we would wish.

It is for this reason that we use this concept in OOPS-Algol.

3. Some Current Object-Oriented Languages

In order to see how OOPS-Algol fits in with other current object-oriented languages we will give a brief description of four of these languages. We begin with Smalltalk-80 as it was the language that popularised object-oriented programming. To demonstrate how object-oriented features are retro-fitted to existing languages we look at Objective-C and C++, two extensions to 'C'. Finally we look at Self, a language that uses classless delegation to implement resource sharing.

We follow a common format for the four languages where possible: demonstrating the syntax of message passing; how instances are created; and how resource sharing is supported.

We conclude our discussion of each language with reference to the ever popular stack which can respond to the following messages:

isMT	This message will return **true** if the stack is empty.

push: pushes an object (passed as an argument) onto the stack.

pop returns the top object from the stack and removes it.

After the stack has been defined, we will define an annotated stack to demonstrate the code sharing aspects of the various languages. The annotated stack behaves in the same way as the normal stack except that we have added two more messages:

attachNote: This will attach the note passed as an argument to the top item on the stack. This note will follow the item around in the stack until that item is removed from the stack.

getNote This will get a note that may have been attached to the top item on the stack.

We do not provide any error checking for the stack, which removes details not necessary to our discussion.

This chapter is not intended to be an exhaustive survey of object-oriented programming languages and we will be ignoring features of the languages not relevant to our discussion.

3.1 **Smalltalk-80**

Smalltalk-80 (which we henceforth refer to as Smalltalk) was the first popular object oriented language. It was developed at Xerox PARC, with work on its design starting in the early 1970's. It was designed specifically as an object-oriented language with no concessions to 'traditional' programming languages.

Smalltalk is more than just a language; it is also a development environment encompassing a browser and other tools to assist the development process. The environment is persistent as it is saved between invocations.

For a full description of Smalltalk see [Goldberg & Robson 83].

3.1.1 **Message Passing**

A message expression in Smalltalk consists of a *receiver*, a *selector* and arguments if required. Following the message analogy the receiver receives the message, the selector is used to select the appropriate method for the message, and the arguments (if any) are the data required for that message. Each message in Smalltalk returns a value.

The message expression syntax in Smalltalk is designed to be readable, and can resemble the syntax of traditional languages in some cases. There follow some sample expressions to give the feel of Smalltalk:

(i) 3 + 4

In this case the receiver of the message is the object 3. The selector is '+' (selecting the addition operation) and the argument is 4. This illustrates the purity of Smalltalk, even integers are objects in the system (there are internal optimisations to speed the usage of integers however).

(ii) quantity sqrt

In this case the receiver is *quantity* and the selector is *sqrt*. The object referenced by *quantity* is asked to return the square root of itself.

(iii) values replaceFrom: 1 to: oldValues size with: oldValues

Here we begin to see the readability of Smalltalk syntax. The receiver here is *values* which is sent a message with the selector *replaceFrom:to:with:*. One of the arguments is the result of another message expression: *oldValues size*. This is evaluated first as it is a *unary expression*, that is, a message with no arguments.

(iv) sizeOfThing <- thing size

In this case we instantiate *sizeOfThing* with the value of the message expression *thing size*.

Smalltalk allows the user to delay a sequence of actions by use of a *block*. A block is treated as any other object, and is sent the message *value* to evaluate it. For example:

```
(number \\ 2) = 0
    ifTrue: [parity <- 0]
    ifFalse: [ parity <- 1]
```

In this example *number* is examined to decide if it is

odd or even. If it is even the block *parity <-0* is evaluated, otherwise the block *parity <-1* is evaluated. Any boolean object can have the message *ifTrue:ifFalse:* sent to it, and the receiver in this case is the *(number \\ 2)=0* expression. Note that the block shares the context (instance variables and arguments) of the expression in which it is evaluated.

3.1.2 **Instance Creation**

Instances of a class are created by sending the message *new* to the instance's class.

```
aNewDictionaryInstance <- Dictionary new.
anMTArray <- Array new: 10
aNewInt <- 1
```

Note the use of literals to create instances of a class. It can be viewed as shorthand for "send the new method to the class, initialise the instance to be the value as indicated by the literal". The type of the destination is not explicitly declared as it is determined by the right hand side of the expression when it is evaluated.

3.1.3 **Resource Sharing**

Smalltalk has a class hierarchy supporting single inheritance. Smalltalk added the concept of an *Abstract Class* which is used to describe a protocol used by similar sub classes, but you can not create an instance of an abstract class. The abstract class is used to combine the common properties of its subclasses.

Smalltalk also has the concept of a *Metaclass* which is a class of classes. This is used to handle messages which are sent to a class, not an instance of that class. This is used when we want to send different initialisation messages to different classes. When we create a new instance of the *Date* class we might want the instance returned to represent today so we send the message *today* to the class *Date*; however if we want to created a new point we will want to give it its initial x and y values. The metaclass allows us to describe the messages applicable to a class in the same way the class describes the messages for an instance of that class.

The hierarchy of instances has a one to one correspondence to the hierarchy of classes. This can cause some logical inconsistencies as the determination of a superclass is based on implementation

considerations. We discuss this problem more fully in Chapter 6.

3.1.4 **Stack**

In the interests of accuracy we present our Smalltalk stack in Little Smalltalk [Budd 87] which is in the public domain. We simulate the use of message categories used in Smalltalk-80 by indicating the categories in comments, which in Little Smalltalk are surrounded by double quotes.

```
Declare Stack Object tos theStack  " This declares Stack as a
                                     subclass of Object, with
                                     the instance variables:
                                     tos and theStack"

Class Stack                        " Signifies definition
                                     of methods for class
                                     stack"
"Message Category: initialisation"
      new
            ^ self initialise  " Send the initialise message to
                               set up the instance variables"
|
      initialise
          tos <- 1.
          theStack <- #(nil,nil,nil,nil,nil,
                        nil,nil,nil,nil,nil)
!
"Message Category: queries"
      isMT
        ^ (tos = 1)           " Returns true or false"
!
"Message Category: Alteration"
      push: aThing
            theStack at: tos put: aThing.
            tos <- tos + 1
!
"Message Category: Accessing"
      pop
            tos <- tos - 1.
            ^ theStack at: tos

]
```

```
Declare AnnotatedStack Stack notes

Class AnnotatedStack

"Message Category: Initialisation"
      new
           ^ super new initialise
!
      initialise
          super initialise.
          notes <- #(nil,nil,nil,nil,nil,nil,nil,nil,nil,nil)
!
"Message Category: Alteration"
      attachNote: newNote
          notes at: tos - 1 put: newNote
!
"Message Category: Accessing"
      getNote
          ^notes at: tos-1
]
```

The following is done from the interpreter

```
globalNames at: #aNewAnnotatedStack put: AnnotatedStack new

aNewAnnotatedStack push: 10

aNewAnnotatedStack attachNote: 'A note'
```

A few things appear to need explanation. The first is the use of the cascaded message expression 'super new initialise' in the *initialise* method for the AnnotatedStack. This is equivalent to :

```
new
     | newOne |
     newOne <- super new.
     newOne initialise
```

The second is how a new instance is declared when running the interpreter. In Little Smalltalk you can compile class definitions to produce an image that is loaded by the interpreter. This is what the first half of our stack example is. When you are in the interpreter however, you have to create an instance by creating a new place in the *globalNames* dictionary, and assigning a new instance to that position. This is what the: 'globalNames at: #aNewAnnotatedStack put: AnnotatedStack new' expression does.

3.2 C++

C++ was designed by AT&T to update and replace C. It was designed to be upwards compatible with existing C code as AT&T did not want to support two languages. Another major design constraint on C++ was that it should have runtime execution speed similar to C's.

Although C++ supports object-oriented programming, it does not force it and the extensions to C are such that the language would be worth using even if one was not using the object-oriented extensions.

In keeping within the design constraints, C++ is generally implemented as a preprocessor which produces C code which is then compiled by the local C compiler. There are a number of C++ compilers that skip the C code generation step and produce object code directly (G++ [GNU 88] and Zortech C++ are such examples).

C++ keeps it execution speed acceptable by not using method invocation as in Smalltalk, but selecting the appropriate function by either:

a) finding the function at compile time if it is not a virtual function.

b) finding the function based on the type of the receiver at runtime by using a compiler computed offset into a table of functions for that object.

The difference between this approach and that of Smalltalk is that Smalltalk **looks up** the table of methods for that object, as it can not know the offset at compile time. For a more detailed discussion of this see [Stroustrup 87b].

No automatic memory reclamation is provided and obsolete objects have to be explicitly destroyed. There is no concept of an 'environment' built-in to the language so considerable housekeeping is required if objects are required to persist beyond one program invocation. The compiler has no memory of previous compilations so shared information has to be handled by included text files.

3.2.1 **Message Passing**

Message passing in C++ is modeled by using structure field selection. That is, you can select a function as part of a structure to execute. This enables compile time checking, and speeds the process of message 'switching' considerably, because the location of the method is determined at compile time (virtual functions are accessed through a runtime set pointer, but the overhead is minimal). Naturally this approach loses some flexibility but since C++ is not really intended as a prototyping language this is not significant. Consider the following example messages:

(i) complexNmr.printOn(ouputStream)

The receiver of the message is *complexNmr*, the selector is *printOn*, and the argument is *outputStream*. This

follows the normal C syntax for structure field access where *complexNmr* is the structure name and *printOn* is the field required. The syntax is extended to make the calling of a function from that structure more palatable (normally C requires the field to be a pointer to a function which requires a rather ugly statement to call).

(ii) currentPC = virtMachine.getPC()

In this case the message *getPC* requires no arguments and the receiver is *virtMachine*. The value returned will be the program counter of the *virtMachine*.

Although the message passing looks like structure field selection it should be remembered that the field need not exist in the class of which instance is a member, but it may exist in one of its superclasses.

3.2.2 **Instance Creation**

Instances are created by the *constructor* of a class in C++. When an instance is no longer to be used the programmer explicitly requests for it to be destroyed using the *destructor* for that class (unless it was created on the stack, in which case it will be destroyed by the system automatically). C++ uses overloading to

provide similar functionality to Smalltalk's *new* messages. In C++ there is no explicit *new* message, instead the declaration of a variable for a new instance is used to call the constructor. Some examples follow:

```
(i)    vector newVector(100);
```

The class of *newVector* is *vector*. When this is executed the constructor for *vector* is called and passed the argument 100 (that is the vector is 100 elements long).

```
(ii)  point newPoint(2,4);
      point copyOfNewPoint = newPoint;
```

In this case *newPoint* will be an instance of *point* with its x value set to 2 and its y value to 4. *copyOfNewPoint* will be a copy of *newPoint*.

This scheme fits in well with the syntax of C but is perhaps a little obtuse if you are used to Smalltalk-like *new* messages.

3.2.3 **Resource Sharing**

C++ supports single inheritance by including the name of a superclass in the class definition. It does not support multiple inheritance (although some experiments have been done on this [Stroustrup 87a]). To

provide the functionality of Smalltalk's Abstract Class, C++ allows a class to be defined with *virtual* functions which are instantiated with the appropriate function when an instance of that class is created.

3.2.4 **Stack**

Here is our stack in C++. This was compiled with the GNU C++ compiler, known as G++. We place reserved words in bold to assist readability.

```
#include <std.h>
#include <stdio.h>
#include <stddef.h>

// Stack object for C++ (Actually G++ - the GNU C++ Compiler)
class Stack {
    int theStack[10];  // Private information

protected:  // Anything following here is accessible by classes
            // derived from this one
    int tos=0;

public:    // Anything following here is publically accessible

    int isMT()          {return(tos==0);}
    void push(int item) {theStack[tos++] = item;}
    int pop()           {return(theStack[--tos]);}
};

// Now we define the annotated stack which derives some of its
// behaviour from Stack
class AnnotatedStack : public Stack {
    char *notes[10];       // Array of strings
public:
    void attachNote(char *note) {notes[tos-1] = note; };
    char *getNote()        {return(notes[tos-1]); };
};

// Create a stack and use it
main()
{
    AnnotatedStack newStack;       // Create a new instance of
                                   // Stack using the default
                                   // constructor.
    newStack.push(10);             // Pushes 10 onto the stack
    newStack.attachNote("was 10");// Attaches a note.
    newStack.push(20);             // Pushes 10 onto the stack
    newStack.attachNote("A note");// Attaches a note.
    fprintf(stderr,"Top Note: %s,",newStack.getNote());
    fprintf(stderr,"pop = %d\n",newStack.pop());
    exit(0);
}
```

As can be seen from the example C++ allows three levels of protection on the members of the class structure. The first part is the private part whose members

can only be accessed by *friend* functions defined within the structure. Then there are the *protected* members which can be accessed by any other class deriving some of its behaviour from the structure. Finally there are the *public* members that are accessible from everywhere.

It should be noted that it is not possible to declare the stack as being able to contain any type of object in C++. Instead it must be built for a specific data type.

3.3 **Objective-C**

Objective-C [Cox 86] is another C hybrid. It supports runtime message switching so is a little slower than C++, but faster than Smalltalk would be on the same processor. It comes with a rich set of classes which C++ does not have. Its syntax is modelled after Smalltalk. As in C++ a precompiler produces C code to be compiled by the normal C compiler on the system. However it does require run time support to execute (which is linked with the object code produced).

Objective-C adds a type, *id*, to the C type system. A variable of this type is used to hold the identification (actually the address) of an object. In comparison, C++ defines the type of each instance to be the instance's class. This difference is necessary as Objective-C performs its type checking at runtime, whereas C++ does it at compile time.

3.3.1 **Message Passing**

Objective-C's message passing syntax is modelled on the Smalltalk language, whilst allowing the compiler to compile existing C programs correctly. Message expressions are surrounded by square brackets. Consider the following examples:

```
(i)  sizeOfaSet = [aSet size];
```

The receiver is *aSet* and the selector is *size*. The result of the expression is used to set the value of *sizeOfaSet*.

```
(ii) if ([virtMachine getWordAt: address] == 0)
        printf("The address is zero0);
```

In this case the receiver is *virtMachine* and the selector is *getWordAt:*. The argument is *address*. The result of the expression is used in the standard C expression

and tested against 0.

As can be seen the syntactic 'sugar' is sweeter than C++, especially if you are used to Smalltalk type languages.

3.3.2 Instance Creation

All objects are of type *id*, which is a pointer to the structure representing the object. The message *new* is sent to the *Factory object* (which corresponds to Smalltalk's Class) which returns a new instance of the object. By convention in Objective-C, factory objects begin with a capital letter, while instances begin with small letters. For example:

```
id aNewSet = [Set new];  /* Set is the factory object */

id anArray = [IdArray new:100];
             /* Create an instance of idArray 100 long */
```

These examples show how we use *id* as the type name of all objects in the system, regardless of what class they belong to.

3.3.3 **Resource Sharing**

Objective-C supports single inheritance in a manner similar to Smalltalk, but has neither metaclasses nor abstract classes. However Objective-C does have what it calls *factory methods* which are basically equivalent to Smalltalk's Class methods. Class variables can be simulated in Objective-C by using normal C global variables.

3.3.4 **Stack**

```
/* Stack object for objective-c */

/* We can create stack of objects here, unlike in C++ */
= Stack : Object { int tos=0; int theStack[10]; }
- (int)isMT {
        return (tos == 0);
}
- (void)push: (int)newInt {
        theStack[tos++] = newInt;
}
- (int)pop {
        return(theStack[--tos]);
}

/* Now we define the annotated class */
= AnnotatedStack : Stack { char *notes[10]; }
- (void)attachNote: (char *)note {
        notes[tos-1] = note;
}
- (char *)getNote {
        return(notes[tos-1]);
}

/* Now we create an instance of the stack and use it */
id newStack =  [AnnotatedStack new];
[newStack push: 10 ];
[newStack attachNote: "A Note"];
```

Objective-C denotes the beginning of a class definition by the equals sign and the definition of each instance method by the minus sign. It also allows the definition of factory methods by the plus sign but this is not shown here.

It would have been possible to declare the stack as being able to contain any type of object by merely making the array type *id* which is a pointer to any type of object.

3.4 **Self**

Self [Ungar & Smith 87] is a recent object-oriented language that is based on three simple ideas: prototypes, slots, and behaviour. Unlike Smalltalk there is no concept of a class, or instance variables. In Self everything is an object but instead of having a class pointer as in Smalltalk a Self object has a pointer to its parent object.

There is no direct way to access a state variable in Self. It is best to explain this with an example. If we have a point object with an *x* and *y* part we would create a self object with a slot with the name of *x* and

a slot with the name *y*. When the message *x* is sent to this point object the object contained in the *x* slot is evaluated, returning the current value of *x*. If we want to change the value of *x* and *y* we also create slots with the names *x:* and *y:* which are methods used to change the variable. This technique allows an ancestor object to replace the state accessing methods of its parent with other methods. One use could be to replace the *x* message with a random x generator.

3.4.1 **Message Passing**

The syntax of Smalltalk has been retained in Self where possible, and extended to allow for creating slot lists. The other difference is that what would be instance variable accesses in Smalltalk are messages sent to *self*. This is illustrated in the stack implementation below.

3.4.2 **Instance Creation**

To create a new instance in Self we merely copy a previously existing object. This is know as cloning. The idea of cloning existing objects from the prototype has the advantage that if we want to create a one-of-a-kind object we do not have to create a class just to support that object. An example follows:

```
aNewDictionary: Dictionary clone.
```

In this case the slot *aNewDictionary* will contain a clone of the *Dictionary* object. Note the use of the colon to assign the value to the slot. Actually we are sending a message with the selector *aNewDictionary:* to *self*.

3.4.3 **Resource Sharing**

Self only allows a single parent so it only implements single 'inheritance'. Self differs from the other languages considered in that it is possible to have objects dependent on any other object, replacing what would be instance variables at will. For instance it is possible to create a point object constrained by the x value of its parent point by overriding its parent's *y* and *y:* messages and leaving the x messages alone. This means that the parent could be moved in the x direction, taking its ancestor along with it.

3.4.4 **Stack**

Before we look at the example we need to explain the way Self denotes inheritance and slots. In Self, passive objects and blocks are enclosed in square brackets, and methods are enclosed in braces. The slot list

is enclosed in vertical bars and each item in the list must be separated from the next by a period. In Self the word *self* can be left out so that saying:

```
theArray: theArray clone
```

is equivalent to

```
self theArray: self theArray clone
```

presuming that *theArray* is a slot accessible from the current object.

There are several forms for slots which we only briefly detail here. For more information see [Ungar & Smith 87]:

- A selector by itself denotes two slots: One initialised to *nil* and one named with a trailing colon initialised to the assignment primitive (<-). In our example the *aNewAnnotatedStack* slot in the *doStackTest* method is like this.

- A selector followed by an equals sign and an expression denotes one slot, initialised to the expression. No assignment slot is created, so the slot is read-only. This is generally used for

method definitions but is also useful for the prototypes.

- An identifier with a trailing colon followed by a left arrow (<-) defines an assignment slot that can be use to change the value of a read-only slot with the same name elsewhere. An example of this can be found after the definition of the *emptyStack*.

Inheritance in Self is denoted by lexical scope. So in the example *emptyStack*'s parent is *Stack*, and *AnnotatedStack*'s parent is also *Stack*. Stack's parent is the root object of the system. We find this notation a little unclear but it does simplify the syntax of the language.

As in our Smalltalk example we indicate comments by preceding them with an exclamation mark.

```
[ |                              ! This is the root object.
  clone = {<primitive>}.         ! Initialises a slot containing the
                                 ! clone primitive.
  nil = [].                      ! Most basic object with no slots.

  ! Now we begin the definition of our stack object
  Stack = [|
      ! Here is the prototypical stack which is assigned to the
      ! emptyStack slot.
      emptyStack = [|
            tos = 1.
            theArray = #(nil,nil,nil,nil,nil,
                         nil,nil,nil,nil,nil)|].

      !Define some slots so that they can be assigned to

      tos:<-.
      theArray:<-.

      ! We clone by cloning our parent object, then each of the
      ! elements of the prototype stack is cloned and placed in
      ! the appropriate slots of the new object.
      clone = {
          super clone tos: tos clone theArray: theArray clone
      }.

      ! Now define the methods
      push: obj = { theArray  at: tos put: obj.
                    tos: tos + 1. }
      pop = { | valueWas. |
          tos : tos - 1.
          valueWas: theArray at: tos.
          ^valueWas
      }
      isMT = { ^tos = 1 }
```

```
        ! Here is the child which will be inheriting the
        ! properties of stack, as it is in the same scope.

        AnnotatedStack = [|

            ! Our prototypical stack, note that we only define
            ! what is different
            emptyAnnotatedStack = [|
                 notes = #('','','','','','','','','','')
            |].

            ! Make notes changeable.
            notes:<-.

            ! We clone our super which is now a Stack and then
            ! place a copy of notes from the prototype object
            ! into our notes
            clone = {
                super clone notes: notes clone
            }.

            ! Here are the methods
            attachNote: newNote = { notes at: tos-1
                                        put: newNote. }.
            getNote = { ^notes at: tos-1. }
        |].  ! End of the AnnotatedStack slot list

    |]. ! End of the Stack slot list

    ! Now we create a slot which will perform some stack
    ! operations when evaluated
    doStackTest = {
        | aNewAnnotatedStack. |

        ! First we create a new stack, note that we have to
        ! get the AnnotatedStack from the Stack object by
        ! sending it the AnnotatedStack message as it is not
        ! visible in this scope. Then we get the prototype from
        ! the AnnotatedStack slot and clone it.
        ! This value is shoved into the aNewAnnotatedStack.
        aNewAnnotatedStack:
              Stack AnnotatedStack emptyAnnotatedStack clone.

        ! Now shove a value on it, and then attach a note.
        aNewAnnotatedStack push: 10.
        aNewAnnotatedStack attachNote: 'A note'
    }
|].
```

3.5 **OOPS-Algol**

We finish off this chapter with a very brief description of the ideas that OOPS-Algol has borrowed from these languages.

OOPS-Algol includes the same message expression syntax as Smalltalk. We use message categories to group our messages as in Smalltalk. OOPS-Algol does not contain the one-to-one instance/class relationship used by Smalltalk but uses an approach similar to that described in [LaLonde, Thomas et al 86].

OOPS-Algol uses structures to hold instance variables as in C++. C++ uses a preprocessor (in some implementations) to extend C, and we do the same with OOPS-Algol. The type of variables is declared in OOPS-Algol, as in C++, but we use subtyping to provide the versatility lacking in C++.

We have used Objective-C's technique of surrounding message expressions with brackets to simplify the parsing of message expressions. We do not use the generic type, *id*, used in Objective-C to specify an object type but use the name of the class of which the object is a

member. This is to add type security.

We had originally considered using messages to access instance variables in OOPS-Algol as in Self but this would have compromised our use of subtyping, as the messages for accessing these variables would have been included in the class definitions. We consider instance variables for a class to be private to all but those classes inheriting from that class, so this was not desirable.

We do have the idea of a prototypical object, as in Self, which is cloned to make new instances of a class. OOPS-Algol calls this prototypical object the *exemplar* for a class.

4. Subtyping in Class Based Systems.

In order to provide some perspective on the separation of conceptual and implementation hierarchies in OOPS-Algol we will review the ways subtype hierarchies are used in other common object-oriented systems. It should be noted that we are considering systems like Smalltalk, Objective-C and C++ where the implementation and conceptual hierarchies are not separated. In these languages the two hierarchies have a one-to-one correspondence with the hierarchies being defined by the subclass/superclass relationships. In this context subclass and subtype are synonymous, as are superclass and supertype.

This chapter is based in part on [Halbert & O'Brien 87] which provides a good description of the use of subtypes in class based object-oriented languages.

4.1 What is subtyping

We use a simple definition of a subtype:

> If B is a subtype of A, B can be used where ever A can be used.

Also, in the systems we are considering here, a subtype may share or inherit characteristics of its supertype. The characteristics that are usually shared include the storage representation of the supertype and the operations provided by the supertype.

The inherited characteristics can be overridden by a subtype. A subtype may reimplement the code to execute upon receipt of a message, or may augment its storage representation with additional information.

4.2 The uses of subtyping

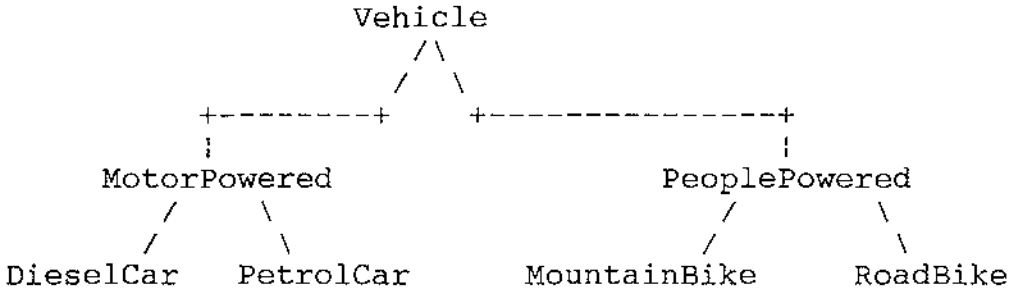
There are two major categories of usage of subtyping in object-oriented languages. The first category, which we call *standard* usage, captures the conceptual relationships between components of a system. The other category, *nonstandard*, uses the type hierarchy to

increase the amount of code sharing.

The standard uses of subtyping are specialisation; interface specification; and combination. The nonstandard uses are generalisation and variance. We consider the uses in that order.

4.2.1 **Specialisation**

In this case the type hierarchy is used to model a conceptual hierarchy, with the most general type on top, and more specialised types below. This hierarchy can be used to represent a model of the real world as in:



Specialisation can be used to capture common behaviour of objects in the system. For example, the supertype *Process* may be used to capture behaviour common to *RealtimeProcess* and *BatchProcess*, which are

specialised versions of *Process*.

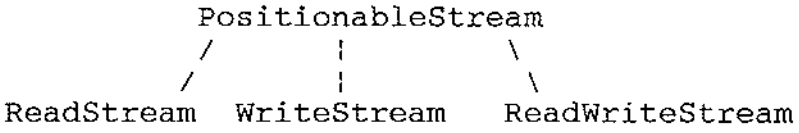
Another use for specialisation is to model intangible concepts that are external to the program. For instance, a hierarchy might model the relationship between fiction and non-fiction books.

In all of these cases the subtype *specialises* the behaviour of the supertype. If we treat types as sets of values the subtype restricts the definition of the supertype in order to create a subset of that set.

4.2.2 Interface Specification

Subtyping can be used to guarantee that instances will present a certain interface to other objects. Here the supertype is an abstract type which is used to define a common interface among its subtypes. We can not actually have an instance of this abstract type. The subtypes thus provide various implementations of their supertype.

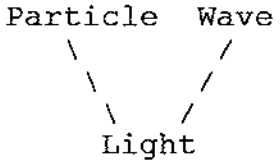
An example from Smalltalk demonstrates this usage:



This describes an abstract class, *PositionableStream*, which defines the interface to various types of streams: *ReadStream*, *WriteStream*, *and ReadWriteStream*.

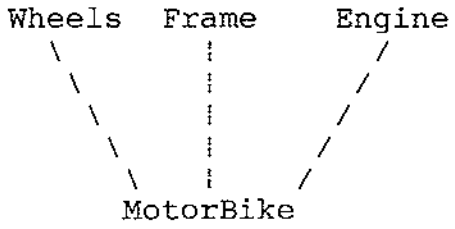
4.2.3 **Combination**

Given the use of multiple inheritance, that is, a type system when a subtype can have more than one supertype, we can use subtyping to combine the properties of types. For example:



In this way we can treat Light as a wave, or as particles.

It is tempting to use multiple supertypes to model the components of an object. Consider this example:

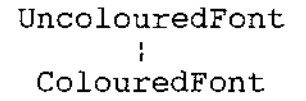


This is incorrect as a *MotorBike* is **composed** of the supertypes in the diagram. The error becomes obvious when you consider the fact that modern motor bikes can have two different sized wheels. That would require changing the type hierarchy so that there were two wheels as supertypes of Motorbike which is clearly wrong as both wheels have the same type. They are in fact different instances. [Blake & Cook 87] describe an extension to Smalltalk to support part hierarchies that does not resort to this use of subtyping.

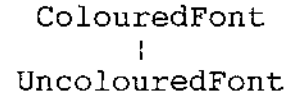
4.2.4 **Generalisation**

Subtyping for generalisation is used to create a more general type of object than its supertype. This usage is based purely on implementation issues as we normally want to generalise an existing type so that we can share some of the implementation of the supertype.

A demonstration of this is our coloured font example presented in Chapter 2. We began with a one colour font and extended it to handle different colours by adding a colour attribute and some messages to change the colour. The hierarchy looked like this:

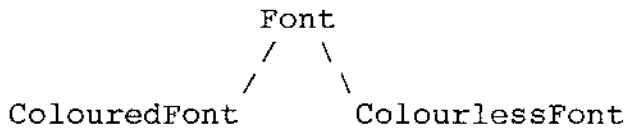


However, we could actually consider the one colour font to be a special case of the coloured fonts, and as such it would be preferable to have the one colour font as a subtype of coloured fonts:



The problem with this solution is that we are allowed to send colour specification messages to a member of *UncolouredFont* because it is a subtype of *ColouredFont*. Sending a colour message to *UncolouredFont* is really an error, and should, if possible, be detected at compile time. In order to detect such an error using the simple hierarchies presented above, *UncolouredFont* must be the supertype of *ColouredFont*.

The way around this problem is to create an abstract type that captures the functionality of both. We then create two subtypes, one for each type of font:



Now protection against inappropriate messages is possible, and the hierarchy follows a more reasonable model.

4.2.5 **Variance**

The final use of subtyping is for variance. Here we make one type a subtype of another solely because the supertype has common code we would like to reuse. Conceptually the two types should be siblings but implementation issues dominate our organisation.

As an example of this usage we may have an input device that can be either a mouse or a trakball (which is basically an upside down mouse). The implementations would be almost exactly the same, as both devices have a ball that rotates in two dimensions. We could decide to make one of the types a supertype of the other so that we can share the common code.

4.3 Subtyping in OOPS-Algol

The various uses of the subtyping mechanism we have described demonstrate that we have to weigh implementation issues against conceptual design issues when defining a type hierarchy. We are forced to shape the type hierarchy either to give us maximum inheritance, or to provide the best logical model of a system.

OOPS-Algol has separated the inheritance hierarchy (which we call the exemplar hierarchy) from the type hierarchy (called the class hierarchy) in an attempt to alleviate the problem caused by such tight coupling. Its type hierarchy specifies the **interface** to instances of that type, and the implementation hierarchy defines various implementations which inherit from each other. This changes the uses of subtyping discussed above as follows:

1. We do not need to use *subtyping* for interface specification as that is the function of type hierarchy.

2. We do not allow multiple inheritance yet so we can not use subtyping for combination.

3. An implementation of a type can inherit from any other implementation, so we do not need to use the type hierarchy for variance because an exemplar can inherit from a sibling in the type hierarchy.

This only leaves two uses of subtyping - specialisation and generalisation. We have seen that we can design around the use of subtyping for generalisation, and specialisation is what we want to use the subtype mechanism for anyway.

5. The OOPS-Algol Language.

In this chapter we illustrate our message expression syntax and provide a brief introduction to defining classes and exemplars in OOPS-Algol. We round the chapter off with the OOPS-Algol implementation of the stack used in Chapter 3.

5.1 Message Expressions

OOPS-Algol's message expressions follow those of Objective-C. We surround a message expression with square brackets and use the message syntax of Smalltalk. We will illustrate this with examples which will also demonstrate important features of PS-Algol. A full definition of PS-ALgol is provided by [PS-Algol 84].

We support two types of message expressions in OOPS-Algol. These are unary and keyword message expressions which we consider separately.

5.1.1 **Unary Message Expressions.**

A unary message expression is the simplest kind of message expression. It has a receiver and a selector. No arguments are passed. Consider the following examples:

(i) **let** aStack = [stackExemplar clone]

This example defines a new variable called *aStack*, initialising the value of *aStack* with the result of the unary message expression *stackExemplar clone*. The expression sends the *clone* message to *stackExemplar* which is our prototypical stack. As in Self we have a prototypical object (called the exemplar) which we clone to make new instances. The message will return a new object with all of the attributes of *stackExemplar*.

In PS-Algol a variable declaration is preceded by **let**. An equals sign (=) following the variable name indicates that the name is a constant and a semi-colon equals (:=) indicates that it is a variable. Thus *aStack* is a constant and can only refer to the object created. It is possible to change the state of the object that is referred to by *aStack* however.

PS-Algol determines the type of a variable by the type of the expression on the right hand side of the declaration. We have implemented OOPS-Algol to support this and in this case it will mark *aStack* as type *Stack* which is the class of *stackExemplar*. This enables us to type check our OOPS-Algol programs in a strict manner, unlike Smalltalk or Objective-C. We discuss this more fully in Chapter 6.

We can use a message expression wherever we can use an ordinary expression in PS-Algol. We can also use a function returning an object as the receiver in a message expression:

(ii) **let** wasItEmpty := [functionReturningStack() isMT]

This example defines *wasItEmpty* to be a boolean variable containing the result of sending the message *isMT* to the object returned by the function. We know that the variable will be boolean because the *isMT* message is defined as returning a boolean value in the class definition of Stack. When OOPS-Algol compiles the message expression it will make sure that *functionReturningStack* will return a Stack or a subtype of Stack so that the *isMT* message will not fail.

5.1.2 **Keyword message Expressions.**

Message expressions that take arguments are known as keyword expressions in Smalltalk. OOPS-Algol retains the same terminology. We present here an example with keyword expressions that illustrates other features of PS-Algol.

Consider a code fragment that prints all of the elements of *aList* onto the file called *aFile*, or until 100 elements have been printed:

```
let outFile := create("aFile",493)  ! Open the file for writing

let executeMe = proc(Object objectToWorkOn)
{
    [objectToWorkOn printOn: outFile]
}

let count := 0

let checkForEnd = proc(->bool)
{
    count := count + 1
    (count > 100)
}

[aList forEach: executeMe until: checkForEnd ]
```

The fragment begins by opening a file for output by calling *create* which is a predefined PS-Algol procedure that returns a file opened for writing. Assigning the file returned by this to *outFile* defines its type to be

file. PS-Algol treats all of its data types as first class, so we define a procedure by declaring a variable or constant that refers to the *closure* of the procedure necessary to execute the procedure correctly. The closure contains two parts: the code to execute the procedure; and the procedure's environment, which contains the local and free variables of the procedure. In *executeMe* the global variable *outFile* is accessed so this is included in the closure along with *objectToWorkOn*. *executeMe* contains one keyword message expression. It sends a message with the selector *printOn:* to the object passed to executeMe as an argument. The message requires one argument, the file to print on. For a more detailed discussion of procedures as first class data see [Atkinson & Morrison 84].

The next procedure defined, *checkForEnd*, returns a boolean value which is the result of evaluating *count > 100*. To return a value from a PS-Algol procedure we place an expression of the appropriate type at the end of the procedure.

The example finishes with a message which uses both of these procedures. The final message expression sends a message with the selector *forEach:until:* to *aList*.

Using OOPS-Algol this message is defined as:

```
forEach: procedureToExecute until: endTestProcedure
                    typeIs proc(proc(Object),proc(->bool))
```

which shows the message selector, together with dummy names to indicate the purpose of each argument. This is followed by the PS-Algol style type definition of the message. This is considered in more detail later.

It is worth noting that when we pass the procedures *executeMe* and *checkForEnd* as arguments to the message they are not evaluated. PS-Algol denotes evaluation of a procedure by following the name with parentheses (which may contain arguments). In this case we are merely passing the closure of the procedure to the method.

5.2 **Creating a Class**

Now we have examined message expressions we will introduce our method of defining a class. In OOPS-Algol we have a class definition which defines the *type* of a member of that class. This type specification is based solely on the interface to an instance. It contains no information about the implementation of that instance. We define the implementation of a member of a class in

the *exemplar* for that class (discussed in the following section).

In OOPS-Algol our Stack class definition would look like this:

```
Class Stack superClassIs Object
{
    "Implements a simple stack containing any object"
    messagesFor "queries"
                isMT typeIs proc(->bool) "True if MT"
    messagesFor "alteration"
                push: newObject typeIs proc(Object)
                                "Pushes an Object"
                pop typeIs proc(->Object)
                            "Pops and returns the top object"
} ! end of Class Stack definition
```

A class definition begins by specifying the name of the class and its superclass. The class description is surrounded by braces (or **begin end** pairs). The first string is the description of the class which is accessible by *oopsdump* (and any future browser). We group the message definitions by category as in Smalltalk but use **messagesFor** to mark the category name. In the class above, *isMT* is in the *queries* category and *pop* is in the *alteration* category. We decided to make the category

name a user defined string for added versatility.

In defining the messages for a class we wanted to denote the message selector; the arguments to the message; the type of the arguments ; and the type of the value returned by the message (if any). We decided on a combination of the Smalltalk approach (for readability) and the type specifications of PS-Algol (to save learning a new way to specify the types). The two parts are separated by the **typeIs** reserved word. We come back to the type specification in Chapter 6.

5.3 **Creating an Exemplar for a Class**

After having defined the class we need to specify how an instance of that class is to be implemented. We do this in OOPS-Algol by associating one or more exemplars with a class. Here is the appropriate exemplar for the Stack class defined above:

```
Exemplar aStack forClass Stack  superExemplarIs anObject
{

    stateIs {
        let stackIs := vector 1 :: 10 of anObject
        let tos := 1
    }

    method isMT typeIs proc(->bool); (tos = 1)

    method push: newObject typeIs proc(Object)
    {
        stackIs(tos) := newObject
        tos := tos + 1
    }

    method pop typeIs proc(->Object)
    {
        tos := tos - 1
        stackIs(tos)    ! Returns the top object.
    }
} !aStack
```

The exemplar definition begins with the name of the exemplar, in this case *aStack*. We then specify the class to which this exemplar belongs and the exemplar from which it is to inherit methods and instance variables. OOPS-Algol checks the exemplar methods to ensure that all definitions match those of the class, and that there is a method defined for every message that is specified for an instance of that class.

The body of the definition begins with the declaration of the instance variables which we call the state. We follow PS-Algol syntax for the variable declarations

to avoid confusion. Then the method for each message is declared. To increase the readability of the exemplar we decided to duplicate the method definition as used in the class definition in the exemplar body. We consider the redundancy presented not to be a problem because if the message definition changes (the type of an argument for instance) the method would almost certainly have to be changed. The body of the method follows all normal PS-Algol rules for a procedure, enhanced by the OOPS-Algol additions.

It is not possible to define a variable local to the exemplar body without including it in the *stateIs* part of the definition. If this was allowed unwanted interaction between clones of an exemplar would occur as we only keep one copy of the closure for each exemplar.

5.4 **Creating an Instance of a Class**

Having defined the class and the exemplar of the stack we can now use it. We create the instance of a class by cloning one of the exemplars for that class. For example:

```
let aNewStack := [aStack clone]
```

It is worth mentioning that the above code fragment can be used in any OOPS-Algol program compiled after the class and exemplars have been defined. This is because the OOPS-Algol system stores all of the classes that have been defined, and all exemplars defined for each class in the persistent store. This is not possible in C++ or Objective-C without including numerous header files. OOPS-Algol ensures that names are kept unique to prevent confusion and can tell from the *aStack clone* expression that the type of *aNewStack* will be Stack.

5.5 **The Stack Revisited**

We now finish off our stack by extending it to include the AnnotatedStack used in Chapter 3.

```
Class AnnotatedStack superClassIs Stack
{
     "Implements a Stack that allows notes to be attached to
      its elements"

     messagesFor "alteration"

         attachNote: newNote typeIs proc(string)
                 "Attachs the string newNote to the top item of
                  the stack"

     messagesFor "accessing"
         getNote typeIs proc(->string)
                 "Gets the note (if any) for the top item of the
                  stack"

} !AnnotatedStack
```

As with the other programming languages we only define

the new messages, together with the class's superclass.

Here is the exemplar:

```
Exemplar anAnnotatedStack forClass AnnotatedStack
         superExemplarIs aStack
{
    stateIs {
        let notes := vector 1 :: 10 of ""
    }

    method attachNote: newNote typeIs proc(string)
    {
        notes(tos-1) := newNote  ! Note that we use aStack's
                                 ! state variable tos
    }

    method getNote typeIs proc(->string)
    {
        notes(tos-1)
    }
}
```

As can be seen this is very similar to all of the other languages we have examined. The difference is that we have separated the class definition from the implementation explicitly.

6. OOPS-Algol's Type System.

It is useful to present our view of the purpose of a type so that the motivations for the decisions we made concerning OOPS-Algol's type system become clear. A type can be viewed as serving two purposes:

1. Defining valid forms of interaction with an instance of a type.

2. Protecting the underlying representation of an instance of a type.

A class in OOPS-Algol defines the valid forms of interaction with an instance by defining the messages that can be sent to that instance. In OOPS-Algol a class contains no information about how an instance should be implemented. As OOPS-Algol's class definition meets the purposes we identified above, we can treat an object's class as its type. We use the terms type and class interchangeably.

Our definition of subtyping in OOPS-Algol is the simple one presented in Chapter 4: Class B can be a subtype of class A if class B can be used in place of class

A. To put this in another way, class B should be able to respond to all of the messages to which class A can respond.

6.1 Limitations On Subtyping in OOPS-Algol

We have the following limitations on subtyping in OOPS-Algol:

1. No automatic subtype determination.

2. Subtypes are restricted to objects.

3. Message selectors must have uniquely determinable types

4. Subclasses can not exclude message definitions.

We will examine these separately in the following sections.

6.1.1 **Subtype Determination**

[Cardelli 84] describes a type system where the subtype relationships can be determined without the programmer explicitly declaring the supertype of a new type. The semantics he describes determine type equivalence by using the names and types of fields in a record. If a type B has at least the field names of another type A, and the types of the fields match (or are subtypes of the field types in type A) then type B can be considered a subtype of A. This can be easily translated into object-oriented programming terms by equating the fields in a record with message selectors, and the types of the fields with the types of the message selectors.

OOPS-Algol forces the programmer to name the superclass of a new class for the following reasons.

(i) Naming the superclass in the subclass is a useful form of documentation. It allows the compiler to mark the new subclass as being dependent on the superclass, thus allowing the user to be warned if a change to a superclass will invalidate that superclass's subtypes.

(ii) Explicit superclass naming helps other programmers

to understand the system.

(iii) Implicit subtyping can cause a class to be erroneously regarded as a subclass of another. We present an example to demonstrate this.

Suppose we have a system where we define a class, *Runner*, which responds to messages to set a runner's name, age, and speed. We could also have a class *Vehicle* which also has a name, age, and speed. With the semantics presented in [Cardelli 84] these two conceptually different entities are treated as being equivalent because they respond to the same messages. Given this we could define a new class, *Car*, which adds information about its engine. The car would now be a valid Runner because it responds to all of the messages of a Runner, with additional messages to handle engine information.

As with all such examples this is perhaps a little contrived, but it is not difficult to imagine a big system where this could become a real problem. This is the main reason why we force explicit superclass naming.

6.1.2 **Subtype Restrictions**

Subtyping does not work for the standard PS-Algol types (*int*, *real* et cetera). It only operates on objects. This is because we are not redefining PS-Algol, we are adding to it. If we wanted a full subtyping system we would have to design a new language, which is beyond the scope of the research leading to this thesis.

An advantage of a hybrid language is that we can escape to the underlying language, allowing us to perform time critical operations efficiently. By using an already existing language to build upon we do not force people to learn YAPL (Yet Another Programming Language), they merely need to build on their already existing knowledge of PS-Algol. We do however lose such useful features as making Integer a subtype of Real.

6.1.3 **Message Selector Types**

As PS-Algol determines the type of a variable by looking at the right hand side of the initialising expression, we have to be able to determine an expression's type at compile time. It is sometimes impossible to determine the class of an object at compile time so we have to be able to determine the return type of a message expression from the selector alone.

This forces the following restrictions on the type of a selector:

1. Where a PS-Algol primitive type is used it must match exactly with all other definitions for the message selector being defined.

2. Where an OOPS-Algol Class is being used, the message selector must follow these rules:

    1. The class of each argument must be the same class, or a superclass of, the superclass's corresponding argument for the same selector.

    2. The class of the return value must be the same, or a subclass of, the superclass's message's return value.

Two points need further explanation here.

1. The OOPS-Algol system remembers details about every class that has been defined since the system was first used. This means that the compiler has information about every possible message selector,

and the types required for a message using that selector. When we say 'with all other definitions for the message selector' we mean all selectors that have ever been defined, not just the selectors that are superclasses of the new class.

2. The class of the subclass's message arguments are not allowed to be a subclass of the superclass's arguments. These rules also apply to procedure arguments of messages with respect to arguments and return values. This is so that the following situation can not occur:

```
Class Number superClassIs Object
{
    " Number Class"
    messagesFor "adding"
       add: addend typeIs proc(Number->Number) "Adds things"
}

Class Integer superClassIs Number
{
    "Integer Class"
    messagesFor "adding"
       add: addend typeIs proc(Integer->Integer)"Adds things too
}

Class Real superClassIs Number
{
   ... defined as above except for reals
}
```

The definition for Integer is invalid because the Number class specifies that a Number should be able respond to the message *add:* with a Number as an argument. This means that we should be able to send an instance of Integer an *add:* message with a Real argument because a Real is a Number. This is not possible as we have defined the type of the argument for *add:* in the Integer class as being an Integer (or a subclass thereof). As Real is not a subclass of Integer, sending this message will fail.

We could eliminate this problem in many cases if we had included the idea of a *coercer* function to convert between types as in [Bruce & Wegner 86]. This changes the meaning of subtypes so that instead of saying 'can be used instead of', we say 'can be coerced into the appropriate type'. This is much more powerful than our simple technique but would require a correspondingly more complex, and slower, system. We decided this was not justified because the main use of this capability is for number representation. PS-Algol already has efficient methods for handling numbers so this capability is not necessary to make OOPS-Algol useful.

To attempt to clarify some of these rules we will present a few arbitrary examples to show how the subtyping works. We will skip some syntactic details required by OOPS-Algol for brevity.

```
Class One superClassIs Object
{
     sel1: anotherOne typeIs proc(One->One)
}

Class Two superClassIs One
{
    sel1: anotherOne  typeIs proc(One->Two)
}
```

The above example is valid because Two's *sel1*: message takes the same argument as One's, and returns a subtype of *One*. Lets add another class:

```
Class Three superClassIs Two
{
    sel1: yetAnotherOne typeIs proc(One->One)
}
```

This is not valid because the return value will be a *One* which is not a subclass of *Two*. Note that the place holder for the argument, in this case *yetAnotherOne*, does not have to be the same as the superclass's place holder.

Now we will look at some examples that have procedure arguments.

```
Class OneP superClassIs Object
{
    sel: aProc typeIs proc(proc(One->Object))
}
```

This definition means that *aProc* is a procedure which takes an object of type *One* and returns an *Object*. Note that the message has no return value. Now we define a subclass of this:

```
Class TwoP superClassIs Object
{
    sel: anIntProc typeIs proc(int->int)
}
```

This is invalid because even though *TwoP* is a subclass of *Object*, not *OneP*, the message selector is the same, so we have to follow the rules we outlined above. They are violated because the *anIntProc* uses PS-Algol base types, and they do not match the types defined for *aProc* in the definition for *OneP*. It would be valid if we did this:

```
Class TwoP superClassIs Object
{
    sel: aProc typeIs proc(proc(Object->TwoP))
}
```

This is valid because *Object* is a supertype of *One* as defined in *OneP*'s message, and *TwoP* is a subtype of *Object* defined in *OneP*'s message.

6.1.4 **Message Definitions**

It is not possible for a subclass to undefine messages valid for its superclass, so we have eliminated the problem of a subclass being unable to respond to a message defined for its superclass.

6.2 **An Example Class Hierarchy**

We will present an example to illustrate the above ideas. The classes we will use will provide a small subset of the Smalltalk's *Collection* subclasses. We will be presenting only a few of the methods that are provided in Smalltalk for clarity reasons.

```
Class Collection superClassIs Object
{
    "Common methods for all collections"

    messagesFor "adding"

        add: newObject  typeIs proc(Object)
                  "Add newObject to the collection"

    messagesFor "removing"

        remove: oldObject ifAbsent: exceptionProc
                typeIs proc(Object,proc())
                      "Remove oldObject from the collection, if
                       it does not exist run exceptionProc"

    messagesFor "testing"

        includes: anObject typeIs proc(Object->bool)
                       "Return true if anObject exists in the
                        collection"

        isEmpty typeIs proc(->bool)
                       "Return true if the collection is empty"

        occurrencesOf: anObject typeIs proc(Object->int)
                        "Return the number of occurrences of
                         anObject in the collection"
}

Class Bag superClassIs Collection
{
    "This is a collection class that allows duplicate
     occurrences"

    messagesFor "adding"
        add: newObject withOccurences: nmr
                       typeIs proc(Object,int)
                       "Add nmr newObjects to this collection"
}
```

```
Class Set superClassIs Collection
{
   "This is a collection that does not allow duplicates"
}
```

The above example shows how we define a class, in this case *Collection*, to contain the most general operations for its subclasses. Notice that when we define the type of an object that can be added into a collection we give it the most general type, *Object*. This means that we can have any mixture of object types in a collection as every class is a subclass of *Object*. However, as previously mentioned, we can not add a PS-Algol base type to a collection, as the base types are not objects.

We then define the classes that can respond to *Collection*'s messages, as well as their own. The first one, *Bag*, allows duplicate entries to be made. To make this easier to use we have defined a message *add:withOccurences:* which allows us to add many copies of an object.

The definition of *Set* is interesting because it shows that it is possible to define a subclass that has no messages other than those provided in the superclass. If we were going to define a Collection that did not

allow duplicates we could make the exemplar a member of the Collection class. However it is much clearer to create a new class that states that it will not allow duplicates. This removes some of the temptation to dig into the implementation details of class exemplars.

The class *Collection* could be considered an abstract class as it is unlikely that an exemplar would be defined that names Collection as its class.

7. The Exemplar Hierarchy in OOPS-Algol.

In the previous chapter we saw how the behaviour of members of a class is defined in OOPS-Algol. In this chapter we show how the behaviour for a class is implemented in OOPS-Algol.

OOPS-Algol allows more than one implementation to be specified for a class, in the same way as the scheme described in [LaLonde, Thomas et al 86]. We associate one or more exemplars with a class and allow the exemplars to inherit methods and instance variables from any other single exemplar.

OOPS-Algol allows the exemplar hierarchy to be independent from the class hierarchy. This means that an exemplar does not need to inherit from another exemplar which belongs to the superclass of the class being implemented by the new exemplar.

When an exemplar is defined it can only respond to those messages that are defined for its class. As an exemplar can inherit from any other exemplar, it is possible that the exemplar being inherited has methods defined for messages that are not permissible for an

instance of the new exemplar's class. The compiler handles this by discarding message implementations (with a warning to the user) that are not defined for an instance of that type. In this way we can guarantee that an instance of class will only respond to messages for which we have all the type information.

7.1 The AnnotatedList Revisited

We are now ready to return to the problem posed in Section 2.4. We defined a List class, and decided, for efficiency reasons, to use a different implementation for empty and non-empty lists. We then created an AnnotatedList and tried to produce a suitable class hierarchy to match the problem. Our attempts failed because we were trying to match the conceptual hierarchy to the implementation hierarchy. We now look at how we would implement this in OOPS-Algol.

7.1.1 The List Class Definition

The first thing we do is to define the class definitions.

```
Class List superClassIs Object
{

    "A List Object"

    messagesFor "Miscellaneous!"

    add: newObject typeIs proc(Object->List)

         "Adds newObject to the list, it returns the list
          object which may be new"

    forEach: procToUse typeIs proc(proc(Object->bool)->int)
          "Runs procToUse on each object in the list, passing
           the object as an argument, until procToUse returns
           false.
           A count of the objects processed is returned"
} !  List
```

The above definition is just enough to give the flavour of our List object. We define the *add:* message to return a *List*. This is to allow us to return a non-empty list from an empty list if we should try to add something to an empty list.

7.1.2 **The List Implementation**

Now we define two exemplars to implement this. One exemplar is intended as an empty list only, and the other is used for a list with data.

```
Exemplar nonMTList forClass List superExemplarIs anObject
{

    stateIs {
        ! Define a structure to link list elements together
        structure listItem(pntr data,next);
        ! Create the head and tails of the list
        let listTail = listItem(nil,nil)
        let listHead = listItem(nil,listTail)
    }

    method add: newObject typeIs proc(Object->List)
    {
        ! Create a structure for the new item
        let newItem = listItem(newObject,nil)

        ! Link the new item onto the start of the list
        newItem(next) := listHead(next)
        listHead(next) := newItem
        self    ! Return this object
    }
    method forEach: procToRun typeIs
                                proc(proc(Object->bool)->int)
    {
        let thisItem := listHead(next)
        let count := 0
        while thisItem ~= listTail do {
            count := count + 1
            if procToRun(thisItem(data)) then
                thisItem := thisItem(next)
            else
                thisItem := listTail ! Force an exit
        }
        count  ! Return count
    } !forEach:
}
```

The above implementation represents a list by linking instances of the *listItem* structure together. We use a fixed head and tail to remove special case insertion and deletion. Notice that we do not use any objects in our implementation of a list. To improve the efficiency

of our implementation we use native PS-Algol constructs to build a list, not other objects.

Now we have the non-empty list exemplar, we can create the empty list exemplar.

```
Exemplar MTList forClass List superExemplarIs Object
{
    stateIs {}     ! Empty state.

    method add: newObject typeIs proc(Object->List)
    {
        ! Here we return a new list that is non-empty
        let newList := [nonMTList clone]
        newList := [newList add: newObject]
        newList
    }

    method forEach: procToRun typeIs
                                proc(proc(Object->bool)->int)
    {
        ! We do not need to run anything as the list is empty,
        ! so we just return a count of zero
        0
    }
}
```

The above example has demonstrated how two exemplars can exist for the same class. When we want to add something to an empty list we create a new *nonMTList*. We do this by sending the *clone* message to the *nonMTList* exemplar. This exemplar does not have to be visible in the scope of the method being defined. OOPS-Algol remembers all previously defined exemplars and returns a copy of the appropriate exemplar when the *clone* message

is sent. This is convenient, but does force the implementor to choose unique names.

Our implementation of *forEach:* is very simple for the *MTList* as we know the list must be empty.

7.1.3 The AnnotatedList Class Definition

Now we can define our AnnotatedList class, and the appropriate exemplars for that.

```
Class AnnotatedList superClassIs List
{
    "This is the same as List except we can add some notes"
    messagesFor "Miscellaneous"
    add: newObject note: aNote typeIs proc(Object,string->List)
      "Add newObject to the list, with the note aNote. It
       will return a possibly new list"
    forEachNote: procToRun typeIs
                          proc(proc(Object,string->bool)->int)
       "Run procToRun on each member of the list. The object
        will be passed as the first argument, with the note
        as the second argument.

        procToRun should return false when it is finished.
        The message will return the number of items processed"
}
```

In our new class we have added two new messages, *add:note:* and *forEachNote:*. We still have the original

messages, *add:* and *forEach:* which remain unchanged, allowing us to treat an *AnnotatedList* as a *List*. Here are the exemplar definitions for the *AnnotatedList*:

```
Exemplar nonMTAnnotatedList forClass AnnotatedList
                            superExemplarIs Object
{
    stateIs {

        ! Define a structure to shove in the list
        structure listItem(pntr data,next; string note);

        ! Create the head and tail of the list
        let listTail = listItem(nil,nil,"")
        let listHead = listItem(nil,listTail,"")
    }

    method add: newObject typeIs proc(Object->List)
    {
        [self add: newObject note: "" ]
    }

    method add: newObject note: aNote typeIs
                            proc(Object,string->List)
    {

        ! Create a structure for the new item
        let newItem = listItem(newObject,nil,aNote)

        ! Link the new item onto the start of the list
        newItem(next) := listHead(next)
        listHead(next) := newItem
        self    ! Return this object
    }
```

```
    method forEach: procToRun typeIs
                                proc(proc(Object->bool)->int)
    {

        !We have to reimplement this as we use a new structure

        let newProc = proc(Object theObject;
                           string aString->bool)
        {
            procToRun(theObject)
        }

        [self forEachNote: newProc ]

    }

    method forEachNote: procToRun typeIs
                        proc(proc(Object,string->bool)->int)
    {

        let thisItem := listHead(next)
        let count := 0
        while thisItem ~= listTail do {

            count := count + 1

            if procToRun(thisItem(data),thisItem(note)) then
                thisItem := thisItem(next)
            else
                thisItem := listTail ! Force an exit
        }
        count  ! Return count
    }
}
```

There are a number of points worth mentioning about the above implementation. Firstly, we have made *Object* the super exemplar. This was done because we can not reuse any code from *nonMTList* as we are using a different structure as our list item. PS-Algol would raise a runtime error if we tried to access a field from this structure with the code that was used in *nonMTList*

because the structure definition is different.

To increase the amount of code sharing we reimplemented *add:* and *forEach:* by sending *self* a message with the appropriate extra arguments set. For *add:* we just set the note to the empty string. In the implementation of *forEach:* we wrapped up *procToRun* in another procedure which took the arguments required for *forEachNote:* and sent the *forEachNote:* message to *self.*

7.1.4 The AnnotatedList Exemplar

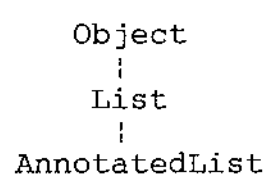
```
Exemplar MTAnnotatedList forClass AnnotatedList
                         superExemplarIs MTList
{

    stateIs {}     ! Empty state.

    method add: newObject typeIs proc(Object->List)
    {
        [self add: newObject note: ""]
    }

    method add: newObject note: aNote typeIs
                                proc(Object,string->List)
    {
        let newList := [nonMTAnnotatedList clone]

        newList := [newList add: newObject note: aNote]
        newList
    }
```

```
    method forEachNote: procToRun typeIs
                        proc(proc(Object,string->bool)->int)
    {
        0 ! Just return the count
    }
}
```

In our above implementation we do not need to reimplement the *forEach:* method from the superexemplar as the *MTList* returns predetermined answers, with no structure references. We do need to reimplement the *add:* message to ensure that the correct exemplar gets returned.

7.1.5 **The New AnnotatedList Hierarchies**

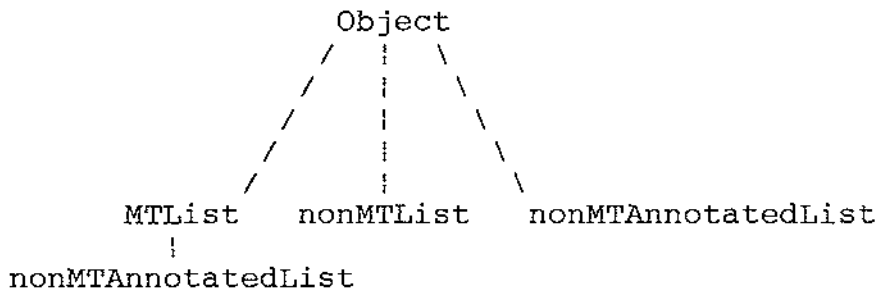
Now that we have defined the classes and the exemplars we present diagrams to illustrate the hierarchies in order to provide a comparison with the alternatives in Chapter 2. Here is the class hierarchy:



This is much more acceptable than the original alternatives from Chapter 2. There are no implementation details to hide the simplicity of the relationships

between the classes.

Now we draw the exemplar hierarchy.



The diagram shows how the complexity has been shifted to the implementation hierarchy. This is acceptable because our implementation design was based on performance considerations, which should only really impact the implementation details of a system, not the conceptual design.

7.2 **Summary**

This example demonstrates how we can hide the complexity of the implementation hierarchy from a user. The user of a class only needs to know the relationship between classes, and the names of the exemplars used to create instances of that class. If we had not separated our two hierarchies in OOPS-Algol the user would have

been burdened with an unnecessarily complex class hierarchy, as in the ones presented in Section 2.4.

OOPS-Algol allows the implementor of a class to make arbitrary decisions about the exemplar hierarchy based solely on implementation considerations. The implementor does not need to consider the impact on the conceptual structure of the classes. We consider this to be an important advantage of OOPS-Algol.

8. Conclusion and Future directions.

8.1 The Work Completed

OOPS-Algol provides the support necessary to program in an object-oriented fashion. We can share code easily in OOPS-Algol which is difficult to do in PS-Algol.

Our type system allows us to separate the logical design of a system from the physical design in a simple and versatile manner. We do not have to compromise the conceptual design of a system in OOPS-Algol in order to maximise the code sharing between classes.

OOPS-Algol's syntactic structure separates the implementation of a class from its description. This property is particularly useful when windowing editors or development environments are used, as we can open a window with only the behaviour of the class visible.

We have removed some of the administrative problems present with other languages that have had objects retrofitted. OOPS-Algol remembers all of the classes and exemplars it compiles. This obviates the need for the

many *include* files required for such languages as Objective-C or C++ when separate compilation is used.

We have retained the flavour of PS-Algol by clearly delimiting our new constructs, and by using as much of PS-Algol's existing syntax as possible. We have the use of the persistent store in which to keep our objects. This gives us a considerable advantage over C++ which has no concept of persistence at all (except that provided by traditional, file based methods), and we do not have to resort to Objective-C's *passivation* mechanism.

All of these factors combine to make OOPS-Algol a major improvement over PS-Algol when object-oriented programming is performed.

8.2 Further Work

Possible future work falls into three broad categories.

1. Performance Improvement.

2. Language Improvements.

3. Support Tools.

We detail these below.

8.2.1 **Performance Improvement**

Most object based systems today cache the results of previous method searches. This gives a performance improvement by preventing the laborious inheritance hierarchy traversal that is necessary when trying to locate a method. The performance of OOPS-Algol could be significantly improved by using this technique.

To give the maximum performance we would need to extend the PS-Algol abstract machine to be able to handle message switching. This would require changing the PS-Algol interpreter and compiler to support this. We would remove the OOPS-Algol preprocessor and replace it with a compiler that produced PS-Algol abstract machine code which would reduce the development times.

8.2.2 **OOPS-Algol language changes**

Multiple inheritance could be added to both the exemplar and class hierarchies. This would not be difficult for the class hierarchy, but due to the type constraints imposed by PS-Algol this would be difficult for the exemplar hierarchy. However, although multiple inheritance is frequently written about, its usefulness seems to be restricted, and as such may not actually be necessary.

An interesting idea was presented in [Kristensen, Madsen et al 87] where *parts* of a method could be inherited, not just a while method. This would be an interesting avenue of exploration as it seems that we spend a lot of time writing similar pieces of code, and this would be a nice way to exploit the commonality between them.

It would be possible to improve our subtyping mechanism by adding coercer functions as described in [Bruce & Wegner 86]. These functions would allow the implementor to define conversions between types so that subtyping could be extended considerably.

In Chapter 7 we showed how OOPS-Algol can have more that one exemplar for a class, and how we select the appropriate exemplar depending on the situation. An interesting avenue of research would be to determine how the appropriate exemplar might be selected automatically.

Currently we can only implement objects with an exemplar that must be associated with a particular class. It would be interesting to extend OOPS-Algol so that we could automatically delegate the responsibility for certain messages from any object, to any other object. As an example, consider a point that does not keep its own *y* coordinate, but relies on some other object to keep that information. Our point would follow the other point around in the y direction whilst retaining independent movement in the x direction. This sort of technique has particular applicability when windowing systems are implemented.

8.2.3 **Support Tools**

It would be desirable to have a number of support tools implemented to make OOPS-Algol more usable.

Unfortunately PS-Algol has no debugger. This makes error tracing difficult, forcing the programmer to insert debugging code whenever errors occur. The implementation of a debugger would make OOPS-Algol a more productive environment. This would require the implementation of a new, interactive interpreter, together with an extension to the PS-Algol abstract machine to include information about line numbers and file names so that source level debugging could be performed.

There is no interactive browser for OOPS-Algol. We have provided non-interactive reports to detail the available classes and their exemplars. Given a browser it would be useful to keep the source code of the exemplars under the control of OOPS-Algol so that the user could peruse the source whilst browsing. Currently we use the standard Unix tools (SCCS) to control versions of exemplars.

OOPS-Algol also needs a class library similar to that of Smalltalk. This would be a significant undertaking because PS-Algol has no built in window support (although it does have built in graphics primitives). The building of a window system in PS-Algol was commenced, but it was the laboriousness of that process

that motivated the creation of OOPS-Algol.

Implementing the full class hierarchy would also be interesting as it would provide a good testing ground for the separate hierarchies we use in OOPS-Algol.

Appendix 1 - OOPS-Algol Syntax

This appendix defines the syntax of OOPS-Algol. We have used the same two level grammar used in [PS-Algol 84], extended for OOPS-Algol. We have indicated the changes to the PS-Algol grammar in italics. The meta rule changes are limited to the productions for *TYPE-expression* and *declaration*. We have added a new type *object* to the hyper rule for NONVOID which we use to represent an object which is a member of any class. We do not express the subtyping rules in the grammar, but assume that when type matching is performed it follows rules presented in Chapter 6 of this dissertation. We have extended the PARAM hyper rule to include *class* and *exemplar* to allow for the specification of the class and exemplar names. The additional rules necessary to support these changes are placed at the end of the modified syntax.

We briefly describe the meta-language used. The description borrows heavily from that used in [PS-Algol 84]. We assume the reader is familiar with BNF syntactic specification and only explain the extensions to this.

The braces '{' and '}' are used in pairs to enclose anything that is optional. If the syntactic object can appear zero or many times the braces are followed by a '*'. The square brackets '[' and ']' are also used in pairs to denote an object that must occur once. When used with a '*' we have one or many times repetition.

An important extension is to allow the specification of type for a production. This is done by separating the type name from the syntactic category with a '-'. Consider:

<pixel-literal> ::= **on**|**off**

<int-literal> ::= [<digit>]*

This example indicates that the terminal symbols, **on** and **off** are a literals. The type of these literals is *pixel*. An integer literal is specified to consist of one or more digits.

By using the hyper rules we have a means of matching the types of productions. For example:

<NONVOID-assignment>::= <NONVOID-assign>:=<NONVOID-clause>

This example shows how we can match types with our meta language. The name *NONVOID* refers to a hyper rule which can represent any type in the language that is not void. Because every instance of NONVOID in the production given above must match, we are indicating that an assignment must involve a variable of the same type as the value of the clause being assigned to that variable.

The example also shows how the use of hyper rules shorten the grammar somewhat, as we do not need a production for every type in the language. NONVOID stands for all possible non-void types.

For a more complete explanation of the meta language used to describe the grammar see [PS-Algol 84].

The grammar for OOPS-Algol follows:

Hyper Rules

```
ARITH      ::= int¦real

COMPARABLE ::= ARITH¦string

SIMPLE     ::= COMPARABLE¦bool¦pixel

LITERAL    ::= SIMPLE¦pntr¦pic

IMAGE      ::= #pixel¦#cpixel

NONVOID    ::= LITERAL¦IMAGE¦*NONVOID¦TYPE.proc¦object

TYPE       ::= NONVOID¦void

PARAM      ::= NONVOID¦structure¦NONVOID.field¦class¦exemplar
```

Meta Rules

```
<void-program>       ::= <void-sequence>?

<TYPE-sequence>      ::= [<declaration>¦<void-clause>];
                                    <TYPE-sequence>¦
                         <TYPE-clause>

<void-sequence>      ::= <declaration>

<void-clause>        ::= if<bool-clause>do<void-clause>¦
                         repeat<void-clause>while<bool-clause>
                              {do<void-clause>}¦
                         while<bool-clause>do<void-clause>¦
                         for<int-identifier>=<int-clause>
                            to<int-clause>
                           {by<int-clause>}do<void-clause> ¦
                         <write> ¦
                         <NONVOID-assignment> ¦
                         <raster.clause> ¦
                         <void-expression>

<NONVOID-assignment>::= <NONVOID-assign>:=<NONVOID-clause>
```

```
<raster.clause>     ::= <raster.op><IMAGE-clause>onto
                                <#pixel-clause>

<raster.op>         ::= ror|rand|xor|copy|nand|nor|not|xnor

<TYPE-clause>       ::= if<bool-clause>then
                          <TYPE-clause>else<TYPE-clause>|
                        case<NONVOID-clause>of
                           [<NONVOID-clause>{,<NONVOID-clause>}
                                              :<TYPE-clause>;]*
                                default:<TYPE-clause>

<NONVOID-clause>    ::= <NONVOID-expression>

<write>             ::= write<write.list>|
                        out.byte<int-clause>,<int-clause>

<write.list>        ::= <SIMPLE-clause>{:<int-clause>}
                                      {,<write.list>}
```

```
<TYPE-expression>    ::= <TYPE-exp3> | <TYPE-mesg.expression>

<bool-expression>    ::= <bool-exp0>{or<bool-exp0>}*

<bool-exp0>          ::= <bool-exp1>{and<bool-exp1>}*

<bool-exp1>          ::= <COMPARABLE-exp3>[<lt>|<le>|<gt>|<ge>]
                                              <COMPARABLE-exp3>|
                         <NONVOID-exp2>{[=|<neq>]<NONVOID-exp2>}|
                         <pntr-exp3>[is|isnt]
                                    <structure-identifier>

<bool-exp2>          ::= {~}<bool-exp3>

<TYPE-exp2>          ::= <TYPE-exp3>

<TYPE-exp3>          ::= <TYPE-exp4>

<ARITH-exp3>         ::= <ARITH-exp4>[[+|-]<ARITH-exp4>]*

<TYPE-exp4>          ::= <TYPE-exp5>

<real-exp4>          ::= <real-exp5>[[<star>|/]<real-exp5>]*

<int-exp4>           ::= <int-exp5>[[<star>|div|rem]<int-exp5>]*

<pic-exp4>           ::= <pic-exp7>[[^|&]<pic-exp7>]*

<pixel-exp4>         ::= <pixel-exp7>&[<pixel-exp7>]*

<TYPE-exp5>          ::= <TYPE-exp6>

<ARITH-exp5>         ::= [+|-]<ARITH-exp6>

<TYPE-exp6>          ::= <TYPE-exp7>

<string-exp6>        ::= <string-exp7>[++<string-exp7>]*
```

```
<TYPE-exp7>          ::= <TYPE-name> ¦
                         <lcb><TYPE-sequence><rcb>¦
                         begin<TYPE-sequence>end

<NONVOID-exp7>       ::= (<NONVOID-clause>)

<LITERAL-exp7>       ::= <LITERAL-literal>

<pic-exp7>           ::= shift<pic-clause>
                                 by<real-clause>,<real-clause>¦
                         scale<pic-clause>
                                 by<real-clause>,<real-clause>¦
                         rotate<pic-clause>
                                 by<real-clause>¦
                         colour<pic-clause>in<string-clause>¦
                         text<pic-clause>
                                 from<real-clause>,<real-clause>
                                 to<real-clause>,<real-clause>¦
                         <lsb><real-clause>,<real-clause><rsb>

<string-exp7>        ::= <string-clause>
                           (<int-clause><bar><int-clause>)]*

<*NONVOID-exp7>      ::= vector<bounds>of<NONVOID-clause>¦
                         @<int-clause>of<NONVOID-type1>
                                <lsb><NONVOID-clause.list><rsb>

<bounds>             ::= <int-clause>::<int-clause>{,<bounds>}

<IMAGE-exp6>         ::= limit<IMAGE-clause>
                             {to<int-clause>by<int-clause>}
                             {at<int-clause>,<int-clause>}¦
                         <IMAGE-exp7>

<#pixel-exp7>        ::= image<int-clause>by<int-clause>
                             of<pixel-clause>

<pixel-exp7>         ::= <pixel-clause>[(<int-clause>)]*

<IMAGE-exp7>         ::= <IMAGE-clause>
                            [(<int-clause><bar><int-clause>)]*
```

```
<NONVOID-assign>     ::= <NONVOID-identifier>¦
                         <NONVOID-vec.exp>¦<NONVOID-struct.exp>

<NONVOID-vec.exp>    ::= <*NONVOID-expression>
                               [(<int-clause.list>)]*

<NONVOID-struct.exp>::= <pntr-clause>
                          [(<NONVOID.field-identifier.list>)]*

<NONVOID-clause.list>::=<NONVOID-clause>
                               {,<NONVOID-clause.list>}

<pntr-name>          ::= <pntr-structure.creation>

<NONVOID-name>       ::= <NONVOID-identifier> ¦
                         <NONVOID-vec.exp> ¦
                         <NONVOID-struct.exp>

<TYPE-name>          ::= <TYPE-proc.call> ¦<TYPE-standard.name>

<TYPE-proc.call>     ::= <TYPE.proc-clause>({<args.list>})

<args.list>          ::= [<NONVOID-clause>¦
                          <structure-identifier>]
                                    {,<args.list>}

<structure.creation>::= <structure-identifier>
                                    {(<NONVOID-clause.list>)}
```

```
<int-standard.name> ::= [lwb|upb](<*NONVOID-clause>)|
                        [readi|read.byte]()

<bool-standard.name>::= [eoi|readb]()

<string-standard.name>::=[read|peek|reads|read.name|
                          read.a.line]()

<real-standard.name>::= readr()

<void-standard.name>::= abort

<pixel-literal>     ::= on|off

<bool-literal>      ::= true|false

<pntr-literal>      ::= nil

<real-literal>      ::= <int-literal>.{<int-literal>}
                                  {e{+|-}<int-literal>}|
                        <int-literal>e{+|-}<int-literal>

<int-literal>       ::= [<digit>]*

<string-literal>    ::= "{<char>}*"
<digit>             ::= 0|1|2|3|4|5|6|7|8|9

<char>              ::= any ascii character

<PARAM-identifier>  ::= <letter>{<letter>|<digit>|.}*

<letter>            ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|
                        Q|R|S|T|U|V|W|X|Y|Z|
                        a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|
                        q|r|s|t|u|v|w|x|y|z

<declaration>       ::= <let.decl>|<structure.decl>|
                        <class.decl>|<exemplar.decl>

<let.decl>          ::= let<NONVOID-identifier>[=|:=]
                                        <NONVOID-clause>

<structure.decl>    ::= structure<structure-identifier>
                                    {(field.list)}

<field.list>        ::= <NONVOID-type1>
                        <NONVOID.field-identifier.list>
                                {;<field.list>}
```

```
<TYPE.proc-clause>   ::= proc[{<TYPE-type.spec>};<TYPE-clause>¦
                             {<TYPE-type.spec>};nullproc]

<void-type.spec>     ::= ({param.list>})

<NONVOID-type.spec>  ::= ({<param.list>}<arrow><NONVOID-type>)

<param.list>         ::= <param.spec>{;<param.list>}

<param.spec>         ::= <NONVOID-type1>
                             <NONVOID-identifier.list>¦
                         <structure.decl>¦
                         <TYPE-proc.type>
                             <TYPE.procedure-identifier.list>

<TYPE-proc.type>     ::= proc({<arg.type.list>}
                              {<arrow><NONVOID-type>})

<arg.type.list>      ::= [<NONVOID-type1>¦
                          <TYPE-proc.type>¦
                          <s.type>] {,<arg.type.list>}

<s.type>             ::= structure{(<NONVOID-type1>
                                    {,NONVOID-type1>}*)}

<PARAM-identifier.list>::=<PARAM-identifier>
                                 {,<PARAM-identifier>}*
```

```
<NONVOID-type1>     ::= {c}<NONVOID-type>
<int-type>          ::= int
<real-type>         ::= real
<bool-type>         ::= bool
<string-type>       ::= string
<pntr-type>         ::= pntr
<pixel-type>        ::= pixel
<#pixel-type>       ::= #pixel
<#cpixel-type>      ::= #cpixel
<pic-type>          ::= pic
<proc-type>         ::= <TYPE-proc.type>
<*NONVOID-type>     ::= <star><NONVOID-type1>
<arrow>             ::= ->
<lcb>               ::= {
<rcb>               ::= }
<lsb>               ::= [
<rsb>               ::= ]
<star>              ::= *
<lt>                ::= <
<gt>                ::= >
<le>                ::= <=
<ge>                ::= >=
<neq>               ::= ~=
```

All of the following productions have been added to support OOPS-Algol.

```
! A message expression that returns any type
<TYPE-mesg.expression> ::= <lsb><TYPE-mesg.expr><rsb>

<TYPE-mesg.expr>      ::= <receiver.expr><TYPE-selector.args>

<receiver.expr>       ::= <object-expression>

<object-mesg.expr>    ::= <exemplar-identifier> clone ¦
                          <receiver.expr><object-selector.args>

<TYPE-selector.args>  ::= <TYPE-unary.selector> ¦
                          <TYPE-keyword.selector>

<TYPE-unary.selector>::= <TYPE-identifier>

<TYPE-keyword.selector>::= {<TYPE-identifier>:
                                <NONVOID-expression>}+

<class.decl>          ::= Class <class-identifier>
                          superClassIs <class-identifier>
                          [<rcb><class.body><lcb>¦
                           begin<class.body>end]

<class.body>          ::= <class.desc>{<mesg.category.body>}*

<class.desc>          ::= string-literal

<mesg.category.body>  ::= messagesFor<mesg.category>
                            {<mesg.type.body>}+

<mesg.category>       ::= string-literal

<mesg.type.body>      ::= <mesg.type><mesg.desc>

<TYPE-mesg.type>      ::= [<TYPE-unary.selector>
                           ¦<TYPE-keyword.selector.decl>]
                           typeIs<TYPE-proc.type>

<TYPE-keyword.selector.decl> ::= {<TYPE-identifier>:
                                      <NONVOID-identifier>}+

<mesg.desc>           ::= string-literal

<exemplar.decl>       ::= Exemplar <exemplar-identifier>
                          forClass <class-indentifier>
```

```
                        superExemplarIs <exemplar-identifier>
                         [<lcb><exemplar.body><rcb>|
                          begin<exemplar.body>end]

<exemplar.body>      ::= stateIs[<lcb><state.body><rcb>|
                                 begin<state.body>end]
                        {method.definition}+

<state.body>         ::= {<let.decl>}*

<method.definition>  ::= <TYPE-mesg.type>
                              [<lcb><TYPE-method.body><rcb>|
                               begin<TYPE-method.body>end]

<TYPE-method.body>   ::= <TYPE-clause>
```

Appendix 2 - Object Representation in OOPS-Algol

This section provides a brief description of how we represent objects in OOPS-Algol, and how we have come to terms with PS-Algol's type restrictions to allow us to use inheritance.

The technique employed in OOPS-Algol to implement objects are very different from those used previously, as described in Appendix 4. In OOPS-Algol we store the instance variables in an instance's structure, and pass references to fields within this structure to the methods for an exemplar.

As a precursor to describing the structures used in OOPS-Algol we need to explain what an *id* is. Previous experimentation with PS-Algol had shown that string comparisons were slow. To overcome this in OOPS-Algol we convert all strings used for message selectors and instance variable names into unique integers, called *ids*. This conversion is done at compile time so that no overhead is present when a program is run. We use a table to map strings to ids and *vice versa*. This table is held in the persistent store so that the numbers remain unique across invocations of OOPS-Algol.

We now describe the most important structures used to represent instances and exemplars in OOPS-Algol.

The Instance Structure

An instance in OOPS-Algol is partly represented by this structure:

```
structure instance.struct(
     int  i.exemplarId;
     pntr i.superInstance;
     pntr i.IVS
     )
```

We prefix each field with *i* to ensure that the structure fields are unique. This approach is also used in the structures described later. The purpose of each field is as follows:

i.exemplarId This is the id of the exemplar for this instance. We use the id instead of a pointer to the exemplar because when we create a new exemplar, its address will be different. This means that the instance that referred to the old exemplar will be referring to an out of date exemplar (PS-Algol will not delete this exemplar in garbage collection as it is still

referenced). This would be repeating one of the problems with our original attempt, so we use the id as an index into a table of exemplars which is kept in the persistent store. When we update an exemplar, we simply replace the old one in the table.

i.superInstance This points to the superinstance for this instance. This is necessary because the superexemplar's methods must still reference a structure containing the instance variables with the same signature (that is, the same fields and types of fields) as when it was originally compiled. We pass the instance variable structure from this instance to any methods defined in the superexemplar.

i.IVS This is a pointer to a structure containing the instance variables for this structure. This contains pointers to structures containing the instance variables. This is a technique used to allow the use of *maiv*s which are described a little later.

The Exemplar Structure

This is the structure used to hold information for an exemplar. Instances of this structure are stored in the exemplar table for reference by other OOPS-Algol programs, and instances.

Here is the definition of that structure:

```
structure exemplar.struct(
                string e.exemplarName;
                int e.exemplarId;
                int e.superExemplarId;
                proc(int->pntr)e.grabEMProc;
                proc(int,pntr->pntr)e.grabIVProc;
                proc(->pntr)e.newProc
                )
```

We examine each field in more detail.

e.exemplarName This is the name of the exemplar. We keep this in case the id table should become corrupted and we need to rebuild it.

e.exemplarId This is the id of the exemplar. This used as the key in the exemplar table.

e.superExemplarId This refers to the superexemplar for this exemplar. It is used when searching for methods that are not implemented in this

exemplar.

e.grabEMProc This is a procedure that returns an instance of *exemplarMethod.struct* for the message with the selector id passed. We describe this structure later.

e.grabIVProc This is a procedure that accepts the instance variable structure (*i.IVS*) of an instance, the id of a particular variable, and returns a pointer to the location of the variable. This procedure is necessary because we have to bind the accessing of instance variables with the object implementation so that we do not have to keep the names of every field of every structure in the system unique.

e.newProc This is the procedure that returns a new instance of the class implemented by this exemplar.

The Exemplar Method Structure

This structure is used to hold the procedures required to be used when we execute the method for a message. This is complicated by the fact that we have to have one message procedure for every possible method. This means that we have to send the arguments for a message in a structure (so that we can use a pointer). It is worth looking at the declaration of the procedure used to send messages at this point.

```
let sendMsg = proc(int selectorId;! Message Selector Id
                   pntr clientI,  ! Original Instance
                        thisI,    ! This Instance
                        args;     ! Arguments to the method
                 string srcFile;  ! Name of source file
                    int srcLine;  ! Line in source file from
                                  ! whence this message is sent
                    ->pntr)       ! Value is returned in a
                                  ! structure.
```

As you can see, we have had to use pointers where ever values of different types might be returned. The value is actually held in a structure created by the OOPS-Algol preprocessor to hold a value of the appropriate type. The arguments *clientI* and *thisI* are used to resolve references to self when we are inheriting methods. *clientI* is the original instance receiving the message, and *thisI* is the current instance being looked

at. These are used to implement the semantics for reference to self discussed in the main body of this thesis.

Now we look at the *exemplarMethod.struct*

```
structure exemplarMethod.struct(
              proc(pntr->pntr)em.method;
              pntr em.maivsList;
              proc(pntr,int,pntr->pntr)em.setMaiv;
              proc(pntr,int->pntr)em.getArg
              )
```

em.method This is the procedure that implements the message. It accepts one argument, the *maiv* structure, which is an abbreviation for Method Arguments and Instance Variables. This is a structure created by the OOPS-Algol preprocessor to hold all the data necessary for the method to execute. It returns a pointer to a structure containing the value returned by the method.

em.maivsList This points to a list describing the fields used in the maiv structure. This is accessed at runtime by *sendMsg* to determine what values need to be injected into the maiv structure.

em.setMaiv This procedure takes the id of a field in the maiv and injects a pointer to the value into it for use by the method. If a null maiv was passed as an argument, it will create one, returning the address of this new maiv.

em.getArg This procedure extracts a member from the arguments structure passed to *sendMsg* so that it can be inserted into the maiv.

An outline of the message switch

To give some idea of how the system fits together, the following sequence of actions occurs when a message is sent.

1. The exemplar chain is followed until we find one with the required method implementation.

2. The maiv for that method is built by iterating over the maiv list, instantiating each field required by the method by either getting the appropriate instance variable, or the appropriate message argument.

3. The method is executed, returning the value returned (if any) to the caller.

The message logic is simple, but has been complicated because of the techniques we have used (in particular the maiv) to circumvent PS-Algol's structure type checking.

Summary

We have only lightly sketched the implementation of exemplars in OOPS-Algol. We have not detailed the implementation of type checking within the class hierarchies as it is straight forward, involving simple list comparison and storage of class descriptions.

Appendix 3 - The OOPS-Algol Environment

This appendix describes the environment under which OOPS-Algol runs, and how the user uses it. OOPS-Algol was built in a Unix environment on Sun 68020 workstations and an NCR Tower 32/600.

We use the persistent store as the OOPS-Algol compiler's repository of information about the classes and exemplars it has compiled. This has obviated the need for large numbers of include files that are used in Objective-C and C++ to transfer information between separate compiles in a system.

How to compile and run an OOPS-Algol program.

An OOPS-Algol program is compiled in the following manner:

```
oopsc aProgram.oop
```

Note that *oopsc* runs the C preprocessor on the program to be compiled so the user has the full functionality of the preprocessor available. This produces a standard PS-Algol executable (called *aProgram.out*) from the source file *aProgram.oop* which we then interpret using

the PS-Algol interpreter thus:

```
psr aProgram.out
```

Time limitations have prevented the implementation of an interactive browser for PS-Algol so we have implemented a utility to produce a listing of all of the defined classes in the system and their exemplars. This is run as follows:

```
oopsdump
```

which produces a listing on the standard output which might look like this:

```
Classes

Object
    "The Root Object"
    exemplars anObject
    messagesFor "inquiry"
        respondsTo: aMessage

Stack
    "Implements a simple stack containing any sort of object"
    superClassIs Object
    exemplars aStack mtStack


    messagesFor "queries"
                isMT typeIs proc(->bool) "True if MT"
    messagesFor "updating"
        push: anObject typeIs proc(Object) "push the object"
        pop            typeIs proc(->Object) "pop an object"
```

The dump names the class, a description of the class, the superclass for the class, and the exemplars defined for that class. It then lists out all the messages defined for that class.

Appendix 4 - Objects in PS-Algol

This appendix describes our initial attempt at implementing objects in PS-Algol.

The Technique

We originally modelled objects in PS-Algol by using a structure to contain the methods for a member of the class. We used the scope rules of PS-Algol to hide the instance variables, which where visible to the method procedures.

We did this by creating a procedure which returned a structure containing a field for each method for a class. Within this procedure the instance variables were defined, and any required initialisation was performed. Message passing was simulated by selecting the appropriate field from an instance's structure and executing the procedure referred to there.

We present a simple example to illustrate this. Suppose we want to create a point. First we define the messages to which a point would respond. Then we create a structure with the names of the methods as fields:

```
structure point.struct(proc(->int)get.x.pos;
                       proc(->int)get.y.pos;
                       proc(int)set.x.pos;
                       proc(int)set.y.pos
                      )
```

Having done this we define a procedure that will create a point by returning an instance of *point.struct*. Here is one possibility:

```
let make.point = proc(int x,y->pntr)
{
    ! Define a variable which represents self. This is now
    ! visible in any of the procedures within make.point

    let self := nil

    ! Here are the procedures which implement the messages.
    ! Notice that they can access x and y as they
    ! are within their scope.

    let get.x.pos.proc = proc(->int); x
    let get.y.pos.proc = proc(->int); y
    let set.x.pos.proc = proc(int newX); x := newX
    let set.y.pos.proc = proc(int newY); y := newY

    ! We create an instance by creating a new instance of
    ! point.struct, filling it with the procedures
    ! we have just defined, and assigning it to self.

    self := point.struct(get.x.pos.proc,
                         get.y.pos.proc,
                         set.x.pos.proc,
                         set.y.pos.proc)

    ! This returns the pointer to our new structure.
    self
}
```

Note that PS-Algol allows us to leave out the braces surrounding the statements making up a procedure when the procedure body consists of only one statement.

The most interesting thing about this technique is that we have used PS-Algol's first class procedures and name scoping rules to make the instance variables visible only to the procedures that should have access to them. This makes our objects totally safe from unwanted interference, intentional or unintentional.

This works because when we make a procedure a member of a structure, as in the *self* assignment above, we are also storing the closure of that procedure. In this case it is the *x*, *y* and *self* locations for this invocation of *make.point*. The next invocation will save new locations for *x*, *y* and *self*.

This is how we would create an instance of point, and how we would send a message to it:

```
! Create a new point with x=10 and y=5
let new.point = make.point(10,5)

! Change the y position
new.point(set.y.pos)(10)

! Get the y position and print it out
write "Y pos is ",new.point(get.y.pos)(),"'n"
```

In PS-Algol we access a field from a structure by naming the structure and following this with the field name in parentheses. As our structure members are procedures we execute the procedure by following the field reference with the parenthesis enclosed arguments for that method. In the above example our structure is called *new.point* and we accessed the *set.y.pos* and *get.y.pos* fields.

We stored our creation procedures (like *make.point* above) in the persistent store, which we then retrieved when necessary to create new instances.

Analysis

This technique was not suitable for the following reasons:

1. Unique names are required.

2. Instance variable inheritance was impossible.

3. Class changes did not propagate to already existing members of the changed class.

We discuss these in more detail in the following sections.

Unique Names

PS-Algol determines the type of an structure field reference by looking at the field name. This means that every field of every structure that is defined within the current scope must have a unique name, otherwise the correct field can not be identified, and the type of that field would be undeterminable. Other languages overcome this by specifying that a variable will only refer to a particular structure type. PS-Algol has no

such concept, and allows a variable of type **pntr** to refer to any structure.

This is a problem because it is not possible to define a message with the same selector (the field name in our technique) to instances of different classes. We had to name each message uniquely, by appending the class name to the end of the field name.

Instance Variable Inheritance

We could not inherit any instance variables as they were not visible outside the creation procedure. This precluded the possibility of using some of the methods from a super class. To model inheritance we have to make the new enhanced object contain an instance of the old object, and then route messages to the base object manually.

Class Change Propagation

If we change a method within our creation procedure this does not change any existing instances of that class. Even worse, when we add a message to a class, this change does not propagate to the already existing members of the class. This means that every PS-Algol program that creates new instances of the changed class has to be recompiled, otherwise the program would fail at runtime because the structure definitions are different.

Conclusion

This approach was employed in building the basis of a windowing system involving thirteen object classes consisting of approximately four thousand lines of code. It produced systems that were difficult to maintain, as every program using a changed class had to be recompiled. It is against this background that OOPS-Algol was developed, and it demonstrated that the technique was impractical. Any form of automatic inheritance was impossible, which negated one of the prime advantages of object-oriented programming - that of code reuse.

References

We use these abbreviations in the following references:

OOPSLA '86 Object-Oriented Programming Systems, Languages and Applications, September 29-October 2, 1986, Portland Oregon, SIGPLAN Notices, Vol. 21, No. 11, (November 1986).

OOPSLA '87 Object-Oriented Programming Systems, Languages and Applications, October 4-8, 1987, Orlando, Florida. SIGPLAN Notices, Vol 22, No. 12, (December 1987).

ECOOP '87 European Conference on Object-Oriented Programming Paris France, June 1987. Springer-Verlag. Lecture Notes in Computer Science 276.

PPR Persistent Programming Research Report. These can be obtained by writing to The Secretary, The Persistent Programming Research Group, Department of Computing science, University of Glasgow, Glasgow G12 8QQ.

[Atkinson, Bailey, et al 83] Atkinson, M.P, Bailey,P.J., Chisholm, K.J., Cockshott, P.W., Morrison, R.: An Approach to Persistent Programming, in *The Computer Journal, Vol 26, No 4.*, 1983

[Atkinson & Morrison 84] Atkinson, M.P. & Morrison R.: Procedures as persistent data objects, *PPR-9-84*, 1984.

[Blake & Cook 87] Blank, E. & Cook, S.: On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk. in *ECOOP '87.*

[Bobrow, Kahn et al 86] Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F.: CommonLoops Merging Lisp and Object-Oriented Programming in *OOPSLA '86.*

[Booch 86] Booch, G.: Object-oriented Development, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986.

[Borning 86] Borning, A. H.: Classes Versus Prototypes in Object-Oriented Languages, *ACM/IEEE Fall Joint Computer Conference*, November 1986.

[Bruce & Wegner 86] Bruce, K.B. & Wegner, P.: An Algebraic Model of Subtypes in Object-Oriented Languages in *SIGPLAN Notices* V21 October 1986, pp163-172.

[Budd 87] Budd, T.A.: *A Little Smalltalk.* Addison-

Wesley, 1987.

[Cardelli 84] Cardelli, L.: A semantics of multiple inheritance, in *Semantics of Data Types*. Springer-Verlag, Lecture Notes in Computer Science, Vol. 173, 1984, pp.51-67

[Cardelli & Wegner 85] Cardelli, L. & Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys*, Vol. 17, No. 4, December 1985 (actual publication August 1986).

[Cox 86] Cox, B.: *Object-oriented Programming, An Evolutionary Approach*. Addison Wesley, 1986.

[DeMarco 78] DeMarco, T.: *Structured Analysis and System Specification*, Prentice-Hall, 1978.

[Ducournau & Habib 87] Ducournau, R. & Habib, M.: On Some Algorithms for Multiple Inheritance in Object Oriented Programming, in *ECOOP '87*.

[Gane & Sarson 79] Gane, C., & Sarson, T.: *Structured Systems Analysis: tools and techniques*. Prentice-Hall, 1979.

[GNU 1988] GNU: Project GNU, Free Software Foundation, 675 Mass. Ave., Cambridge, MA 02139, USA.

[Goldberg & Robson 83] Goldberg, A. & Robson, D.: *Smalltalk-80: The language and its Implementation*. Addison-Wesley, 1983.

[Halbert & O'Brien 87] Halbert, D.C. & O'Brien, P.D.:

Using Types and Inheritance in Object-Oriented Languages, in *ECOOP '87*.

[Kristensen, Madsen et al 87] Kristensen, Bent Bruun, Madsen, Ole Lehrmann, Nygaard, Kristen: Classification of actions, or Inheritance also for methods, in *ECOOP '87*.

[LaLonde, Thomas et al 86] LaLonde, W. R., Thomas D. A., Pugh, J.R.: An Exemplar Based Smalltalk, in *OOPSLA '86*.

[Lieberman 86] Lieberman, H.: Using Prototypical Objects to Implement Shared Behaviour in Object Oriented Systems, in *OOPSLA '86*.

[Moon 86] Moon, D.A.: Object programming with FLAVORS, in *OOPSLA '86*.

[Morrison, Brown, et al 86] Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T & Dearle, A.: A persistent graphics facility for the ICL PERQ, *Software Practice and Experience*, Vol.14, NO.3, (1986)

[Morrison, Brown et al 87] Morrison,R., Brown, A., Connor, R., Dearle, A.: Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment. *PPR-32-87*.

[PS-Algol 84] PS-Algol 84: *Persistent programming research group PS-Algol reference manual*, 2nd ed.

References

University of Glasgow and University of St. Andrews, Rep. PPR-12, 1984.

[Ritchie & Thompson 74] Ritchie, D.M & Thompson, K.: The UNIX Time Sharing System, *Communications of the ACM*, pp365-375, 1974.

[Stein 87] Stein, L.A.: Delegation is Inheritance, in *OOPSLA '87*.

[Stroustrup 86] Stroustrup, Bjarne: An Overview of C++, in *SIGPLAN Notices*, Volume 21, October 1986.

[Stroustrup 87a] Stroustrup, Bjane: Multiple Inheritance for C++, in *Proceedings of the Spring '87 EUUG Conference*. Helsinki, May 1987.

[Stroustrup 87b] Stroustrup, Bjarne: What is Object-Oriented Programming?, in *ECOOP '87*.

[Ungar & Smith 87] Ungar, David & Smith, Randall B.: Self: The Power of Simplicity, in *OOPSLA '87*.

[Wegner 87] Wegner, P.: Dimensions of Object-Based Language Design, in *OOPSLA '87*.