This paper should be referenced as:

Atkinson, M.P. & Morrison, R. “Integrated Persistent Programming Systems”. In Proc. 19th International Conference on Systems Sciences, Hawaii (1986) pp 842-854.

Integrated Persistent Programming Systems

Malcolm P. Atkinson and Ronald Morrison

University of Glasgow, Glasgow, Scotland G12 8QQ. Tel. 0413398855

and

University of St Andrews, St Andrews, Scotland KY16 9SX. Tel. 033476161

Abstract

We contend that future programming languages should contain orthogonal persistence of data, interactive graphics support and a mechanism which permits the programmer to specify incremental binding. Persistence should permit data of any type or size to exist for arbitrary lengths of time. Our graphics proposals include bit map and vector models of pictures and control of colour maps. Name spaces provide the mechanism for binding control. It is the interplay and introduction of these language constructs that greatly extends the repertoire of programming activities supported by the programming language. A prototype language which includes these concepts has been implemented and used extensively.

1. Introduction

In this paper we set out to report our current experiments in language design and to outline our views on the necessary developments in programming languages and environments for the 1990's based on these experiments and surveys we have done[6,8]. Present dependence on a plethora of languages and mechanisms such as command languages, file systems, compilers and interpreters, linkage editors and binders, debuggers, DBMS sub-languages, graphics libraries, etc. increases the cost of understanding and maintaining software and training programmers, for even the simplest of programming activities. Axiomatic in our approach is that we are searching for a language or a family of languages that are capable of supporting **all** programming activities varying only in their universe of discourse defined by their respective type systems. With this one mechanism we envisage a saving throughout the life cycle of program development and use.

Inherent in our method of language design is that the language itself should be simple. Merely adding features to languages without integrating them into some overall design simply increases the complexity of the system and can often lead to it being beyond the intellectual capacity of the programmer and sometimes even the compiler writer. We subscribe to the view of van Wijngaarden[34] which is epitomised in the maxim

> "In order that a language be powerful and elegant it should not contain many concepts"

He argues that languages are too complex and that complexity is due, in part at least, to being too restrictive. Power through simplicity, simplicity through generality is the message. This forces the language designer to formulate the fundamental concepts behind the language and to generalise these wherever possible.

Our languages are designed using three principles due to Landin[15] and Strachey[27] and further developed by Tennent[29]. They are

1. The principle of correspondence: the use of names should be consistent within a language. In particular there should be a one to one correspondence between the method of introducing names in declarations and in parameter lists.

2. The principle of abstraction: all the major syntactic categories should have abstractions defined over them. For example, functions are abstractions over expressions.

3. The principle of data type completeness: all data types should be first class without arbitrary restriction on their use.

We have described our design philosophy fully elsewhere [20] and will not labour it here. However the essence is that there are no exceptions to these rules since having such exceptions makes the language more complex in terms of defining rules and usually less powerful since there are restrictions. Given this general framework in which to define our languages we have identified a number of requirements we feel are desirable for modern programming languages. We do not regard this list as exhaustive. Indeed we would expect that having achieved these requirements new ones will emerge in the pursuit for better programming systems. They are

a A mechanism for the uniform treatment of data independent of the length of time for which it exists. We call this orthogonal persistence.

b A type mechanism that meets all programmer requirements, is strict and employs a checking method that is eager. That is, it reports errors in the programming process as early as possible. Strong typing allows this to be at static analysis time with some concessions for I/O. However, if the type system is to encompass all constraints on the use of data, such as the integrity constraints in a database system, then the checking may have to be performed as late as transaction commit.

c The ability to define and use interfaces to system components built with technologies other than this language, both hardware and software.

d Support for system evolution. For very large scale, widely used or continuously used systems any alteration to part of the system should not require total rebuilding. We require a mechanism which will allow the programmer to control the size of the units of reconstruction.

e Immediate execution of programs with a ramp of useable concepts, so that simple things can be programmed without having to know all the features of the language. This is important in making the language accessible and we aspire to an autodidactic language processor that makes the training system part of the language and is responsible for correct and sensible use of the language.

These requirements lead us to search for constructs that handle the activities above which are not currently supported in programming languages. In this paper we will describe our results and some projections for the future on orthogonal persistence, uniform treatment of program and data, type systems, support for graphical data and large scale system construction. We have designed and implemented the language PS-algol[5] as a testbed for these experiments and are currently building its successor in a search for a sufficiently powerful tool to contain all programming activity in one language.

2. Orthogonal Persistence

The long term storage of data has been of concern to programming language designers for some time. Traditional programming languages provide facilities for the manipulation of data whose lifetime does not extend beyond the activation of the program. If data is required to survive a program activation then some file I/O or database management system interface is used. Two views of data evolve from this. Data can either be classed as short term data and would be manipulated by the programming language facilities or data would be long term data in which case it would be manipulated by the file system or the database management

system (DBMS). The mapping between the two types of data is usually done in part by the file system or the DBMS and in part by explicit user translation code which has to be written and included in each program.

That there are two different views of data has certain disadvantages. Firstly in any program there is usually a considerable amount of code, typically 30% of the total, concerned with transferring data to and from files or a DBMS. Much space and time is taken up by code to perform translations between the program's form of data and the form used for the long term storage medium. For example, we normally have to explicitly flatten and rebuild graphs or trees modelled in the programming language in order to write them out or read them back in from the file store. In such systems there are three models of data, one in the real world, one in the database system and one in the runtime system. This is unsatisfactory since the programmer or system designer has to visualise and maintain all of these mappings correctly. This leads to an intellectual overhead in using these mappings and preventing them from becoming mutually inconsistent.

The second major disadvantage is that the data type protection offered by the programming language on its data is often lost across the mappings. The structure that might have been exploited in a program to aid comprehension is neither apparent nor protected and is soon lost.

We seek to eliminate the differences between the DBMS and programming language models of data. This can be done by separating the issue of what data structures are best for a particular problem from the issue of identifying and managing the data. We call this second property of data persistence which we have defined to be the period of time for which the data exists and is useable[4]. A spectrum of persistence exists and is categorised by

1. transient results in expression evaluation.

2. local variables in procedure activations.

3. own variables, global variables and heap items whose extent is different from their scope.

4. data that exists between executions of a program.

5. data that exists between various versions of a program.

6. data that outlives the program.

The first three persistence categories are usually supported by programming languages and the second three categories by a DBMS, whereas filing systems are predominantly used for categories 4 and 5.

We regard the provision of persistence as orthogonal to all other properties of data. For this we define the Principle of Persistence Independence as

> The persistence of a data object is independent of how the program manipulates that data object and conversely a fragment of program is expressed independently of the persistence of data it manipulates.

That is, all code should be written so that it will work with the same interpretation independent of the persistence of the data on which it operates. This reduces the number of conceptual mappings from three to one - the one between the real world and the program model and thus simplifies the whole system.

We also identify an extension to the Principle of Data Type Completeness to handle persistence. It is

In line with the principle of data type completeness all data objects should be allowed the full range of persistence.

We note that such orthogonal persistence is rare even in languages that support database features. In Pascal[31] only files of some types persist, in Pascal/R[25] only relations of a few scalar types persist, Taxis[22] provides no short term data and Adaplex[26] uses entities/entity classes as persistent objects but they are not uniformly available as transient objects.

All large systems need persistence since they all need to store their code and data. Computing systems that involve humans must support long term activity. Since the attention span of the human is short in computing terms then the storage of the present state of an attempted solution to a problem is essential. To achieve this in an integrated manner the system should provide persistence designed by the above principles.

We identify the following as outstanding problems for which solutions will need to be found in order to accommodated persistence correctly. They are

i) there should be mechanisms to permit and control shared and concurrent usage;

ii) it should be possible to encapsulate any sequence of operations on the store into a transaction, which behaves as a single operation;

iii) there should be privacy and access control mechanisms;

iv) there should be a mechanism for the long term control, manipulation and use of names.

As can be seen from the above we see the most pressing problem in the provision of a totally integrated persistent system as solving the complex relationship between concurrency, transactions and the long term use of names.

We have experimented will several versions of persistence[1,2,12]. Each one has a notion of transaction which allows a group of operations on data to appear atomic. The effects of the transaction are only visible to other transactions that start after the transaction has committed. Transactions should not be confused with mechanisms to make store reliable. They are a mechanism for viewing and controlling change which is a quite separate issue from reliability.

The concept of a transaction is clearly recursive with the fixed point being the machine's atomic store update. Nested transactions may be formed by grouping more than one store update into a single transaction. The semantics of these nested transactions are reasonably well understood[3] for a sequential process. There are however some outstanding difficulties. For example if a transaction is to leave the store unchanged when it aborts, how does it communicate an arbitrarily sophisticated reason for its failure to the outer transaction. There is also the question of whether the outermost transaction is a special case as in Argus or is just the lifetime of the system. At present we know of no clear semantics for nested transactions in a concurrent environment.

We are experimenting with models of concurrency in a persistent environment[36] but it is too early to present any results.

3. Uniform Treatment of Program and Data

We have demonstrated[7] that in a persistent environment, the provision of procedures as data objects allows us to implement abstract data types, modules, separate compilation, views and data protection with the one mechanism - that of the persistent procedure.

Furthermore we believe procedures to be essential in finding simple methods for system construction and version control. We call such procedures, first class procedures and by the principle of data type completeness they must also be higher order, that is, they are first class data objects.

Most programming languages provide facilities for abstractions over expressions and statements. Indeed these abstractions, functions and procedures, are often the only mechanisms for abstraction in a programming language. The power of the mechanism is derived from the fact that the user of the procedure does not require to know the details of how the procedure executes, only its effect. We use the word 'procedure' to represent both procedure and function when it is not necessary to differentiate between them.

The procedures of Algol 60[23] and Pascal can only be declared, passed as parameters or executed. However, as has been pointed out by Morris [19] and Zilles [33], to exploit the device to its full potential it is necessary to promote procedures to be full first class data objects. That is, procedures should be allowed the same civil rights as any other data object in the language such as being assignable, the result of expressions or other procedures, elements of structures or vectors etc. Lisp[17] was the first language with first class procedures and other languages include Iswim[15], Pal[14], Gedanken[24], Sasl[30], ML[18] and with some restrictions Euler[32] and Algol 68[35]. Of course the applicative programming technique revolves around the ability to have first class procedures in the language and central ideas such as partial application are not properly implementable otherwise.

The main advantage of having first class procedures in a persistent environment is that there is a simple and well understood mechanism for system construction - that of the procedure call. The safety of the system is provided by the type mechanism of the language which must guarantee type checking for data of all persistence. Hence, under orthogonal persistence, no other arrangement for storing procedures and programs need be made. This removes the need for separate provision of libraries and library maintenance systems. The loading and type checking of the persistence system removes the need for a linker, for separate loaders and for type checking system builders. Separate compilation of program parts is achieved by compiling and running programs which leave procedures in the persistent store. Subsequent programs identify and use these procedures just like any other data thus yielding a mechanism for incremental system construction.

The power and facilities of abstract data types and modules can be obtained from encapsulating data in a procedure that yields as its result a record containing other procedures to manipulate this data. The data may then only be accessed by the yielded procedures thus providing the protection features of abstract data types and modules. Separate compilation of these modules may be achieved by storing the procedures in the persistent store.

The provision of procedures and procedure activations that can exist in a database opens up a number of interesting avenues of exploitation, some of which we have introduced in the paper:

i) Software tools: managing versions, program assembly, program and data dictionaries, reporting on data and programs etc. can be written entirely within such languages. There are two advantages - the complexity of going outside the language or of building interpreted representations is avoided and the portability of implementations is improved.

ii) Views and complex mechanisms of authorisation can be coded within the language. New features are first needed to allow views to be bound to existing data if all the functions provided in databases are to be accommodated. View mechanisms are used in databases to perform two roles; to provide a stable and appropriate view to the programmer, and to implement protection and privacy controls. The first class functions, together with partial application perform both of these roles. Stability

means that the underlying data may be changed without impact upon programs it was not intended to alter apart from possible changes in performance. The person who changes the underlying data is usually responsible for redefining the mapping that provides the view except where the only available mappings are so simple that the new mapping may be inferred. If we interpose a set of functions, then redefinition of these functions will provide the required stability. Similarly, they can be defined so as to provide the appropriate view and the access controls. We have discussed this use of first class persistent functions elsewhere[7].

iii) Dynamic mechanisms to be invoked on database operations can be implemented. This would include those proposals such as triggers in System/R[10] and conditions in CODASYL DBTG[28] neither of which were implemented, we suspect because the proper binding of procedures within the database had not been developed at that time. This could lead to various forms of active and deductive databases.

iv) Integrated interactive environments for general or special programming may prove easier to construct and to evolve given such persistent procedures.

Since readily understood and easily implemented languages are needed as a foundation for software engineering, we argue that serious consideration should be given to languages which support procedures as data objects, which have an orthogonal provision of persistence and which are not overgrown with numerous other concepts.

4. Type Systems for Persistent Data

Ideally we would like a simple set of types, and a type algebra, so that by a succession of operations of the algebra, and provision of parameters, we could define a data type equivalent to any data model or conceptual data model. Parameterisation of such types yields schemata, i.e. the types of databases in the model, and each database is then an instance of such a type. We call this the "type alchemist's dream" as we do not yet know how to do it. In consequence the language designer has to opt for a particular model. Contrary to our original beliefs, this has to contain types motivated by long term data, because it tends to become large, and hence bulk operations and describing its regularity become important.

Based on an extensive survey[8] of existing, research and proposed languages together with experience of PS-algol, we have come up with the following general advice with regard to programming environments and persistent data.

i) the data types should include a mechanism to capture user communication including graphical and audio facilities;

ii) the data types should be expressive enough to capture the structure and regularity that will be required of the data;

iii) the data types should include program, as procedures and abstract data types;

iv) the data types should include a general purpose indexing mechanism, both polymorphic and variadic;

v) the data types should include a bulk data type with operations on collections of data;

vi) the data types should include some form of inheritance to model specialisation;

vii) the types and bindings used should allow system evolution on an incremental and localised basis.

Our own preferences for a type system are defined elsewhere[9] but two aspects of our type system are presented in the next two sections.

5. Support for Graphical Data

The inclusion of graphics facilities in a programming language that supports an integrated persistent environment yields an ideal base for building large and complex graphics systems. Picture libraries of complete or partial pictures may be built in the persistent store in the same manner as any other collection of data or program libraries. Thus the users have a well defined and well understood mechanism in which to access and store pictures. In turn this provides a well structured method of building complex picture systems out of parts of pictures in the same manner that might be used when constructing complex programs out of procedures in a library.

Fourth generation seamless systems such as the software for the Apple Lisa with picture editors, menu systems and mixed picture, text and program documents may be built out of the integrated persistent environment. We expect to use the persistent store as an enabling technology for a graphics software tools exchange[37].

In order to make use of the persistent store facility in PS-algol pictures must be a proper data type in the language. Indeed PS-algol supports two separate types for graphics and a method of relating them. These two types correspond to the two main types of graphics devices available, i.e. calligraphic and raster. Line drawings are supported by the data type picture and bitmaps by the data type image.

As well as making the persistent store available, having the graphics facilities supported by data types has another main advantage. We can define infix operators on pictures and images and have syntactic support for the manipulation of these objects. This means that programs which use these objects will be considerably shorter, with all the attendant benefits, than programs constructed from subroutine libraries, for example.

This section describes the facilities available in PS-algol for graphics, both for line drawing and for bitmap operations, in the persistent store.

5.1. Pictures

The picture drawing facilities in PS-algol are a particular implementation of the Outline system[21] which allows line drawing in an effectively infinite two dimensional real space. Altering the relationship between different parts of a picture is performed by mathematical transformations which means pictures are usually constructed from a number of sub-pictures. In the PS-algol system picture description and picture drawing are separated. Picture description is supported by operators over pictures and picture drawing by mapping the picture to an image. In this manner pictures are described in a device independent manner.

In PS-algol the picture descriptions are represented by the data type picture. The simplest picture is a point. For example,

```
let point = [ 0.1,2.0 ]
```

represents the point with x-coordinate 0.1 and y-coordinate 2.0 in two-space. All the operations on pictures provided return a picture as their result, so arbitrarily complex pictures may be described and operated on.

Points in pictures are implicitly ordered. The binary operators on pictures operate between the last point of the first picture and the first point of the second picture. In the resulting picture the first point is the first point of the first picture and the last point is the last point of the second picture.

There are two binary operators on pictures, join (^) and combine (&). The effect of the join operator is to give a picture that is made up of its two operands with a line from the last point of the first operand to the first point of the second operand. Combine operates in a similar way without adding the joining line.

In addition to the binary operators pictures may also be transformed by shifting, rotating and scaling. For example:

shift p **by** x.shift, y.shift

will produce a new picture by adding x.shift to every x-coordinate and y.shift to every y-coordinate in the picture p.

rotate p **by** no.of.degrees

will produce a new picture by rotating the picture p no.of.degrees degrees clockwise about the origin.

scale p **by** x.scaling, y.scaling

will produce a new picture by multiplying the x and y-coordinates of every point in the picture p by x.scaling and y.scaling respectively.

Text can be included in pictures using the text statement. This takes a string of characters and a base line and constructs the picture of those characters along the base line.

let p = **text** "hello !" from 1, 1 to 2, 1

The characters will always be drawn from the first to last point of the base line. As a consequence text can be inverted by ending the base line on the left of its starting position.

Colour can also be specified in a picture but, unlike the other picture operations, the effect of this will depend on the physical output device used.

These are the basic support facilities for line drawing. Particular applications packages built on these facilities, for example curve fitting or 3-D modelling, may be stored in and retrieved from the persistent store as pictures themselves or procedures that produce or manipulate pictures. The choice is made according to the requirements of the application.

5.2. Images

In general an image is a 3 dimensional object made up of a rectangular grid of pixels. A pixel has a depth to reflect the number of planes in the image and the image has an X and Y dimension to reflect its size. In its most degenerate form a pixel is one spot which is either **on** or **off**. Thus

let a = **on**

creates a pixel a with a depth of 1. To form a pixel of depth 4 say we could write

let b = **on**, **off**, **off**, **on**

which creates b this time with a depth of 4. To form an image with an X and Y dimension different from 1 we could write

let c = **image** 5 **by** 10 **of on**

which creates c with 5 pixels in the X direction and 10 in the Y direction all initially on. The origin of all images, which is at the bottom left hand corner of the image, is 0,0 and in this case the depth is 1.

Full 3 dimensional images may be formed by, for example

let d = **image** 64 **by** 32 **of on**, **off**, **on**, **on**

which would create d as an image of depth 4 with 5 pixels in the X direction and 10 in the Y direction all initialised to the pixel value **on**, **off**, **on**, **on**.

In order to introduce the concept of and operations on images gently we will restrict ourselves for the present to images with a pixel depth of 1. Everything that we say will be true for images of greater depth as we will see later.

Images are first class data objects and may be assigned, passed as parameters or returned as results. e.g

let b = a

will assign the image a to the new one b. In order to map the operations usual on bitmapped screens, the assignment does not make a new copy of a but merely copies the pointer to it. This as we will see later is consistent with update in place raster operations on most raster devices.

There are 8 raster operations which may be used as described in the following BNF.

<void-clause> ::= <raster.op><image-clause>onto<image-clause>

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

thus

xor b **onto** a

performs a raster operation of b onto a using xor. It should be mentioned that a is altered in situ as would be expected on a raster device. Both images have origin 0,0 and automatic clipping at the extremities of the destination image is performed.

The limit operation allows the user to set up windows in images. e.g.

let c = **limit** a **to** 1 **by** 5 **at** 3, 2

sets c to be that part of a which starts at 3,2 and has size 1 by 5. c has an origin of 0,0 in itself and is therefore a window on a.

Rastering sections of images on to sections of other images can be performed by for example

```
xor limit a to 1 by 4 at 6, 5 onto
  limit b to 3 by 4 at 9, 10
```

Automatic clipping on the edges of the limited regions is performed. If the starting point of the limited region is omitted 0,0 is used and if the size of the region omitted then it is taken as the maximum possible. That is from the starting point to the edges of the host image. Limited regions of limited regions may also be defined.

The standard identifier screen is an image representing the output screen. Performing a raster operation onto the image screen alters what may be seen by the user. e.g.

xor a **onto limit screen to** 4 **by** 5 **at** 4, 7

will raster a onto the defined section of the screen. This will be visible to the user.

The standard identifier cursor is an image which represents the cursor. The cursor may be altered in the same manner as any image. e.g.

copy b **onto cursor**.

For hardware with more than one colour plane, the pixels can also have depth. As we have seen before

let a = **image** 64 **by** 32 **of on**, **off**, **on**, **off**

is a 64 by 32 image with a depth of 4(i.e. 4 planes). The planes of the pixel are numbered from 0 and so a above has planes 0,1,2 and 3.

In systems that support multiple planes the standard identifiers screen and cursor will have a depth greater than 1. All the operations that we have already seen on images(raster, limit and assignment) work more generally with depth. Thus the raster operations perform the raster function plane by plane in one to one correspondence between source and destination. Automatic depth clipping at the destination is performed and if the source is too small to fill all the destination's planes then these planes will remain unaltered. The limit and assignment operations also work with the depth of the image.

The depth of the image may be restricted by the depth selection operation. For example

let b = a(1|2)

yields b which is that part of a which has the two depth planes 1 and 2. b has depth origin 0 and dimensions 64 by 32. Also

let c = **limit** a **to** 32 **by** 16 **at** 8, 8 (2|1)

yields c which is plane 2 of a of size 32 by 16 starting at point 8,8 in a. c has origin 0,0 and depth 1.

Standard functions provided by the system include facilities to find the size of images, extract pixels from images, make the cursor invisible and defining which point is the tip of the cursor. Other standard functions provided include a seed fill operation, a pop up menu mechanism, fonts and functions to attach the mouse to the screen. The facilities are stored in the persistent store and may be called by the user when required.

Finally the PS-algol system provides two functions for manipulating the colour map of the device. The first is,

colour.map(**pixel** p ; **int** i)

This functions sets the integer produced by the colour map when pixel p is displayed to be i. The second function allows the user to interrogate the colour.map and is,

colour.of(**pixel** p -> **int**)

This function returns the integer corresponding to the pixel p in the colour map.

5.3. Mapping Pictures and Images to Output Devices

As we have seen, images use standard names, screen above, of which there may be many to define the device to be used. Pictures may also be drawn on calligraphic devices of standard names. However if such a device is not available or we wish to draw a line drawing on a raster device then we may map a picture on to an image. e.g.

draw(an.image,a.pic,0.0,3.2,1.5,3.9)

will draw the section of the picture a.pic bounded by the box specified by the points (0.0,3.2) and (1.5,3.9) on the image an.image. Automatic clipping of the line drawing is performed to make it fit the bounding box.

The picture may be drawn directly on to the screen or any part of it. Once the line drawing has been mapped on to an image the image may be manipulated by any of the image operations. Notice also that we can now easily mix pictures and images on a screen choosing which ever one is appropriate for each section of the screen.

The ubiquity of graphical devices both calligraphic and bit map and the dominance of human computer interfaces in applications programs, makes it essential that the programming language supports these activities correctly. It is also important that the graphical data is supported by the persistence mechanism since it is most likely to be used on a start stop basis over a long period of time.

6. Large Scale System Construction

In this section we turn our attention to mechanisms that may be used for large scale

system construction. We have already described how the persistent store may be used to simplify the mechanisms for binding. Fundamental to this is that large systems must be capable of evolving and to accommodate this we first discuss the tradeoffs between static and dynamic binding.

With static binding some errors may be detected early in the software life cycle. This is so important to some language designers such as those of ML, Argus[16] and Pebble[11] that they insist that all binding be static. Where it is possible to perform static checking it is also possible to make the programs more efficient since checks can be factored out.

In large scale systems the issue of binding is of major importance and it is in such systems that static binding has hidden costs. Any change to program or data form requires a recompilation to re-establish the bindings.

For large systems involving software evolution or the distribution of software this recompilation may be prohibitive in cost.

It is interesting to note that to overcome these costs and to accommodate dynamic evolution of programs and data, most programming languages allow file names to be bound dynamically and read statements to involve a dynamic type check. This is true even in, so called, strongly typed languages such as Pascal. Indeed it is not well recognised that these languages allow static binding within the program and resort to the operating system, file system or database management system to provide dynamic binding.

We assert that both methods of binding are necessary for large scale system construction and evolution. Static binding would interpret the above file names in the compilation environment, thus freezing the program and data form. The second type of binding would interpret the file names in the context of the run time environment. This is dynamic binding. We recognise that it is advantageous to statically bind wherever possible with dynamic

binding available where necessary. We would expect small objects to be statically bound to form larger objects, with interfaces between those objects being dynamically bound. We coin the term flexible incremental binding (FIB) to describe this mixture of bindings, which we expect to obey the Principle of Correspondence.

6.1. Name Spaces

To accommodate these flexible incremental bindings we have invented the name space. This is an environment mechanism that permits the following:

a) the storage of bindings in a name space;

b) the dynamic use of names from a name space;

c) the static use of names from a name space;

d) the evolution of the names available in a name space;

e) safe exchange of arbitrary data between parts of the system - especially between the permanent store and programs.

A name may be statically or dynamically bound, and the system may determine the type and object associated with the name. However, we wish to compile most code in contexts where there is enough information to statically bind and statically type check. Even when dynamic binding or checking the strictness of type checking will not be compromised.

Name spaces are data objects which act like abstract data types to yield the values of the name space. These values are bindings between name and object and the name may be used in the current environment. To statically bind to a name space we use

with name-space-expression **compile** statement

which provides an environment for the statement containing all the names in the name space. The name space may be an object in the persistent store and the statement is bound in the compile time environment. All binding is on a need to know basis and only what is used in the statement needs to be checked when the program is run to ensure that the bindings are still valid. This can be reduced to the comparison of two integers when the name space is first used. If they are not recompilation is necessary.

To bind a name space dynamically we use

using name-space-expression **with** signature **do** statement

In this the name space yielded by the expression must have the properties (name-type pairs) described by the signature in order to match correctly. It is a partial match and is the same mechanism that we use to match actual to formal type parameters in procedures and abstract data types to achieve parametric dependent polymorphism as first used in Russell[13].

This dynamic interfacing is under the system builder's control. Most interfaces are statically bound to form reasonable units but the dynamic boundaries identify recomposition points for economic rebuilding.

In passing it should be noticed that name spaces can be used as the conventional interface between system components of different technologies.

To accommodate evolution, name spaces may grow and contract. For example

extend name-space-expression **with** identifier-list **from** statement

adds the new definitions of the identifier names to the name space. To contract the name space we use

drop identifier-list **from** name-space-expression

which will remove the identifier names from the name space.

As a name space is itself an object in the language, a name in a name space may be bound to a name space. Hence hierarchical naming structures similar to directory hierarchies are possible.

Program libraries are obtained by putting procedures or abstract data types in name spaces. The programmer can choose whether to statically bind to a routine in the library and thus find that certain library maintenance operations result in the program needing recompilation, or to bind dynamically so that the program will operate as long as the library provides a routine of the expected name and type. The type checking, binding and loading for all data objects, which has been described and prototyped[7], replace such things as: type checking linkers, consolidators, APSE databases and associated utilities. Separate compilation is achieved by running programs which leave program parts in such a program library.

Permanent storage is usually provided by the filing system of an operating system. Programs use this storage to convey results to later programs. Even applicative languages, and those that save workspaces, perform updates on such a store. By distinguishing one or more name spaces, say PermanentStore etc, which for a new system starts empty, and arranging to preserve all data reachable from them, as we have explored in PS-algol, then permanent storage is provided and the interprogram communication previously provided by the file system may now be coded within the language but now it is properly type checked. Furthermore different users may adopt different naming schemes and organisations of permanent store.

A common requirement is to name a complex data structure, so that an instance of it can be identified for an activity with a given program. For example, a CAD system may be used for many projects. The data associated with each project may be a name space, and all of these may be named in PermanentStore. The current programs constituting the CAD system may be in another name space, say CAD, also made accessible by being bound in PermanentStore. The user may apply any program (procedure) from CAD to any of the project databases. The vendor of the CAD system may provide replacement versions from time-to-time, which will be installed, replacing the previous value of CAD. They can be written so as to bind to the existing projects, but so as to extend them to contain new types of data. Bindings via name spaces of customised programs, the package user has written, will still run. Thus revised software or other data may be delivered to a customer without revealing the program source or proprietary data structures, and this can be used with existing data and software, which may represent a substantial investment by the customer. Such dynamic reassembly is essential with large commercial systems, and is not provided by systems dependent only on static binding, when they claim to provide persistence.

6.2. Outstanding problems

With file directories it is possible to write general utilities which scan a directory, often interactively, revealing the directory's contents and possibly allowing user action - e.g. delete - on each file. Some equivalent operations on name spaces may be useful, however, it is difficult to identify a primitive in terms of which they may be written. For example, how does one type the control variable of an iteration over a name space and how does one represent and manipulate names? This problem is not peculiar to name spaces, it also arises in scanning other structures that yield an environment, for example when writing general

utilities for browsing, diagnosis, statistics collection and generalised data input and output. Current practice is to step outside the type system or to depend on calling the compiler. Our desire to support more of the total programming activity within the strictly typed language leaves us confronting this problem.

7. Conclusions

We stated that we were searching for a language or a family of languages that are capable of supporting all programming activity. By simplifying the number of concepts in a language and integrating them using our design methodology we will achieve economies in the production of software using our systems. Whether we will achieve this one language is an open question but it is quite clear that we can travel a long way along this road and achieve most of our aims.

We hope we have convinced any language designer reading this, who is engaged in programming environment design, that orthogonal persistence is essential as an enabling technology. We have noted that there are outstanding problems of the semantics of nested transactions in a concurrent environment and the controlled use and manipulation of names. We think solutions to these are possible by the 1990's.

With orthogonal persistence we have argued for higher order functions to integrate the use of data and program. Our type alchemist's dream is the focus of much research, not only by us but by many other as well, and we are hopeful of good results.

We presented our results in graphical data and the advantages of such data in a persistent environment. Finally, we introduced name spaces as a controlled mechanism for binding. Most of what we have talked about has been implemented in PS-algol and we are currently engaged in the building of its successor to implement the rest. It only remains for us to popularise the concepts! Before 1990?

8. Acknowledgements

Our dependence on a working implementation of PS-algol, from which we have learnt much, cannot be overstated. It was built largely by the following members of our team: Pete Bailey, Fred Brown, Paul Cockshott and Al Dearle. We also benefited from suggestions and ideas arising in discussions with Peter Buneman and Tony Davie.

The work is supported at Glasgow by SERC grants GRC 21977 and GRC 21960 and at St Andrews by SERC grant GRC 15907. The work is also supported at both Universities by grants from ICL.

9. References

1. Atkinson, M.P., Cockshott, W.P. & Chisholm, K.J. CMS : A chunk management system. Software, Practice & Experience 13, 3 (1983), 273-286.

2. Atkinson, M.P., Cockshott, W.P., Chisholm, K.J. & Marshall, R.M. Algorithms for a persistent heap. Software, Practice & Experience 13, 3 (1983), 259-272.

3. Atkinson, M.P., Cockshott, W.P. & Chisholm, K.J. NEPAL - The New Edinburgh Persistent Algorithmic Language. DATABASE Infotech State of the Art Report 9, 8 (1982), 299-318.

4. Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. An approach to persistent programming. Computer Journal 26, 4 (1983), 360-365.

5. Atkinson, M.P., Bailey, P.J., Cockshott, W.P. & Morrison, R. PS-algol reference manual. Universities of Glasgow and St Andrews PPRR-12 (1984).

6. Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. Progress with persistent programming, Database - role and structure, Cambridge University Press, Cambridge, 1984.

7. Atkinson, M.P. & Morrison, R. Procedures as persistent data objects. ACM.TOPLAS 7, 4 (1985).

8. Atkinson, M.P & Buneman, O.P. Database programming language design. in preparation.

9. Atkinson, M.P. & Morrison, R. Types, bindings and parameters in a persistent environment. Proc. Appin workshop. Universities of Glasgow and St Andrews PPRR-16.

10. Blasgen, M.W. et al. System R : an architectural overview. IBM systems journal 20 (1977), 41-62.

11. Burstal, R. & Lampson, B. A kernal language for abstract data types and modules. Proc. international symposium on the semantics of data types, Sophia-Antipolis, France (1984).

12. Cockshott, W.P., Atkinson, M.P., Bailey, P.J., Chisholm, K.J. & Morrison, R. The persistent object management system. Software, Practice & Experience 14 (1984).

13. Demers, A. & Donahue, J. Revised report on Russell. Technical report TR79-389, (1979), Cornell University.

14. Evans, A. PAL a language designed for teaching programming linguistics. Proc. ACM 23rd. Nat. Conf. Brandin Systems Press (1968), 395-403.

15. Landin, P.J. The next 700 programming languages. Comm.ACM 9, 3 (1966), 157-164.

16. Liskov, B., Herlihy, M., Johnson, P., Leavens, G., Scheifler, R. & Weihl, W. Preliminary Argus manual. Technical Report Momo 39 (1983), M.I.T.

17. McCarthy, J. et al. Lisp 1.5 Programmers manual. M.I.T. Press Cambridge Mass. (1962).

18. Milner, R. A proposal for standard Ml. Technical Report CSR-157-83 University of Edinburgh. (1983).

19. Morris, J.H. Protection in programming languages. Comm.ACM 16, 1 (1973), 15-21.

20. Morrison, R. Ph.D. Thesis. University of St Andrews (1979).

21. Morrison, R. Low cost computer graphics for micro computers. Software, Practice & Experience 12, 8 (1982), 767-776.

22. Mylopoulos, J. & Wong, H.K.T. Some features of the Taxis data model. 6th International conference on Very Large Data Bases. (1980). Montreal.

23. Naur, P. et al. Revised report on the algorithmic language Algol 60. Comm.ACM 6, 1 (1963), 1-17.

24. Reynolds, J.C. Gedanken a simple typeless language based on the principle of completeness and the reference concept. Comm.ACM 13, 5 (1970), 308-319.

25. Schmidt, J.W. Some high level language constructs for data of type relation . ACM.TODS 2, 3 (1977), 247-261.

26. Smith, J.M., Fox, S. & Landers, T. Reference manual for ADAPLEX. Computer Corporation of America, Cambridge, Massachusetts (1981).

27. Strachey, C. Fundamental concepts in programming languages. Oxford University Press, Oxford (1967).

28. Taylor, R.C & Frank, R.L. CODASYL database management systems. ACM Computing Surveys 8, 1 (1976), 67-103.

29. Tennent, R.D. Language design methods based on semantic principles. Acta Informatica 8 (1977), 97-112.

30. Turner, D.A. SASL language manual. University of St.Andrews CS/79/3 (1979).

31. Wirth, N. The programming language Pascal. Acta Informatica 1, 1 (1971), 35-63.

32. Wirth, N. & Weber, H. EULER a generalisation of algol. Comm.ACM 9, 1 (1966), 13-23.

33. Zilles, S.N. Procedural encapsulation : a linguistic protection technique. ACM Sigplan Notices 8, 9 (1973).

34. van Wijngaarden, A. Generalised algol. Annual Review of automatic programming 3 (1963), 17-26.

35. van Wijngaarden, A. et al. Report on the algorithmic language Algol 68. Numerische Mathematik 14,1 (1969), 79-218.

36. Krablin, G.L. Building flexible multilevel transactions in a distributed persistent environment. Proc. of the Appin worshop. PPRR-16 (1985).

37. Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P. An integrated graphics support environment. Universities of Glasgow and St Andrews. PPRR-14 (1984).