On the Construction of Persistent Programming Environments

Alan Dearle

Computational Science Department
University of St. Andrews
St. Andrews

March 1988

Acknowledgements

A number of people must be thanked for their help during this project. Firstly, Professor Ron Morrison, my supervisor, for his constant patience, advice, encouragement and good humour. Fred Brown, my sparring partner for the last five years, who is always ready to "fix" problems that arise. Richard Connor, for his part in implementing the Napier type checking module and for his suggestions on parametric polymorphism. The final member of the PISA project at St. Andrews, Ray Carrick, must also be thanked for his part in the research effort.

Others that must be mentioned are: Professor Malcom Atkinson for, amongst other things, inventing persistence, Dr. Paul Cockshott for his advice in "the early days" and Pete Bailey whose mark is unmistakeable on parts of the PS-algol and Napier abstract machine.

The Apple Computer Corporation also deserve a mention for producing an excellent document production vehicle.

Finally, I would like to thank my wife, Fiona, who proof-read this thesis and put up with my elations and depressions during the period of writing.

Abstract

This thesis presents research into the construction of persistent programming systems. Much of the thesis is concerned with the design and implementation of persistent programming languages, in particular PS-algol and Napier. Both languages support machine independent vector and raster graphics data types. Napier provides an environment mechanism that enables the incremental construction and binding of programs. Napier has a powerful type system featuring parametric polymorphism and abstract data types.

The machine supporting Napier, the Persistent Abstract Machine, is investigated. The machine supports an efficient implementation of parametric polymorphism and abstract data types.The Persistent Abstract Machine has a layered architecture in which permits experimentation into language implementation and store design.

The construction of compilers in a persistent environment is explored. A flexible compiler architecture is developed. With it, a family of compilers may be constructed at relatively little cost. One such compiler is the callable compiler; this is a first class data object in the persistent environment. The uses of such a compiler are explored, in particular in the construction of an object browser.

The persistent object browser introduces a new software architecture that permits adaptive programs to be constructed incrementally. This is achieved by writing, compiling and linking new procedures into an executing program. The architecture has been sucessfully applied to the construction of adaptive databases and bootstrap compilers.

Contents

1 Introduction

The constant decrease in the cost of hardware components has given rise to a proliferation of computer systems in every aspect of everyday life. Kettles, toys, aircraft, libraries, banks and nuclear power stations are all affected by the so called computer revolution. The dramatic reduction in the cost of hardware is unparalleled in the industrial revolution. In no other area have production costs halved and power doubled in an unerring four year cycle [bro87]. Consequently, it should be of no surprise that the software industry has failed to match this explosive growth.

The software crisis of the 1960's [bux69] brought into focus the fact that hardware development was advancing faster than the ability to produce software. The crisis was catalysed by the availability of the newly available third generation computer hardware. It was capable of providing hardware support for sophisticated systems at moderate costs. However, the production of software systems could not match the demand caused by the arrival of these systems.

Since the 1960's the problem has not diminished, in fact, software production costs have increased in the intervening period. This is largely due to the labour intensive nature of software production. Software cannot be mass produced like cars[weg84]. Unlike cars, no two pieces of software are identical, since if they are, they are merged into a single abstraction.

The real costs of software production are now astronomical. It has been estimated [joe83] that, in America, software production and maintenance now costs 2% of the gross national product. Therefore even small savings made to the software life cycle will result in a vast reduction in economic expenditure.

Boehm [boe86] gives four strategies for improving software productivity:

1. Write less code
2. Get the best from people
3. Avoid re-work
4. Develop and use integrated project support environments

Getting the best from people is the domain of managers. The other three topics provide the thrust for the work documented in this thesis. This view point is retrospective since Boehm had not given his address when the work was started.

1.1 Writing Less Code

A program satisfying some specification may be produced in less fewer of code if written in a high level language rather than a low level one. For example, a ten line program written in Ada may be equivalent to a few hundred lines of assembly code. An IBM survey [ibm78] found that programmers produce approximately the same number of lines of code per day no matter what languages are used. This in turn means that the cost of program is directly proportional to its size. Therefore if a high level language is used to produce a product fewer programmer days are required to produce it - resulting in lower costs.

The provision of programming languages with a high degree of compile time type checking allows errors in programs to be detected early in the software life cycle. Systems written in languages with strong type systems take longer to write but contain fewer semantic errors [boe87]. This leads to less time being needed in the testing and debugging phase of software development and, perhaps most importantly, the resulting code requires less maintenance.

The production of a piece of software is highly labour intensive and its cost is directly proportional to the amount of programmer time that is required to produce it. It is therefore essential to find ways of reducing the amount of programmer time required to produce software. The use of strong type systems is costly in terms of machine time. Machines must perform more checks to ensure a program is well typed but this results in less time being spent by the programmer during the potentially expensive debugging cycle.

The advantages of using very high level languages motivated the U.S. Department of Defense to adopt Ada [ada83] as a standard language. One of the design aims of Ada was to provide a language in which the specification of programs was indistinguishable from their implementation. This has also been the aim of researchers designing so called executable specification languages [gog82,kre80].

The use of prototyping has been compared with specification techniques in experiments [boe87]. In these experiments it was found that the use of prototyping resulted in 40% less code being produced with 40% less effort. Furthermore, the resulting products were easier to understand and therefore maintain.

Balzer [bal87] stresses the need to be able to evolve prototypes into an operational system rather than discarding prototypes. This result suggests that the programming systems we develop should support a smooth transition between prototypes and products.Therefore, an incremental development facility is required to support prototyping Prototypes are highly complex, structured objects requiring more sophisticated support tools than a mere text editor. In general, such incremental development requires an integrated project support environment (IPSE).

The production of software is an expensive labour intensive activity. Consequently, any technology that allows a given problem to be solved by writing less code than other methods will lead to the production of cheaper software. Good candidate technologies are: strong type systems, high level languages and integrated project support environments.

1.1.1 Language Design

The provision of high level languages leads to a reduction in programmer time spent on a given problem. A high level programming language should be capable of supporting the development of a broad spectrum of applications,

1. development of data models
2. generic toolsets
3. object based systems
4. adaptive systems
5. user interfaces

In order to perform these tasks most programming systems depend on a plethora of different mechanisms, these include:

1. command languages
2. filing systems
3. compilers
4. interpreters
5. linkers
6. symbolic debuggers
7. DBMS sublanguages
8. graphics libraries

The diversity of these mechanisms increases the cost of maintaining even the simplest software systems. The approach taken in designing PS-algol [ps87] and its successor,

Napier [mor88b] is to attempt to provide a language capable of supporting all of the needs of the application programmer. The resulting simplification should result in an overall saving throughout the life cycle of the program [atk83].

Simplicity is the cornerstone of programming language design. The addition of new features without integrating them into an overall framework merely increases the complexity of the system. This complexity often overloads the programmer beyond his or her capability. In contrast, a simple language allows the programmer to concentrate on the inherent problems of the task and not on the mapping between the solution and the programming vehicle. This view was epitomised by van Wijngarden [vw69] who states,

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

This message is restated by van Wijngarden as,

> *power through simplicity, simplicity though generality*

He argues that languages are too complex and that complexity is due, at least in part, to languages being too restrictive. The PISA [atk86b] languages are designed using three principles attributed to Landin [lan66], Strachey [stra67], Tennant [ten77] and Morrison [mor73].

> ***The Principle of Correspondence****: the use of names should be consistent within a system. In particular there should be a one to one correspondence between the method of introducing names in declarations and parameter lists*
>
> ***The Principle of Abstraction****: all major syntactic categories should have abstractions defined over them. For example, functions are abstractions over expressions.*
>
> ***The Principle of Data Type Completeness****: all data types should be first class without arbitrary restriction on their use.*

The power is derived from the generality of these three principles, the simplicity from the lack of deviation from them. The ultimate goal is to produce a totally integrated environment capable of supporting the needs outlined above. In such an environment the user never has to step outside it for any computational task. Central to this ideology is the provision of orthogonal persistence, which is discussed below.

1.2 Avoiding Rework

Parnas cites information hiding as one of the most effective ways of avoiding rework [par79]. If implementation decisions are hidden inside module interfaces, ripple-through effects may be avoided when changes are made to one part of a system. This approach has proved extremely effective in eliminating rework during system evolution.

The concept information hiding is central to the object-oriented methodology which is based on ideas originally developed in Simula [nyg70]. In this paradigm, methods may produce or modify members of a class of objects. Wegner [weg87] defines object-oriented languages as having three essential features:

1. the ability to define objects as a set of operations and a state that remembers the effect of operations;
2. objects may be categorised by type; and
3. there is an inheritance mechanism for defining superclasses and subclasses.

The support of objects, which contain both methods (procedures) and instances (data) is in contrast with the traditional separation of code and data.

1.2.1 Uniform Treatment of Program and Data

In many programming languages such as Algol 60 and Pascal [wir73] procedures may only be declared, passed as parameters or executed. As Zilles [zil73] and Morris [mor73] point out, in order to exploit the device to its full potential it is necessary to promote procedures to full first class data objects. That is, to give them the same civil rights as any other data object in the language such as being assignable, the result of expressions, procedures and blocks and being elements of data structures. This is in accordance with the principle of data type completeness.

The power of first class functions has long been known to lisp [mcc62] programmers, Lisp was the first programming language to have first class procedures. The technique of applicative programming depends on the ability to have procedures as first class data objects.

The main advantage of providing first class procedures as data objects is that there is a simple and well understood mechanism for system construction. Therefore no special provision needs to be made for the provision of libraries and loaders. Separate compilation may also be achieved by running procedures (a compiler) that introduce procedures into the environment. This technique is discussed fully in chapter 6. The power that may be extracted from first class functions is fully discussed in [atk84] and [mor85].

The implementation of languages with first class functions provides some problems not otherwise encountered. Solutions to these problems are fully discussed in chapter 4

1.2.2 Software Reuse

The cheapest way of obtaining software is to reuse code that has already been produced. The most extreme form of this reuse is purchasing "off the shelf" packages. If software is to be reused successfully, it is important that mechanisms are provided in the environment to support reuse. Several questions arise over how this support may be provided, some of these are:

1. How do you write programs that may be reused?
2. How do you store reusable programs?
3. How do you catalogue reusable programs?
4. How do you find a program to reuse?

The provision of a polymorphic type system may be used to facilitate the writing reusable software [mor87a]. To support the reuse of software the system must be capable of storing the polymorphic and other functions that are produced. It is important that the storage of code preserves the type information associated with it or many of the advantages of using a strong type system may be lost.

Storing a potentially reuseable piece of code is of little use unless someone else may find it and reuse it. One of the most effective ways of discovering what code is available to reuse is by browsing over it. This technique has been used successfully in the design of the Smalltalk-80 [gold83] system and much its success may be attributed to it. Browsing has proved to be of utility in browsing strongly typed objects in PS-algol. The PS-algol object browser is discussed in chapter 7.

1.3 Using Integrated Project Support Environments

If IPSE's make software cheaper to produce it must be ascertained that the definition of an IPSE is clear. Boehm distinguishes an IPSE from a collection of ad-hoc tools by the

amount of gross integration available [boe86]. The tools available in toolkits such as Unix work in isolation. Tools operate in isolation with no knowledge of other tools or of any special data structures being used. On the other hand in an IPSE the tools share common schemata. Each has specialised knowledge of the data being manipulated and tools may even use other tools in the system. The IPSE operates as a central depository where objects being manipulated may be found and where schema information is stored.

Systems vary considerably in the amount and nature of integration that they support. Consider the Macintosh toolset [mac86] and the Unix programming environment [ker79].

The Macintosh toolset has regular sub-systems with well documented interfaces and conventions. For example, a set of facilities are provided for managing graphics data and another for managing system events. The application developer may combine these facilities to produce complex applications with ease. The toolbox functions interface most smoothly with Pascal programs although they may be used from programs written in other languages with a little more trouble.

The Unix environment lacks the fine grain system integration of the Macintosh. Libraries of functions are provided to support various programming activities but do not have the coherent structure of the Macintosh toolkit. However, Unix provides much more global integration than the Macintosh. Tools are provided for a variety of tasks, for example,

C specific tools:	cc, dbx, lint
documentation tools:	spell, troff, wc
configuration management tools:	make,SCCS
help tools:	apropos,man

These tools may be combined by the extremely powerful mechanism of the pipe. Pipes allow the output of one process to be connected to the input of another. Programmers may chain tools together to provide yet more powerful ones.

Unix does provide libraries of functions, typically these libraries provide mechanisms to support tasks such as mathematical functions and I/O. When a programmer wishes to use these libraries a linker is employed to resolve addresses. The linker produces a new program with copies of the library functions bound into the program. This means that if a function is heavily used many copies of it exist in the system resulting in large modules of executable code. Furthermore, if a library function is changed for any reason all the code that uses that function must be relinked. This is an expensive mechanism for supporting software reuse.

The success of Unix is largely due to the uniform I/O structure of which pipes are a part. However, with this I/O structure comes one of the weaknesses of Unix. That is the fact that the data supported by the I/O system is not structured. The character stream is an extremely low level communication medium. If two tools are to operate on the same structured data the data must be linearised, passed down a pipe and then reconstructed. This process is expensive in both CPU time and programmer time since the mappings involved are often complex. The restrictions imposed by this architecture are one of the major motivations in moving towards object stores.

The need to map structured data to flat data and vice versa is aggravated in Unix by the lack of any ability to share data in primary memory. In most Unix systems it is impossible to share data structures in RAM between different processes. The first level of sharing data is therefore at the file system level which requires data to be mapped via the I/O system.

Due to the relative expense of process invocation in Unix, tools tend to be large objects. The tools are generally written in the language C and invoked from the command language known as the shell [bou78]. This results in many useful pieces of code being

trapped within tools. Reuse in Unix is therefore good at the global level but poor at the subsystem level.

1.3.1 Public Common Tools Interface

The Public Common Tools Interface (PCTE) is an architecture designed to provide several important facilities to application developers, these include:

1. Reduce development costs of tools.
2. Facilitate the exchange of software tools.
3. Allow integration of tools in comprehensive, uniform and homogeneous Software Engineering Environments

The above considerations lead the designers naturally to the realization that a unified framework is required to support such an architecture. Furthermore, this architecture needs to be based on powerful mechanisms, especially in the area of object management. However, one of the criteria placed upon the designers of PCTE was that a smooth transition must be possible between current programming practices and the use of PCTE. Therefore, the system had the limitation that initially tools had to operate in the Unix environment. The limitations of this design decision have already been discussed. Nevertheless, the realisation that a comprehensive, uniform and homogeneous environment is needed to support IPSE's is important.

1.3.2 Ada Programming Support Environment

The developers of Ada recognised that in addition to a high level language a programming support environment was also necessary. Without such an environment, programs written in a high level language must depend on operating system facilities. This in turn reduces the portability advantages of using a high level programming language.

Another strong motivation for the provision of a common support environment was the observation that the size and complexity of support software often exceeds the size of the embedded system being supported. If the environment is shared between many developers this high support cost may also be shared resulting in a reduction of the total software cost.

A number of constraints were placed on the design of the APSE Architecture [dod83]. These constraints are similar to the constraints placed upon the design of the Persistent Abstract Machine although the motivations were slightly different. To achieve portability the following constraints were made upon the APSE design:

1. All tools must be written in Ada
2. The APSE must be structured in layers, each layer being dependent on only the subsidiary layer.
3. One layer, the kernel layer, must provide access to a system database and facilitate communications

1.4 Object Storage

Boehm cites the provision of a project master data base or persistent object base as one of the important features distinguishing an IPSE from a collection of ad-hoc tools. The need for a uniform, homogeneous object storage facility is also identified by the PCTE and the ADA Apse designers.

Programming languages normally have little support for the maintenance of long term data. The only concession made to this requirement is usually the provision of a file data

type. Therefore, the programmer is faced with the task of mapping data onto long term storage; this is usually provided by the file system or DBMS. The mapping of data between long and short term storage is expensive, both in terms of programmer design time and program run time. In 1978, Atkinson [atk78] recognised this problem and isolated a property of data known as persistence.

Persistence is defined to be the length of time for which data exists and is usable [atk83]. It is therefore an abstraction over a physical property of data; the length of time it is kept. Traditionally, programming languages partition data into two categories:

1. data whose lifetime does not exceed program invocation
2. data whose lifetime does exceed program invocation

If data falls into the first category, it is managed entirely by the programming language. Programming language designers have developed many different methods of structuring this data, these vary from relatively simple objects such as arrays to more complex ones such as abstract data types.

However, if data falls into the second category it is managed by a file system or a database management system. Curiously, a completely different set of modelling techniques have developed to structure long term data. In this category, the methods adopted include relational [cod70], hierarchical [loc78], network [tay76] and functional data models [shi81].

These two views of data have certain disadvantages. Firstly, there is usually a considerable amount of code, typically 30% of the total [ibm78] concerned with transferring data to and from files or DBMS. A large amount of space and time is taken up by code to perform translations between the program's form of data and that used for the long term storage of data. For example, programmers normally have to flatten and rebuild graphs or trees modelled in the programming language in order to write them out or read them in from a file system or DBMS. The time inefficiency incurred includes both programmer design time and the run-time efficiency of the program.

A second more serious disadvantage of maintaining two forms of data is that data type protection is lost across the mapping. A structure used to aid comprehension in the programming language domain may not be available in the DBMS or file system.

In persistent programming languages the issue of what data structuring techniques are required is separated from a concept known as persistence. For example, in a persistent system, relations could be used as a logical structuring technique without concern with their long term storage.

In accordance with the rules of Strachey, Landin and Tennant given above, three new rules for persistent data have been given in [atk83]. They are:

> ***The Principle of Persistence Independence****: 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. For example, it should be possible to call a procedure with its parameters sometimes objects with long term persistence and at other times only transient.*
>
> ***The Principle of Persistent Data Type Completeness****: In line with the principle of data type completeness all data objects should be allowed the full range of persistence.*

In a persistent system the use of all data is independent of its persistence. This notion of persistence may be extended to abstract over all the physical properties of data, for

example, where it is kept, how long it is kept and in what form it is kept [mor87b]. The use of the persistence abstraction removes the need to explicitly program for the differences in the use of long and short term data.

Thus, a programming language that supports persistent data objects provides the homogeneous object storage facility required in order to construct integrated project environments. Indeed, the ability of a persistent store to transparently manage the long term storage of complex data objects makes it an ideal vehicle for building many applications and indeed, Atkinson's original motivation for investigating persistence was to support computer aided design (CAD) systems.

The problems in CAD systems are similar to the ones facing the designer of an IPSE. The most obvious similarity is the need to share potentially large amounts of structured information which may persist over arbitrary periods of time. This ability of a persistent store is in marked contrast to the unstructured byte streams supported by Unix. Another similarity is the inherent complexity of the problem domain. In such systems the management of long term data often makes a clean design, with the corresponding lower software maintenance costs, extremely difficult.

The problems associated with scale are often overlooked. Both CAD systems and IPSE's are large, complex pieces of software. If a system is small it is easy to maintain and understand. Systems such as those discussed are so large and complex that it is almost impossible to understand all of them except at the highest levels of abstraction. A persistent system must therefore provide support for programming in the large as well as programming in the small.

The persistent store serves as a unified repository of the entities created during the software development process. A persistent store is therefore an ideal vehicle for the construction of IPSE's [mor85] and other large systems.

In order to provide the necessary savings in software costs the programming languages used must be as high level as possible. Persistence should, therefore be an orthogonal property of data. The language must support the reuse of code. This may be achieved by the provision of procedures as first class data objects that may be stored in the persistent store combined with a type system that includes parametric polymorphism. The language should also provide mechanisms such as abstract data types to allow information hiding. Incremental system construction and mechanisms to control change must also be supported if the language is to be used for the construction of large software systems.

Two programming language systems have been used as vehicles to explore the problems outlined above. Both languages, PS-algol and Napier, treat persistence as an orthogonal property of data. The following sections introduce the research areas addressed in this thesis.

1.4.1 Identification of Persistence

There are three main methods for identifying this persistent data:

1. All data persists.
2. Only data explicitly marked persists.
3. All data reachable from one or more roots will persist.

The languages PS-algol [ps87] and Napier [mor88b] both use the third method of identifying persistent data, the reasons for this are discussed in [bro88]. The implementation mechanisms behind this are discussed in chapter 4. How persistence appears at a language level is discussed below.

1.4.2 Persistence in PS-algol

In the language PS-algol, persistence is provided by an extendible number of roots known as databases. Every object reachable from the transitive closure of a database is persistent.

A named database may be opened for reading or writing by a function called open.database. It is defined as follows

```
let open.database = proc( string name, password, mode -> pntr )
```

This says that the procedure open.database takes three parameters of type string. The first is the name of the database which is to be opened, the scond is the database password. The third is the mode in which it is to be opened. Acceptable modes are "read" or "write". The procedure returns a pointer. In PS-algol the datatype **pntr** is comprised of the infinite union of all labelled cross product types. By convention databases always point to an associative structure, implemented within the language, known as a table.

In PS-algol a primitive transaction mechanism is provided called commit. Commit makes all changes made to the persistent store permanent. If a user operates against the persistent store and does not call commit no permanent changes are made to the store. Therefore not calling commit from within a program is equivalent to aborting a transaction. Commit is an atomic action - that is it either happens or it does not. Commit is defined as follows,

```
let commit = proc( -> pntr )
```

The pointer returned by commit is nil if the commit is successful or a pointer to a structure class, called error.record, containing error information. The class error.record is defined in PS-algol by writing down the following structure class definition.

```
structure error.record( string   error.fault;
                                 error.explain ;
                                 error.reason )
```

In practice, this is predefined by PS-algol's prelude code. If a commit fails the fields of this structure contain the reason for the failure. This includes trying to write to a database opened in read mode, databases being locked by other users and various system errors such as running out of disk space.

1.5 The Napier System

The persistent system is itself a large piece of software. The methodologies that apply to systems constructed within the persistent environment also apply to the construction of that environment. During the development of the PS-algol system boundaries between different parts of the system became blurred. This was mostly due to the fact that the system was a research vehicle and grew in an uncontrolled manner. Development and research using the PS-algol system eventually became difficult due to this.

Much of the Napier system is similar to the PS-algol system, however, the system has been re-engineered breaking it into individual modules. Each of the modules in the system presents a functional interface to other modules in the system and uses the functions presented at the functional interface of other modules.

For example, in the PS-algol system all the modules in the system knew about the structure of objects. The compiler needed to know about them in order to plant code, the interpreter to execute instructions, the garbage collector so it could reclaim space and the persistent object manager so that objects could be saved. This meant that in order to add a new data structure to the system all the modules had to be changed. This task was

aggravated by the inherent complexity of some modules, in particular, the persistent object manager.

If the observations of Boehm and Parnas are correct, the modularisation of the persistent system will result in reduced maintenance costs throughout the software life cycle. However, this re-engineering process was not performed purely as an exercise in software cost saving. The Napier system is an experimental one, it is not intended as a complete commercial implementation. The Napier system must therefore act as a research test bench on which various experiments may be performed. The modularisation of the system allows experiments to be carried out in language design, type systems, programming environments, abstract machine design, garbage collection, compiler design, optimisation techniques and object management to be performed concurrently.

The Napier system achieves this by providing a framework of plug compatible coherent subsystems. Each layer in the system is insulated from each other layer in the system by the functional interface it presents. Components may therefore be freely substituted for each other provided that the functional interface is met. The experimenter may therefore set up any mixture of the components that are available. Thus, not only facilitating experimentation in each of the fields individually but also allowing assessment of the interaction of different strategies. This architecture may be viewed diagrammatically below.

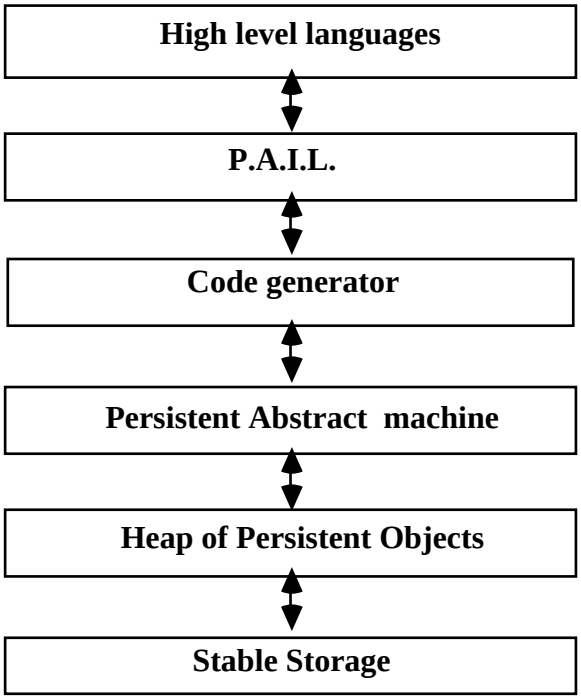


figure 1
layers in the Napier system

1.6 Persistent Information Space Architecture

The PISA architecture may be viewed as being divided into four domains [atk86b]:

1. The Store domain
2. The Language Domain
3. The System Building Domain
4. The Application Domain

The Store Domain is at the lowest level in the architecture. The storage architecture in PISA is stable, that is it is resilient to faults such as hardware failure and power loss. The store is object oriented in that it supports the storage of autonomous objects. Objects may reference other objects and in general the store will form a graph structure. It may be implemented on disparate computing engines, therefore the store domain implements a distributed stable object store.

The language domain provides the facilities to construct applications in the applications domain. The languages must support all programming activity including the control of the programming environment. The major research issues in persistent languages are: finding type systems rich enough to capture all uses of data, discovering binding mechanisms suitable for modelling adaptive long lived data and discovering ways of overcoming the complexity inherent in large systems.

The system building domain supports the construction of the persistent language environment. The tools currently available in this domain are:

1. Compiler Components
2. Support for compilation and execution merging
3. Support for abstract program graphs
4. Persistent Abstract Machine
5. Demand driven optimisation

The top level in the architecture is the applications domain. The applications domain provides support for the construction of application programs. This domain includes generic tools which operate over a range of different types and data. It also provides adaptive programs that may modify their behaviour to suit data that had not been constructed when the component was implemented. These adaptive components are considered to be of importance for large scale data manipulation.

1.7 Thesis Browsing

The thesis is broadly divided into three sections corresponding to the language domain, the system building domain and the application domain. Chapters two and three discuss the language domain, four, five, and six discuss the system building domain and chapter seven discusses an experiment in the application domain.

1.7.1 Language Domain

The Persistent Information Space Architecture (PISA) languages are strongly typed languages with a high degree of compile time type checking. They are also required to support their own environment. The requirements for such a class of languages are investigated in chapters 2 and 3.

The PISA languages provide machine independent raster and vector graphics. Chapter two describes the support for graphical operations in the PISA languages and discusses the data types provided and the operations that may be performed on them.

One of the aims of the PISA project was to provide better control of complexity in large systems, in particular, to support evolution of programs and data. One important aspect of this is the control of names and bindings within the system. Chapter three addresses these problems showing how they have been tackled in the past and proposing a new method of tackling the problem.

1.7.2 System Building Domain

Chapter four introduces the system building domain. This thesis concentrates on three areas within that domain they are: abstract machine design, intermediate code

requirements and compilation system architecture. These are discussed in Chapters four, five and six respectively.

Chapter four concentrates on the design of the Persistent Abstract Machine (PAM). The design decisions incorporated in the machine are outlined. Of particular importance is the modular design of the machine allowing concurrent experiments in several areas without major reconstruction. For example, the languages supported are independent of the machine, as is the storage architecture discussed in [bro88]. PAM supports languages with block level retention. This style of architecture is ideal for supporting object oriented languages.

Chapter five introduces the Persistent Architecture Intermediate Language (PAIL). The provision of such a language is an engineering decision in order to support many activities carried out within PISA. It is shown how PAIL may be used as a protection mechanism, as an optimisation tool and as a debugging aid.

The architecture is based upon the persistent languages. These languages are supported by compilers resident in the persistent information space and are therefore written in the persistent languages. The implementation of this reflexive system is discussed in chapter six. In it, compilers that may be called as functions from within the persistent environment are discussed.

1.7.3 Applications Domain

The applications domain comprises all the programs written within the persistent environment. For example, the compilation architecture discussed in the system building domain may be considered to be an application program. Another architecture that hovers between the application domain and the system building domain is the browsing architecture [dea88] which is discussed in chapter seven. The browser uses the compilation toolset to incrementally construct programs that operate against arbitrary data taken from infinite type space of the persistent languages.

1.8 Conclusions

The Napier system is still incomplete, the techniques discussed in this thesis provide the framework for the construction of a totally integrated environment. The lessons learned from building the PS-algol system are now being put to good use in this task. The final chapter provides a slightly speculative view of how these components may be combined to provide a fully integrated environment.

2 Environments

2.1 Introduction

The primary objective of the research of the Persistent Information Space Architecture (PISA) project is stated, in [atk86b], as:

> *the exploitation of the opportunity provided by the dramatic shift in the cost of hardware relative to software, the removal of the incoherence between the various programming mechanisms and thus, the provision of a better environment for exploiting new computer systems. Specific technical objectives thus include:*
>
> ***1.*** *Controlling complexity by establishing consistent rules which apply throughout the design and system, and, by introducing new concepts into the architecture only very parsimoniously, preferring those new concepts which encapsulate or abstract existing concepts.*
>
> ***2.*** *Introducing persistent data and separating the issue of what data structures are best for a program from the issues of identifying and preserving the data; thereby allowing most file and database data to be processed using the same language constructs as those used for a program's local data.*
>
> ***3.*** *Controlling system evolution even though the nature of data including program is that it's uses are neither parochial nor predictable. In particular, persistent systems of data and program are to be partially reconstructable and thus incrementally enhanceable.*

In this chapter the language construct, environment, written **env** will be introduced. Objects of type **env** are collections of bindings that have first class data rights. As such they provide the programmer with a mechanism to control bindings in the system. Such a mechanism provides a conceptually simple framework for manipulating bindings thus controlling complexity and system evolution from within a unified language framework.

2.2 Contextual naming

The persistent store is a conceptually unbounded space populated by objects. In such a space the naming of objects becomes a problem. This may be observed in programming systems which adopt a simple, flat object naming strategy such as Smalltalk-80 [gold83]. These systems have a single name space in which names may be introduced, resulting in the use of names being highly restricted by the names that have been previously used.

This problem may be overcome if contextual naming is used. In a contextual naming scheme names are introduced within some context. Names may be used many times within a system, one name denoting different things in different contexts. Contexts are used in everyday life to overcome naming problems, for example, when I say Ron has a beard to one of my colleagues they know by context to whom I am referring. The same sentence may mean something different or be meaningless to someone else. For example, to someone who knows a beardless person called Ron the sentence could be untrue. To a non-English speaking alien the sentence could contain no meaningful information at all.

Many different contextual naming strategies may be found in the computer systems of the present day. Some examples of these strategies are:

1. block structure in programming languages;
2. file directories in filing systems; and
3. segments in operating systems.

Usually these contextual naming schemes impose a tree structure on naming. For example, in a block structured programming language the programmer may write,

```
begin
        let a = 7
        write a
        begin
                let a = "hello"
                write a
                begin
                        let a = 1.23
                        write a
                end
        end
        begin
                let a = "hi"
                write a
        end
end
```

example 1
block structures contexts

This static piece of program may be represented by the following tree:

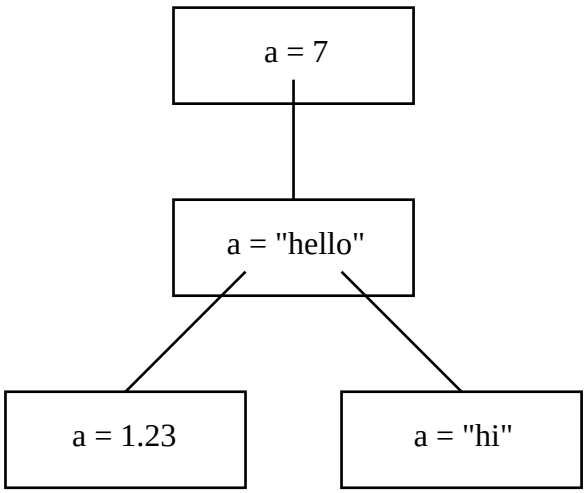


figure 2
A graph of
block structures contexts

Notice that each instance of the clause "write a", displays a different value due to the imposition of the context tree.

Clearly, this is a special case of the more general case, that of a graph. In general the persistent object space comprises a graph of objects. The environments which are discussed in this chapter provide extensible contextual naming on that graph. The understanding of binding mechanisms and their significance in the persistent environment is an important consideration in the provision of such contextual naming strategies. Binding mechanisms are discussed below.

2.3 Bindings

Bindings are comprised of a four-tuple [atk85a,mor86a] consisting of:

1. name
2. value
3. type
4. constancy

Thus the constant binding

let a = 7

may be written,

{a,7,int,true}

and the variable binding

let b := "hello mum"

may be written,

{b,"hello mum",string,false}

Bindings may be categorised by the following four properties,

1. whether the binding is to location or value;
2. when the binding is performed;
3. when is type checking performed (if at all); and
4. what scoping is performed.

These categories are examined below.

Bindings may be made either to locations or values. When a binding is made to a location, it is traditionally known as an L value binding [stra67]. In this kind of binding, the location does not change although different values may be stored in it. Sometimes, bindings are made to values which are immutable, this type of binding is known as an R value binding. Applicative languages such as SASL [tur79] only have this kind of binding.

Bindings may be instantiated statically by the compiler or dynamically by the run time system. If systems are bound statically many errors may be detected early (at compile time). Some languages designers consider this safety element so important that their languages only contain static binding.

However, in order for a dynamic system to evolve a measure of dynamic binding must exist in the system. If a program is entirely statically bound any change to the program or data requires the entire system to be recompiled to establish new bindings. This cost is prohibitively high for large systems. The system must, therefore, accommodate some dynamic binding in order to accommodate change.

There is a delicate balance between static binding for safety and dynamic binding to provide for evolution. Both methods of binding are necessary for large scale system construction and evolution. Consequently, the system must provide for both static and dynamic binding. In general, one would expect small objects to be statically bound and large objects to be bound dynamically.

Type checking, like the instantiation of bindings, may be performed statically by the compiler or dynamically by the run time system. Static type checking is generally performed for one of two reasons; firstly, as an optimisation, checks may be factored out and types are known, and more efficient code may be produced; secondly, and perhaps more importantly, as a safety measure. Early type checking detects many erroneous programs which may cause damage to a system. However, in order to accommodate change, systems must provide some dynamic typing, in particular, all projections from union types require a dynamic check.

2.4 Programming in the large

The size of applications that may be constructed using any methodology is limited by the size of programs we can debug and maintain. When any program reaches a certain size it is extremely difficult for anyone to understand it. At that point, if not before, the system must be broken down into separate components, the idea being that each of the components is of a manageable complexity. Hopefully, someone will be capable of understanding how these components may be fitted together in order to construct the required system. This task has become known as "programming in the large".

If programming systems are to be used to support the construction of large systems, they must provide modular construction facilities. Furthermore, these systems must also provide easily understood mechanisms for binding modules together. These mechanisms must be capable of accommodating change.

Software systems are constantly subject to pressures of change. Erroneous systems need to accommodate change because they require maintenance. Successful software products are used by people for tasks they were not originally intended to support. Users who like the basic functionality of a product often bring pressure to bear on designers to support new tasks which lie outwith the original product specification. Advances or change in hardware functionality also require change to software products. Often software is ported to a machine other than the one for which it was first written. It is imperative that software systems, especially large ones, support incremental change.

2.5 Static Contexts

The block structured tree shown in example 1 is static in nature. That is, the contextual structure is assembled by the compiler at compile time. Block structure hides information contained within a block from objects external to that block. Uses of an identifier are bound to the innermost textually enclosing definition of that identifier. This is known as static scoping and is in contrast to the dynamic scoping found in Lisp systems [mcc62].

The block structuring paradigm imposes visibility conditions on identifiers. In many languages, principally in the algol family, block structure and procedural abstraction are the only mechanisms provided for program structuring. This static imposition of structure is not sufficiently powerful to support the incremental construction of systems or incremental change.

The environments of Napier support programming in the large by providing incremental program construction mechanisms and contextual naming facilities. This problem is currently being tackled by other researchers. Some of the approaches taken, in the field of databases and programming languages are discussed below.

2.6 Language Approaches

In the specification language Clear [bur84a], Burstall and Goguen identify the three main operations on environments as:

1. create an empty environment;
2. extend an environment with a name value pair; and
3. find the value associated with a given name.

The operation of adding a new name value pair to an environment is extended to the addition of a binding quadruple by Atkinson and Morrison in [atk85a]. These operations may be illustrated by the block structure of the algols. The following examples are in Napier. A new environment is created in a block structured language by the introduction of a new block as follows,

```
begin
end
```

This is the empty environment containing no bindings. A binding may be introduced into the environment,

```
begin
        let a = 7
end
```

The environment now contains the single binding,

{a,7,int,true }

The value associated with a given name is found by looking up that name in the environment. In Napier, this is achieved by writing down the name of the identifier to yield the value. In this example, the programmer may write,

```
begin
        let a = 7
        a
end
```

which would yield 7 as the value of the block, the value being determined by the static scope of the block.

In a later language, Pebble [bur84b,bur84c], Burstall states that programming in the large will become merely typed functional (applicative) programming. In Pebble, bindings are treated as first class data objects, where a binding is defined as either a name bound to a value or a tuple of bindings. However, it is not obvious how applicative languages may help in the field of incremental system construction since by their very nature they must be statically bound. For this reason, there will be no further discussion of applicative languages.

2.6.1 Galileo

The language Galileo [alb85] recognises the need for control of names and values in a database context. In Galileo, a run time environment is defined to be a mapping from identifiers to denotable values. Such an environment is obtained by evaluating an environment expression. For example,

```
use a := 3 and b := 4 in a + b
```

yields the value 7. Here, the expression

```
a := 3 and b := 4
```

is an environment expression that yields an environment containing the bindings,

{a,3,num,true} and {b,4,num,true}

in which the expression a + b may be evaluated.

The above example introduces two of the environment operations provided by Galileo, namely the introduction of new bindings using ":=" and the combination of environments using **and**. Galileo provides other mechanisms that allow the programmer to select single bindings from environments, recursively introduce names and values and to remove names from environments.

Galileo provides persistence by having an environment called the global environment that always persists. The global environment may contain bindings including other environments. Galilieo is in an interactive system in which every expression entered by the user is evaluated with respect to an environment, initially the global one. The user may evaluate expressions with respect to another environment using the command **enter**. This command allows the user to traverse the tree of environments that may be found in the global environment. For example, in Galilieo the following dialogue may be carried out,

use anenv := (a := 3 **and** b := 4)
! this defines an environment called anenv in the global environment
enter anenv:
! now the current environment is anenv
a + b
! yields 7 as before

The designers of Galileo suggest that environments help the user to develop and test database schemata incrementally or to express the overall structure in terms of smaller related parts. They also suggest that they may be used as a modularisation mechanism in a manner similar to that suggested by Burstall and Goguen in Clear.

2.6.2 Name Spaces

In [atk85a] Atkinson and Morrison introduce name spaces. Name spaces are an environment mechanism that permit the following:

1. the storage of bindings in a name space;
2. the dynamic use of names from a name space;
3. the static use of names from a name space;
4. the evolution of names available in a name space; and
5. safe exchange of arbitrary data between parts of the system.

A name space is created by the following expression,

ns < identifier list> **from**
<sequence>
end from

This is best illustrated with an example,

let new = **ns** a,b **from**
    let a = 7
    let aa := a * a
    let b := **proc**(-> int) ; aa
end from

example 2
namespace instantiation

Here *new* has as its value the set of bindings (name space), in the notation used earlier:

{ { a,7,int,true },{ aa,49,int,false },{ b,(proc(-> int) ; aa),proc(-> int),false } }

Notice that unlike the environment expressions of Galileo, namespace is a first class data object. In order to accommodate change, namespaces provide a mechanism to add new bindings and to remove old bindings. A new binding may be added to a namespace using following construct:

extend <namespace expression> **with** <identifier list> **from**
<sequence>
end from

This is similar to the instantiation of a name space. Bindings may be dropped from name spaces by the use of the drop construct,

drop <identifier list> **from** <namespace expression>

In order to use a name space the user may write,

using <namespace expression> **with** <signature> **compile**
<sequence>
end compile

This notation is used to denote dynamic binding to a name space. The expression yields a value of type namespace. The type of the name space is checked dynamically to ensure that it satisfies the interface specified in the signature. If the type checking is successful, the sequence is evaluated in the new environment which is formed by enriching the static environment with the bindings in the namespace.

Thus, the signature specifies a formal store and the namespace expression provides an actual store each time the statement is executed. The dynamic binding is thus localised to the scope of the **using** .. **compile** construct.

Namespaces may also be used statically, the notation for this is,

with <namespace> **do**
<sequence>
end

Here the sequence is statically bound to the namespace. However, despite the static nature of the binding, the namespace must still be checked to ensure that it contains the bindings required of it. The reason for this is that the bindings may have been removed from the namespace using drop. Indeed, an error condition or exception may arise at run time due to a required binding not being present in the namespace. This is shown in the following example,

```
let new = ns a from                    ! define a namespace containing
        let a = 7                      ! one binding a~7
end from

let useNew = proc()                    ! define a procedure which
begin                                  ! uses new by binding to it
        with new do                    ! statically and writes out the
                write a                ! value of a
        end
end

drop a from new                        ! drop the binding from the
useNew()                               ! namespace, calling the
                                       ! procedure will cause an
                                       ! exception when a is checked.
```

example 3
using a namespace

This seems to contradict the idea that the binding is a static one. Careful analysis of the situation reveals that the problem is in the nature of the binding. The static bind is made to the namespace itself and not to the bindings contained in the namespace.

2.6.3 Program Editors

Namespaces influenced the work of Buhr and Zarnke [buh87]. They describe a programming system that allows the manipulation of bindings. However the environments in their system are not first class data objects and therefore may not be manipulated by the language. Instead, the user may change environments using a program editor which is essentially a symbol table browser. The system suffers from the same binding problems as those encountered with namespaces. Datestamps are used to maintain integrity and preserve static type checking. When datestamps are found to be in an incorrect time order the compiler is called to automatically correct the situation.

2.7 Environments

The programming language Napier introduces the concept of an environment in order to provide support for the control of names and to allow incremental system construction. This is achieved by providing an extensible mechanism that permits the storage of bindings. The environments provided in Napier satisfy the three main operations defined in Clear with the addition of one new operation. The operations on Napier environments are:

1. create an empty environment;
2. extend an environment with a name value pair;
3. find the value associated with a given name; and
4. remove a binding from an environment.

Environments, written env, are introduced using a predefined function called *environment*. It has the following form,

```
let environment = proc( -> env )
```

The function *environment* returns a new empty environment. That is an environment containing no bindings. In Napier, bindings are always introduced with the word *let*. When bindings are declared within a block the programmer may write something like,

```
begin
        let a = 7
        let b = proc( -> int )  ; a
end
```

example 4
block structure

Similarly, bindings are introduced to environments using the word *let*. The user must also specify the environment in which the binding is to be made. The syntax of binding introduction is therefore,

in <environment-clause> <declaration>

The environment in the environment clause may be statically or dynamically determined. The programmer may therefore write,

```
let e = environment()
in e let a = 7
```

example 5
static use of environments

The environment *e* contains one binding,

{ a,7,int,true }

In this example, the first let declaration introduces the name *e* into the static environment of the program. In the second line the environment to which *e* is bound is enriched with the new binding,

{ a,7,int,true }

The static environment of the program remains unchanged. An exception is generated if a name is added that has already been used to identify another binding in the environment. The **in .. let** construct therefore serves the same purpose as the **extend** and the name space instantiation of namespaces. The example shown in example 2 above would be written in Napier as,

```
let new = environment()
in new let a = 7
use new as a : int in
in new let aa := a * a
```

example 6
instantiation of environments

Bindings in environments may be manipulated with the **use** clause. It has the following syntax,

use <environment clause> **as** <signature> **in** <clause>

for example to use the environment defined in example 6 and write out the values associated with a and aa the programmer would write,

```
use new as a,aa : int in
begin
        writeint( a )
        writeint( aa )
end
```

example 7
using values stored in environments

The signature need only specify a partial match on the bindings stored in the environment. The environment may therefore contain bindings other than the ones specified but must contain at least the bindings denoted in the signature. If any of the bindings are not present an exception is raised.

Values may be exported from an environment by returning a value from the clause bound to the use statement. For example, if the programmer wished to extract the value associated with a value from the environment the following could be written,

```
let valueOfa = use new as a : int in a
```

Bindings may be removed from an environment using the drop construct. This has both the same same syntax and semantics as namespaces, namely,

drop <identifier> **from** < environment clause>

This expression removes the binding containing the identifier from the environment specified in the environment clause. Note that the binding is not deleted, merely removed from the environment. This distinction is important as will shown below. Another example, clarifies this,

```
let new = environment()
! new is an empty environment
in new let a = 7
! new now contains the binding a ~ 7
use new with a : int in
in new let aa := a * a
! new now contains the bindings a ~ 7 and aa ~ 47
drop a from new
! new now contains only the binding aa ~ 49
```

example 8
dropping values from environments

2.7.1 Type checking

All environments are of type **env**, this is regardless of what is stored in them. This is in sharp contrast to the structure type of Napier. Structures are type checked using structural equivalence. In order to pass them as parameters the user must specify the names and types of the fields of the structure. Thus, the programmer is provided with a choice of whether to model using environments which allow flexibility but delay type checking or structures which are statically strongly type checked.

The type environment is the infinite union of all labelled cross products. The **use** statement projects bindings out of the infinite union. The flexible binding mechanism provided by environments in no way weakens the type system. The program is still strongly typed, however, it is no longer statically typed. Furthermore, the programmer must specify the types associated with the bindings that are to be used in an environment.

This specification allows the segment of code within the use clause to be statically type checked with respect to the projection.

The provision of environments and the other infinite union type provided in Napier, **any**, allow the programmer to choose to delay some type checking until execution time. Such a mechanism is extremely important in an otherwise strongly typed persistent environment. If a point of dynamic type checking is not provided in a statically typed persistent environment, the user would have to specify the type of the entire persistent store every time he or she wanted to interact with it. Furthermore, the type of the store is constantly changing as users add or remove objects of different types from it. The types **env** and **any** allow the user to partially specify the type of the store.

It is expected that programmers will statically bind data structures used within "programs" and use the environments to structure the information space in the manner that files and directories are used to structure the stores provided in today's operating systems.

The code within the use clause is statically bound to the bindings projected from the environment. The following occurs when a use is executed:

1. the bindings are looked up in the environment;

2. the type of the bindings are checked against the signature;

3. if either the types do not match or the bindings are not present an exception is raised;

4. if an exception is not raised the bindings are introduced into the environment - this constitutes dynamic binding; and

5. the clause associated with the use is executed with the bindings already instantiated in the environment. All further uses of the projected values or locations are statically bound.

Notice that since the projection is from an infinite union, it is always necessary to specify the types of the bindings that are to be used. The use of a unification algorithm, such as the one used in ML [har86], will not help here since we must specify all the necessary type information. This is necessary if strong static type checking is to be retained everywhere apart from at the time of projection. The mechanism provides the maximum amount of static type checking whilst retaining the flexibility required for incremental system evolution.

The need to specify potentially large amounts of type information in order to use an environment is worrying. Although not visible to the user, type information must be encoded into the environments so that type checking may be performed at the time of projection. This information is stored in the implementation of the environment, this allows a reversal from traditional type checking to be made.

In traditional type checking systems, the user writes down a program associated with type definitions. The program is then submitted to the compiler which tells the user whether the program is correct or not. In the system described above, the user may traverse the information space using a browser, similar to the one described in chapter 7. This browser may report the types of the objects stored in the environments. If it were used in conjunction with a callable compiler, like the one described in chapter 6, the user could interactively construct programs to operate against data held in the information space. In such an environment the distinction between browsing and compiling becomes blurred, since different tools in the support system are being combined transparently to provide a high degree of programmer support.

2.7.2 Binding to environments

Recall that in example 3, a difficulty arose concerning the use of **using** and **drop** with namespaces. The problem with namespaces was that the binding was always to the namespace and not to the bindings stored in the namespace. The example shown below is semantically equivalent to example 3. As in example 3, it will cause an exception to occur on the last line.

```
let new = environment()                 ! define an environment containing
in new let a = 7                        ! one binding a~7

let useNew =
proc() ;      use new as a : int in     ! define a procedure which writes
        write a                         ! out the value of a

drop a from new                         ! drop the binding from the
useNew()                                ! namespace. calling the
                                        ! procedure will cause an
                                        ! exception when a is checked.
```

example 9
example 3 revisited

However, using environments this example may be rewritten as,

```
let new = environment()                 ! define an environment containing
in new let a = 7                        ! one binding a~7

let useNew =                            ! define a procedure
use new as a : int in                   ! writes out the
        proc() ; write a                ! value of a

drop a from new                         ! drop the binding from the namespace
useNew()                                ! calling the procedure will cause
                                        ! 7 to be written out.
```

example 10
example 3 with desired semantics

This example will write out the value 7 rather than raising an exception. The difference between the two examples is in the following lines,

```
let useNew =
proc() ;        use new as a : int in
            write a
```

and,

```
let useNew =
use new as a : int in
        proc() ; write a
```

In example 10, the use clause is within the procedure body. This means that every time the procedure is called, the use clause is executed. It then dynamically performs type checking and checks to ensure that the desired binding is in the environment. When the procedure is called the binding is no longer in the environment and an exception will be raised.

In the second example, the projection out of the environment to yield the binding is performed only once - before the procedure closure is formed. The binding, {a,7,int,true} yielded by projection from the environment is then bound into the closure of the procedure. The value (the binding itself) is never again looked up in the environment, so the drop operation has no effect on the procedure. It will be shown in the next section that environments exhibit the same semantics as block structure.

2.7.3 Simulation of scope

The semantics of composition of environments is equivalent to the more familiar block structure in programming languages, for example in a block structured programming language such as PS-algol the programmer may write,

```
let a := 7
begin
        let a := 6
        write a                         ! writes out 6
        a := 4
end
write a                                 ! writes out 7
a := 32
write a                                 ! writes out 32
```

example 11
scope and block structure in algol

Similarly, in Napier the programmer may write,

```
let env1 = environment()
in env1 let a := 7
let env2 = environment()
in env2 let a := 6

use env1 as a : int in
begin
        use env2 as a : int in
        begin
            writei( a )                 ! writes out 6
            a := 4
            writei( a )                 ! writes out 4
        end
        writei( a )                     ! writes out 7
        a := 32
        writei( a )                     ! writes out 32
end
```

example 12
scope and block structure using environments

This use of environments in this way will be familiar to programmers who have programmed in block structured programming languages. It is no accident that environments should exhibit the same semantics as block structure, it is a consequence of the language design principle of only introducing a few powerful concepts.

The binding mechanism used in environments is also the same as that used in the block structure of Napier. In Napier, variable binding is by L-value and constant values are by R-value. The bindings stored in an environment exhibit the same behaviour - all variable bindings are to locations and all constant bindings are to values.

2.7.4 Binding to the persistent store

The root of persistence in Napier, called *ps*, is of type **env**. Any data that is reachable from *ps* is persistent. Making any data structure persistent is then simply a matter of binding that data structure to something reachable from *ps*. For example, suppose that in a program a binary tree, for simplicity over integers, has been defined. An instance of such a tree is then to be made persistent. This may be performed as follows,

```
rec type Tree is variant(     tip : null ;
                              node : structure( val : int ;left,right : Tree ) )

let twig = Tree(tip : nil )
let atree = Tree( node : structure(  val = 7,
                                     left = twig,
                                     right = twig ) )

in ps let savetree = atree
```

example 13
binding to the persistent store
using environments

The final line of this program binds data structure, bound to *atree* in the local environment, to *savetree* in the persistent environment *ps*. In order to use this data structure in another program the user may write,

```
rec type Tree is variant(     tip : null ;
                              node : structure( val : int ;left,right : Tree ) )

use ps as savetree : Tree in
if savetree is node then
begin
        writes( "it as a node with value : " )
        writei( savetree'node( val ) )
end
else writes( "it was a tip" )
```

example 14
binding to the persistent store
using environments

Notice how the type definition of *Tree* serves to unify the types across the persistent store and allows type checking to be performed statically and separately in each of the programs. The check that the type in the persistent store is the expected one is performed in the use statement.

In general, the persistent store will form a graph consisting of environments and data bound to those environments. Such a graph may be viewed as,

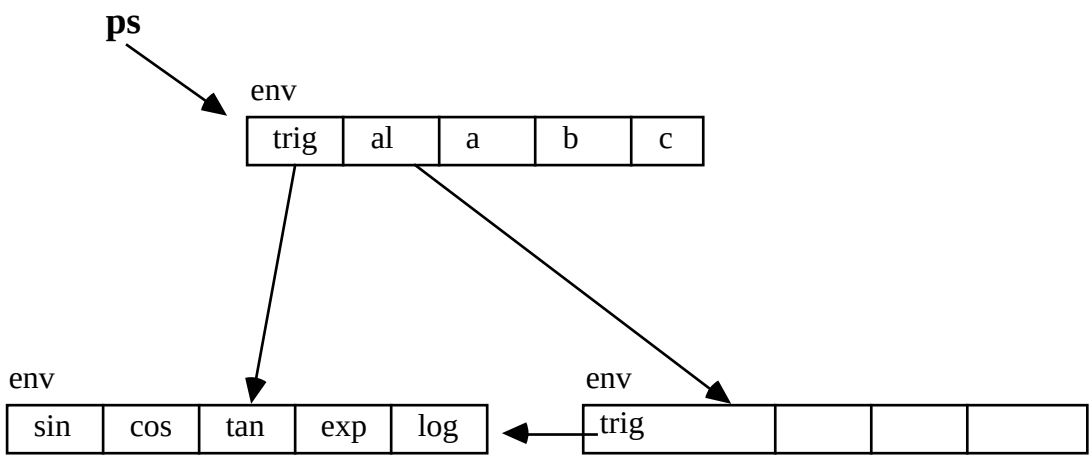


figure 2
the persistent store
as a graph of environments

The graph of environments subsumes the function of traditional file storage replacing it with a strongly typed data structuring mechanism. This mechanism unlike files, may be used to store structured data of arbitrary complexity. This ability is stated by Balzer in [bal86] as being one of the most important features required of new generation operating systems.

2.7.5 Supporting incremental construction

The use of environments to support incremental system construction has been strongly stated in this chapter. The way in which this is achieved is examined below. Suppose that the persistent store is arranged at some time in the manner shown graphically in figure 2. The user may then carry out the following dialogue through an interactive compiler.

```
! only ps is in scope at the beginning of the session
! first introduce the environment trig into the local environment

let trig = use ps as trig: env in trig

! next declare square in the local environment
! square uses exp from the environment trig.

let square = use trig as exp:proc( int,int -> int ) in
        proc( a : int -> int ) ; exp( a,2 )

writei( square( 7 ) )   ! test out square

49
! system writes out 49 - satisfied so save it in environment

use ps as al : env in
in al let square := square
```

example 15
incremental construction of
a program

After the completion of the dialogue the persistent store will be arranged as follows,

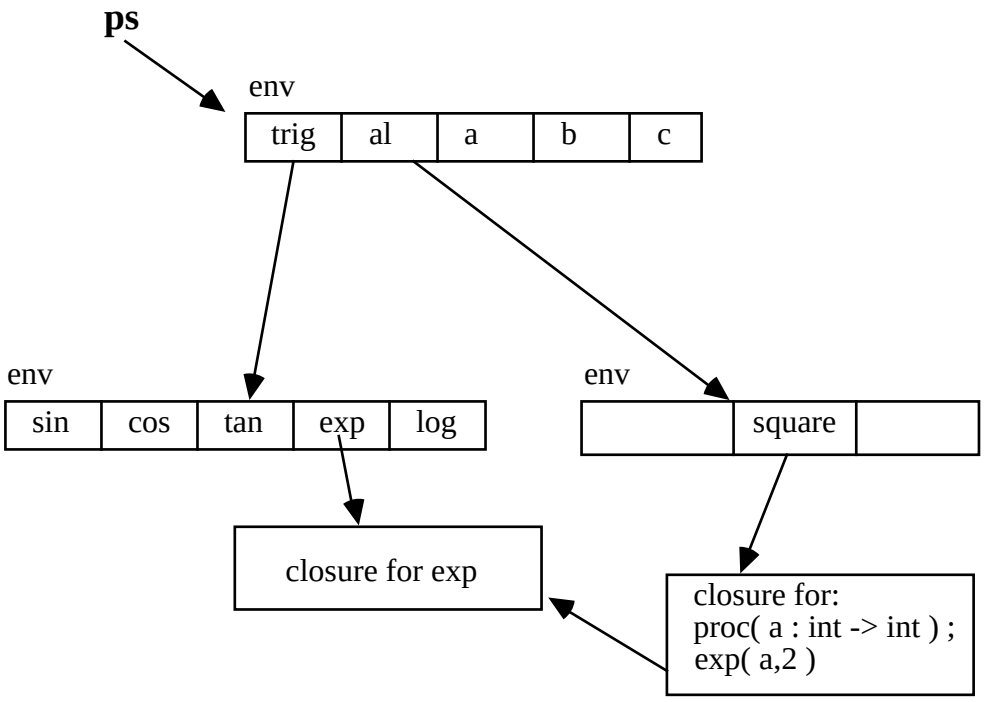


figure 3
persistent store
after interaction

Notice that the procedure called *square* has the location of the procedure *exp* bound into its closure. Thus, if the programmer assigned another value to the location *exp* the function square would also change. Sometimes this behaviour is undesirable and the programmer may wish to ensure that future changes to the system cannot affect the program he or she has constructed. In such a case, the programmer would project out of

the environment to yield a value rather than a location. This would allow a static R-value binding to be made. In such a case example 15 could be rewritten as,

```
let trig = use ps as trig: env in trig
let exp = use trig as exp:proc( int,int -> int ) in exp

let square = proc( a : int -> int ) ; exp( a,2 )

use ps as al : env in
in al let square := square
```

example 16
incremental construction of
a program

Here the value stored in the location associated with *exp* is first projected out of the environment and is then bound into the closure of the procedure square. If R-value bindings are used in this way, the procedure closure cannot be affected by changes to the environment. This style of binding is therefore safer than the L-value binding shown earlier but the hidden cost is that it cannot be maintained incrementally and rebinding is necessary to accommodate change.

The store shown in example 3 not only exhibits graph structure in terms of data structures and environments but also in code. Procedures in a persistent environment, such as the one described here, also form a graph structure. One procedure may be bound to many programs. This kind of architecture allows for a higher degree of software reuse [mor87a] than conventional software architectures.

Example 15 shows that by a short interaction with the system, new definitions may be incrementally added to it. Similarly, if the programmer wished to change a definition this could be achieved by assigning to a location within an environment. For example the programmer may wish to change the implementation of square, defined in example 15. This may be achieved by the following interaction,

```
use ps as al : env in
        use al as square : proc( int -> int ) in
                square := proc( a : int -> int ) ; a * a
```

example 17
incremental changing of
a program

2.7.6 An Implementation of Environments

The Persistent Abstract Machine does not support the extension and reduction in the size of an instance of any data structure. Consequently, environments are not directly supported by the machine. Instead, environments are constructed within the Napier system. The implementation of environments is, of course, hidden to the user by the compiler.

Environments are an infinite union of labelled cross products. The machine provides support for another union - the data type **any**. A value of any type may be injected into the type **any** which results in an object of type **any**. For example, the user may write the following,

```
let anint = 4                          ! of type int
let anany := any( anint )              ! of type any
anany := any( "a string" )             ! of type any
anany := any( proc( -> int ) ; 3 )     ! of type any
```

A value of type **any** may be projected from to yield a value of the type that was originally injected into it. For example, if the user had declared *anany* in the example above, the following expression would project the value out of the union,

```
project anany onto
int             : writes( "it was an integer" )
string          : writes( "it was a string" )
proc( -> int )  : writes( "it was a proc( -> int )" )
default         : writes( "it was something else" )
```

Details of the implementation of the type **any** may be found in the Persistent Abstract Machine Manual which appears as Appendix 1. Notice that the type rules of the language may not be broken, but, like the projection out of environments, type checking must be delayed until run time.

The data type **any** may be used to provide an implementation of environments. This is achieved by implementing a mapping from strings to the type **any**. The way in which this mapping is implemented is unimportant, for example, it may be implemented by a balanced binary tree, by a vector or by hash tables. The most efficient implementation will depend on how users utilise the environment facility, in particular, with respect to the number of bindings in an environment. This has yet to be measured. The implementation must provide (at least) the following functional interface,

```
insert   : proc( string, any )
remove   : proc( string )
lookup   : proc( string -> any )
```

An implementation of such a system is (almost) already provided within the compilation environment. The symbol tables supported by PAIL (see chapter 5) are defined as follows,

```
rec type symbolTable is variant(  Empty : null ;
                                  Table : symTab )
&
SymTab[ t ] is structure(         lookupLocal( string -> t )
                                  lookupRec( string -> t )
                                  InsertEntry( string,t )
                                  EnclosingScope( -> symbolTable )
                                  EnterScope( symbolTable- > symbolTable )
                                  ScanScope( proc( t ) )
```

This interface may be parameterised by the type **any** and extended with the remove operation to provide the necessary support.

Whenever the user creates a new binding in an environment the value is injected into the type **any**. This value may be associated with the appropriate name represented as a string using the procedure *insert* .

The *remove* procedure merely removes the string from the index associated with the implementation. The location in which the value is stored may be reclaimed by the garbage collector only if it is rendered inaccessible by the *remove* operation.

When a **use** clause is executed two different operations take place. Firstly, the name specified in the **use** clause must be looked up in the implementation of the environment. If the name is not present an exception will be raised. If this is successful the value associated with that name will be returned by the lookup function wrapped in an **any**.

In order to use a binding in an environment, the user must specify the type of the value that is being projected. This type may be used to project the value from the **any** in which it is stored. If this projection is unsuccessful an exception will be raised, if not the value may be placed on the stack.

2.8 Conclusions

At the beginning of this chapter, the control of complexity was stated to be one of the aims of the PISA project. The difficulty in the control of names and the need for controlled system evolution was cited as a particular area of difficulty.

The introduction of context was demonstrated to be one method of managing complexity. In particular, block structure in programming languages was a typical method of controlling the use of names. This technique, and others used to control complexity have been explored.

The data type environment has been shown to be a flexible method of controlling system evolution by allowing large systems to be incrementally developed in an interactive environment. The parallels between block structure and environments have been shown. This is in line with the principle of parsimoniously introducing new concepts and only introducing concepts that encapsulate or abstract existing concepts. Finally, a possible implementation of environments was suggested and shown how it could operate.

3 Graphics

3.1 Introduction

A total persistent environment is required to support all programming activity. The advantages of such an environment is that it relieves the programmer from the burden of managing the mapping of data from one environment to another. One of the potentially most expensive activities in this area is in the field of man machine interaction (mmi). The programming of user interfaces is an inherently complex task. This complexity is often greatly magnified by having an alien toolset with which to program mmi.

Packages of library functions such as those provided by Ghost [cal77], Suntools [sun86] and GKS [gks82] often do not interface smoothly with the programming language being used to program an application. This creates a situation in which the programmer is manipulating two languages, the application language and the graphics sub system language. This increases the complexity of the task presented to the programmer. Complexity is also increased if the graphics objects may not be stored in the same manner as the objects in the programming language.

Often, graphics systems such as those mentioned above are imperative in nature. That is, it is only possible to express commands such as,

> do this, then this, then this

which may draw a picture. If the picture has to be stored, the programmer must create a data structure representing the picture and then traverse the structure calling the appropriate graphics language commands during the traversal. Many applications are required to run on more than one machine, say an Apple Macintosh and Sun workstation. In this case, the programmer must repeat this complex task - mapping one onto the Macintosh graphics toolbox [mac86] and once onto the Sun library, Suntools. This is made more complex by the number of different pieces of hardware available today - all offering different facilities.

If the picture is to be saved on backing store, as is often required in CAD packages, a further representation of it will be required. This is the long term data structure and will typically either be a byte or record stream in a file system or some relational structure in a database management system.

Libraries of functions provided by graphics sub systems are often ad-hoc in nature. Some libraries provide output devices such as windows, others have more complex features such as the canvases of Suntools. The facilities provided also vary, ranging from simple line drawings to the esoteric functions such as polyline facility of GKS. Some systems allow the creation of picture libraries. However, if these libraries of pictures are expressed in a different language to the application language, it is difficult for the programmer to manipulate these pictures.

Clearly, what is required is a language for manipulating graphics entities which is integrated with the application programming language. It must contain simple building blocks that provide the ability to build abstraction level upon abstraction level in order to provide the complex user interactions required. It must also be machine independent and provide the same power for manipulating graphics objects as the other data types. In other words, graphical objects must have the same civil rights as other data types - that is the graphical data types must be able to be stored, passed as parameters and have full rights to persistence.

The graphics facilities of PS-algol were designed on these grounds. It provides orthogonal persistence and two graphics data types one for manipulating line drawings, the other for

manipulating raster graphics. Using these basic building blocks complex event driven systems [cut86] and graphics database systems with inheritance [ben86] have been constructed.

3.2 Pictures

The picture drawing facilities in PS-algol are a particular implementation of the Outline system [mor82b] which allows line drawing in an effectively infinite two dimensional real space. The Outline system was originally designed and implemented by Professor Ron Morrison of St Andrews University as an extension to the language S-algol [mor82a]. Outline is itself derived from GPL/1[smi71]. Some of the original outline facilities provided in this system have since been abandoned - but the spirit of the original system lives on. The integration of Outline into a persistent language provides the programmer with more power than was available in the original. Altering the relationship between different parts of a picture is performed by mathematical transformations which means that pictures are usually constructed from a number of sub-pictures.

In the Outline system, picture description and picture drawing are separated. Picture description is supported by the programming language and picture drawing by mapping the picture to an image or output device. In this manner, pictures are described in a device independent manner.

In PS-algol, the picture descriptions are represented by the data type picture, written in language as **pic**. 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 return a picture as their result. Therefore, arbitrarily complex pictures may be described by the application of the operations described below.

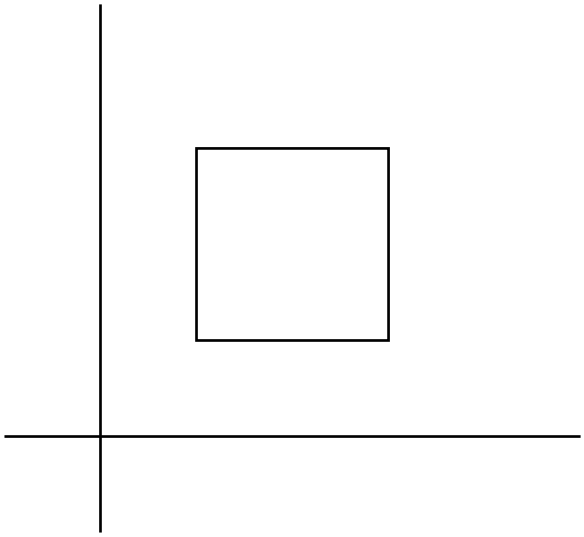
There are two binary operators on pictures, join '^' and combine '&'. The effect of the join operator is to yield a picture that is made up of its two operands with a line from the last point of the first operand to the first point of the second operand. Combine operates in a similar way without adding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

let *box* = [1,1] ^ [2,1] ^ [2,2] ^ [1,2] ^ [1,1]

will produce a unit square with its bottom left hand corner positioned at the point [1,1]. This is illustrated in figure 1.

Points in pictures are implicitly ordered. Join and combine operate on the last point of the first picture and the first point of the second picture. In the resulting picture, the first point of the first picture is the first point and the last point of the second picture is the last point.

(note that the axes are put in for reference and are not part of the picture)



the box
figure 1

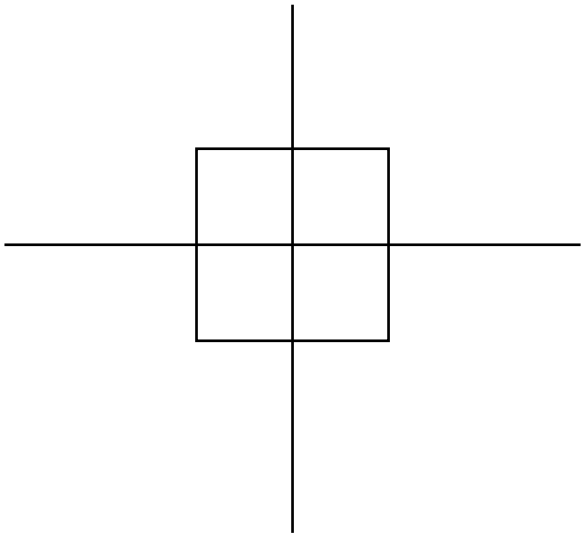
In addition to the binary operators, pictures may also be transformed by shifting, rotating and scaling.

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*. For example,

let *new* = **shift** *box* **by** -1.5,-1.5

Will initialise the value of the identifier *new* to be the picture shown in figure 2.



the shifted box
figure 2

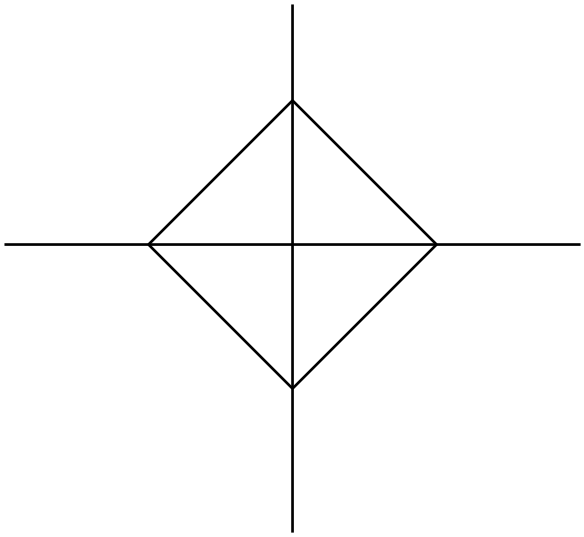
Rotation can be achieved by

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

which will produce a new picture by rotating the picture *p no.of.degrees* degrees clockwise about the origin For example,

rotate *new* **by** 45

will produce the picture given in figure 3



the rotated box
figure 3

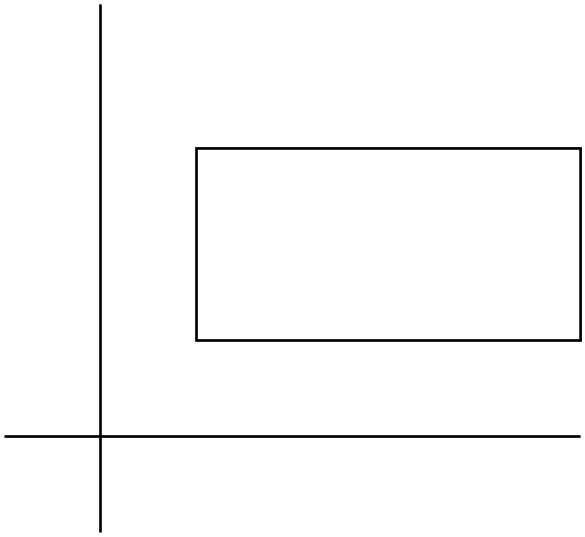
Finally scaling can be obtained by

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

which 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. For example,

scale *box* **by** 2,1

yields figure 4

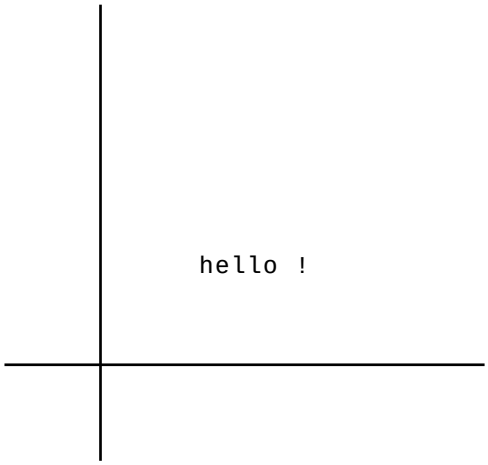


the scaled box
figure 4

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. For example,

text "hello !" **from** 1,1 **to** 2,1

yields figure 5.



some text
figure 5

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

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

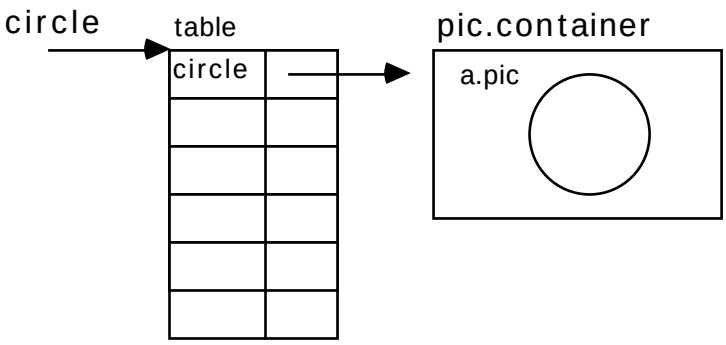
3.2.1 Storing a Picture in a Database

To show how pictures may be stored and retrieved from the persistent store an example is given of a program to calculate the unit circle at the origin and store it in the database. In this example, it is assumed that the database root is a pointer to a data structure for associative storage and retrieval, supported by PS-algol, called a table. Entries are placed in the table using the procedure s.enter which takes the associative key, the table, and the value to be stored. The procedure s.lookup retrieves a value from the given table using the given key.

```
structure pic.container( pic a.pic )          ! used to store the picture

let db = open.database( "a pic","pass","write" )
if db is error.record do
begin                                 ! if db points to an error.record the open failed
        write "Unable to open database because: ",
        db( error.explain ),'"n"
        abort
end

let circle =                          ! this block yields a picture describing a unit circle
begin
        let no.of.sectors = 10
        let angle = 90 / no.of.sectors
        let quadrant := [0,1]
        let segment := [0,1] ^ rotate [0,1] by angle
        for i = 1 to no.of.sectors do
        begin
                quadrant := quadrant & segment
                segment := rotate segment by angle
        end
        let semi = quadrant & scale quadrant by -1,1
        ! below is the value of this block expression
        semi & scale semi by 1,-1
end

! a structure containing the circle picture is
! associated with the key "circle"
s.enter( "circle",db,pic.container( circle ) )

! the database "a pic" is now updated
if commit() = nil do write "Circle entered in the data base'n"
```

A program to store a picture of a unit circle in a database.
example 1

The database called "a pic" now contains a table with a key "circle" which has an associated value of a structure that contains the description of the circle picture. This is shown pictorially below.



Pictorial representation of the database "a pic" after the transaction is committed
figure 6

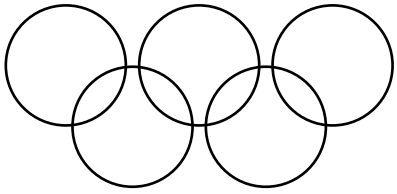
3.2.2 Retrieving a Picture From a Database

The next example retrieves the picture description from the database and uses it to define another picture which is the Olympic Games logo.

```
! this structure will be used to hold pictures kept in this database

structure pic.container( pic a.pic )

let db = open.database( "a pic","pass","read" )
if db is error.record do
begin
        write "Unable to open database because: ",
        db( error.explain ),'"n"
        abort
end

let circle = s.lookup( "circle",db )( a.pic )

let olympics = circle &
        shift circle by 2.2,0 &
        shift circle by -2.2,0 &
        shift circle by  1.1,-1 &
        shift circle by -1.1,-1 &
        text "OLYMPICS" from -1.5,-2 to 1.5,-2
```

A program to retrieve the circle from the database and define an Olympic Games logo.
example 2

The picture olympics now contains the following,



The Olympic games logo
figure 7

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.

3.3 Raster Graphics

In addition to the Outline line drawing system PS-algol also supports raster graphics. The facilities described in this section were designed and implemented by Professor Ron Morrsion, Fred Brown and myself in consultation with Professor Malcolm Atkinson [mor86b]. Two data types are provided for this purpose, a base type pixel and a type image constructed from pixels.

3.3.1 Pixels

Two pixel literals **on** and **off** are provided by the system. In their 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 pixels of greater depth, pixels may be concatenated using the operator '&'. To create a pixel of depth 4, called *b*, the user could write,

let b = **on** & **off** & **off** & **on**

which creates a pixel b with depth 4. Arbitrary pixels expressions are permitted, therefore expressions such as

b & **on** & **a**

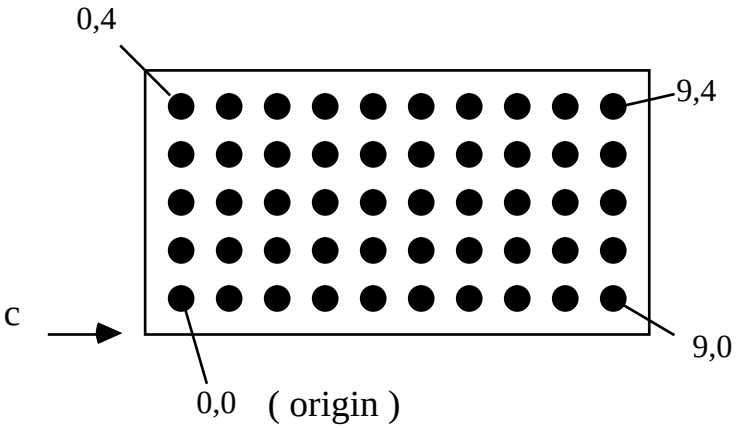
are permissible, and would yield a pixel of depth nine.

3.3.2 Images

Images are rectangular objects comprising of pixels with an X and Y dimension to reflect their size. Images also have depth which determines the number of planes they have. The depth of an image is determined by the depth of the pixel used in the initialising expression. To form an image the user could write,

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

which creates c with 10 pixels in the X direction and 5 in the Y direction all initially **on**. All images, have an origin at the bottom left hand corner which has coordinates 0,0. The image c is represented pictorially below.



an image
figure 8

Full 3 dimensional images may be formed by expressions like the following,

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

which would create an image d of depth 4 with 64 pixels in the X direction and 32 in the Y direction all initialised to the pixel value on & off & on & off. This is illustrated in figure 8.

An image with 4 planes
figure 9

In order to introduce the concept of and operations on images gently, the following discussion will be restricted to images with a pixel depth of 1. Everything that is stated is true for images of greater depth.

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

let b = a

will assign the image a to the identifier 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. In other words images exhibit pointer semantics.

Three standard functions are provided to interrogate an image to discover its size; they are X.dim, Y.dim and depth each being of the following type,

proc(**image** -> **int**)

These functions return the *x* dimension, the *y* dimension and the depth of the image respectively.

3.3.3 Raster-op

PS-algol supports 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

The semantics of these operations may be expressed by representing **on** and **off** as **true** and **false** respectively and the following set of rules for combining pixels. The source, the first operand is represented by S, the destination, the second operand is represented by D. The symbol ~ represents logical negation.

ror	D := S **or** D
rand	D := S **and** D
xor	D := (S **and** ~ D) **or** (~S **and** D)
copy	D := S
nand	D := ~ (S **and** D)
nor	D := ~ (S **or** D)
not	D := ~ S
xnor	D := (S **or** ~ D) **and** (~S **or** D)

Thus,

xor b **onto** a

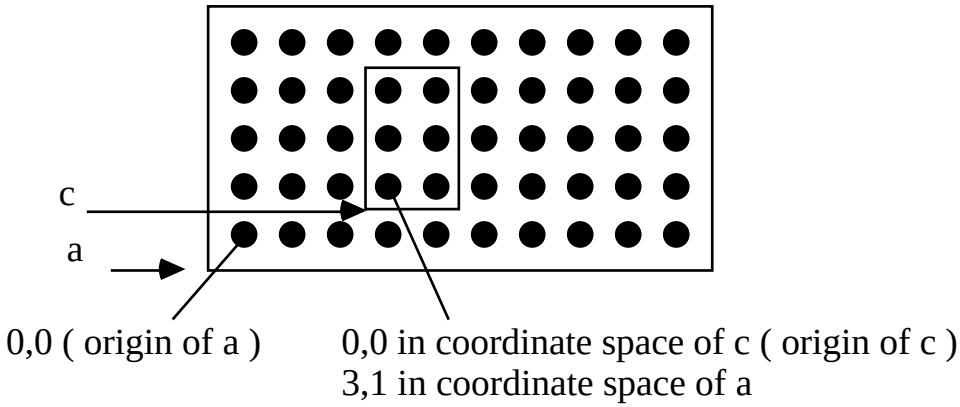
performs a raster operation of bitwise combining b and a using the xor rule above and assigning the result to a. It should be noted that a is altered in situ as would be expected on a raster device. Both images have origin 0,0 and automatic clipping at the extremities of the destination image is performed.

3.3.4 Aliasing

The limit operation allows the user to set up windows in images. For example,

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

sets c to be that part of a which starts at 3,1 and has size 2 by 3. c has an origin of 0,0 in itself and is therefore a window on a. This is illustrated in figure 10 below,



A limited region of an image
figure 10

Once a limit been taken, the resulting image is indistinguishable from any other in the system. It may therefore be passed as a parameter or stored. However, when operations such as raster-op are performed on that image changes will also be propagated to the image from which the image was limited.

Rastering sections of images onto sections of other images can be performed by expressions like the following,

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.Limited regions of limited regions may also be defined

A vertical slice of pixels from an image may also be extracted. This operations is semantically equivalent to the limit operation but in the third dimension. That is like limit it yields an alias to part of an image not a new one. For example,

let b = a(1|2)

yields b which is that part of a which has the two depth planes 1 and 2. The depth origin, like the (*x*,*y*) origin in images is normalised to zero.

3.3.5 Colour Mapping

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**)

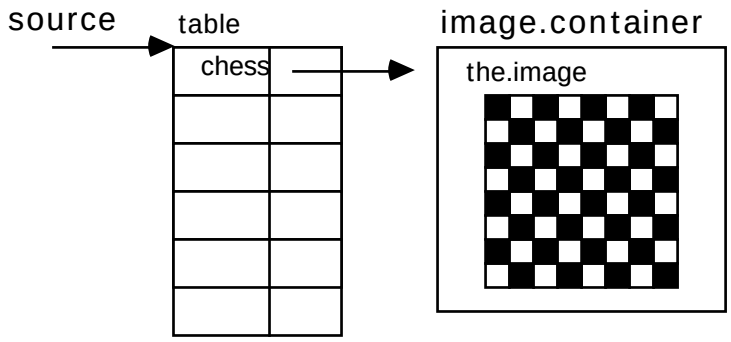
This function returns the integer corresponding to the pixel p in the colour map.

An example of the use of images may be seen from the program to draw a chess board and store it in a database in example 3.

```
!This structure will be used to hold images in this database

structure image.container( cimage the.image )

write "Please input the basic size of the squares "

let size = readi()

let black = off ; let white = on

!define a black square
let black.square = image size by size of black

let size.8 = size * 8 ; let size.2 = size * 2

!define the chess board image
let chess.board = image size.8 by size.8 of white
for i = 0 to size.8 - 1 by size do
for j = 0 to size.8 - 1 by size do
        if i rem size.2 = 0 and j rem size.2 = 0 or
        i rem size.2 ~= 0 and j rem size.2 ~= 0 do
                copy black.square onto limit chess.board at i,j

let source = open.database( "raster.demo","friend","write" )
if source is error.record do
begin
        write "Error opening database : ",source( error.fault ),'"nbye'n"
        abort
end

!a structure containing the chess board image is associated
!with the key "chess"
s.enter( "chess",source,image.container( chess.board ) )

let done = commit()
if done is error.record do
        write "Sorry - commit failed: ",done( error.fault ),'"n"
```

A program to store a chess board image in a database
example 3

The pictorial representation of the database after the transaction has committed is given in figure 11.



Pictorial representation of the chessboard in the database
figure 11

Images may be stored in and retrieved from databases in the same manner as pictures and thus provide the same facilities for providing libraries of images or procedures that manipulate images.

3.3.6 Mapping Pictures and Images to Output Devices

The standard identifier *screen* is an image representing the output screen. Performing a raster operation onto the image screen alters what is viewed by the user. For example,

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 also bound to an image which is mapped to the cursor. When the user moves the mouse or pointing device this image moves accordingly. The cursor may be altered in the same manner as any other image. For example, we may say,

copy b **onto** cursor

The resulting change in the cursor will be visible to the user.

Line drawings may be mapped onto an image using the standard function draw. For example,

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

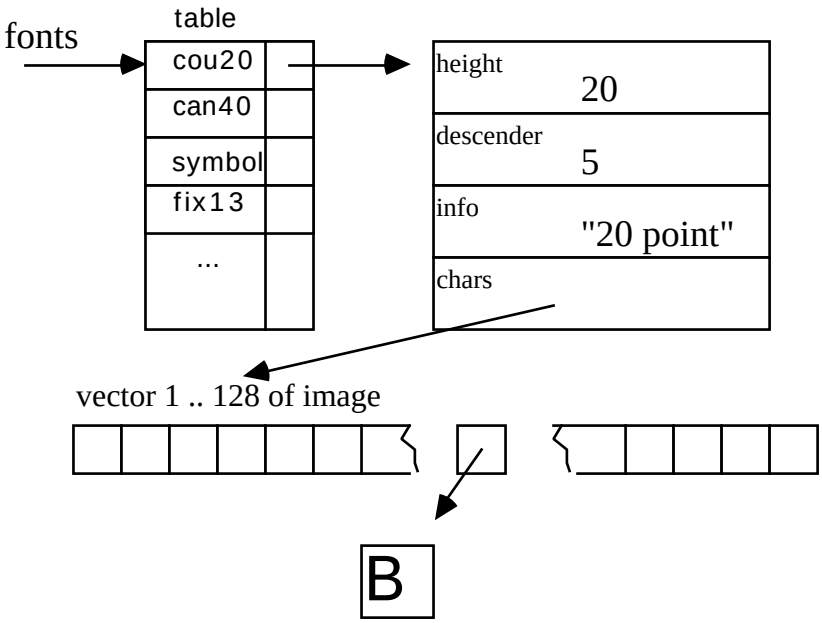
will draw the section of the picture *a.pic* on the image *an.image*. The picture is clipped by the region specified by the points (0.0,1.5) and (3.2,3.9) in the picture coordinate space. Clipping may be performed on the image by specifying a limited area of the image in the usual manner. Automatic clipping of the line drawing is performed to make it fit the image.

Drawings may also be mapped onto other devices. A standard function is provided by the system that allows a draw function for a particular device to be obtained from the database. The technique of storing abstract data types for devices was first used in an earler version of PS-algol [mor86c]. Draw functions are provided for tektronix compatible devices, plotters and various laser printers.

Pictures may be drawn directly onto an image or any part of it. Once the line drawing has been mapped onto an image, the image may be manipulated by any of the image operations. Notice that both pictures and images may be mapped onto a screen, allowing the programmer to choose which paradigm or combination of paradigms is appropriate for the application.

3.3.7 Fonts and menus

From the building blocks many useful functions may be built. One example of this is the ability of the language to manipulate fonts. Fonts are stored in a database which may be freely interrogated by the programmer. The layout of the font database is given in figure 12.



The font database
figure 12

The programmer may not want to have to deal with the fonts database directly so a standard function written in PS-algol is provided by the system. It is called string.to.tile and is defined as following,

let string.to.tile = **proc**(**string** source,font -> **image**)

The procedure returns an image containing a representation of the string source in the font specified by the parameter font. These images can then be used for putting messages on the screen, on the cursor or as part of pictures being built up.

This facility proved to be so heavily used that it warrented syntactic support in the language. To support the manipulation of text and images the print statement was added to the i/o facilities by Livingston [liv87]

Images containing text are often used in conjunction with the pop up menu mechanism which is also provided by the language. Like string.to.tile, the menu function is not a primitive feature but is written using the features we have already seen and another function which allows the programmer to interrogate the pointing device. The menu function has the following definition,

```
let menu = proc(          image title ;
                *image entries,              ! vector of images
                bool vertical ;
                *proc( image,int ) actions,  ! vector of procedures
                -> proc( int,int -> bool ) ) ! returns a procedure
```

The menu which the user sees will have a title corresponding to the image *title* and will have entries corresponding to the vector of images, *entries*. The function menu returns a function which when called will put a menu on the screen at the specified position and allow the user to make a selection from it. If an entry is selected the corresponding procedure from the vector *actions* is executed, the entry and position of the entry used to select it is passed to it as a parameter. If the user does make a selection the procedure will return true otherwise it will return false. In this way, many of the costly calculations that need to be made by the menu function need only be done once. This may be prior to the execution of the main program if the function returned by menu is stored in the persistent store.

3.4 User Interaction

In order to write a function like the menu function shown above, it is necessary to interact with the pointing device and cursor.

As the pointing device (usually a mouse) is moved around, the cursor follows it (unless the standard function which switches off tracking has been called). In order to find out the position of the cursor, the system provides a standard function called *locator*. *locator* has the following form,

let locator = **proc**(-> **pntr**)

it returns a structure of the following type,

structure *mouse*(**int** *X.pos,Y.pos* ; ***bool** *the.buttons*)

The fields *X.pos* and *Y.pos* give the position of the mouse relative to the standard identifier screen. The vector of booleans show the current status of the mouse buttons. One of these structures is returned every time locator is called.

One problem in writing code which interacts with the user via a pointing device and the keyboard is that it is often necessary to know if the user has typed something or not. In order to discover this, the system provides a standard function called *input.pending*. This function returns a boolean which indicates if there is any input waiting to be read from the keyboard. This function allows applications to be written in which the user may non deterministically type or perform a mouse event. A good example of such a program is the Macintosh editor Macwrite.

If the cursor is to be used as a pointing device the programmer must be able to specify which pixel in the cursor is the pointing tip. In PS-algol the function cursor.tip provides this, it has the following type,

let *cursor.tip* = **proc**(**pntr** *new.tip* -> **pntr**)

In order that the old tip may be reinstated, this function returns the old cursor tip. In this way, cursors which look like arrows and cross hairs may be used with the appropriate pointing tip.

3.5 Implementation

The implementation of such a system is highly dependent on the facilities provided by the hardware [bro86] and therefore has not been discussed here. However, the abstract machine must provide support for the graphical objects being manipulated, this support is described in Appendix 1 and in [ps85].

3.6 Conclusions

This chapter has presented the graphics facilities available in one persistent language, PS-algol. These graphics facilities have proven so successful that thay have included, unchanged, in the language Napier. The way in which graphics objects may be manipulated like any other object in the system has been shown. In particular, it is shown how graphical objects may interact with the persistent store. Such an integration provides a powerful vehicle with which applications with sophisticated mmi may be provided.

4 The System Building Domain

4.1 Introduction

The PISA system building domain supports the construction of the persistent languages and environment. The major system construction tools in this domain are:

1. The Abstract machine
2. Abstract program graphs
3. Compiler componentry
4. Support for compilation and execution merging
5. Support for optimisations

Much of the support is provided by tools that are constructed within the persistent environment and therefore supported by the system building domain. At one level, these tools may be viewed as applications making requirements on the system building domain, at another level, these tools will be regarded as part of that domain. A good illustration of this is in the compilation systems, these are written in the persistent languages and, therefore, use the language and environment support provided by the architecture. However, at the same time they provide support for other applications using the architecture.

4.2 History

The abstract machine that supports the language Napier is a refinement of the PS-algol abstract machine. This machine is, in turn, a modification of the machine that supported the language S-algol. Some historical perspective helps understand the structure of the Persistent Abstract Machine.

4.2.1 The S-algol abstract machine

The language S-algol is the predecessor of PS-algol. Although the language is not persistent some important features of the current Persistent Abstract Machine have their roots in the S-algol Abstract Machine [mor79,dea85].

The language S-algol may be implemented using conventional stack techniques. However, the language allows the construction of objects whose size is not known at compile time. The extent of these objects is different from their scope. Consequently, a heap is used in addition to a stack to store objects. Pointers to these objects are stored on the execution stack.

During garbage collection it is necessary to distinguish pointers from scalars. In order to simplify this process the S-algol abstract machine implements two stacks - one for pointer items and one for scalars. Stack frames have a pointer from the main stack to the pointer stack in the mark stack control word to keep the pointer stack frame and the main stack frames logically together. This is necessary to implement return from procedure calls and intermediate free variable access.

4.2.1.1 Object management

Heap objects in S-algol abstract machine are all typed - that is the type of the object is encoded in the headers of objects. Consequently, object management routines in the machine have to check what type the object is before it can be manipulated. For example, the machine needs to know that strings contain no pointers and that their length field is in bytes.

The garbage collector in the S-algol machine keeps a free list of unused space and uses a first fit algorithm for the allocation of space to new objects.

4.2.2 The PS-algol abstract machine

The PS-algol abstract machine is a refinement of the S-algol machine. It needed major revisions in order to implement first class procedures and persistence.

PS-algol is a block structured language that is well suited to a stack implementation technique. However the language has procedures that are first class citizens and store semantics. Therefore, objects in the scope of a block may still be accessed after that block has returned. This is easily illustrated in a small PS-algol example,

```
let counter =
begin
        let count := 0

        proc( -> int )
        begin
                count := count + 1
                count
        end
end
```

example 1
block retention in PS-algol

In this example, the procedure returned as the value of the block yields unique integers, each the successor of the previous one. In order to do this it uses an encapsulated value, *count*. When the block returns, the value of the block, the procedure, is assigned to *counter*. This procedure will access *count* when it is called, consequently the location to which *count* is bound must be retained. The class of languages in which this phenomenon may occur are known as block retention languages.

A conventional stack architecture is not rich enough to support block retention languages. A stack architecture known as a cactus stack is necessary. Since most programs do not require block retention it is tempting to use a stack for efficiency and only do something special when retention occurs. Such a strategy is sometimes called a mixed mode strategy. The something special could be to allocation of space for frames with retention on a heap.

If a mixed mode strategy is used it is possible for the system to run out of space in two ways - by running out of heap space and by running out of stack space. Thus the system can run out of space in one partition whilst unused space exists in the other.

One of the aims of the PISA project is to provide a robust system with stable storage. A persistent store presents the user with a stable, conceptually infinite object space. Thus, it is undesirable for a system to stop working by running out of space when free space exists in the system. Ideally, the system will only stop when all the free space in the system has been used up - this includes both RAM and disk space.

The solution is to have only one dynamic storage system - the heap. All data objects have space allocated for them on the heap. In this way, the system will run out of space only when the heap is full. The PS-algol system does not entirely satisfy this desire as the heap may become saturated with persistent objects. Persistent objects are not written back to disk unless a commit occurs. This problem is corrected in the Napier system.

4.2.2.1 Frames

A stack of procedure frames is simulated in the heap with each frame being a separate heap object. Within each frame two stacks are simulated - one for pointers the other for scalar items. The scalar objects, integers; pixels; reals and booleans, all reside on the main stack, whereas pointers to images, strings, vectors, structures, closures and files reside on the pointer stack.

Each frame has a display that points at the static environment of the procedure. The display may be considered as comprising the bottom of the pointer stack. The display and dynamic links to other frames are also implemented by pointers to other frames also on the heap.

In each frame the main stack grows up memory, the pointer stack grows down memory. This allows all stack addresses to be assigned statically in one pass by the code generator as an offset from one of two machine registers - the local main stack base and the local pointer stack base. The compiler must calculate the maximum stack sizes in order to ensure that the stacks will not collide at execution time.

4.2.2.2 Addressing

The S-algol abstract machine uses procedure level addressing, whereas the PS-algol machine uses block level addressing. This is a requirement if a language with first class functions and store semantics is to be supported. This requirement is illustrated in the following PS-algol example,

```
let avec = vector 1:: 10 of proc( -> int ) ; 1          ! a vector of procedures
for i = 1 to 10 do
        avec( i ) := proc( -> int ) ; i
```

example 2
a vector of procedures

In this example, a vector of procedures called *avec* is declared. The procedures in the vector are all of the type

proc(-> **int**),

the vector initially has the procedure that returns one assigned to each location in it. In the for loop, a procedure is assigned to each location in the vector. Each of these procedures encapsulates one of the values of the control variable *i*. Thus, the procedure which is assigned to location *i* in the vector will also return *i* when called. The control variables in each invocation of the loop body must therefore have a different location.

4.2.2.3 Objects

Like the S-algol machine all objects have their type encoded in the format of the object. The object coding is known by all modules in the system; the garbage collector needs to have knowledge of the encodings to garbage collect objects, the persistent object manager to move objects in and out of RAM and even the compiler so that it may plant code for object literals. This breech of modularity makes it difficult to maintain the system and perform experiments.

This problem made it difficult to do experiments with the PS-algol system. If a new data type was introduced into the language, many modules in the system had to be changed consistently. The code that needed to be changed was written in different languages - the compilers in PS-algol and the run time system in the implementation language. The

volume of code, constructed in different technologies, that needed modification made change difficult and error prone.

The PS-algol machine allocates space for frames and graphics objects (bitmaps) on the heap. This extra utilisation of the heap led to performance problems with the free list method of space allocation used in S-algol. For this reason, the garbage collection technique was changed to a compacting garbage collector [mor78] and the free list discarded. This change improved performance dramatically.

4.2.2.4 The Standard Frame

The PS-algol machine supports a standard frame. There is only one standard frame in any PS-algol system. It provides an environment for all predeclared identifiers in the language such as cos and sin and literals like pi and maxint.

The compiler is provided with a description of this frame in the form of a file of declarations known as the standard declarations file. The run time system is responsible for filling in the locations in the standard frame with the correct values. A set of instructions is provided in the machine which loads objects from the standard frame onto the frame of the currently executing procedure.

The standard frame proved to be another difficulty in providing an experimental framework. The main problem is that different experiments required the addition of different standard functions - functions written in the implementation language. If a change is made to the standard declarations file without the correct change being made to the standard frame, corruption of the whole system is possible.

4.2.2.5 The I/O system

The PS-algol system supports a complex set of I/O instructions which support the buffered I/O in the language. A large part of the underlying support system is devoted to implementing buffered I/O. This is partly due to the I/O system of PS-algol being typed It is also due to instructions that could be written in PS-algol being written in the implementation language for performance gains.

4.2.2.6 Persistent Object Support

Persistent Object Management in the PS-algol system is provided by a module in the interpreter called the POMS (Persistent Object Management System). This module is responsible for implementing a transactionally secure persistent object store.

Originally it was thought that the programmer would use the POMS to support CAD/CAM style of applications. That is they would read in some data, do some processing on it and then either save the changes or discard them. To support this the POMS operates on large disjoint units known as databases. Databases provide a mechanism for identifying persistent data. Each database has a root. Any data object reachable from the root will also persist.

The POMS is part of the run time support system, so there may be multiple invocations of it running on a computer at any time - one for every PS-algol process. Databases are passive objects on which invocations of the POMS operate. In order to ensure that PS-algol programs do not interfere with each other a multiple reader/single writer protocol is imposed on databases. Therefore, a database may only be open for one writer at a time or many readers (but not both).

In order to ensure that data held in the persistent store is updated in a self consistent manner it is necessary to impose a protocol for the update of changes. The PS-algol system provides this by a function called *commit*. A copy of data touched by a program is

loaded into the programs local heap. When a *commit* is invoked by the program, data that has been changed is written back to the databases. The commit algorithm guarantees that either all of the changed objects are written back or none of them. In addition, no objects are copied back to a database unless it has been opened for writing.

Three functions are provided by PS-algol to interface with the persistent store. The functions create.database, open.database and commit allow a program to create a new database, open a database in either read or write mode and invoke the commit algorithm respectively.

4.2.2.7 Pids and Lons

A persistent object is identified by a persistent identifier known as a pid. A pid is the same size as a pointer and may be distinguished from a pointer by having its most significant bit set. A heap pointer is known as a local object number or a lon.

Persistent objects are identified as such by being pointed at by a pid rather than a lon. The abstract machine cannot process pids therefore pointers must be checked before their use. If the pointer is a pid, then the POMS must be called upon to translate the pointer into a lon. Once this translation has been performed the pid is overwritten by the lon

When the POMS is called upon to translate a pid it looks up the appropriate database to find the object to which the pid points. This object is then copied into local memory and the local object number returned. In order to prevent pid translation from being repeated many times a table is kept of all the pids translated during the current interpreter invocation. This table is known as the PIDLAM (pid - local address map). When a pid is first used and translated to a lon an entry is put in this table to memorise the address of the object in local memory. All pid-lon address translations check this table first and only if a lon is not found does the translation take place.

4.3 The Persistent Abstract Machine

The Persistent Abstract Machine (PAM) supports the execution of programs that are written in the Persistent Architecture Intermediate Language (PAIL). These programs are translated from PAIL code into PAM code by the code generation module of the compilation system. The abstract machine relies on the store domain to provide a persistent heap of objects.

4.3.1 Design Principles

Module independence has been the guiding principle in designing the system building domain. Whenever possible the design of one part of the system has been decoupled from the design of other parts. This lesson has been learned from experiments using the PS-algol system. In this system, information escapes from one module into another, making change and maintenance difficult.

This problem is best illustrated in the interface between the language and the persistent store. In the PS-algol system type information is encoded in the abstract machine representations of objects. This means, that to introduce a new type into the system, changes have to be made to the compiler, the abstract machine, the garbage collectors and the persistent object manager.

The separation of the system into layers or modules is essential for several reasons. Firstly, the construction of any system is aided when clear boundaries are formed. These boundaries enforce the demarcation of responsibilities in the system both between programs and programmers. This modularity leads to a benefit in system maintenance. If errors or even design flaws can be localised the volume and complexity of code that has to be changed may be minimised. Secondly, and most importantly in the research field

modularity aids experimentation. The Napier system has been constructed in such a way as to allow experimentation in any of the system construction areas. It is possible to experiment with persistent object managers, abstract machines, code generators, compilers, languages and type systems independently.

The system may be viewed diagrammatically as,

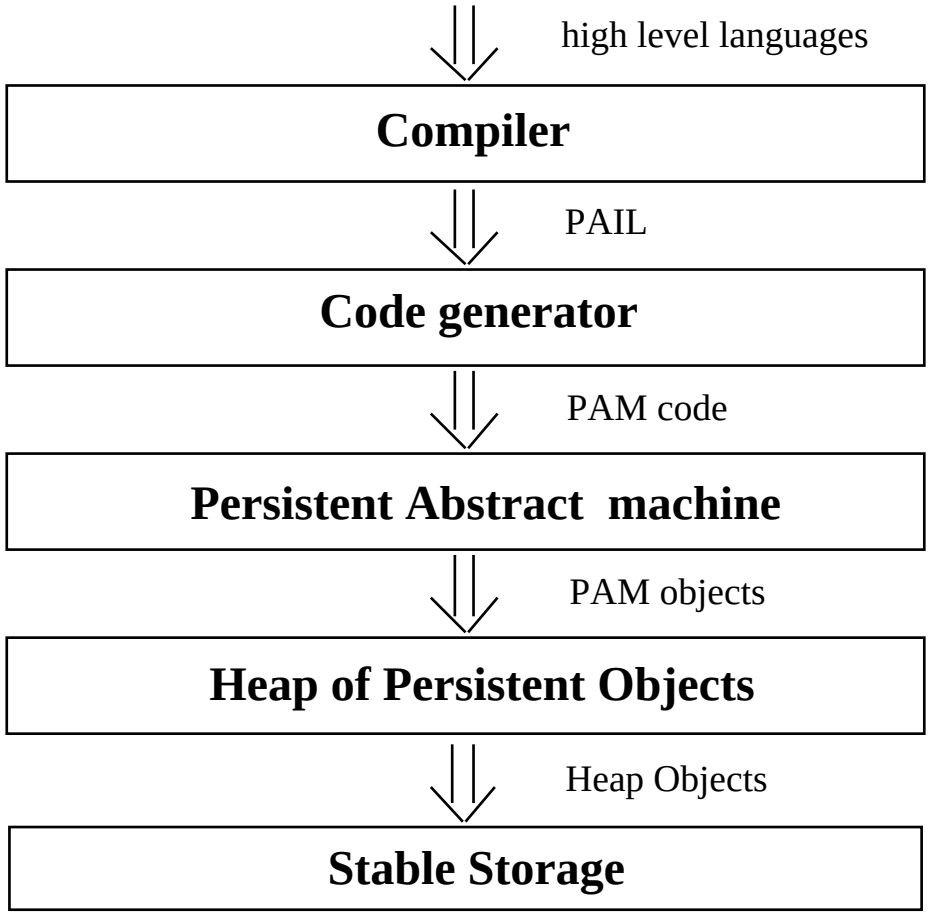


figure 1
The layered Architecture

In this system any of the modules may be replaced by another implementation with the same interface, allowing experiments to continue in parallel on all the areas of interest to the PISA project. The architecture provides the experimental testbed for future experiments in the PISA project. These experiments will include distributed secure object stores and concurrency experiments in both the language and system building domains.

It is our task as system designers to find the correct levels of abstraction so that fire walls are protected; whilst ensuring that these barriers do not adversely effect efficiency. Much of the design effort in the system building domain has focused on finding these correct levels of abstraction.

4.3.2 Heap Objects

The heap is the only dynamic storage system supported by the system. All objects in the system reside on the heap; simple data objects such as integers, pixels or reals reside within objects; complex data objects such as images, vectors and frames are single heap objects.

All heap objects have the same format (a word is a 32 bit integer)

word 0	header
word 1	the size in words of the object
word 2..n	the pointer fields
word n+1..	the non pointer fields

Word 0 has the following interpretation

bits 8-31	the number of pointer fields in the object
bit 7	if set first byte in a short integer is least significant
bit 6	if set first short in an integer is least significant
bit 5	remote address object flag
bit 4	translated bit - if set a field has been changed to a pid
bit 3	written bit for persistent object manager
bit 2	mark bit for garbage collection
bit 1	constancy bit for validating updates in vectors **or** trace bit for use by special return instructions in frames
bit 0	header bit - header or ram address

where bit 0 is the most significant bit of the word.

Thus, all objects are partitioned into pointer and non-pointer fields with the pointer fields preceding the non-pointer (scalar) fields. This minimises the potentially high cost of garbage collection and persistent object management in the system. By separating the pointer and scalar fields, garbage collectors may easily find the number of pointers in an object and then iteratively process them. This decision has serious implications in the rest of the machine design. However, this cost is justified by the simplification in object management which is one of the most expensive tasks performed by the machine. [lob87]

In order to simplify garbage collection, pointers in the architecture only ever point at the beginning of objects - never into them. This has the effect on the instruction set that the addresses of many objects are given as an object address and an offset with the object. This technique is common in segmentation systems.

The object management modules in the system all conform to the basic object convention. Furthermore, they may only manipulate objects in accordance with the information contained in their headers. The high level information described below may not be used by the store level object manipulation code.

4.3.3 PAM Object Formats

The Persistent Abstract Machine imposes higher level conventions on objects. These are not part of the primitive object format. The store level system utilities do not know or care about these higher level conventions since they are only concerned with the basic object format. This allows experimentation in the field of garbage collection, persistent object management and abstract machine design to be carried out independently.

In order to support infinite unions every object has an associated type description. This is pointed to by the object's first pointer field and will be in a canonical form constructed by the compiler. The abstract machine does not have knowledge of how this field is laid out. Type representations are created by the type checker which is part of the high level language implementation. The abstract machine merely treats this field as a pointer. How this pointer is used is discussed later.

Each type of object in the machine is laid out differently, but in accordance with the basic object format. That is that they carry a header and that pointers come before non-pointers. Only the instructions that deal with a particular type have knowledge of how that type is

arranged internally. For example, unlike the garbage collector, the equal string operation needs to know the layout of a string.

Each PAM object class will now be discussed.

4.3.3.1 Strings

word 2	a pointer to the type descriptor for a string
word 3	number of characters in the string
word 4..	the characters 1 per byte padded with zeros up to a 4 byte boundary.

4.3.3.2 Files

There are 5 kinds of files that are supported by the Napier abstract machine; disk files, terminals, mice, tablets and raster displays. Each file kind is represented differently. In all the file types, the internal file number and associated flag bits are represented as follows:

if raster file	bit 8
if tablet	bit 9
if mouse	bit 10
if terminal	bit 11
if disk file	bit 12
if closed	bit 13
if writable	bit 14
if readable	bit 15
file number	bits 16-31

Disk Files

word 2	a pointer to the type descriptor for a file
word 3	a pointer to the file's name
word 4	an internal file number and associated flag bits
word 5	the current position in the disk file (byte offset from the start)

Terminal Files

word 2	a pointer to the type descriptor for a file
word 3	a pointer to the file's name
word 4	an internal file number and associated flag bits
word 5	the terminal modes currently selected

Mouse and Tablet Files

word 2	a pointer to the type descriptor for a file
word 3	a pointer to the file's name
word 4	an internal file number and associated flag bits
word 5	the X dimension of the tablet, 0 for a mouse
word 6	the Y dimension of the tablet, 0 for a mouse
word 7	the X position, absolute for a tablet, relative for a mouse
word 8	the Y position, absolute for a tablet, relative for a mouse
word 9 + n	state of the nth button, numbered from 0

Raster Files

word 2	a pointer to the type descriptor for a file
word 3	a pointer to the file's name
word 4	an image representing the raster device's screen
word 5	an image representing the screen's cursor
word 6	an internal file number and associated flag bits
word 7	the X position of the cursor on the screen
word 8	the Y position of the cursor on the screen
word 9	the raster rule used to display the cursor on the screen (see rasterop)

4.3.3.3 Vectors

word 2	a pointer to the type descriptor for the vector and its elements
word 3..n	the elements
word n+1	lower bound
word n+2	upper bound

4.3.3.4 Images

HEADER	SIZE	TYPE	@ Bitmap Vector	File Descriptor	X-OFFSET	Y-OFFSET	D-OFFSET	X DIM	Y DIM	DEPTH

word 2	a pointer to the type descriptor for an image
word 3	pointer to the vector of bitmap vectors
word 4	pointer to the file descriptor (if a cursor or screen of a raster device otherwise nil)
word 5	X offset into the bitmap vector
word 6	Y offset into the bitmap vector
word 7	depth offset into the bitmap vector
word 8	X dimension of the image
word 9	Y dimension of the image
word 10	depth of the image

The bitmap vector for an image is laid out as follows:

word 2	a pointer to the type descriptor for a vector of integers
word 3	X dimension of the bitmap
word 4	Y dimension of the bitmap
word 5	depth of the bitmap
word 6	number of bits per pixel
word 7	number of pixels per scan line
word 8	offset to start of the image from the start of the object.
word 9..n	bits
word n+1	lower bound
word n+2	upper bound

4.3.3.5 Structures

word 2	a pointer to the type descriptor for the structure
word 3..n	the pointer fields
word n+1..	the non-pointer fields and constancy bitmap

Every structure is assumed to contain a constancy bitmap of one bit per word. It should be checked whenever a word in a structure is to be updated. However updates to the words containing the bitmap are not checked to allow the constancy of fields to be altered. For structure fields of two words only the bit for the first word is used. For a structure of length L the starting word (S) of the bitmap can be calculated as follows:

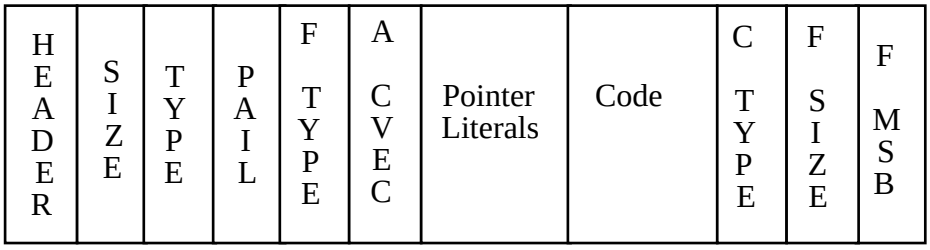
S = L - (L + 30) div 32

The word (W) within the bitmap containing the bit for a given field index (I) and the field's bit (B) within that word can be calculated as follows:

W = I div 32
B = 31 - (I rem 32)

To test if a field is constant bit B in word S + W of the structure is tested. The field is constant if the bit is set. Note that the bits are numbered in decreasing significance from bit 0 to bit 31.

4.3.3.6 Code Vectors



word 2	a pointer to the type descriptor for this code vector (TYPE)
word 3	a pointer to the pail tree for the code vector's procedure (PAIL)
word 4	a pointer to the type descriptor for the frame created when the code vector's procedure is applied (F TYPE)
word 5	a pointer to an alternative code vector (A CVEC), this has the same functionality but contains different code, the code may be a different type
word 6..m	any pointers to objects that are used by the code vector's procedure
word m+1..n	the code to be executed
word n+1	the type of code, 0 if the code is Napier code (C TYPE)
word n+2	the size of the frame (in words) to be created when the code vector's procedure is applied (F SIZE)
word n+3	the offset to the main stack (in words) for the frame (F MSB)

Code vectors are used to store object code for procedures and blocks. They must contain all the information necessary to execute the procedure or block that they represent. The information includes the size of the frame needed and the main stack offset. Debugging information is included in the code vector in the form of the type of the code the vector implements. This is held as a symbol table holding all the address information for the declarations made in the procedure or block. An abstract form of the source code is also held in the code vector in the form of an abstract syntax tree, allowing the source code to be reproduced at run time.

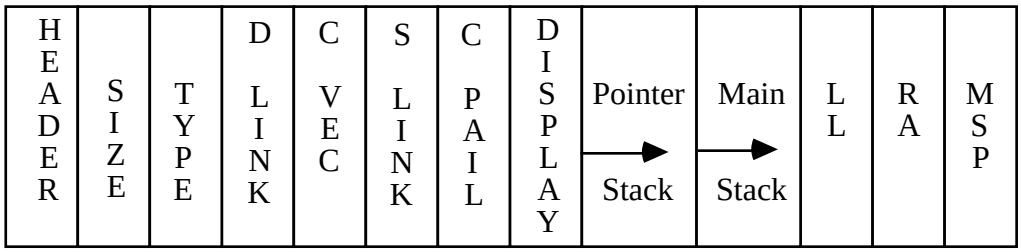
PAM code vectors provide an alternative code vector field. This field may have a code vector of native machine code assigned to it. The assignment may be performed at compile time or at some later time as an optimisation. It is possible to perform the assignment lazily [car87] since code vectors reside in the persistent store and because PAM code vectors contain a pointer to the source of the block or procedure in the form of a PAIL tree.

PAM code must be preserved in this optimisation since code vectors may be executed on different processors in a heterogeneous environment. The abstract machine code definition provides a machine independent program representation. This code may be viewed as an optimised version of the PAIL code.

Code vectors in PAM contain pointer literals. These pointers reference the compile time type representations of objects that may be created at run time. These may be used by the type checking procedure found in the root object. The pointers also store the PAIL source code for the procedure. In this way the source code is bound to the executable code, allowing better diagnostic information to be produced.

4.3.3.7 Stack Frames

Each procedure and block activation is implemented on the heap by a stack frame. In addition to house keeping information, each frame contains two stacks, a pointer stack and a scalar stack shown below,



word 2	a pointer to the type descriptor for this frame, it includes a symbol table for this frame (TYPE)
word 3	the dynamic link (D LINK)
word 4	a pointer to the code vector for the frame's procedure (C VEC)
word 5	the static link for the frame's procedure (S LINK)
word 6	a pointer to the pail currently being executed (C PAIL)
word 7..l	the display for the frame's procedure (DISPLAY)
word l+1..m	the pointer stack frame's procedure
word m+1..n	the main stack frame's procedure
word n+1	the frame's lexical level (LL), the number of entries in the display
word n+2	the return address for the frame's procedure (RA), an offset (in bytes) from the start of the procedure's code vector
word n+3	the saved offset (in words) of the LMSP from the LFB (MSP)

Frames are allocated on the heap and therefore must conform to the basic heap format which dictates that pointer and scalar fields must be partitioned. The machine, like the PS-algol machine, simulates two stacks - a pointer stack and a scalar (main) stack. However, in this machine the pointer fields must precede the scalar fields. The code generators calculate the maximum sizes of these stacks allowing space for all intermediate values in addition to locations which have identifiers bound to them. This information is contained in the code vector. Scalar objects all reside on the main stack; these are integers, pixels, reals and booleans. Pointers to heap objects exist on the pointer stack; they are images, strings, vectors, structures, closures, files, abstract data types, variants and unions. The bottom of the pointer stack implements a display of pointers to the environment.

In PAM, both stacks both grow in the same direction. All addressing is base and offset addressing relative to one register - the local frame base. This scheme makes addressing simpler at the expense of complexity in the code generator. In the code generator no scaler stack addresses may be allocated until the number of pointers in the frame is known. This requires either the code generator to back patch main stack addresses at the end of a block or do another pass allocating addresses.

The technique of making the machine simple at the expense of the code generators has been followed whereever practical. Calculations made in the code generator are made once - at code generation time rather than many times during procedure invocation. Therefore, there is good reason for making the machine as simple as possible from a performance point of view. Another reason for shifting complexity from the machine to the code generators, is that the code generators are implemented in a high level language, whereas the machine is implemented in a low level language or even silicon. This upward movement makes maintenance much more manageable.

4.3.3.8 Abstract data types

word 2	a pointer to the type descriptor
word 3..n	the remaining pointer fields
word n+1.m	the nonpointer fields
word m..	the type keys for the witness types and constancy bitmap

Note that all fields of witness types are implemented as polymorphic objects. Therefore each is assigned space for a double length scalar and double length pointer object. This allows the calculation of field addresses to be performed statically. The dynamic type information of the witness types is stored in the last scalar fields of the object. The implementation of A.D.T.'s is described fully in section 5.3.10.

4.3.3.9 The Root Object

In the PS-algol machine the standard frame became a major obstacle to change. Revisions of the standard frame meant that programs compiled with the old version were no longer executable because offsets into the old frame had been *hard wired* into the code.

Many of the functions in the PS-algol standard frame were not primitive machine instructions. One example of this is the lwb function which returns the lower bound of a vector. This function is implemented in the machine implementation language in the PS-algol system.

In PAM, most of the functions which were implemented in implementation language in the PS-algol machine have been replaced with short sequences of abstract machine code These functions are known as primitive instructions. Functions, like lwb, may easily be implemented in this way, resulting in a simpler underlying machine.

Not all functions may be implemented in this way. Some functions like sin are true primitive machine operations. These have been implemented as abstract machine instructions in the Persistent Abstract Machine. Using these techniques we have removed the need for a standard frame. A fixed point is still needed in the machine -this is provided by the root object. The root object provides a set of known addresses in PAM. There is one root object per machine invocation. It contains literal values such as nil, pi, nullimage and maxint.

A pointer to a vector of single character strings is included in the root object. This was an optimisation first used in the S-algol abstract machine to lessen heap fragmentation and speed up string manipulation.

The root object also contains some procedures used by the machine. These include the startup procedure which is run when the machine is invoked. A type checking procedure that checks if two type representations are the same is also stored here.

A pointer to a vector of error processing procedures that are called when hard errors occur in the machine and a pointer to the vector of event handlers are also included in the root object. These procedures are ordinary procedures written in Napier. A set of special functions, written in PAIL, are provided for assigning values and retrieving values from these locations.

word 2	the pointer literal **nil**
word 3	the code vector for the startup procedure
word 4	the static link for the startup procedure
word 5	the logical root of persistence
word 6	the file literal **nullfile**
word 7	the string literal ""
word 8	a pointer to the vector of all 128 single character strings
word 9	the image literal **nullimage**
word 10	the code vector for the error processing procedure
word 11	the static link for the error processing procedure
word 12	a pointer to the vector of event handling procedures
word 13	a pointer to the vector of error handling procedures
word 14	a pointer to the vector of open files
word 15	a pointer to the frame of the currently executing procedure
word 16	the code vector for the type checking procedure
word 17	the static link for the type checking procedure
word 18	a pointer location for use in comparing variants, **nil** if not in use
word 19	a pointer location for use in comparing variants, **nil** if not in use
word 20	a pointer location for use in comparing variants, **nil** if not in use
word 21	a pointer location for use in comparing variants, **nil** if not in use
word 22	the error number for the last I/O instruction executed
word 23	the integer literal **maxint**
word 24,25	the real literal **maxreal**
word 26,27	the real literal **pi**
word 28,29	the real literal **epsilon**

4.3.4 Stable Storage

Stable storage in the persistent abstract machine is provided by the object management module. This module is a module within the abstract machine and implements the heap. The heap is the top layer of a hierarchy of levels that implements the persistent store. The heap interface consists of eight functions. These are the only functions which deal with object management and operate on basic object formats. That is, they can only determine the size of objects and how many pointers they have.

The interface functions to the heap are:

1. Initialise_Heap
2. Shutdown_Heap
3. Create_Object
4. Destroy_Object
5. Illegal_Address
6. Root_Object
7. Stabilise_Heap
8. Garbage_Collect

The functions Initialise_Heap and Shutdown_Heap are used to initialise and shutdown the persistent store.

Create_Object is the only mechanism provided in the system for object creation. All high level functions that create objects therefore use this function.

The function Root_Object returns a pointer to the root object. Illegal_Address is the function that translates pids into lons, both these functions have already been discussed.

The function Destroy_Object is provided as an optimisation tool. In certain cases the code generator can detect statically when an object is no longer reachable. This system may be used to release space used by these objects. This function is particularly useful for optimising recursive function calls. The usual way of reclaiming space is by calling the garbage collector. This is usually called when a Create_Object fails.

The function Stabilise_Heap causes all new and changed objects to be copied to stable storage. This also causes the store to checkpoint itself causing the store to move into a stable state. A fatal failure in the system such as power loss causes any data changed or modified since the last stabilise to be lost, but data changed or modified before the last stabilise will be retained. Stabilise_Heap is an atomic operation.

4.3.5 The Instruction Set

PAM code is a byte-code instruction set comprising of instructions with between zero and four parameters which may be either bytes, short integers, words or double words.

Objects are represented on stacks by different numbers of stack elements. Therefore, some instructions need to be parameterised by stack size. However, some instructions need to perform different code depending on the object type. Therefore, instructions have two styles of instruction modes, being parameterised either by stack size or by type.

For example, the load instruction performs the operation of loading an object onto one of the stacks. There are six modes of this instruction. These are:

wload -	main stack 1 word
dwload -	main stack 2 words
pload -	pointer stack 1 word
dpload -	pointer stack 2 words
wpload -	main stack 1 word & pointer stack 1 word
dwdpload -	main stack 2 words & pointer stack 2 words

Although the abstract machine does not have any knowledge of the languages' type system, some instructions do need to be parameterised by implementation type. For example, the less than instruction has three typed modes:

lt.i	less than integer
lt.r	less than real
lt.s	less than string

These instructions operate on the top of either the main stack or the pointer stack depending on the type of the arguments.

4.3.6 Types

The Persistent Abstract Machine has no knowledge of the type systems of the languages that it supports. This allows system building domain to be decoupled from the language domain, and illustrates another example of the modularity of the system. However, the machine does store type representations supplied by the high level system and call functions that manipulate these representations.

For the purpose of the discussion below, it is necessary to know that the compilation system builds graph structures representing high level types. The building of graphs rather than having a simpler representation is necessary in order to support the recursive and unbounded nature of the languages supported by the machine - in particular the Napier type system.

4.3.7 Support for Infinite Unions

The languages supported by PAM have infinite unions in their type systems. In the case of Napier, two infinite unions are supported; **env** and **any**. For the purpose of this discussion, only **any** is discussed here. A value of any type may be injected into the type **any** in Napier. The result of the injection has type **any**. This is shown by an example,

```
let astring = "a string "                    ! has type string
let anany := any( astring )                  ! has type any
let aproc = proc( x : int -> int ) ; x       ! has type proc( int -> int )
anany := any( aproc )                        ! still has type any
```

example 3
use of any

Here, the identifier *anany* has type **any**, furthermore the value stored in *anany* may take any value. Values of type any are stored on the pointer stack. If the value is a scalar or a double length object, it must be wrapped in a container. In other words, a pointer (or the value in the scalar case) must be stored in a heap object. This container has the type of the original object as its type in the object type field. This technique insures that the type of an any may always be found in the type field of the object which represents it.

When a value is projected out of an any, the expected type of the projection must be specified. This is best illustrated by an example. Suppose that the above lines of code have been executed. The projection in Napier is as follows

```
project anany as this onto
string              : writes( "It was a string" )
int                 : writes( "It was an integer" )
proc( int -> int )  : writes( "It was a proc( int -> int )" )
default             : writes( "Don't know what it is" )
```

example 4
projection from any

Unions require that some type checking be done at the time of projection i.e. at run time. This is a particular problem for infinite unions since we are forced to check the graph representations of the types dynamically since the members of the union cannot be enumerated.

A solution to this problem has been known for some time. It involves ensuring that all type representations used are identical. That is, they are the **same instance** of the same graph. This could be achieved by having a type server in the persistent store that hands out unique type representations. Doing this allows the run time type check to be simply pointer equality. However, this solution was not considered to be a good one, since major problems may arise in a distributed environment.

Another solution would be to write the graph checker in the implementation language i.e. have the checker as an abstract machine op-code. This solution requires unbounded space, since a stack is needed to check graph structures. This solution has no intrinsic problems, however, simultaneously maintaining both a low and high level implementation is expensive in terms of software engineering costs.

The best solution is to use the same type checker (written in Napier) that the compiler uses to check the graphs. The problems which arise here are the linking of the Napier code into the machine and the writing of the type checker.

The Napier type checker must be written in a subset of Napier without the use of infinite unions, this establishes a fixed point in the type checking system. Given that this may be achieved, the code needs to be located somewhere that the run time system can find it.

A set of fixed locations exist in the system in the form of the root object. A special function, written in pail, is provided which will take the type equality function as a parameter and put it into the root object. When the runtime system needs to check the equality of two types, it may call this function with the type representation graphs as parameters. Since the type checking system is itself written in Napier, the use of unbounded space does not present a problem because space may be allocated from the heap.

4.3.8 Implementation of Variants

The Napier language supports a tagged variant type. Any object may be a member of a variant and any object may be a member of more than one variant at a time. This is a more general variant type than is provided in other languages [har86,mat85] and causes some implementation problems. These problems will be illustrated by example (in Napier)

```
type air is variant(  balloon   : structure( speed : int ) ;
                      plane     : structure( speed : int ) )

type transport is variant(  train     : structure( speed : int ) ;
                            plane     : structure( speed : int ) )

let concorde = structure( speed = 5000 )        ! of type structure(speed:int )
let anair = air( plane : concorde )             ! of type air
let atransport = transport( plane :  concorde )  ! of type transport
```

example 5
use of variants

Here the object bound to the identifier *concorde* is injected into two variant types; *air* and *transport*. This means that the tag information needed for projection may not be stored with the object. Instead, it must be stored with the location associated with the variant, in other words with reference to the object.

The second example is illustrates that any type may be in a variant,

```
type afewshapes is variant(  aproc     : proc( int ) ;
                             anint     : int ;
                             astring   : string ;
                             areal     : real )
```

example 6
a difficult variant
to implement

In this second example, the type described is a variant of four different types. The types of the objects in this variant are all implemented in PAM by objects of different sizes. To make matters worse, the objects may also reside on different stacks as in this case.

Equality is the final problem with variants, suppose we have the above type defined and we have two objects of type afewshapes. The desired semantics for variant equality is that

the objects must be in the same branch of the variant and they must be equal. Since equality is a type dependent operation, the real type must be stored with the object.

In PAM variants are implemented as a double word object consisting of one scalar and one pointer. The scalar field contains an encoding of the type and the variant branch. This allows projection to be performed by comparing this word with a supplied tag. Variant branches are sorted into name order and enumerated. The type encoding is a five bit code which indicates the type of the variant branch. In order to do this, the machine only needs to be able to differentiate between double and single length objects and strings. Thus, encoding removes the need for the potentially expensive graph checking needed in the infinite union case.

4.3.9 Polymorphism

A function is said to be polymorphic if it can accept arguments of more than one type; for example, the identity function, shown in example 7 accepts parameters of any data type. Two forms of polymorphism exist, known as universal polymorphism and ad-hoc polymorphism.

```
let id = proc[ t ]( x : t -> t ) ; x
```

example 7
the identity function

When ad-hoc polymorphism is employed the type of code that is executed depends on the type of the argument. For example, in PS-algol,

```
write "hello"
```

involves different code from

```
write 5.
```

In contrast, the essence of universal polymorphism is that the same code is executed regardless of the types of the arguments. Two different forms of universal polymorphism exist, parametric polymorphism and inclusion polymorphism [card85]. In parametric polymorphism a polymorphic function has an implicit or explicit type parameter, which determines the type of the argument for each application of that function. In inclusion polymorphism an object may be viewed as belonging to many different classes which need not be disjoint. Cardelli and Wegner point out that the two forms of universal polymorphism are not disjoint but are sufficiently different to deserve different names.

The function shown in example 7 is said to be quantified by the type *t*. In the language Napier, procedures may be quantified by any number of types, giving the programmer power to abstract over many types.

The parametric polymorphism provided in Napier is explicit. That is, quantifiers must always be specified when a polymorphic function is defined. Similarly, functions must be specialised to some concrete type before they are applied. No type inferencing is performed by the type checker. For example, the identity function shown in example 7 may be specialised to the integer identity function by writing,

```
id[int]
```

or to the string identity function, by writing,

```
id[string].
```

Each specialisation of a polymorphic function creates a new procedure closure instance. The closure comprises the code for the polymorphic function and the new environment. A single instance of a polymorphic procedure may be specialised to many different non polymorphic procedure instances. This is important in a persistent environment where many users may wish to use a single polymorphic procedure.

In Napier, polymorphic procedures have full civil rights in both their specialised and unspecialised form. Therefore the integer identity function, shown above, may be passed as a parameter or returned by a function. If it is assigned to a location, as follows,

```
let idint = id[int]
```

he function bound to *idint* is indistinguishable from the function

```
let idint = proc( x : int -> int ) ; x
```

Before examining possible implementation strategies, some of the potential problem areas will be examined, they are:

1. first class procedures;
2. passing parameters;
3. returning values; and
4. structure creation.

4.3.9.1 First Class procedures

If procedures are first class citizens in a programming language, on application it is generally impossible to statically determine which closure is being used. Consequently, polymorphic procedures must be indistinguishable from *ordinary* procedures.

4.3.9.2 Parameter Passing

Parameters passed to procedures must behave normally whether they have been declared as polymorphic or not. Four different cases must be considered, they are:

1. ordinary values passed to ordinary procedures;
2. polymorphic values passed to ordinary procedures;
3. polymorphic values passed to polymorphic procedures; and
4. ordinary values passed to polymorphic procedures.

Examples of each of these cases are given below.

```
let id = proc( x : int -> int ) ; x
id( 3 )
```

example 8
ordinary parameters with
ordinary procedure

Example 8 shows a normal procedure definition and call. The formal parameter x is of concrete type (integer) as is the actual parameter, 3. In example 9, the identity function shown above is passed to procedure *p* as a parameter. In the body of function *p*, *y* is called with the quantified object, *x*, as a parameter. The call shown in the second line results in the procedure *id* being called. *P* is supplied with a concrete instance (3) of the formal quantified parameter, x.

```
let p = proc[ t ]( x : t ; y : proc( t -> t ) -> t ) ; y( x )
p[ int ]( 3,id )
```

example 9
a polymorphic type
as a parameter
to a non polymorphic procedure

Example 10 shows a polymorphic procedure that has another procedure defined within its scope. Notice that the function *r* is a polymorphic procedure but is not explicitly quantified by any type. The type of *r* depends on the type to which *q* is specialised. Furthermore, the formal parameter of *r*, *z*, is defined as being of the quantified type *t* and the procedure is always applied with a value, *x*, of the same type.

```
let q = proc[ t ]( x : t -> t )
begin
    let r = proc( z : t -> t ) ; z
    r( x )
end
```

example 10
a polymorphic type
as a parameter
to a polymorphic procedure

The case illustrated in example 11 has already been seen in the other examples but is included for completeness. Here, a polymorphic procedure, *s*, is applied with a actual parameter of a concrete type.

```
let s = proc[ t ]( x : t-> t ) ; x
s[ int ]( 3 )
```

example 11
ordinary parameters with
ordinary procedure

4.3.9.3 Returning values

Values returned by polymorphic procedures must also be carefully considered. Polymorphic procedures may return:

1. ordinary values;
2. polymorphic values; or
3. objects which encapsulate polymorphic values.

The first two cases are straight forward. Polymorphic procedures that return polymorphic values have already been seen. The function *s* shown in example 11 is of this kind. Example 12 shows an equality function.It returns a non polymorphic value (a boolean) indicating if the two objects are identical.

```
let identical = proc[ t ]( a,b : t -> bool ) ; a = b
```

example 12
polymorphic function
returning non polymorphic value

Objects that encapsulate polymorphic values must be treated with caution, the potentially most difficult area is in returning structured objects. These are discussed below.

4.3.9.4 Structure Creation

Consider the following procedure,

```
let p = proc[ t ]( x : t -> structure( y : t ; z : string ) ; struct( x,"hello" )
```

example 13
polymorphic function
returning a structured object

it returns a structure containing an object of polymorphic type *t* and a string. If structure classes are normalised, as they are in the Persistent Abstract Machine, some special action must be taken, since the offsets of *y* and *z* are not known statically.

4.3.9.5 Implementation Approaches

Burstall and Lampson in the language Pebble [bur84b] define universal polymorphic languages to be ones in which the same code is executed regardless of the type of the argument, and that different types of data have uniform representation. In their paper they excuse this operational definition on the grounds that a sound mathematical basis is lacking, although they refer to a mathematical definition by Reynolds [rey83]. It is not clear from the dialogue, if uniform representation is intended to apply to the conceptual or implementation level.

At the implementation level uniform representation is a restrictive and highly inefficient form of parametric polymorphism. The potential problem areas outlined above present no problems if uniformly sized objects are used. However, this restriction makes it impossible to implement double length objects such as real numbers. This explains the lack of floating point objects from the language ML [har86], which uses this implementation method.

Recall that in order to simplify garbage collection, the Persistent Abstract Machine separates pointers from non-pointers. This design decision conflicts with the uniform object strategy proposed by Burstall and Lampson. In order to satisfy the requirement of having uniform objects either all objects would have to be implemented as references or another object management strategy would have to be developed. Neither of these options are acceptable in terms of machine efficiency.

Another method of implementing polymorphic languages is to used a tagged architecture [org73] The provision of tagged objects allows the same code to be executed irrespective of object types. Type dependent operations may interrogate object tags to determine the types on which they operate. However, objects of different sizes still present a problem. Hardware implementations of tagged architectures are relatively rare and software simulations of them are inefficient.

4.3.9.6 P.A.M. Implementation of Polymorphism

The implementation of polymorphism utilises the block retention architecture of the Persistent Abstract Machine. The technique used is a hybrid solution. In order to gain high performance, tagging is only performed when necessary. Instead of tagging individual objects, tags are held in procedure closures at known addresses.

The compiler cannot statically determine which procedure is being specialised. However, it can plant code to perform some operation on whatever closure is on the stack at

specialisation time. Therefore, when the identity function shown in example 7 is compiled the code produced by the code generator is for the following function,

```
let idgen = proc(   key : int -> proc( x : poly -> poly ) )
                    proc(  x : poly -> poly ) ; x
```

It is this function that is applied when the procedure is specialised. An integer encoding of the type is supplied as a parameter. This encoding is encapsulated in the frame of the function bound to *idgen*.

The integer key that is planted is known as the dynamic tag of the polymorphic objects. When the specialised function is executed, the encapsulated values in the static environment may be interrogated to discover the dynamic tag of the polymorphic objects being manipulated.

A distinction is drawn between language level and machine level types. The type checker in the Napier compiler represents types as graphs. These graphs may not be checked quickly enough to produce an efficient abstract machine. This is because the information contained in them is too general. Therefore, the notion of dynamic tag is introduced into the abstract machine. The dynamic tag of objects is the only notion of type in the Persistent Abstract Machine. It allows pointers to be distinguished from non-pointers, the size of the stack object and whether the object is a string to be determined.

This information is sufficient to ensure that the same code run is for polymorphic procedures irrespective of the specialised type, without having to enforce the restriction of uniform object sizes.

A five bit encoding is used for dynamic tags, the bits have the following significance:

bit 0	set if the object is a single word scalar object
bit 1	set if the object is a double word scalar object
bit 2	set if the object is a string
bit 3	set if the object is a single word pointer object
bit 4	set if the object is a double word pointer object

This results in the following encoding for the dynamic tags of Napier objects:

object	bit pattern	integer code
integer,pixel or boolean	00001	1
real	00010	2
string	01100	12
vector, structure, abstract data type, file, image	01000	8
procedure	10000	16
variant	01001	9

In the Persistent Abstract Machine all polymorphic operations take the address of dynamic tag as a parameter. The address of the dynamic tag may always be determined statically even although the polymorphic type may not. The dynamic tag is held in the frame of the generating function, within the static scope of the polymorphic function.

The Persistent Abstract Machine does not enforce uniform object sizes. However, in order to ensure an efficient implementation it is necessary to statically calculate the stack addresses of objects. This allows base and offset addressing to be performed. In order to achieve this within polymorphic procedures all objects of quantified type are allocated two words on the scalar stack and two words on the pointer stack. At most, two words of this

space will be used, the rest being filled in with dummy values. At first, this strategy may appear expensive, but, space is only allocated on stacks and extra heap objects are never created.

The Persistent Abstract Machine supports operations that manipulate these double length objects. These operations are given in the Persistent Abstract Machine reference manual which is supplied as Appendix 1.

Since polymorphic procedures are indistinguishable from *ordinary* procedures all parameter passing and return must manipulate ordinary objects. That is, the double scalar, double pointer objects are never passed as parameters or returned as results. Consequently, polymorphic procedures must perform house keeping operations when they are called, when they call other procedures, when they return values and when structures are created. Each of these cases is dealt with below.

When a polymorphic procedure is applied the calling procedure will have initialised the parameters in the polymorphic procedures frame. Each of the parameters that have been declared as being of quantified type in the polymorphic procedure must be turned into a double length scalar and pointer stack object. An instruction called expandPoly performs this operation. Like all the polymorphic operations in the Persistent Abstract Machine, this operation refers to the dynamic tag information to determine its course of action. Once this has been performed, all polymorphic values are treated as double length objects within the scope of the procedure.

When a polymorphic procedure calls another procedure the reverse of the above process must be performed. Parameters must be passed in there concrete form. Therefore, an operation called contractPoly retracts the objects on the stack frame. This operation essentially removes the dummy values from the stack. Some optimisation is possible when polymorphic procedures call other polymorphic procedures as in example 10 above. However, analysis of these situations is sometimes complex. Consequently these optimisations are not currently performed.

Values are always returned by polymorphic procedures in their concrete form. Therefore, if a polymorphic value is being returned by a procedure, the contractPoly instruction is used to translate a double stack object into its concrete representation.

Structure creation is perhaps the most difficult of the polymorphic operations to deal with. The procedure shown in example 14,

```
let p = proc[ t ]( x : t -> structure( a : int ; b : string ; c : t )
                        struct( 3,"hello",x )
```

example 14
polymorphic function
creating a structure

creates a structure containing three values, a string, an integer and a polymorphic object *x*. This presents special problems in the Persistent Abstract Machine because structures are normalised into alphabetical order with pointers proceeding non pointer objects. If the procedure *p* is specialised so that the quantified type *t* is an integer then the field *c* will precede field *b*, however, if *t* is another string then field *b* will precede field *c*. Consequently, it is impossible to statically determine the address of fields *b* and *c*.

The abstract machine must therefore calculate the addresses of these fields at run time. This calculation could be implemented by a sequence of instructions, but, since the abstract machine is interpreted, it is implemented by a single PAM instruction called polystructaddress which takes the number of fields in the structure as a parameter.

The statically known size of the structure is loaded onto the main stack followed by the known number of pointers and two words for each field in the structure. Each pair of words is initialised to contain 0 and the dynamic tag of the field. Executing polystructaddress replaces these values with the pointer and non-pointer address of the fields respectively. The algorithm used during this calculation may be found in Appendix 1.

4.3.10 Abstract Data Types

Napier has a powerful abstract data type (adt) construct based on the existential types of Plotkin and Mitchell [mit85]. It allows an adt. to be manipulated without being able to find out its implementation or representation. The problems that arise in implementing adts are similar to those encountered implementing parametric polymorphism. Before examining the solutions to these problems some examples are given to illuminate them.

In Napier, adts are described by type. For example, an abstract type may be defined as follows,

```
type number is abstype[ i ]( value     : i ;
                             increment : proc( i -> i ) ;
                             display   : proc( i ) )
```

example 14
the definition of an
abstract data type

The type *i* is known as the witness type. Once an abstract type has been created, it is impossible to discover what this type is in reality. The type defined has three fields: a *value* field of the witness type *i*; an *increment* field which is a function from *i* to *i*; and a display field which is a procedure that takes an *i* as a parameter.

In an analagous manner to the creation of structures, the name of the type is used to create an instance of an abstract data type. Example 15 shows the creation of an instance of the type *number*, in this case the witness type is integer.

```
let adt1 = number[ int ](  1 ;
                           proc( x : int-> int ) ; x + 1 ;
                           proc( x : int ) ; writei( x ) )
```

example 15
the creation of
an abstract object

Similarly, another instance of the same type, *number*, could be created, with string as the witness type. The implementation shown in example 16 uses tabular (base 1) arithmetic.

```
let adt2 = number[ string ](  "1" ;
                              proc( x :string-> string ) ; x ++ "1" ;
                              proc( x : string ) ; writei( length( x ) ) )
```

example 16
the creation of
another abstract object

The two objects denoted by adt1 and adt2 both have the same type, that is,

```
abstype[ i ]( value     : i ;
              increment : proc( i -> i ) ;
              display   : proc( i )
```

or number for short. They are assignment compatable, may be passed as parameters in place of each other and so on. Furthermore, the two implementations are indistinguishable since they exhibit the same operational semantics.

The types described are truly abstract, they exhibit the same semantics yet have totally different implementations. Obviously, disaster would ensue if a witness types from one implementation were supplied to a function from another. Consequently, Napier utilises a scoping mechanism to ensure that this may never happen. Witness types may roam free of abstract data types only within a restricted scope.

In example 17, below, a function called *useabs* is defined. It takes an instance of the abstract data type *number* as a parameter. In the body of the function, a *value* of witness type is extracted from the abstract type, as is the *display* and *increment* proedures. These are extracted to avoid repeated dereferencing of the object. The function *display* is called to show the value of *anumber*. The value *anumber* is then incremented using the *increment* function. The value of the function is again displayed before the value *incremented* is finally assigned to the loaction *value* in the abstract type.

```
let useabs = proc( anum : number )
use anum as this in
begin
    let anumber = this( value )
    let display = this( display )
    let increment = this( increment )
    display( anumber )
    let incremented = increment( anumber )
    display( incremented )
    this( value ) := incremented
end
```

example 17
a procedure that uses
an abstract object

The function defined in example 17 may be supplied with the objects *adt1* or *adt2* as parameters since these are both of type *number*. The fact that the implementations of *adt1* and *adt2* are different does not matter; it is this implementation that has been abstracted over. In both cases, the first call of *display* will write "1" and the second call will write "2". In both cases, the abstract value "2" is assigned to the field *value* in the abstract type.

The key to the protection mechanism is the renaming that is performed.The object bound to the identifier *adt2* is bound to the identifier *this.* The location denoted by *this* is constant. Furthermore, the scope of *this* is limited to the block associated with the use clause.This is the only way in which abstract objects may be dereferenced ensuring safety.

This mechanism may seem unnecessarily restrictive. However, it is necessary if static type checking is to performed on abstract data types.

4.3.10.1 Potential Problem Areas

The first potential problem area is in abstract data type creation. In order to produce an efficient implementation it is desirable for addresses to be calculated statically whenever possible. When the fields of objects may be of different sizes, as is the case with abstract types, this is clearly a problem.

The use of abstract data types also creates problems. Notice that in example 17 the compiler can not determine statically which implementation of the abstract type is being used. The objects referred to anumber and incremented may be of different sizes and reside on different stacks. This is the same problem as that encountered with parametric polymoirphism.

4.3.10.2 P.A.M. Implementation of abstract data types

The mechanisms used to implement parametric polymorphism may also serve to implement abstract data types. Since abstract data types are first class data objects in Napier, they may be passed around freely and placed in the persistent store. Therefore, the technique of holding dynamic tag information in procedure closures is not sufficient to implement abstract data types. However, the tags may be carried around in the object that implements the abstact type. Furthermore, it is possible to calculate the addresses of the tags statically.

It is desirable to be able to calculate not just the addresses of the tags, but the addresses of all the fields of an abstract data type. Clearly, since the objects may be of differenrt sizes, and may be pointer or non pointer types this creates a problem. The solution is to use the double scalar, double pointer technique discussed in the previous section in connection with polymorphic objects.

Abstract data types in Napier are implemented as structures with each field of witness type being allocated two addresses, one for the pointer type and one for the non-pointer type. The last fields in the structure contain the dynamic tags for the witnesses followed by the constancy bitmap found in all structures. This technique allows all the addresses to be calculated statically.

When a use clause is executed the dynamic tags are extracted from the abstract data type. These are placed on the execution stack and the block associated with the use clause executed. Once the dynamic tags have been placed on the stack the situation is implementationally identical to the situation found in polymorphic procedures. The fields in the abstract type may be dereferenced in order to extract or assign values using the poly subscript instructions already provided to support parametric polymorphism.

4.3.11 Debugging Support

The predecessors of PAM, the S-algol and PS-algol abstract machines both have an instruction called line number. This instruction takes as a parameter an integer representing the line number of the current instruction sequence. When this instruction is executed the parameter is saved in a register. Thus, the line on which an error occurs may be displayed. This scheme may be extended so that when a procedure call is executed the line number of the current line is saved in the calling frame and a new line number stored in the register. Therefore, a complete calling sequence may be reported by traversing the dynamic chain.

In an environment where only one source program is running, this technique works well. However, in PS-algol, the use of first class functions and separate compilation means that line numbers do not uniquely identify lines of source code. Consequently, a more sophisticated mechanism is required, and the Persistent Abstract Machine is designed accordingly.

When PAIL code is generated a literal pointer instruction is planted in the code stream. This instruction operates in a similar manner to the new line instruction but places a pointer to some abstract source from the code stream into a location in the currently active frame. This links the source code with the currently running procedure. When an error occurs it is possible to display the source code of where the error has occurred to the user.

Since the source is stored in the currently active frame the dynamic call chain may be displayed to the user. The abstract code PAIL contains context information so that the static environment may also be shown.

PAIL code is decorated by the compiler with address information of the identifiers in the current procedure. This information is all that is required to allow the user to browse over the name-value bindings stored in the frame. When and how this may be done remains unresolved.

If a programmer has encapsulated information within an abstract data type it is safe to assume that data was intended to be hidden. If a fault occurs in that abstract data type, should a debugging system be allowed to examine the contents of that abstract data type? It is not clear what the answer to this question is. On one hand there is a practical argument that says if data which may have been expensive to gather is held in an erroneous program there should be some mechanism to retrieve that data. On the other hand there is the purist view that says that if data has been encapsulated any discovery of hidden types is a breach of type security and hence modularity.

4.4 Conclusions

This chapter has described the important features of the Persistent Abstract Machine. The predecessors of the current machine have been examined briefly. The good parts of machines predecessors have been retained or modified in the new machine and the bad parts discarded. It is only through implementations that it is possible to make this distinction between good and bad.

The Persistent Abstract Machine is an implementation vehicle for language and system experimentation. As such it has been designed with modularity in mind and therefore does not contain any language specific support. Instead, it supports a wide range of typed algorithmic languages. Most importantly, the machine is independent of the type system of the languages which it supports. It is also independent of the persistent object management system that supports it, allowing experimentation in this field to continue independent of abstract machine design. This decoupling will aid our future experiments.

Perhaps the most important part of the machine is the way in which it implements polymorphism. The machine implements universal polymorphism over objects of different sizes in an efficient manner. The scheme used to support polymorphism may be extended without modification to support a powerful notion of abstract types. It is thought that this mechanism may also be used to support inclusion polymorphism. This will be the subject of further investigation.

5 Abstract Program Graphs

5.1 Introduction

During the compilation process a compiler collects and collates a vast amount of information about a program. This information is gleaned from the program source and from a few rules that have been programmed into the compiler itself. All the information gathered by the compiler is therefore contained in the original source program. The internal form of the information created by the compiler is a convenient form for the compiler to manipulate. The program source has been designed for use by human beings.

The information gathered by compilers is varied in nature. Much of the information is concerned with the use of names: such as where is a name introduced; where is it used; what type is associated with a name; what values are associated with a name and so on. The compiler also holds context sensitive information, that is, information that cannot be extracted from the source without knowledge of the semantics of the programming language. The use of names is context sensitive since it is only by context that the compiler or human reader can tell which name is being referred to.

The traditional view of the compilation process is that compilers are gathering information so that a semantically equivalent form of the source may be created by the compiler. This form is usually represented in a lower level language such as assembly language or abstract machine code.

5.2 Traditional Compilation Systems

Once the compiler has achieved its goal and created another form of the program all the information that has been gathered by the compiler is usually discarded. The reason for this is probably due to technological problems such as efficiency considerations. For example, in a traditional system the compiler reads in a source file and produces another file usually containing assembly language. Both the source and the result of the compilation are linear streams of information suitable for storage in a file.

This is in sharp contrast to the highly structured information created within the compiler. Generally, this information may be a graph structure containing symbol tables of name information, perhaps lexicographically scoped and abstract syntax graphs denoting operations on data.

Traditional architectures provide support for manipulating complex data structures in RAM but do not provide support for saving these data structures on disk. In such a system it is cheaper to reconstruct information from the source file rather than attempting the saving and restoration of structured abstract information.

Most operating systems do not provide any binding mechanism between files. This makes it almost impossible to bind the source to the executable version of a program. Thus, when presented with an executable program the user has to trust that it will do what he or she expects. Preferably, the source would be tightly bound to the executable program so that a user could check that the procedure source corresponded to the program intended.

A number of consequences arise from this. Perhaps the worst is the poor diagnostic information given by these systems. A typical Unix error message is:

> segmentation violation - core dumped

The user may examine the core file produced by the system but this is of little help unless the program has been compiled with the debugging options set on the compiler and the

user has the source of the program. Often the user is left to guess what the error was and where it occurred.

Another problem with discarding information gathered by the compiler is that some optimisations are not possible. A class of optimisations, known as peephole optimisations, may be performed on executable code. To do these optimisations no knowledge is needed of the source code. A much larger class of optimisations require information contained in the source program. A convenient form of this information is contained in the data structure created and discarded by the compiler.

5.3 Persistent Systems

In persistent systems any data may persist for as long as it is reachable. This data may include structured data such as the abstract program graphs constructed by the compiler. In a persistent system, it is possible to save the abstract program graphs created by a compiler. This is also possible in a conventional architecture it is, however, neither practical nor convenient. The abstract representation of the program may be bound to the executable form produced by the compiler tying a representation of the program source to the executable code.

Since the information stored in a persistent system is potentially long lived a canonical form of the abstract data graphs is required. This form may then be manipulated not just by the compilers in the system but by the optimisers and the diagnostic and utility programs in the system, for example, by syntax directed editors. In the Persistent Information Space Architecture (PISA) this canonical form is provided by the Persistent Architecture Intermediate Language, known as PAIL.

5.4 Persistent Architecture Intermediate Language

Abstract program graphs are not a fundamental requirement of system construction. It is possible to compile languages directly, perhaps using multiple passes, into a lower level code. Diagnostic information may be provided from the source rather than structured information. However, the provision of an abstract form of programs is an efficient and convenient way of representing programs more suited to manipulation by a program than by human being. The decision to have PAIL in the system is therefore an engineering decision and not a necessity.

The compilation system uses PAIL to store structured information about a program. PAIL graphs are much more structured than source code programs. This allows information to be gleaned from them much more readily. For example, it is easy to find where objects are declared and where they are used from the PAIL graph. It is essential that this kind of information is easily obtainable if good optimisers are to be written.

If diagnostics are to be produced by a compiler or run time system, complete information is required. However, only the source of the original program has any meaning to the user. Therefore the original source must be reproducible from the intermediate form. This is an important consideration when performing optimisations. Poor diagnostic information in systems often results when an optimising compiler is used. Thus, the source program is always reproducible from a PAIL program.

5.4.1 PAIL graphs

PAIL is not a textual language, that is, it does not have a concrete textual linear syntax. PAIL consists of a number of structure classes or types. A valid PAIL program comprises a collection of instances of these classes linked together to form a graph structure. Any valid PS-algol or Napier program may be represented by a PAIL graph.

PAIL comprises of thirteen semantic classes, described below. These classes support all aspects of computation supported by PISA.

1. Basic tree structure
2. Symbol table entries
3. Control
4. Assignment
5. Store Allocation
6. Indexing
7. Aliasing
8. Scoping
9. Store to Store operations
10. Literals
11. Application
12. Comments
13. Optimisations

These classes are described fully elsewhere [dear87]. To give the reader a flavour of PAIL some of the more important PAIL classes are described below using the Napier type system.

5.4.1.1 Basic tree structure

The basic unit node in a PAIL tree is a node of type tree:

```
rec type  tree[t ] is structure(  Type    : TYPE,
                                  Code    : t,
                                  Parent  : Parent[t ] )
&
          parent[t] is variant(   Empty   : null ;
                                  Tree    : tree[ t ] )
```

This structure class is used to hold type information, abstract code and context information together. The field *Type* holds an encoding of the type of the subgraph referred to by *Code*. Types are represented by abstract types that are created by the type checker. The typing of graphs is crucial to the integrity of the system and to the production of efficient code. The field *Code* refers to an arbitrary piece of PAIL code. Since PAIL is a graph structure, a PAIL program may share common sub structures. However, PAIL may also have tree structure imposed on it. By following the *Code* fields of tree nodes from the root, a tree will be traversed that includes every tree node in the PAIL graph. The field *Parent* is a reference to the node immediately above the current node in this tree. This allows an entire PAIL graph to be reached from a leaf node in the tree providing contextual information.

5.4.1.2 Symbol tables

The class *link* below is used to hold information on names that have been declared. The fields *manifest*, *retained* and *primitive* are used by code optimisers. *Manifest* is set if the value is constant and known at compile time. *Retained* is set if the value is referred to by a value that escapes the current scope, that is if block retention is required. The field *Primitive* is true if the value is implemented by the architecture.

```
type  link is structure(     Name       : string ;
                             Type       : TYPE ;
                             Initial    : PAIL ;
                             Manifest,
                             Retained,
                             Primitive  :
                             Constant   : bool
                             Addr       : location )
&
      location is variant(   New        : null ;
                             Stack      : StackPos )
&
      StackPos is structure( Frame,MSoffset,PSoffset  : int  )
```

Links are stored in symbol tables. Symbol tables are implemented as structures with the following type description:

```
rec type symbolTable is variant(  Empty : null ;
                                  Table : symTab )
&
SymTab is structure(  lookupLocal( string -> link )
                      lookupRec( string -> link )
                      InsertEntry( string,link )
                      EnclosingScope( -> symbolTable )
                      EnterScope( symbolTable- > symbolTable )
                      ScanScope( proc( link ) )
```

All contextual information is stored in symbol tables. The lookup functions allow names to be looked up in the current scope or in lexicographically nested scopes. Declarations are made in a scope level using the *insertEntry* procedure. The *EnterScope* procedure allows new scopes to be created at any scope level. The enclosing scope may be retrieved using the *EnclosingScope* procedure. A final procedure *ScanScope* is provided which applies the function supplied as a parameter to every link in a symbol table.

5.4.1.3 Control

The PAIL classes in this section all influence program flow control. There are constructs here that allow sequencing, choice, repetition and exceptions to be expressed, **and** and **or** are also included in this section since they are not strict in their second argument and thus, also affect flow control.

5.4.1.4 Assignment

```
type assign is structure( Lhs : link ; Rhs : tree[PAIL] )
```

This class denotes assignment. All assignments in the system are represented using this class. The field Lhs is a link denoting a location in the system.

source: E1 := E2

PAIL code:

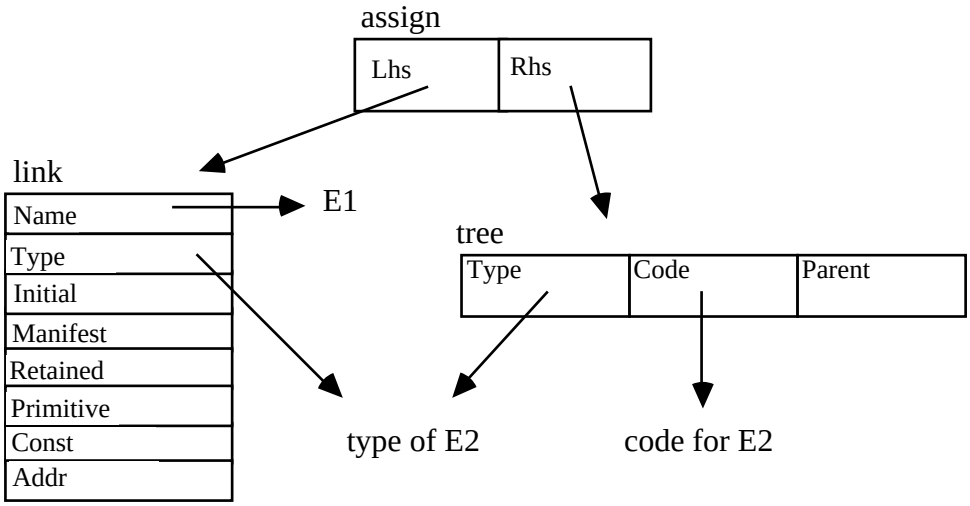


figure 1
PAIL for an assignment

5.4.1.5 Store Allocation

PAIL provides mechanisms to construct all the higher order objects in the system. This includes structures, vectors, abstract data types and images. One class is provided that represents the construction of each of these types.

Also included here is the class that denotes declarations. The example below shows how the PAIL code forms a graph allowing important information to be reached by more than one route. In general, the declaration will be part of the body of some scope level. That scope level will be represented by a *symbolTable* described above. The link for the declaration will be inserted in the symbol table using an *InsertEntry* procedure and will therefore be reachable from both the symbol table representing the current scope and the declaration. Thus, from a symbol table it is possible to find all the declarations made in that scope. Although this kind of information is contained in the source code it is tedious to extract.

```
type  Decl is variant(  simple : SimpleDecl ;
                        recursive : RecDecl )
&
      SimpleDecl is structure( Exp : PAIL ; Symbol : link )
&
      RecDecl is list[ Decl ]
```

source: **let** I = E

PAIL code:

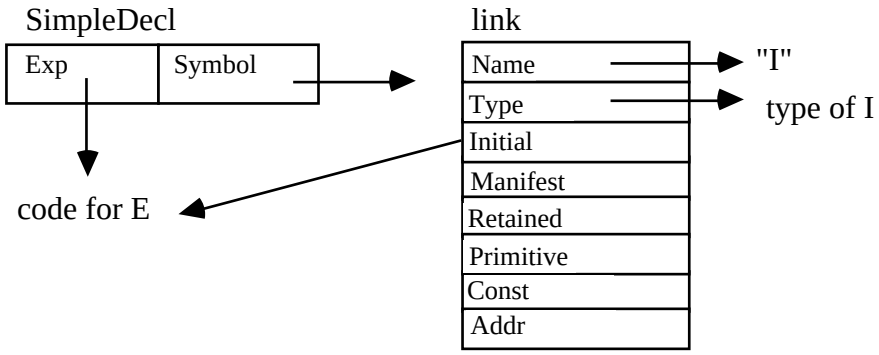


figure 2
PAIL representing a declaration

5.4.1.6 Indexing

The classes in this section allow objects to be indexed. Many different objects may be indexed in PAIL including structure, vectors, images, pixels and strings. Two indexing operations are provided these denote the indexing of objects to produce values and locations. Locations are yielded when a value is assigned to part of an object.

5.4.1.7 Scoping

Two mechanisms exist in PISA to introduce a new scope level, these are by the introduction of a new block or procedure literal. These are echoed in PAIL by the two classes, *block* and ProcDesc. The first of these modeling block entry and exit the second procedure declarations. Examples of these classes are shown below:

type block **is structure**(Symbols : symbolTable ;
Blockbody : PAIL)

source: **begin** E **end**

PAIL code:

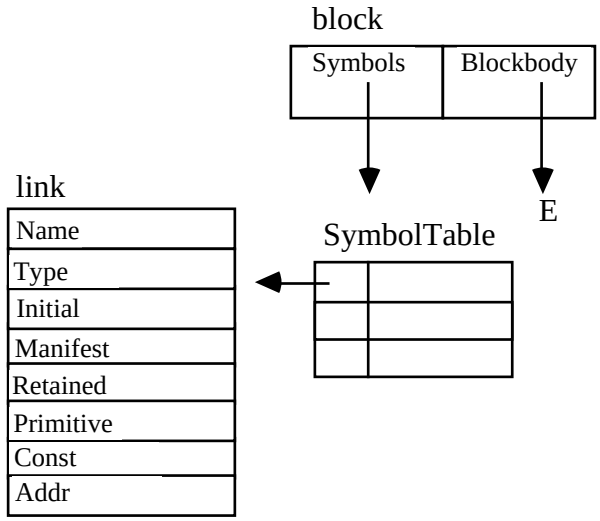


figure 3
block structure

```
type ProcDesc is structure(  Resultype   : TYPE
                             Parameters  : cons[ link ]
                             Body        : tree
                             Symbols     : symbolTable )
```

source: **proc**(E1,E2 -> E3) ; E4

PAIL code:

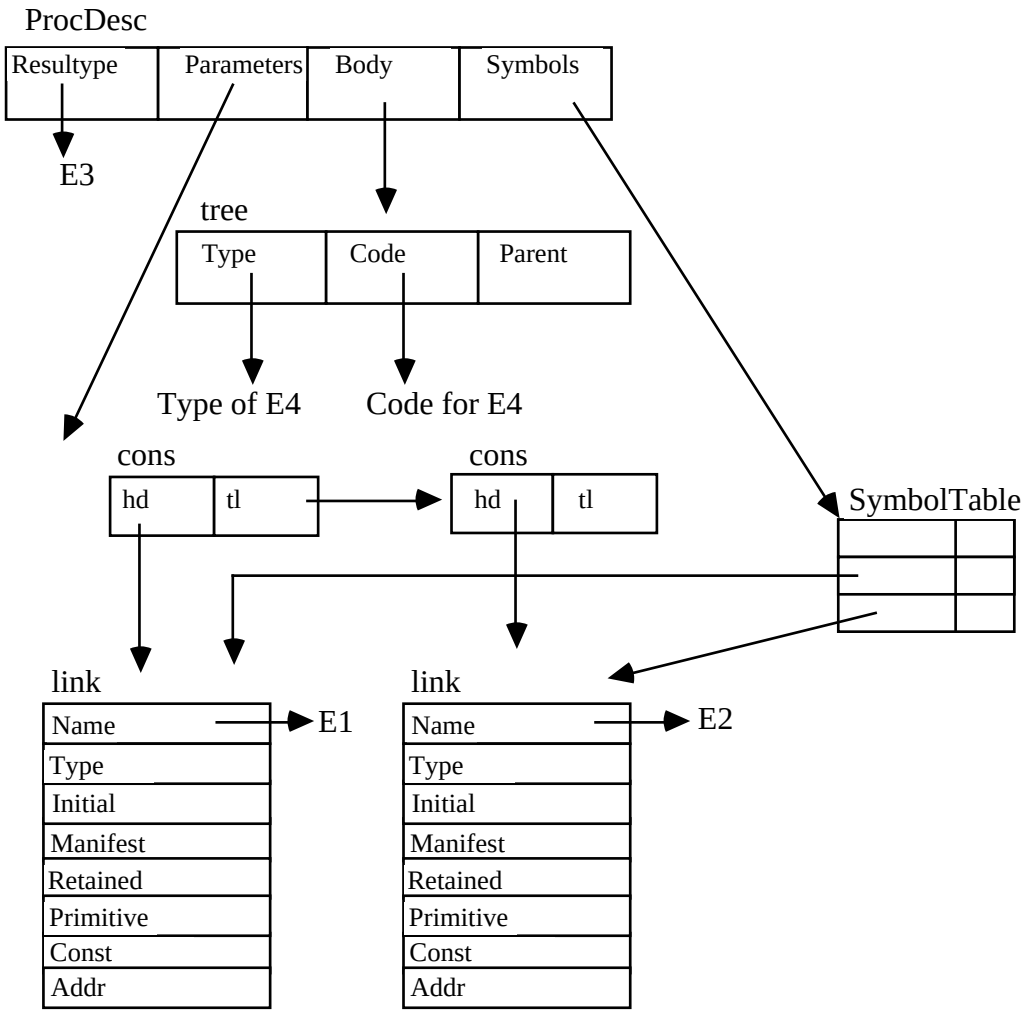


figure 4
procedure definition in PAIL

5.4.2 Support for system building

Since PAIL code is part of a persistent system it is potentially persistent. This means that rather than merely be an intermediate form used in the compilation system it may support many activities performed within the PISA system building domain. PAIL provides support in the following categories:

1. Code generation
2. Debugging
3. Optimisation
4. Syntax Directed Editing
5. Distribution
6. Protection

Each of these are discussed below.

5.4.2.1 Code Generation

Code generators in PISA take as a parameter PAIL code and produce executable code in some lower level language. During this process code generators may decorate PAIL trees with additional information. For example, during code generation the run time stacks are simulated allowing the address of identifiers to be calculated. This information is added to the address.info field of links whenever a declaration is encountered in the PAIL graph. PAIL aids the code generation process where multiple passes are required over data. An example of this is in the **rec let** construct of Napier,

```
rec   let a = proc( -> string ) ; b() &
      let b := proc( -> string ) ; c &
      let c := "hello"
```

In order to generate code for such a construct, it is necessary first to traverse that PAIL graph calculating the addresses of all the declarations. Once this has been achieved the code for the constructs may be generated. This code will, in general, involve the use of the locations whose addresses have been calculated during the first pass.

The provision of PAIL also aids the generation of efficient code. An example of where this is useful is in the code generation of polymorphic functions with type declarations in them. The following Napier example illustrates this,

```
let aproc = proc[ t ]( x : t -> structure( a : int ; b : t ) )
begin
      type tricky is structure( a : int ; b : t)
      tricky( 3,x )
end
```

This program declares a procedure, called *aproc*, quantified by a type *t*. Within the procedure body a structure class called *tricky* is declared. This structure has two fields, *a* and *b*, being of type integer the quantified type t respectively. The procedure creates a structure of type *tricky*. This structure contains the literal 3 and the polymorphic value *x* that is supplied as a parameter to the procedure. The structure created is returned by the procedure. In Napier polymorphic procedures are always specialised before they are used, for example writing,

```
aproc[ string ]
```

specialises the procedure to a procedure of type,

```
proc( string -> structure( a : int ; b : string ) )
```

and writing,

```
aproc[ real ]
```

specialises the procedure to one of type,

```
proc( real -> structure( a : int ; b : real ) )
```

In each one of these cases the addresses of the fields *a* and *b* are different (see chapter on PAM). As a consequence, code must be planted to dynamically calculate the addresses of *a* and *b*. This calculation would normally be planted in the code at the position corresponding to the position of the type declaration. However, this is not the most efficient place to perform the calculations. All the information necessary is known when

the procedure is being specialised. The calculations could therefore be performed at specialisation time, once, rather than potentially many times during each call of the procedure. This technique is sometimes known as hoisting and is commonly practiced by writers of applicative systems.

The provision of PAIL allows the necessary PAIL code to be referenced at a position in the code corresponding to specialisation. The code generator will calculate address information when it first traverses the type declaration in the specialisation code. When the code is reached a second time, in the procedure body, the address fields of the link structures will have already been decorated and no further code will be generated.

5.4.2.2 Debugging

Good diagnostic information is extremely important in any system. When an error occurs the user wants to know what happened and why. In order to do this several pieces of information are required. The first of these is what piece of code was running when the error occurred. This is especially important in a persistent system where it is possible to write seamless systems [mor85] in which the flow of control may transparently move from one compilation unit to another.

The second piece of information the user will require is the state of the machine when the error occurred. The state of the machine includes the values in locations and the dynamic call sequence leading up to the error.

In many systems, when an error occurs, the user is provided with a symbolic debugger which allows debris to be examined. Sometimes, even this is not possible unless the program has been compiled with flags set on the compiler. Newer systems recently appearing on the Apple Macintosh [thi86a,thi86b] provide an integrated environment that allows the writing, development and testing of programs. However, these systems only allow relatively small programs to be developed. In the Napier system we wish to provide good diagnostic information all the time - not just whilst the program is an experimental phase.

Diagnostic information is supported in the Napier system by PAIL. PAIL code contains all the information contained in the original source program. This information is augmented by the code generators which decorate PAIL code with address information. Thus, once the PAIL code has been code generated it contains all the information required by a diagnostic program.

References to the PAIL code are planted in code vectors containing the abstract machine code (see chapter 4). This binds the PAIL code and the abstract machine code together. When the code is executed a reference to the PAIL code is loaded into the currently active frame. This PAIL code includes the symbol tables discussed above and therefore contains the address information for the frame. When an error occurs it is possible to reconstruct the source code from the PAIL code allowing the user to examine what code was running when the error occurred. The symbol tables provide the necessary address information so that the diagnostic program may display the values bound to locations in the frame.

The frames on the dynamic call chain also contain pointers to the PAIL code for their source. This allows the diagnostic program to display the calling sequence to the user. Values in the static scope of procedures may be accessed via the display held in each frame. The address of these values may also be determined by the address information bound to the PAIL code.

5.4.2.3 Optimisation

PAIL may be used to assist program optimisation at various times throughout the program life cycle. The first of these is compile time. Many programmers add extra declarations to

programs for clarity of reading. These declarations do not necessarily map onto a location at run time. The best example of this is manifest constants. The programmer may write something like the following program segment,

```
let debug = false
 ...
if debug
then something
else somethingelse
```

The pail for this construct is as follows,

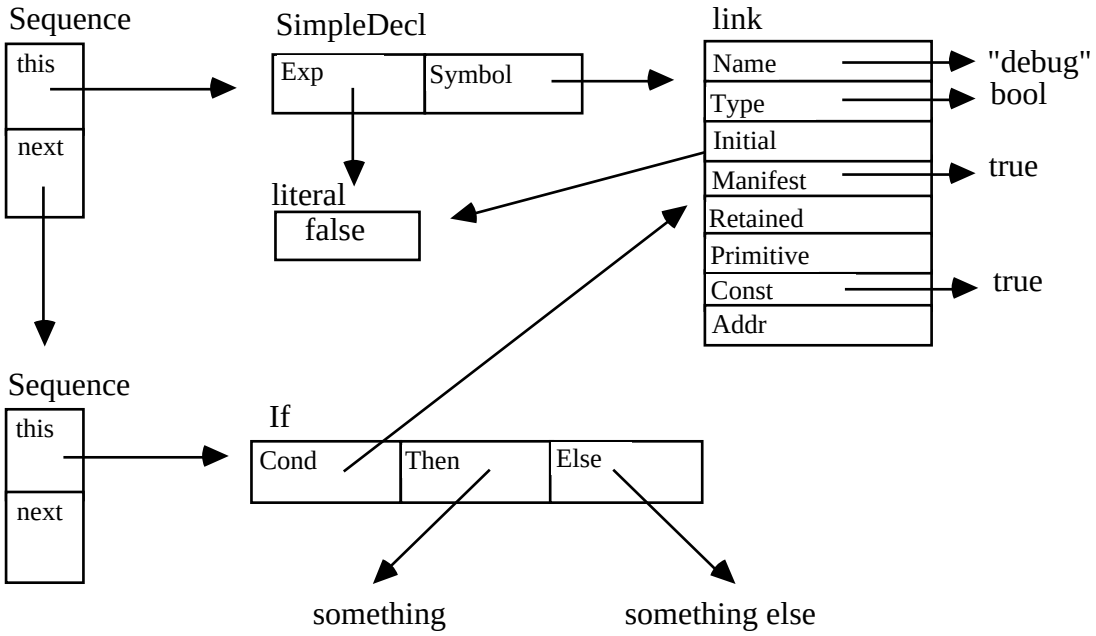


figure 5
unoptimised PAIL

Here, the result of the test for debug being true is always known statically i.e. at compile time. There is no reason to compile the code for the *something else* clause since it can never be reached. Likewise, the code for the test does not have to be planted at compile time nor executed at run time. Finally, no location associated with debug needs to be created. This technique is known as constant folding.

The detection of manifest constants may be performed statically in a single pass over the data. However if the code is compiled into the following program segment,

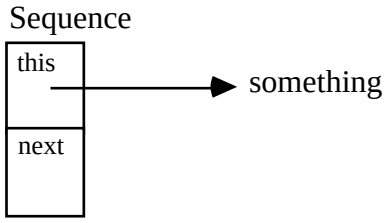


figure 6
optimised code

and an error occurs, the user cannot be shown the original code that he or she wrote. This leads to poor diagnostic messages being produced by the system. Poor diagnostic messages are often associated with optimising compilers and is a symptom of compilers discarding information. PAIL provides an optimisation node of the following type,

```
type optimised is structure(   Optimised,NonOptimised : PAIL
                               Info : string )
```

The field *Optimised* contains optimised code semantically equivalent to the original PAIL code contained in the *NonOptimised* branch. This class allows the original information contained in the program to be retained whilst providing improved code sequences that the code generators may follow.

The optimised PAIL for the above clause is as follows,

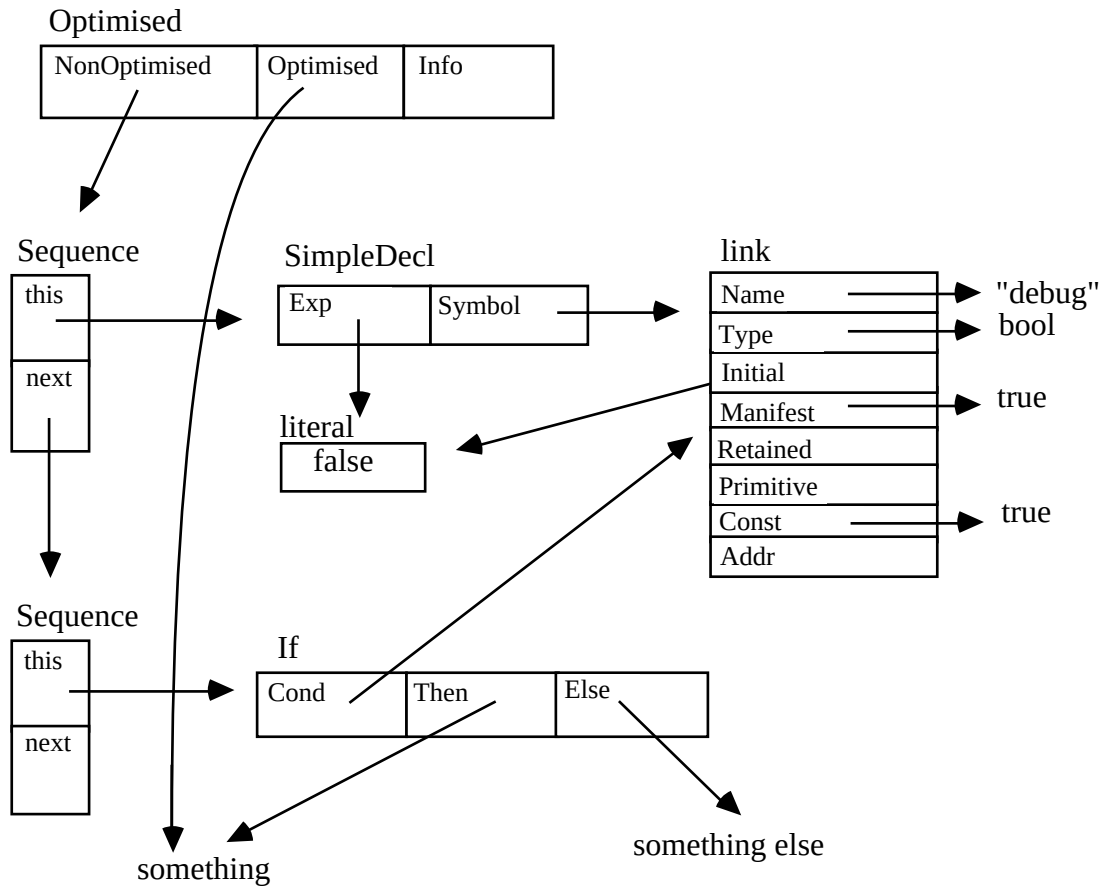


figure 7
optimised PAIL code

Some compile time optimisations cannot be performed in a single pass. In these cases, PAIL provides a framework on which to hang optimised sequences of code. One example of this is the hoisting technique discussed above.

PAIL code also provides support for late optimisation. PAIL code is referenced by Persistent Abstract Machine code. This allows optimisations to be made to the machine code very late. It has been suggested by Carrick and Munro in [car87] that this optimisation could be performed lazily by the system during quiescent periods. Late code optimisation could also be performed by a background process with the optimisation only being performed on frequently used pieces of code. In this way, the system could tune

itself dynamically depending on usage. This technique is only practical if complete source information is available at run time.

5.4.2.4 Syntax Directed Editing

PAIL is the abstract syntax for the languages supported by the Persistent Abstract Machine. The discussion above has centred on traditional compilers transforming source code into PAIL code. However, PAIL may be produced by programs other than compilers. In particular, it may be produced by syntax directed editors.

Syntax directed editors are tools which assist programmers to construct valid syntactic programs. They do this by providing the user with templates. Each template corresponds to one construct in the abstract syntax of the language. For example the user may be presented with the following template,

if <*clause:bool*> **then** <*clause:T*> **else** <*clause:T*>

representing an if clause in the language. The parts in italics are known as stubs and may be selected by the user and expanded into some concrete syntax. The syntax directed editor ensures that the user many only assign valid clauses to these stubs. In some syntax directed editors this is limited only to syntax checking in others such as the Cornell Program Synthesiser [tei81] type checking is also performed on the clauses substituted for stubs.

Templates like the one shown above may be mapped directly onto the abstract syntax of a language. For example, the construct above maps directly onto the PAIL choice construct,

type If **is structure**(Choice : PAIL ; Then,Else : tree[PAIL])

The following PAIL data structure may be constructed by a syntax directed editor for the program,

if E1 **then** E2 **else** E3

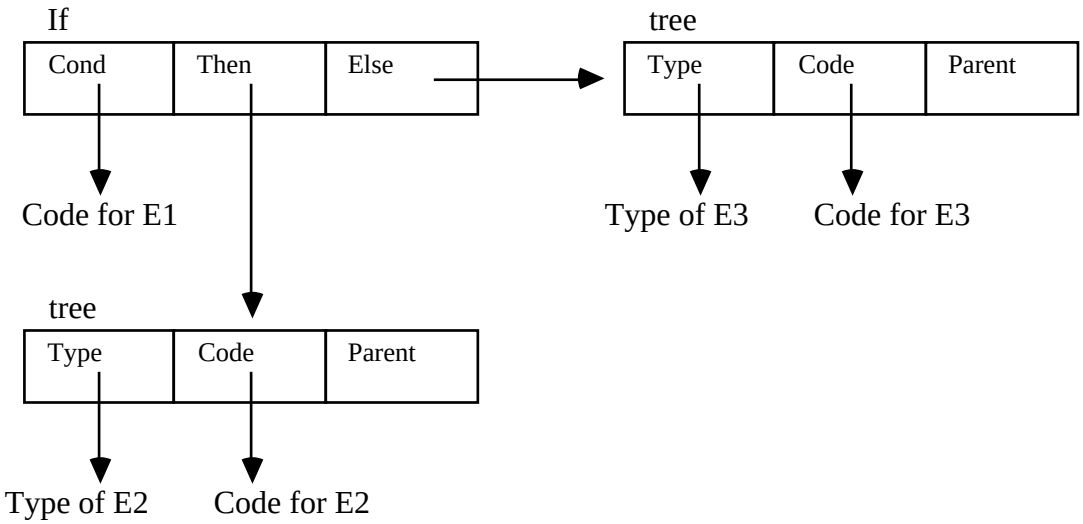


figure 8
if .. then .. else
represented in PAIL

Similarly, all the constructs in the Napier language may be mapped onto PAIL constructs. PAIL forms the ideal data structure on which to base a syntax directed editor for the PISA languages. This is expected to be the subject of future research.

5.4.2.5 Distribution

In a distributed persistent environment objects may be transparently moved from one machine to another. The principle of orthogonal persistence implies that users do not know where or how their data is stored. In other words, data is manipulated independently of the storage mechanism. In a distributed network of non-homogeneous machines this has serious consequences for the design of the architecture. Since the architecture supports procedures as first class data objects one of the objects that may be moved from machine to machine is code. This code must be capable of being executed on any of the machines in the network. If native code on one machine is moved to a different machine this clearly is not possible.

There is a need for a machine independent network language that describes procedures. PISA provides two of these. The first of these is Persistent Abstract Machine code. This code may be executed on any machine that has a PAM implementation. This code is not optimal since, in general, it will need to be interpreted. For this reason, a location exists within a PAM code vector to which alternative code vectors may be assigned. As an optimisation, native code vectors for any of the machines in the network may be assigned here. This location may be a reference to a vector of alternative code vectors if more than one alternative is required. In order to compile optimal native code for a machine, a second higher level representation of the procedure is required. PAIL provides a high level, machine independent description of procedures.

5.4.2.6 Protection

Protection of data from corruption and misuse is important in any system, however, it is especially important in a persistent system. The data on which a persistent program operates is not merely local data loaded into RAM. It may be long lived data that has been expensive to collect, this data is equivalent to data stored in a conventional database. It is essential that this data may not be corrupted by erroneous programs or malice.

The most common method employed to protect data, is the use of capabilities [nee74,wul74]. A capability gives a program the ability to perform some operations on a collection of data. The data may be viewed as a segment or object. The capability may then be considered as an access mechanism for a particular object. That is, it gives the program the ability to operate on data within a particular segment or object. The kind of operation the program may perform on the segment depends on the type of the capability. Capabilities come in different flavours such as read or write capabilities. A program cannot read some data within an object without a read capability for that object.

In capability based systems the protection of capabilities themselves becomes crucial. Some protection must exist in the system to ensure that capabilities are not forged either accidentally or deliberately by programs. The results in the protection mechanism having to be protected, resulting in considerable complexity in the architecture.

When a program attempts to perform some operation on an object the capability that the program has must be checked. The more complicated the protection regime provided by the system, the more complicated and therefore expensive this checking will be. This expense is extremely costly in terms of program execution time. Research performed on the Cambridge Capability Machine [nee74] estimated that 1000 operations were necessary between capability checks to obtain acceptable performance. This is required in order to keep the cost of context switching small in comparison with the amount of computation that is performed in a context.

In order to achieve this efficiency target, compilers are required to compile checks away by that coalescing small objects into larger ones. In this way, one capability may protect many small objects. If this is possible many objects may be accessed with only one capability check. However, this may only be achieved if objects can be grouped statically. Whilst this is true for some objects such as code vectors, it is not generally true, many objects are bound dynamically and must therefore have separate capabilities. This is an intrinsic problem of capability systems and cannot be overcome.

Another common solution to the protection problem is for each process within the system to have its own address space. Each process may only access data within this address space. The machine architecture prevents processes from accessing any data outside their own address space, protecting other data from misuse or corruption. This solution is common in modern operating systems like Unix.

The problems with each process having its own address space is twofold. Firstly, process creation is an extremely expensive operation. This expense has lead to the name heavyweight processes being adopted for this solution. Secondly, and more seriously, this solution complicates the sharing of objects between processes.

The protection of data from corruption is only necessary if the executing code cannot be guaranteed to operate safely. Code that has been produced by low level languages such as C or assembler may violate data. A simple example of this is shown in the following segment of C code,

```
disaster()
{
        *int a = 0 ;
        while( ++a ) *a = (int) a;
}
```

This procedure will overwrite all the addresses in the address space with their own address. If such programs are prevented from occurring the protection mechanisms described above may be discarded. This allows processes to share one address space without fear of objects being corrupted by rogue processes.

In PISA, the protection of data is achieved by a high level protection mechanism, this is provided by PAIL. All programs that wish to access the persistent store must be compiled into PAIL. PAIL is the lowest level at which access to the store is provided. The integrity of a PAIL program is checked by the code generator. If the program attempts some illegal operation on data the program will not be accepted by the code generator.

Not all languages may be mapped into PAIL, for example, languages that are not typed such as C or assembly language. The restriction of only allowing languages that may be compiled into PAIL into the architecture may seem restrictive but this is more than compensated for by the simplification of the underlying architecture.

5.5 Conclusions

PAIL is not an intrinsic requirement of the system, it is provided for engineering reasons. It has been shown how the provision of PAIL may support many different activities within the persistent architecture. These activities are wide ranging and include code generation, debugging, optimisation, syntax directed editing, distribution and protection.

6 The Compilation Environment

6.1 Introduction

The languages in PISA are supported by the system building domain. In turn, this domain is largely written in the PISA languages. This chapter will describe the most reflexive layer in the architecture - the compilation environment. The compilers in the system are written in and compile the PISA languages. Their construction heavily utilises the facilities available within the persistent environment.

The compilation system described consists of a number of different modules each of which are specified by a type signature. Many different instances of particular modules may exist side by side in the persistent store. Each implementation of a module may present a different user interface to the outside world. An instance of a compiler may be constructed by composing instances of the different components. Compilers that appear to the user to be quite different, for example, that compile different languages, may share much of the same code.

The system is a complete software architecture for the creation and construction of compilers. The paradigm used is that of a tool set comprising of many different components each satisfying an interface specification. The components are all plug-compatible with the architecture framework. That is any instance of a module may be replaced by any other satisfying the same interface. Components may be mixed and matched to build a compiler of any flavour required.

Components are all inwardly compatible, that is they present the same interface to the compiler architecture. However, they may present completely different interfaces to components outside the compiler architecture. This is achieved by wrapping all components in generator functions.

6.2 Advantages of using a persistent environment

The decomposition of a compiler into different sub-tasks has been well known for many years. Typically, compilers are described to undergraduate students as consisting of a lexical analyser, a syntax analyser, a code generator and so on [dav81]. However, these theoretical methods of constructing a compiler are often not strictly followed in order to gain improved performance. It will be shown that using a persistent store no significant overhead is incurred in space or time in building a flexible, modular system.

The space arguments will be examined first. Many programming languages [wir73,ker78] allow libraries of functions to be constructed. These are fragments of code that have been separately compiled and are thought of as being useful to a community of programmers. When an individual wishes to use one of these functions a binder is employed to **copy** the code in the library and bind it into a new program. Thus, if a compiler is constructed by building library functions every compiler constructed will have its own copy of the code. This is a crude form of software reuse. In such a system only copies of code are being shared and not instances of modules as in a persistent system.

In a persistent system, such as PS-algol [ps87], procedures may be stored in a type secure manner. Programs may link dynamically or statically to code in the database simply by indexing a structure class. In this way different programs may share instances of code rather than merely own a copy of the code. This method of building large systems is much more persistent space efficient than building libraries of functions and using a linker to obtain copies of the code. Furthermore, as the usage of a library function increases the greater the benefit from using the persistent store, since in a library system more and more copies will exist in the system but in the persistent system only one version of the code

will ever exist. Clearly, the persistent store subsumes the role of a conventional procedure library [atk85b,mor85]. The time overheads of using such a system will now be explored.

Since PS-algol supports first class procedures, closures may be stored in the persistent information space. The user finds a procedure by navigating the persistent graph from a root of persistence. Once a procedure has been found in the persistent store it is indistinguishable from one declared in the main program. In other words, there is no difference between a closure retrieved from the persistent store and one declared locally. This implies that by using the persistent store no penalty is paid for decomposing large programs into modules - provided that a functional interface is used.

One time penalty that may be incurred in using the persistent store is the time taken to navigate the store to find the appropriate procedure. This operation is equivalent to linking in a conventional system. The navigation of the persistent graph may be performed at many different times. If the navigation is performed at a time earlier time than call time, the user does not have to pay this time penalty every time the code is used. The time at which binding is performed is discussed later.

The previous paragraphs compare the persistent store to conventional technology but other benefits may be gained by using a persistent information space.

Complex data structures may be created without concern of how to map them onto a linear secondary storage medium. A good example of this, is the symbol table package discussed in the chapter on PAIL. The symbol tables model lexicographical scope and contain both complex type information and initialising code for declarations. All this information automatically persists because it is reachable from the executable code. If such a system was constructed using conventional software technology it would be extremely complex. A further benefit of using a persistent system, is the ability to control the way modules are bound together. This is discussed fully later.

6.3 Architecture Composition Rules

During the construction of the compiler tool kit a set of rules evolved. These rules may be considered to be a paradigm for constructing systems in a persistent environment. The rules are:

1. I/O independence
2. plug compatibility
3. binding independence
4. information hiding
5. encapsulation.

Each of these is examined below.

6.3.1 I/O independence

The rule of I/O independence states that only one module should directly perform I/O. The input and output of information should be routed via single input and output modules. If this rule is followed, modules constructed will have a much higher degree of usability. To demonstrate this, imagine a lexical procedure to parse a real number. Such a procedure may be found in most compilers. Typically, if the procedure encounters an error it will display an error message. The use of the procedure is therefore limited to applications where the displaying of an error message is permissable. The procedure could not, for example, be used in a desk top calculator since it would destroy the display. If the procedure takes an error displaying procedure as a parameter it will be of greater utility, since an appropriate error procedure may be supplied by each application using the procedure. Careful control of the hidden interface of a module, not just the published

apparent interface creates an environment in which more reuse is possible and therefore one in which software is cheaper to produce.

6.3.2 Plug Compatibility

The second rule of plug compatibility states that each module should have a well defined interface and that modules with the same interface may be freely substituted for each other. It is this rule that allows us to create a whole family of compilers by specifying interfaces and by having a number of different instances which conform to those interfaces. Generators for the various compiler modules are placed in the persistent store independently. Typically, more than one instance of each module can be found in the persistent store. In order to construct a particular compiler, these components need to be joined together. A good analogy is having several jigsaws all cut using the same pattern. A new jigsaw may be constructed by selecting pieces from different jigsaws. Provided that the pieces are placed in the correct positions, a jigsaw displaying a new picture may be created. One configuration of the architecture may be viewed pictorially in figure 1.

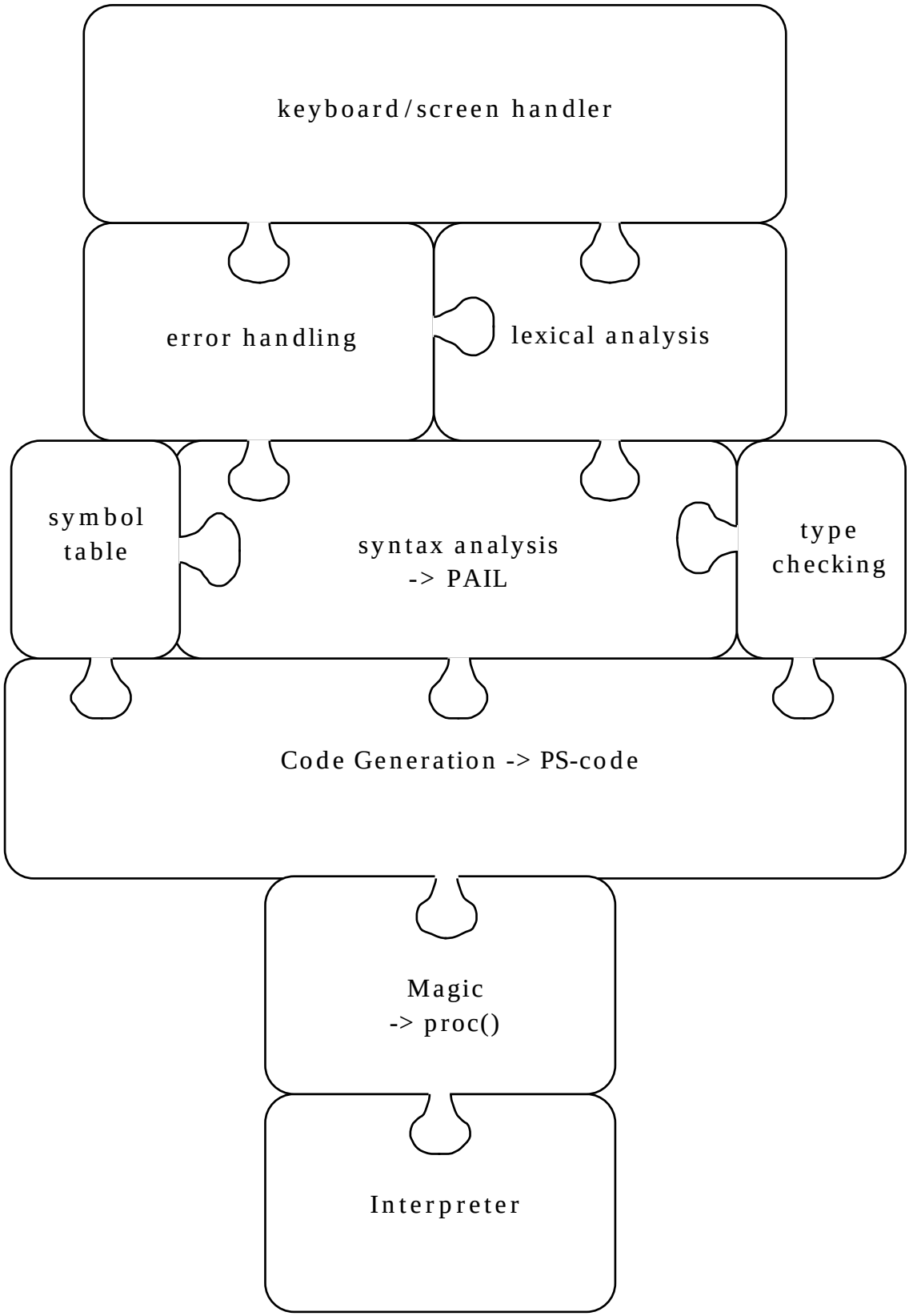

keyboard/screen handler
error handling
lexical analysis
symbol table
syntax analysis -> PAIL
type checking
Code Generation -> PS-code
Magic -> proc()
Interpreter

figure 1

The compiler tool set consists of many components. The interface of each component is well defined so that new versions of any of the components may be easily created and used in a compiler. The modules that comprise the components of the Napier compiler discussed below.

The modules provided within the architectural framework include: input and output modules; error handlers, lexical analysers, syntax analysers, type checkers, code generators and symbol table packages.

The input and output modules are responsible for providing an abstraction over the file system or persistent storage facility. These modules essentially implement a functional interface to input and output streams. Thus, by substituting the input and output modules, one compiler may operate against the file system whilst an otherwise identical compiler may operate within the persistent environment.

This is a good example of how, using functional abstraction a module may exhibit different interfaces. The procedures supplied to the compiler tool kit must be of a certain type to satisfy the interface requirements. However, here we require one procedure to take a file as a parameter and the other to take say a string as a parameter. This may be achieved by encapsulating the required procedure in a generator function and partially applying it. For example, in this case we may have two generators hat return functions of the required type which have either a filename or the string bound into their environments. The generators are of different types, but that is not important since it is only the generated functions that must satisfy the architectural requirements The technique of using generator functions is discussed later in this chapter.

A single module is responsible for noting and reporting all compilation errors. This module interfaces with the output module which displays all output to the user.

The lexical analyser interacts with the input module to deliver a stream of tokens to the syntax analyser. The lexical analyser is parameterised by the terminal symbols of the grammar, the syntax analyser is parsing. This ensures that efficiency is preserved whilst maintaining module independence.

The syntax analyser does not interface with the primitive input or output systems. Instead, the most basic input is provided by the lexical analyser The interface between the syntax analysis module and the code generation module is provided by the Persistent Architecture Intermediate Language, PAIL [dea87]. Thus, the syntax analyser does not interface with the output system .

A type module is responsible for creating representations of programming language data types. A set of constructors and selectors are provided to create and decompose complex data types. The ability to perform type checking is also provided by this modules. To do this, the module provides predicates to test things, such as, equality of types. The implementation of the type system is completely contained within this module. Outside this module, nothing has knowledge of how types are implemented.

The code generators in the system accept PAIL from syntax analysers or other tools such as syntax directed editors. The provision of PAIL makes it possible to write language independent code generators. It also allows experiments in language design and language implementation to be carried out independently and in parallel. This is only possible because the machine has been decoupled from the language. The code generator does not output its results directly to the file system or persistent store. Instead, is produces a vector of integers containing PAM code. The PAM code is processed by one of two modules. The first of these is a code planter which outputs the code to the file system. The other, called magic, turns code into a closure within the system. This is the fixed point in the system and has to be written in a lower level of language, it is discussed later.

The compilation components, described above, are viewed as aids to the construction of a total compiler. Of course, they do not have to be used as such, being applicable in a number of other applications including spreadsheets (parsers and lexical analysers) and word processors (lexical analysers). Partial compilers may also be constructed, for example, merely as syntax checkers. Such tools have proved highly useful in the development of new language processors and type checkers.

6.3.3 Binding Independence

The third rule of system construction, binding independence, is not commonly practised by programmers in persistent systems. In order for us to understand this rule let us consider two procedures, written in PS-algol.

```
let example1 = proc()
begin
        structure container( proc() somethingUseful )
        let aContainer = s.lookup( "usedByExample1","database" )
        let usedByA = aContainer( somethingUseful )
        usedByA()
end
```

example 1

In this example the procedure first declares a structure type. This introduces a class along with some selectors and a constructor. In the second line of the procedure an object is looked up, using the key "usedByExample1", from a database called "database". Databases in PS-algol provide a persistent root. They have an associative lookup table attached to them by convention. The *s.lookup* function returns a pointer to the object associated with the named parameter. In the procedure, we are assuming that *aContainer* points to an object of class *container* defined in the first line. The procedure stored in the object is retrieved by indexing the structure. Lastly the procedure is applied. Notice that this procedure dynamically looks up the database to retrieve the procedure every time it is called. Such an action may be required in a development system where the most recent version of a procedure from a library is required. This code is typical of code written by PS-algol programmers.

```
let example2 = proc( pntr aContainer )
begin
        structure container( proc() somethingUseful )
        let usedByA = aContainer( somethingUseful )
        usedByA()
end
```

example 2

example2 is similar to *example1*, it also applies a procedure obtained by dereferencing a structure of class *container*. There are a number of important differences between *example1* and *example2*. These are examined below.

example1 has the strings "usedByExample1" and "database" bound into it - leaving the user with no option but to use the procedure stored in the appropriate table. It also leaves the user of the procedure, with no option other than to bind dynamically to the data every time it is executed. In the second procedure no information, apart from the structure class, has been bound into the procedures' closure. This allows the procedure to be used in a number of different ways which will now be examined. If the semantics of the first example were required the user could write,

```
let synthsiseExample1 = proc()
begin
        let aContainer = s.lookup( "usedByExample1","database" )
        example2( aContainer )
end
```

example 3

As in *example1*, this procedure looks up the database every time, thus retrieving the most recent version of the procedure.

A user may, however, wish a static system with which to experiment without having possible changes to other modules affect the experiment. In such a case a static bind is required. This may also be written using *example2* by producing a new procedure with the original function bound into its closure. This is illustrated in example 4,

```
let staticExample =
begin
        let aContainer = s.lookup( "usedByExample1","database" )

        proc() ; example2( aContainer )       ! this is the result
end
```

example 4

In this example, the container is looked up only once.The procedure in the final line is exported as the result of the block and bound to the identifier *staticExample*. The object pointed at by *aContainer* is statically encapsulated in the scope of the procedure. This is another example of the power of block retention.

The procedure *synthsiseExample1* looks up the database every time - giving us a dynamic bind whereas the procedure *staticExample* has the structure instance *aContainer* in its closure, giving a static bind. In the compiler architecture all the modules have been written in the style of *example2* giving the user the option of composing programs statically or dynamically. This is illustrated with examples later

6.3.4 Information Hiding

The fourth rule, is that of information hiding. This has been known for many years and is commonly practiced by users of abstract data types [lis74].

The rule that modules should be totally encapsulated is best understood by example. Consider the following program segment, once again in PS-algol. The procedure saves an integer and returns the last integer saved.

```
let saved := 0

let saver = proc( int this -> int )
begin
        let temp = saved
        saved := this
        temp
end
```

example 5

If two processes were to use such a procedure concurrently, the outcome would be nondeterministic. This potential problem may be overcome by encapsulation. One way of doing this is by wrapping the procedure in a generator as in example 6.

```
let saverGen = proc( -> proc( int -> int ) )
begin
        let saved := 0                  ! this is encapsulated

        proc( int this -> int )         ! this is the procedure returned
        begin
                let temp = saved
                saved := this
                temp
        end
end
```

example 6

As a result of this mechanism every procedure wishing to use saver, may do so safely by calling *saverGen* to obtain a saver with its own store. In this way, procedures may share code without having to share state. Notice that every saver produced has its own copy of the variable saved. Although this technique has been well known to researchers in persistent languages for some time it is, unfortunately, not commonly practiced. In the compiler tool set all of the modules are encapsulated in a generator so that each instance of a module operates entirely within its piece of store.

The rules of I/O independence, plug compatibility, binding independence, information hiding and encapsulation have proved to be invaluable in constructing the compiler tool kit. In the following sections we will see how these rules have been applied in practice.

6.4 Compiler Composition

Below a compiler constructed from components is shown. The components are provided as parameters. The compiler is a conventional file based compiler, it reads input from one file and produces another file containing executable code, providing that the compilation was successful. It reports the compilation errors if it was not.

```
let compiler = proc(   input                          ! the types of these
                       errors                         ! parameters are not given
                       types                          ! for brevity
                       lex                            ! these are all the generator
                       syntaxGen                      ! functions.
                       codeGen
                       planter
                       environment
                       filename )
begin
      let source = input( filename )                  ! get an input pack
      let syntax = syntaxGen(  options(),             ! compiler options
                               source,                ! the input pack
                               errors,                ! error reporting module
                               types,                 ! the type checker
                               lex )                  ! lexical ananyser
      let local.env = environment( Create.scope )( environment )

      let this.code = syntax( local.env )             ! syntax analysis
      ! this code is either an error.pack or PAIL
      if this.code is error.pack  ! check errors
      then  write "**** Compilation fails ****'n",
            "No of errors = ",this.code( No.errors )(),"'n"
      else
      begin
            write "***** Compilation Succeeds *****'n"
            let c.file = codeGen( this.code,global.env )        ! code generation
            planter( code.f.name( file.name ),c.file )    ! output code to file
      end
end
```

example 7

The compiler, shown above, is a typical compiler that may be found in the PISA system. The simplicity of this complete compiler, is due to the fact that it operates by taking as parameters the plug in components provided by the architecture. First - an input module is generated by applying a generator function with a filename as a parameter. This is in turn supplied as a parameter with other generator functions to the syntax analyser generator. The syntax analyser generator is applied to provide a syntax analyser, with the components supplied as parameters to the generator bound into it. When this function is applied with an environment, the program is parsed and some output is generated. This is then tested to find out if the compilation was successful. If it was unsuccessful an error message is displayed and the compilation is terminated. If not, code is generated by the code generator, which is then passed to the planter which outputs the code to the file system.

Compiler components may be bound together statically or dynamically depending on the choice of the programmer. In some cases, for example when a new language system is still being developed, the user may want the most recent version of a particular module to be used. In this case, the programmer would use dynamic binding to combine the components as follows.

```
let dynamiccompiler = proc( string filename )
begin
        let input = s.lookup( "input",comp.db )( Input.gen )
        let errors = s.lookup( "error",comp.db )( Error.gen )()
        let types = s.lookup( "types",comp.db )( Type.gen )()
        let lex = s.lookup( "lex",comp.db )
        let sa = s.lookup( "sa",comp.db )
        let cgen = s.lookup( "cgen",comp.db )
        let planter = s.lookup( "planter",comp.db )( Planter.gen )()
        let global.env = s.lookup( "global.symbol.table",comp.db )

        compiler(  input,errors,types,lex,sa
                   cgen,planter,global.env,filename )       ! call compiler in fig 7
end
```

example 8

At other times, a static system is required. This may be achieved by statically combining components into a compiler. This is shown in example 9,

```
let staticcompiler =
begin
       let input = s.lookup( "input",comp.db )( Input.gen )
       let errors = s.lookup( "error",comp.db )( Error.gen )()
       let types = s.lookup( "types",comp.db )( Type.gen )()
       let lex = s.lookup( "lex",comp.db )
       let sa = s.lookup( "sa",comp.db )
       let cgen = s.lookup( "cgen",comp.db )
       let planter = s.lookup( "planter",comp.db )( Planter.gen )()
       let global.env = s.lookup( "global.symbol.table",comp.db )

       proc( string filename )        ! this is the result of the block
       begin
             compiler(  input,errors,types,lex,s                    ! call the
                        cgen,planter,global.env,filename )          ! compiler above
       end
end
```

example 9

Often, both of the binding mechanisms shown above are required in the same system. An example of this is when a compiler under development is released for others to use. A release compiler would consist of the modules bound statically together, as shown in example 9. Changes made to database modules cannot affect this release version, since it has a copy of the code bound into its closure. In the same system a development compiler like the one in example 8 may exist. In this compiler changes made to the persistent store will affect the compilers execution immediately.

The need for control over binding is of major importance in a persistent environment. The insight gained here, provided the motivation for the design and implementation of environments discussed in chapter 2**.**

6.5 First Class Compilers

The compilers we have seen so far all accept input from and return output to the file system. The PISA architecture, is totally self contained with no reliance on the external operating system. It must, therefore, support all programming activities. To this end, a

mechanism is required to introduce compiled programs into the current environment. Doing this introduces a new set of problems to type checking and machine design.

The first problem which must be overcome with compilers that introduce new programs into the system is assigning a type to the compiler. Since there are an infinite number of valid programs, the compiler must produce a value that is a member of an infinite union. The bindings to members of unions are dynamic bindings, this fits well with our requirements for typing the compiler. PS-algol contains one infinite union called **pntr**. It is the infinite union of all labelled cross products. The newer language, Napier, provides two infinite unions **any**, the infinite union of all types, and **env** the infinite union of labelled cross products. These allow a type to be prescribed to a compiler of this kind.

A compiler that introduces the compiled code into the environment is provided in the PS-algol system as a standard function. It is defined as follows,

```
let compiler = proc(  cstring filename,
                      cpntr proc.holder
                      -> pntr )
```

This compiler, takes as a parameter, the name of a file containing source code. Using the compiler tool set, the source could be contained in a string or any other data type. The compiler is also supplied with a pointer to a structure. The class of this structure indicates the expected type of the compiled code. If the code contained in the file is of this type and the compilation is successful the compiler will place the result of the compilation in a field of the structure and return a pointer to it. Otherwise an error structure is returned. An example clarifies this,

Suppose that the file XXX contains the following string,

```
"let aprocedure = proc() ; write "hello" ?"
```

The user may call the compiler, from another program, as follows,

```
structure procontainer( proc() aprocedure )

let dummy = proc() ; {}
let holder = procontainer( dummy )
let compiled = compiler( "XXX",holder )
if compiled is procontainer do
        compiled( aprocedure )()
```

example 10

The first line of the program declares a structure class capable of storing a void procedure. A structure of this class is passed to the compiler as a parameter. If the compilation of the program in file XXX is successful; a procedure containing the code described in the file XXX will be assigned to the field aprocedure of the structure. The success of the compilation is tested using the is predicate in the fifth line. If it is, the procedure is extracted from the structure and applied. This example will therefore write "hello".

During a compilation the compiler, written in one of the PISA languages must turn a sequence of bytes representing a code vector into a procedure in the language. In the compiler tool set a module called *magic* turns a vector of op-codes into a procedure. This capability must be carefully protected in the system. The magic module has the following type in the PS-algol compiler,

```
structure PScode(  *int Code.vec;
                   *string String.vec ;
                   *pntr Proc.vec ;
                   int Ms.size,Ps.size )

let magic = proc( pntr PScode -> proc() )
```

This procedure takes as parameters, a vector of integers containing byte codes and some house keeping information, produced by the code generator and returns a procedure.

In order to coerce the function produced by magic into a procedure of any type a function called *coerce.proc* is used. It has the following type,

```
let coerce.proc = proc(  proc() Proc ;
                         cpntr result
                         -> pntr )
```

Once the compiler has checked the integrity of the source code produced and created a closure using *magic* it uses this procedure to put the closure into the structure provided. The two functions, *magic* and *coerce.proc* implement the functionality described in example 10. Both these functions are written in low level code since they cannot be written in PS-algol. The compiler provided as a standard function in PS-algol system only compiles procedures. A better system would compile code to produce objects of any type. This ability is provided in the Napier system.

The more sophisticated type system provided in Napier makes it easier to type the compiler. There is no need to declare a structure type in order to introduce a new type into the system, instead the type **any** is used. The Napier compiler with the same functionality as the PS-algol compiler described above has the following definition,

```
let compiler = proc( source : file -> any )
```

The user may project from the any using a project clause. The example above with the same file XXX would be written,

```
let compiled = compiler( "XXX",holder )
project compiled as choice onto
proc()   : choice( aprocedure )()
default        : {}
```

example 11

This example is less cumbersome than the corresponding PS-algol example. The project statement does case analysis on the type that has been injected into the any. The renaming of the value being projected is a mechanism to provide a constant binding on which to project. A file or any other source medium containing the source of any data type may be compiled using this compiler.

Compilers that introduce values into the system are known as callable compilers, since they may be called from within the persistent environment. Such compilers have proved to be of great utility in providing a richer programming environment; whilst maintaining a strict type regime. Callable compilers have been used to provide an object browser [dea88b] discussed in the next chapter and an adaptive relational database implementation [coop87].

6.6 Interactive Compilers

A Callable Compiler which reads from the users console and immediately executes the code produced is known as a compile and go compiler. Such a compiler may be used in a way which resembles a shell [bou78,joy80]. This shell is of much greater power due to the support of a full programming language. Compile and go environments have been provided for applicative languages, like SASL, and so called A.I. languages such as LISP, for a number of years. These environments allow programs to be developed only using one language without the need for a command language. A compile and go environment to support the PISA languages introduces new problems discussed below.

To clarify, the aim is to provide a system in which the following dialogue is possible interactively,

```
user>       let a = 3
system>     <nothing>
user>       let b = 7
system>     <nothing>
user>       write( a * b )
system>     21
```

example 12

The global naming of data is one of the problems originally intended to be tackled by PISA. Global naming is provided in PS-algol by the named databases in the system, and in Napier, by the provision of a persistent root. Only the Napier case will be considered here for simplicity. The persistent root, called *ps*, is the only global name provided in the system.

The compiler must be called every time the user enters a clause. In practice the user must specify a terminator to indicate to the system that he or she wishes the entered text to be processed by the system. If the terminator was a full stop character the interaction above would look as follows,

```
user>       let a = 3.
system>     <nothing>
user>       let b = 7.
system>     <nothing>
user>       write( a * b.)
system>     21
```

example 13

In such a system, every clause entered by the user is a separate compilation. Therefore, no global naming is available on a compilation unit basis. The only global naming is via the persistent root *ps*. It would be extremely tedious for the user to always have to specify the full path to the data. Such a system is shown below,

```
user>     let new = environment()  ! create an environment
          in new let a = 3         ! put binding <a:3> in it
          in ps let new = new.     ! put environment in ps
system>   <nothing>
user>     use ps as new : env in   ! specify where the env is
          in new let b = 7.        ! add binding <b:7> to it
system>   <nothing>
user>     use ps as new : env in   ! specify where the env is
          use new as a,b : int in  ! project  bindings from env
          write( a * b.)           ! write out the added value
system>   21
```

example 14

This kind of interaction with a system would persuade the user that interactive systems were not for him or her. However, the interactive system may transparently perform the activities shown above. An environment is used to support top level bindings in the interactive system. It serves the same purpose as the environment called new above. Whenever a user makes a top level declaration in the system, it is transformed into a declaration in this environment. All other declarations, for example declarations within blocks or procedures are unchanged. This environment provides us, with the unbounded space we require to support declarations in an interactive system.

The interactive system must record all declarations made in the top level environment. When an unresolved name is found in the interactive compilation system, the name is checked in the outermost environment. If the name is in the environment and has the appropriate type, the value associated with that name is used in the code produced. In this way, the top level declarations are transparently simulated using environments. This allows dialogues like the one shown in example 13 to be supported.

There is a small time cost associated with looking up values in environments and performing type checking. However, the time cost is small in comparison to the real time domain of the user. The retrieval of values from environments, and the addition of bindings to environments is only required for top level user definitions. All other declarations, including those made within top level definitions, will have space allocated for them in the usual manner

6.7 Conclusions

By combining instances of modules with slightly different functionality a rich set of tools may be provided. These modules may be bound together in different ways to provide an environment suitable for experimentation or production.

The compiler toolset has been implemented in PS-algol. The current tools available built from the components include callable compilers, batch compilers, interactive compilers and persistent information compilers. Currently, syntax analysers exist for PS-algol, Napier and Hope+ [per87,mcn87].

7 Browsing

7.1 Introduction

A mechanism to display data structures is often required in database and programming language systems. Usually this requirement is satisfied by a tool known as a browser. Browsers are used extensively to traverse through the data structures found in database systems, often to gain insight into the behaviour of a complex and highly dynamic system. They are also of great use in debugging and, if powerful enough, can be used to repair erroneous data structures which may contain valuable information.

Browsers which operate on programming language data structures are equally useful for the same reasons. Unfortunately, they are seldom provided as programming language support tools. In a persistent environment, the data structures of the programming language and the long term data structures are the same. In such an environment, browsing tools have been observed to be especially useful. The chapter describes one method of constructing of such a tool.

In most powerful programming and database systems there are a potentially infinite number of types which may occur in the system. This presents a problem when writing a program to browse over them. In general, one cannot write a static program to anticipate all of the types that may occur without resorting to some magic or a second level of interpretation. Object-oriented programming languages [gold83,bob81] with a few exceptions [sch85] avoid this problem by resorting to a combination of conventions and dynamic typing. For example, one solution to this problem would be for every instance of a class to have a print method. This is not a safe solution to the problem since a print method may be overwritten by a method which performs a completely different function.

In a persistent programming language PS-algol [ps87], it is possible to write a browsing program which displays the language's own data structures. This may be achieved without resorting to conventions, having built-in functions or using second level interpretation. Using a mechanism in the language, the program is, however, allowed to discover the types of objects. The technique demonstrated utilises a compiler which is a *first class citizen* in the language environment. This kind of compiler is discussed in the chapter 6.

PS-algol has powerful raster and vector graphics facilities which are an integral part of the language, these are discussed chapter 3. Only one graphics procedure, the *menu* function is used in the browser, this is discussed below for completeness.

7.2 Graphics

The *menu* function, like many of the predefined functions, is written in PS-algol. The procedure *menu* generates another procedure which interacts with the user by displaying a menu on a bit mapped screen at the coordinates supplied as a parameter. This menu will have title *title* and entries taken from the vector of strings called *entries*. When the user makes a selection from the menu the corresponding procedure from the vector of procedures *actions* will be called. *menu* is defined as follows:

```
let menu = proc(  string title ;
                  *string entries ;
                  *proc() actions
                  -> proc( int,int ) )
```

7.3 A Simple Browser

When presented with a pointer to an instance of a structure class such as:

structure *x*(**int** *a* ; **string** *b* ; **pntr** *c*)

the browser will present the user with a menu like the one in Figure 1 which allows the user to examine the values of *a* and *b* and allow the pointer *c* to be browsed. The entry with the stars allows the user to return to the previously examined data structure (if any).

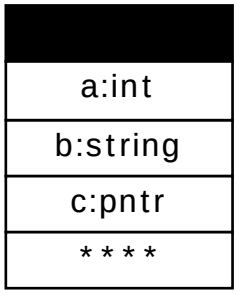


figure 1

A PS-algol program to draw the menu shown in Figure 1 may look like the program below:

```
let traversex = proc( pntr p )
begin
        structure x( int a ; string b ; pntr c ) ! declare the structure class
                                                 ! which this proc displays

        let return = proc() ; {}                 ! a do nothing procedure

        let strings = @1 of string [  "a:int",      ! declare a vector of strings
                                      "b:string", ! with lower bound 1
                                      "c:pntr" ,  ! for the menu entries
                                      "****" ]

        ! Next declare a vector of procedures - the menu actions

        let procs = @1 of proc()  [  proc() ; write p( a ),     ! display the int a
                                     proc() ; write p( b ),     ! display the string b
                                     proc() ; Trav( p( c ) ) ,  ! browse the pntr c
                                     return                     ! return-do nothing
                                  ]

        let this.menu = menu(   "x",          ! the title
                                strings,      ! the entries - a vector of strings
                                procs )       ! the actions - a vector of procs

        if p is x
        then this.menu( 20,20 )               ! display menu at 20,20
        else Error()                          ! take some error action
end
```

example 1

The procedure *traversex* will display any structure of class :

x(**int** *a* ; **string** *b* ; **pntr** *c*)

but will fail with any other structure class. If *x* is a member of some finite union , this procedure could be generalised to handle any of the members of that union. However, if *x* is a member of an infinite union, such as the PS-algol type **pntr**, all the structure classes that the procedure may encounter can never be statically anticipated. The procedure *Trav* which is called from the menu is faced with this problem since the member of the infinite union to which *c* refers is unknown.

If a mechanism existed to discover what class a pointer is pointing at then a procedure of the appropriate type could be selected and called in order to display that instance. One way of engineering this in PS-algol would be to maintain a table containing procedures indexed by the appropriate class. This table could be indexed by the structure class that the procedure could display. Notice that although the procedures in this table operate on different classes their interface is the same; that is they are all of type:

```
proc( pntr )
```

In PS-algol a predefined function *class.identifier* is provided which allows the structure class that a pointer is pointing at to be discovered. It returns a string representation of the class and is defined as follows:

```
let class.identifier = proc( pntr p -> string )
```

For example, if the following program is executed,

```
structure x( int a ; string b ; pntr c )
let p = x( 7,"abc",nil )
write class.identifier( p )
```

the string,

```
x( int a
string b
pntr c
)
```

would be written out.

Suppose that a table called *trav.table* has been created which contains associations between class identifier strings and pointers to structures of class,

```
structure trav.pack( proc( pntr ) trav )
```

which contain a procedure to display an instance of the appropriate class. This may be viewed pictorially in figure 2.

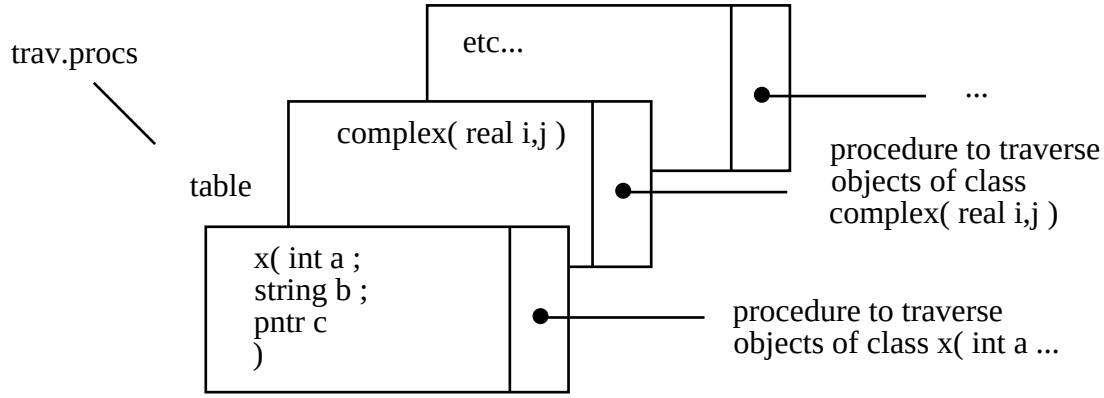


figure 2

A generic *Trav* procedure capable of traversing any data structure may be written using the technique described above like so,

```
let Trav = proc( pntr p )
begin
        structure trav.pack( proc( pntr ) trav )

        let class = class.identifier( p )
        let look = s.lookup( class,trav.table )
        if look is trav.pack  then look( trav )( p )
                              else Error()
end
```

example 2

This browsing procedure can now display and browse over any class whose display procedure is contained in the table. The procedures in the table look like the procedure shown in example 1. Notice that new procedures may be added to this table without altering this program.

It would be preferable if the traverser program could do better than simply report an error when a new structure class is found - but what options are open to it? The procedure could prompt the user of the browser to write a procedure which traverses the new structure class. If the procedure displayed the structure class of the new structure to the user, all the information needed to write such a procedure would be available. This procedure would need to be edited, compiled, debugged and entered into the *trav.table* table (equivalent of linking) by the user. This process is tedious and repetitive since the same procedure must be written each time with small variations. If the user were traversing a graph in a development environment this problem would be heightened since the user may be changing the structure classes frequently as a design was refined.

7.4 A First Class Compiler

A better solution to the problem is for the traversal procedure to write the procedure to traverse over the new class. It has all the information necessary to construct a procedure to display the new class. However, it must be able to compile and link the new code into the running program. In order do this another function is required in persistent environment - the compiler. The callable compiler is discussed in the previous chapter.

In order to simplify the following discussion it is assumed that the PS-algol callable compiler is of the following form,

```
let compiler = proc( string source; pntr p-> pntr )
```

The compiler is passed the source code to be compiled. It is also passed a pointer to a structure class which should have a field of the same type as the compiled code. If the procedure is type compatible with the structure class, and the compilation is successful, the compiler will put the compiled procedure into that structure instance and return a pointer to it, otherwise it will return a pointer to an error structure.

7.5 Binding

The traverser procedure *traversex* needs to access the generic pointer traversing program *Trav*, in order that the pointer fields in the structure may be traversed. This may be achieved without resorting to the use of globals by *wrapping up* the procedure inside a generator procedure. This would take the procedure *Trav* as a parameter like so:

```
let traverser.gen = proc( proc( pntr ) Trav  -> proc( pntr ) )
begin
    proc( pntr p )
    ! procedure body as traversex in example 1 above
end
```

example 3

Therefore a procedure of the following type must be compiled,

```
proc( proc( pntr ) -> proc( pntr ) ),
```

like that in example 3 which returns a procedure capable of displaying a structure of a particular class.

Using a first class compiler, it is possible to write a procedure, *mk.trav.proc*, that generates a traversal procedure for a class when supplied with a representation of that class. This procedure returns a pointer to a structure class that contains a procedure like *traverser.gen* described above,

```
let mk.trav.proc = proc( string class -> pntr )
begin
    let last := ""                          ! last character read
    let pos := 0                            ! index into class string

    let next.ch = proc( -> string )
    begin
        pos := pos + 1                      ! takes a sub string length 1
        class(pos|1)                        ! from string class at position pos
    end

    let lex = proc( -> string )             ! gets next lexeme from
    begin                                   ! the class identifier string
        let str := ""
        repeat
                last := next.ch()
        while   last ~= "(" and
                last ~= ")" and
                last ~= " "
        do      str := str ++ last          ! ++ is concatenation
        str                                 ! return str
    end
```

```
        let strings := "let strings = @1 of string [ "      ! build string vector
        let procs := "let procs = @1 of proc() [ "          ! build procs vector
        let title := lex()                                  ! build menu title
        let name := "structure " ++ title ++ "("            ! build class name

        repeat
        begin
            let type = lex()                                ! '"n" is the newline character
            let field = lex()
            name := name ++ type ++ " " ++ field ++ " ; "
            strings := strings ++ field++ ":" ++ type ++ ",'n"
            procs :=  procs ++  if type ="pntr"
                                then "proc() ; Trav( p( "++ field++") ),'n"
                                else "proc() ; write p( " ++ field++ "),'n"
        end
        while last ~= ")"

        name := name ++ ")'n"                   ! list part of structure name
        strings := strings ++ """****"' ]'n"    ! last entry of strings vector
        procs := procs ++ "proc() ; {} ]'n"     ! last entry of procedures vector
        ! next create string containing program representation

        let prog :=  "proc( proc( pntr ) Trav -> proc( pntr ) )'n" ++
                     "proc( pntr p )'nbegin'n" ++
                     name ++ strings ++ procs ++
                     "let this.menu = menu( ",title,",strings,procs )'n" ++
                     "if p is " ++ title ++ " then this.menu( 20,20 ) else Error()'n" ++
                     end'n"

        structure gen( proc( proc( pntr ) -> proc( pntr ) ) maker )
        let S = gen( proc( proc( pntr ) t -> proc( pntr ) ) ; nullproc )

        compiler( prog,S )              ! return the result of compilation that is
    end                                 ! S containing the required procedure
```

example 4

The procedure *Trav* can now be refined to use this procedure. Whenever a class is found for which no traversal procedure exists in the *trav.procs* table *mk.trav.proc* will be called to create a traversal procedure. The generator procedure is then extracted from the structure and called with the generic pointer traverser (*Trav* itself) as a parameter. The resulting procedure can then be stored in the table and finally called to traverse the structure that caused the procedure to be generated. The *Trav* procedure will therefore look something like this,

```
let Trav = proc( pntr p )
begin
        structure gen( proc( proc( pntr ) ->proc( pntr ) ) maker )
        structure trav.pack( proc( pntr ) trav )

        let key = class.identifier( p )                 ! get class of instance
        let traverser. := s.lookup( key,trav.procs )    ! look for display procedure
        if traverser is trav.pack                       ! found one so
        then traverser( trav )( p )                     ! call it with p as a parameter
        else
        begin
                let package = mk.trav.proc( key )       ! create a display package
                let T = package( maker )                ! get generator from package
                let bound = T( Trav )                   ! generate a display proc
                traverser := trav.pack( bound )         ! re-package display procedure
                s.enter( key,trav.procs,traverser )     ! and put it into the table
                bound( p )                              ! finally call it
        end
end
```

example 5

The browser is now complete. The traversal procedure *Trav* maintains and uses the *trav.procs* table which is used to store the procedures that display particular classes. Whenever a display procedure cannot be found by *Trav*, the procedure *mk.trav.proc* is called to generate the necessary compiled code. This code may need to have access to the *Trav* procedure, therefore, the *mktrav.proc* procedure returns a display generating procedure which is passed to *Trav* as a parameter. This step is equivalent to linking in a conventional system. The newly generated procedure is then put into the table so that it can be called to display subsequent instances of that structure class.

7.6 Fire Walls

The language type rules have not broken in the browsing program. However, the discovery of the structure class types using the *class.identifier* procedure has been permitted. The procedure closure has remained sacrosanct and has provided a fire-wall through which this program cannot penetrate. Nevertheless, the need to see inside a closure or, indeed, an activation does arise, for example, when a symbolic debugger is used. The need to see inside such objects also arises when a system is in need of repair. This is seen as being equivalent to the hardware engineer placing probes on a board to identify faults within it. The scheme described does not handle such cases which are clearly in need of more investigation. It is thought that different levels of object interpretation may be needed in this case.

7.7 Performance

The alternatives approaches to the above scheme will now be considered. The first is to halt the system with an error message when a structure class for which no traversal procedure exists is found. The user would then have to write,compile,debug and enter into the table a procedure to traverse the object. The solution outlined in above is several orders of magnitude faster than this. The second alternative would be to write the browser in a lower level language - not a viable compromise in terms of software engineering costs.

The procedure shown in example 1 to traverse the class,

```
structure x( int a ; string b ; pntr c )
```

takes the browser 4.5 seconds of user time to write, compile, enter into the *trav.procs* table and display the menu on a SUN 3/260.If the procedure is already in the table, the combined time required for look up and display time takes less than a sixtieth of a second.

7.8 Persistence

In a conventional programming system the scheme described would be very expensive. The traversal program would have to recreate the traversal procedures in every invocation. In a persistent programming language the table *trav.procs* may reside in the persistent store and therefore any changes made to the tables will exist as long as they are accessible. Consequently the traverser program **never** has to recompile traversal procedures. The program in effect *learns* about new data structures. It does so in a lazy manner, as it only learns how to display the classes that it is actually required to display. This may be viewed pictorially in figure 3.

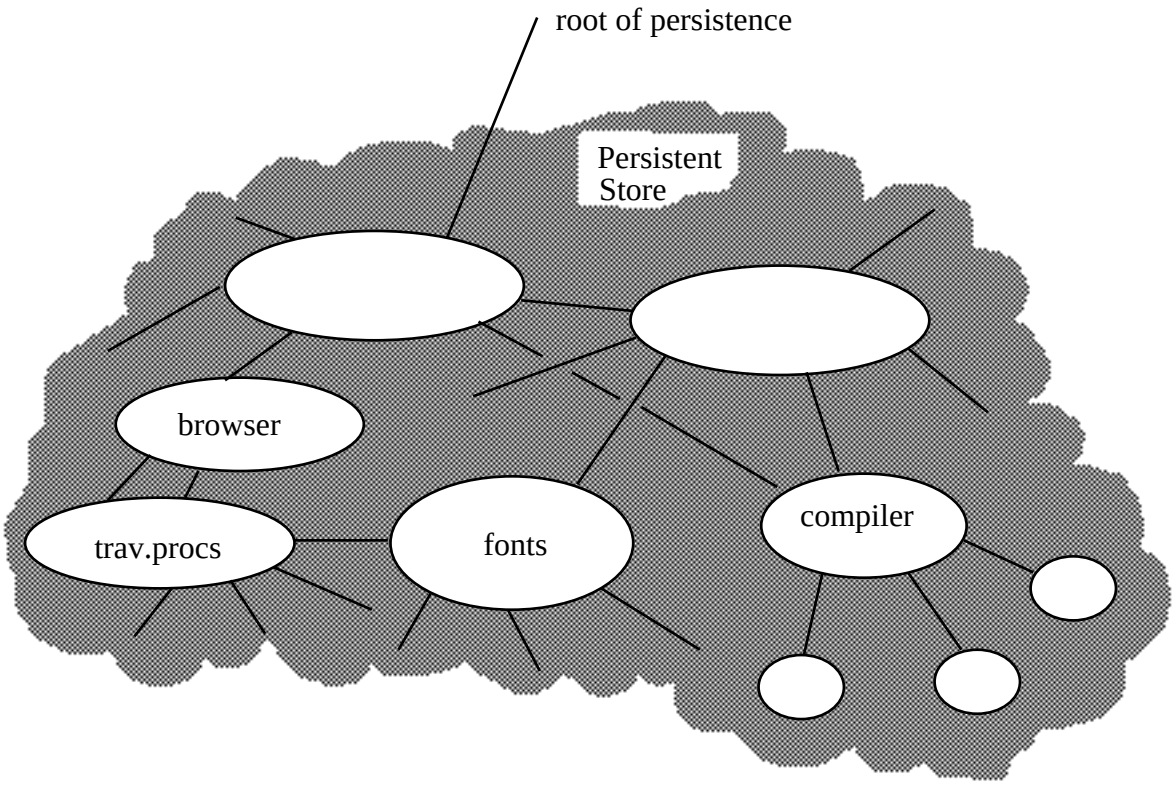


figure 3

7.9 Browser Software Architecture

As the browser evolved it became apparent that it was more important than merely a method of traversing data structures. What had evolved was a new software architecture. The important features of the architecture are:

1. strong (static) type checking
2. late (dynamic) demand driven binding
3. dynamically linking of code
4. adaptive in nature.

These features are discussed below.

The browser is built entirely using the mechanisms provided by the PS-algol language. The language is statically type checked, apart from the projection out of the infinite union **pntr**, where dynamic type checking is employed. The procedures written by the browser

are type checked by the callable compiler and only syntactically correct programs are admitted into the architecture.

The binding of procedures to the architecture is performed extremely late. Indeed, procedures that traverse a particular class are not even written unless required. The storage of procedures in an extensible data structure with dynamic lookup on class is necessary in order to permit the flexibility required. The kind of binding performed here is the weakest possible in a strongly typed system.

The procedures written by the browser and compiled by the callable compiler are dynamically linked into running code. The linking is performed when the procedure which is returned from the callable compiler is entered into the table.

The system is adaptive in nature. The browser can traverse any data structure composed from any of the infinite number of types in the system. These types do not have to have been declared at the time the browser was written. The system adapts itself to operate on new data structures as required.

Two brief examples, showing how this architecture has been exploited in the fields of bootstrapping and databases, are discussed below.

7.10 Browsers as a bootstrapping tool

Code files for the Persistent Abstract Machine consist of a heap of objects prefixed by five words of header information. In order to bootstrap the system, such a file must be created by the bootstrap compiler. The problem is the mapping of the complex graph structure, which comprises a Persistent Abstract Machine heap, onto the file system. This task is normally carried out by the POMS, but in the bootstrap system the compiler is written in PS-algol and no Persistent Abstract Machine code is running.

The problem may be solved by a similar solution to that used in the browser. A set of rules exists for the creation of valid objects. The object management system must keep maintain not one but two tables. The first is similar to the one maintained by the browser - that is a table of output procedures for each object class encountered. The second maintains an address mapping table. This maps object pointers in the address space of the bootstrap compiler to addresses within the code file. The first table may be persistent but the second table is recreated with the production of every heap.

This solution allows the object formats to be easily changed since they are controlled by the browsing program. No programmer time is required in the production of a heap containing a different set of object classes. Therefore, the decision of which objects are in the heap is inexpensive. The production of a virtual image using this technique is much cheaper than hand building a virtual image or even writing a program to produce a custom made one.

7.11 Adaptive Databases

The technology used in the browser has also been used in the production of a relational database system [coop87]. Traditionally, databases are implemented by creating a canonical relation structure [cod70]. Relations introduced by the programmer of the system are then mapped onto this canonical representation. Relations then require a level of secondary interpretation at run time.

The database system constructed by Cooper uses the techniques first invented in the PS-algol object browser. When a user of the database system defines a new class of relation the system generates a set of creation and selector functions for a data structure. The programs are compiled using the callable compiler and entered into a table. Whenever the relation is accessed the appropriate selector functions are used. In this way each data

structure is stored in the most appropriate manner for that relation without the need for secondary interpretation of the data structures. Furthermore the expensive task of programming the movement of objects to and from backing store is performed by the POMS.

7.12 Conclusions

It has been illustrated how a browser may be written in a closed strongly typed environment. This has been achieved without having to use dynamic typing, or make the requirement that every data structure has to have a printString method as in the Smalltalk-80 system. In the system described the programmer may still write a display procedure manually thus specializing the programs default action as in the Smalltalk case. It is also possible to have different display formats for objects by having more than one display table.

The program is allowed to discover the type of objects, even when the type of an object may have been deliberately hidden by the programmer. This raises the issue of who should be able to break these fire walls? The browser needs to be able to see inside objects if it is to be used as a debugger but the programmer may not want the contents of say, an abstract type discovered.

The architecture developed in the browser has been explored and two further examples of how the architecture may be exploited have been given.

8 Conclusions

This thesis presents research into the design and construction of persistent programming systems. This work has been performed as part of the Persistent Information Space Architecture (PISA) project [ack86b].

The main areas in which research has been performed are :

1. programming language design ;
2. programming language implementation ;
3. compiler construction ;
4. abstract program graphs ; and
5. adaptive object browsers.

8.1 Programming Language Design

The importance of good programming language notations cannot be overstated. The provision of a good notation permits the programmer to concentrate on the complexities of a given task rather than the mapping of that task onto a particular language. Research into programming languages has been explored using the persistent languages PS-algol and Napier.

The main areas explored in the language domain are:

1. machine independent graphics ;
2. environments ; and
3. polymorphism.

8.1.1 Graphics

When the work documented in this thesis was started, PS-algol had no raster graphics. It did not therefore provide any means of utilising the power of the graphics facilities provided on the then new ICL Perq computers [icl83]. Several experimental language implementations [mor86c,mor86b] were constructed in order to discover how graphics facilities could be integrated with a high level language.

The integration of graphics facilities into a high level language permits sophisticated machine independent user interfaces to be constructed. Graphics objects are language objects with full civil rights, this means that they may be be stored in the persistent store and manipulated by procedures. For example, the persistent object browser makes use of the menus provided by the graphics facilities. Menus are held within procedure closures in the persistent store, allowing them to be rapidly displayed when required.

The PS-algol graphics facilities have been used to build the front ends to a number of sophisticated applications including a windowing system [cut87] and an object oriented database with inheritance [ben87].

8.1.2 Environments

Much of the work has involved discovering what special problems arise in persistent systems. One of the problems that emerged early during the research was the need to control complexity in large systems. Indeed, the problems of building large systems have been known for many years.

The way in which objects are bound together in a large system is especially important. During the development of the browser a design flaw was discovered and the knowledge that the browser had gained had to be discarded. This was necessary because of the way

that the system had been bound together. In this case, too much static binding had been used which did not permit enough change.

This kind of problem lead to the conclusion by myself and others [atk85a] that control over binding mechanisms is extremely important in large persistent information spaces. The datatype environment was introduced in order to provide a mechanism that would allow incremental system construction and change within a large system.

The environment datatype is a simple mechanism with clean semantics that are easy to understand. Environments provide a way of smoothly integrating the programming language with the programming environment. They also provide a structuring mechanism over the name space which is similar to the structure imposed by directories on a file system. Problems still remain in this area, in particular, how functions like *ls* in Unix may be expressed in a strongly typed system [atk87].

8.1.3 Polymorphism

The cheapest way to build a software system is to construct from components already written [brk86]. In order to achieve this, a type system is required that is powerful enough to describe all the objects in a system. Polymorphism provides the mechanism for abstracting over types. However, the search for an all powerful type system is not an end in itself. One mechanism, the type system, may be used for checking the legal composition of objects makes the system simpler with the attendant cost benefits.

8.2 Abstract Machine Design

To design programming languages and not implement them is pointless, yet this often happens. It is only through implementations that engineering lessons are learned. Sometimes paper designs cannot be realised by current implementation technologies and the design has to be revised - this is part of the design process. Much has been learned from implementing PS-algol, the first persistent programming language. Without the implementations of PS-algol, the language Napier would not have evolved. Many important, though small, advances have been made during the research into machine support :

1. modularisation ;
2. uniform object formats ; and
3. efficient implementation of non uniform parametric polymorphism.

8.2.1 Modularisation

The Persistent Abstract Machine, like all the components of the Napier system, is constructed in a modular fashion. Each layer in the machine presents a well defined interface. This has two main benefits: the first is in maintenance costs, the second is as a research vehicle.

Parnas cites information hiding as one of the most effective ways of avoiding rework [par79]. The PS-algol machine has proven expensive to maintain. This is partly due to its size and partly due to its complexity. Much of this complexity has arisen in this machine due to its nature - that of a research vehicle.

The Napier support environment is also a large, complex piece of software, as such requires maintenance. It is hoped, that the modular design of the architecture will result in lower costs in the future. More importantly the modular design of the architecture allows experimentation into language implementations to be performed independently. For example, it is possible to change the persistent object management strategy without changing the Persistent Abstract Machine. This will allow the interactions between different parts of the system to be explored.

8.2.2 Uniform Object Format

One of the biggest advances in the Persistent Abstract Machine is a simple one. The abstract machine has no knowledge of the type systems of the languages that it implements. One result of this is that the machine has a uniform object format. The heap is the only dynamic storage system supported by the system. Objects are partitioned into pointer and non-pointer fields, minimising the potentially high cost of garbage collection and persistent object management.

8.2.3 Parametric Polymorphism

The parametric polymorphism provided by Napier has a large impact on the machine design. One consequence of polymorphism is that the compiler cannot tell statically how big a polymorphic object is or whether it is a pointer type or not.

An efficient implementation of first class polymorphic procedures has been achieved without adversely affecting the performance of non-polymorphic ones. The implementation is novel in that it implements parametric polymorphism for non-uniformly sized objects. The technique makes use of the block retention architecture provided by the Persistent Abstract Machine. The technique may be extended without modification to support a powerful notion of abstract types. It is thought that this mechanism may also be used to support inclusion polymorphism.

8.3 Compiler construction

During the development of the Napier language many benefits have emerged from using the persistent store as a compiler construction vehicle. The most obvious benefits are:

1. modularisation ;
2. provision of PAIL ;
3. optimisation ; and
4. callable compilers.

8.3.1 Modularisation

The compilation system comprises many different modules. Like much of the persistent system it is written using the persistent languages that it supports. The modules provided in the compilation system provide the language implementor with a toolset. Using this toolset many different compilers may be constructed cheaply. These compilers may compile different languages or provide different interfaces to a compiler for a single language. The compiler toolset therefore provides an architecture that is highly flexible and given to software reuse.

8.3.2 PAIL

Much of the language implementation depends on the Persistent Architecture Intermediate Language - PAIL. The provision of this level in the system is not an intrinsic requirement but rather an engineering decision. PAIL provides support for many of the activities performed within the persistent architecture. In particular, PAIL acts as a protection mechanism, an optimisation aid and a debugging aid. PAIL also provides an abstraction level between layers in the compilation system.

8.3.3 Optimisations

The provision of PAIL permits a certain class of code optimisations to be made. Some of the optimisations performed are common; these include optimisations such as manifest elimination and constant folding. A complex optimisation strategy is used to produce an

efficient implementation of parametric polymorphism. During the compilation of polymorphic functions, code is hoisted to lexicographically previous blocks.

8.3.4 Callable Compilers

One class of compiler that may be constructed using the compiler toolset is the so called callable compiler. A callable compiler may be passed program source in some form and returns an executable version of it. In order that this kind of compiler is useful it has to be a first class data object in the persistent universe. The dynamic introduction of code into a running system introduces problems in a strongly typed environment, however, these are easily overcome. This mechanism is extremely important if the persistent information space is required to be a self-contained closed universe. The provision of a callable compiler means that no linker or loader is required in the system. There is also no need to provide an explicit language mechanism for separate compilation.

Many classes of applications can benefit from the provision compiler that is a first class data object. One of these applications is the persistent object browser. The object browser [dear88b] uses most of the facilities provided by the architecture including the graphics and a callable compiler.

8.4 Adaptive Object Browsers

The persistent object browser introduces a new software architecture that permits adaptive programs to be constructed incrementally. This is achieved by constructing new programs, compiling them and calling them from a running process. Constructed programs are memorised by storing them in the closure of the browser. This is only a viable prospect due to the provision of the persistent store. The architecture has been successfully applied to the construction of adaptive databases and bootstrap compilers.

8.5 Future Research

At the time of writing the total Napier system is incomplete. The design of the language and the machine to support it is finished and an implementation of the compilation system, the Persistent Abstract Machine and the persistent storage manager is complete. The implementation of processes has only just begun therefore the system may only be used by a single user at any time. Currently, the Persistent Abstract Machine does not support distributed object stores although provision for this has been made in the store design [bro88].

Much of the thrust of this thesis has been concerned with the construction of an integrated programming environment. To conclude, let us examine how the parts of the system discussed in this document may be unified and utilised. This is the only part of the work discussed that has not been implemented, as such it is partly speculative.

Users of the Napier programming environment will never have to use the facilities provided by an operating system, file system or database management system. The programming environment is a self-contained world containing all the facilities necessary to support application development. The environment will be provided by applications written in Napier. It will therefore be tailorable by applications programmers. It is important that some facilities should be judiciously protected from misuse in order to protect the environment. The most important example of this is the ability of the compiler to turn a byte stream into a closure.

8.5.1 Windows

When the user sits down at a workstation to interact with the Napier system he or she will be faced with a window based environment. A prototype window-based environment for the persistent information space has been implemented by Cutts and Kirby [cut87]. The

implementation of this system relies heavily on the graphics facilities provided by the language.

It is important that the windowing system should preserve the user dialogue. In other words, a window should provide an approximation to an infinite buffer that contains everything written on the screen by both the user and the system. This may support complex dialogues.when combined with the cut and paste facility epitomised by editors available on the Apple Macintosh.

The Napier environment is required to support the development of applications. For this to be successful the system must allow the user to gain access to all the information stored about applications. It should not be necessary for program developers to remember information about applications. For example, the system must be capable of showing the user what types are being used to model aspects of applications. If this is displayed on the users screen there is no reason why the user should have to type the information back into the system and the system re-parse it. Instead, the user should be able to select displayed information and say "yes that is correct I want to use it".

8.5.2 Browsing Information Graphs

The persistent information space forms a graph structure emanating from a single root called *ps*, which has type **env**. The graph comprises of environments and objects constructed using Napier. This graph structure may be examined using a browser similar to that described in chapter 7. The use of a browser allows the user to discover what objects populate the information space. In particular, the application writer may wish to locate procedures stored in environments that may be reused in the application he or she is developing.

The system will not provide a command language other than the Napier language. Of course, other languages will be provided, for example the window dialogue interaction language. However, these are not supported by a syntactic form. Instead they rely on the use of a mouse and interaction with light buttons and dialogue boxes.

In some of the windows in the system an interactive compiler that will execute instructions immediately will be available. Such a compiler may also be used to browse the persistent information space by executing statements that operate against environments and perhaps calling a browser. In such an environment there is little distinction between browsing and interactive compilation.

Browsing a complex information space is equivalent to browsing a hypertext graph. It is expected that techniques developed to traverse hypertext graphs may be of utility in traversing the information space and program graphs.

It is expected that the unit of change in the system will be very small. The provision of interactive compilers and environments will encourage the incremental development of systems from prototype through to complete applications.

In a truly integrated environment the use of the word "program" and the word "procedure" are indistinguishable. Procedures may be viewed at one level as complete programs, may be viewed at another as tools contributing to a larger application. This is in marked contrast to traditional systems such as the Unix environment where a program is a markedly different entity from a procedure. Procedures may use other procedures in the information base. The provision of environments and the binding mechanisms over them will encourage the reuse of code.

In the Napier system the value of a procedure will always have its source code bound into it. This will result in simpler management of code since there will be no need to remember

which source belongs with which executable code. A polymorphic function, called *source* will be provided to yield the source of any procedure.

Once a programmer has obtained the source of a procedure it may be edited using tools provided by the windowing environment. It may then be recompiled, using a callable compiler. If the resulting code is of the same type it may be assigned to the location in the environment from where it came. If it is not, it may have a new location created for it.

The source code that is bound into the executable version is in the form of a PAIL graph. The information contained in the PAIL graph will allow sophisticated tools to be developed that operate against program source. For example, program editors may be constructed that provide the user with much better facilities than found in simple text editors.

8.5.3 Debugging

The Napier system has had debugging support built into its design from the first stages. This is essential if good debugging information is to be provided. Attempting to retro-fit debugging support to the PS-algol system proved to be almost impossible. The integration of source and executable code is the key to good debugging information. The machine keeps a note of the source code being executed in a manner similar to the maintenance of a code pointer. The source position information is stored in closures when procedure calls are made so that an activation record may be displayed.

The debugging system will be constructed as a self contained module in a similar manner to the compiler. Entry points to this module will be placed in the exception handlers so that when an error occurs the debugging system will be called interactively. The debugger may then call an editor and the compiler in order to present the user with a smooth transition from one tool to another.

8.6 Finally

This thesis presents research into the world of persistent programming systems. Hopefully, orthogonal persistence will become common in the computer systems of the future. If not, so be it, the lessons learned in constructing the PS-algol and Napier programming systems will live on in systems as yet figments of the imagination.

Appendix 1
Persistent Architecture Intermediate Languge
Napier Definition

```
!************************** PAIL Structures ****************************

rec type  list[ t ] is variant(  tip      : null ;
                                 cons   : cell[ t ] )
&
          cell[ t ] is structure( hd : t ; tl : list[t ] )

type pair[ t ] is structure( fst,snd : t )


rec type  PAIL is
          variant(  Empty            : null ;
                    Control          : Control ;
                    Assign           : structure(  Lhs : Typed[ index ] ; Rhs : tree ) ;
                    Alias            : structure(  Subject,Origin,Length : PAIL ) ;
                    Overwrite        : structure(  Rule : string ; Source,Destination : PAIL ) ;
                    Application      : structure(  Function : PAIL ;
                                                   Arguments : list[ PAIL ] ) ;
                    Comment          : structure(  Code : PAIL ; Comment : string ) ;
                    Optimisation     : structure(  Optimised,NonOptimised : PAIL ;
                                                   Info : string ) ;
                    NamedAddress     : link ;
                    Literal          : lit ;
                    Cons             : Constructor ;
                    Indx             : Index ;
                    Scoping          : Scope ;
                    Exception        : Exception )

&
!************************* Basic Tree Structure ************************

          tree[ t ] is structure( Type : TYPE ; Code : t ; Parent : Parent[ t ] )
&
          Parent[ t ] is variant(  Empty : null ;
                                   Tree  : tree[ t ] )
&
!******************************** Types ********************************


          Typed[ t ] is variant( Link : link ;
          Tree : tree[ t ] )
&
          TypeDescriptor is structure(  TheType : TYPE ;
                                        Descriptor : TypeConstructor )
&

          TypeConsParam is structure(  Type : TypeDescriptor ;
                                       ParameterList : list[ TypeDescriptor ] )
&

          TypeConsVec is structure( Elements : TypeDescriptor )
&
          TypeConsStruct is structure(  Fields : list[ link ] ;
```

```
                                        total,pntrs : Offset ; bitmap : *int )
&

        TypeConsProc is structure( quantifiers,parameters,result : TypeDescriptor )
&

        TypeConsAbstract is structure(  Witnesses  : list[ TypeDescriptor ] ;
                                        Abstype    : TypeDescriptor )
&

        TypeConstructor is variant(  Parameterised  : TypeConsParam ;
                                     Vector         : TypeConsVec ;
                                     Structure      : TypeConsStruct ;
                                     Proc           : TypeConsProc ;
                                     Abstract       : TypeConsAbstract )
&

        baseTypeContainer is structure( BaseType : string )
&
        StoredType is structure( Type : TYPE )
&
        TypeDecl is structure( Type : TYPE  )
&
        TypeRecDecl is structure( Types : list[ TypeDecl ] )
&

        Parameterise is structure(  Parameterised : TypeDescriptor ;
                                    TypeParameters : list[ TypeDescriptor ] )
&

        Specialise is structure(  Source : PAIL ;
                                  TypeList : list[ TypeDescriptor ] )


!*************************** Symbol Table ******************************

&
        stackUse is structure( Cvec.indirect : structure( Cvec : *int ) ; Caddr : int )
&
        stackPos is structure( Frame,MSoffset,PSoffset : int ; Uses : list[ stackUse ] )
&
        heapPos is structure( MSoffset,PSoffset : int )
&

        location is variant(  New    : null
                              Stack  : stackPos ;
                              Env    : stackPos ;
                              Heap   : heapPos )
&

        Offset is variant( static : int ; dynamic : link )
&

        link is structure(  Name        : string ;
                            Type        : TYPE ;
                            Initial     : PAIL ;
                            Manifest,
                            Retained,
```

```
                        Primitive,
                        Const       : bool ;
                        Addr        : location )
&

        SymbolTable is
        variant(  Empty       : null ;
                  Table       : structure( Create : proc( SymbolTable -> SymbolTable ) ;
                  InsertEntry : proc( string,link ) ;
                  LookupLoc   : proc( string -> link ) ;
                  LookupRec   : proc( string -> link ) ;
                  Enclosing   : proc( -> SymbolTable ) ;
                  Scan        : proc( proc( string,link ) ) ) )
&

!******************************** literals *********************************

        lit is variant( Boolean : bool ;
                        File    : file ;
                        Int     : int ;
                        Real    : real ;
                        Pixel   : pixel ;
                        String  : string )


!*************************** Control ******************************
&
        CaseChoice is structure( Patterns : list[ PAIL ] ; Action : PAIL )
&

        Control is
        variant( And      : structure(  And1,And2 : PAIL  ) ;
                 Or       : structure(  Or1,Or2 : PAIL ) ;
                 Sequence : structure(  This,Next : PAIL ) ;
                 If       : structure(  Cond : PAIL ; Then,Else : tree ) ;
                 Loop     : structure(  Repeat,Cond,Do : PAIL ) ;
                 For      : structure(  Symbols : SymbolTable ;
                                        Iterator,Start,Stop,Step,Do : PAIL ) ;
                 Case     : structure(  Switch : PAIL ;
                                        Branches : list[ CaseChoice ] ; Default : PAIL ) )
&

        assign is structure( Lhs : link ; Rhs : tree )

!*************************** Constructors *******************************
&
        initialiser is structure( field : link ; value : PAIL )
&
        SimpleDecl is structure( Exp : PAIL ; Symbol : link )
&
        RecDecl is structure( Decls : list[ Decl ] )
&

        Decl is variant(  simple    : SimpleDecl ;
                          recursive : RecDecl )
&
```

```
        Constructor is
        variant( MakeVector     : structure(  Start : PAIL ; Elements : list[ PAIL ] ) ;
                 MakeStructure  : structure(  Type : TYPE ;
                                              initial : list[ initialiser ] ;
                 Constructor    : TypeDescriptor ) ;
                 Declaration    : Decl ;
                 MakeImage      : structure( XDim,YDim,Initial : PAIL ) ;
                 EnvExtend      : structure( Source : PAIL ; Decl : Decl ) ;
                 MakeAbstract   : structure(  Decl      : link ;
                                              speclist  : list[ TypeDescriptor ] ) ;
                 Constructor    : PAIL ) )
&
```

```
!***************************** Indexing ******************************

        Index is
        variant( Subs          : variant(   value     : structure( Subject,
                                                                   Origin,
                                                                   Length : tree[ PAIL ) ;
                                            address   : structure( Subject,
                                                                   Origin,
                                                                   Length : tree[ PAIL ] ) ) ;
                 EnvProject    : structure( Source : PAIL ;
                                            Signature : list[ link ] ; Body : PAIL ) ;
                 UseAbstract   : structure( Source : PAIL ; Signature : link ;
                                            Body : PAIL  ) )


!***************************** Scoping ******************************
&
        Scope is variant( Block      : structure( Symbols : SymbolTable ;

                                                  Block.body : PAIL ) ;
                          ProcDesc   : structure( Resultype   : TYPE ;
                                                  Parameters : list[ link ] ) ;
                          Body       : PAIL ;
                          Symbols    : SymbolTable ) )


!***************************** Exceptions ******************************
&
        Exception is variant( Catch   : structure( Handler : PAIL ; Code : PAIL ) ;
                              Raise   : structure( Event : PAIL ) )
```

References

[ada83] The Programming Language Ada Reference Manual. ANSI/MIL-std-1815a-1983. Lecture notes in Computer Science, Springer Verlag (1983).

[alb85] Albano A., Cardelli L. & Orsini R. Galilieo: A Strongly Typed, Interactive Conceptual Language. ACM TODS 10,2 pp 230-260 (1985).

[atk78] Atkinson M.P. Programming Languages and Databases. Proc VLDB pp 408-419 (1978).

[atk83] Atkinson M.P. Bailey P.J., Chisholm K.J. Cockshott W.P. & Morrison R. An Approach to Persistent Programming. The Computer Journal (1983). 26,4 pp 360-365 (1983).

[atk84] Atkinson M.P. & Morrison R. First class functions are enough. Foundations of Software Technology and Theoretical Computer Science. Lecture Notes in Computer Science 181, Springer Verlag (1984).

[atk85a] Atkinson M.P. & Morrison R. Types, bindings and parameters. Proceedings of the 1st Appin workshop on persistent object systems. Universities of Glasgow and St Andrews PPRR-16 (1985).

[atk85b] Atkinson M.P. and Morrison R. Procedures as persistent data objects. ACM.TOPLAS 7,4 (1985).

[atk86a] Atkinson M.P. & Morrsion R. Integrated Persistent Programming Systems. Proc. Hawaii International Conference on System Sciences. (1986).

[atk86b] Atkinson M.P., Lucking J., Morrsion R. & Pratten G. Persistent Information Space Architecture Club Rules, Universities of Glasgow and St Andrews PPRR 47 (1986).

[atk87] Atkinson M.P. & Morrison R. Polymorphism, Type checking and Lables in a Persistent Object Store. Proceedings of the 2nd Appin workshop on persistent object systems. Universities of Glasgow and St Andrews PPRR-44 (1987).

[bal86] Baltzer R. Living in the next generation operating systems. Proceedings of the IFIP 10th World Computer Congress, pp 283-291 (1986).

[ben87] Benson P.J., D'Souza E.B., Rennie I.J. & Waddell S.J. An Implementation of multiple inheritence in a persistent environment. Universities of Glasgow and St Andrews PPRR 49 (1987).

[bob81] Bobrow D. G. and Stefik M. The Loops manual. Tech Rep.KB-VLSI-81-13, Knowledge Systems Area. Xerox Palo Alto Research Centre (1981).

[boe84] Boehm B., Gray T.E & Seewaldt T. Prototypings vs. specification: A multi-project Experiment. IEEE Transactions on Software Engineering. May,1984 pp 133-145 (1984).

[boe86] Boehm B. Understanding and Controlling Software Costs. Proceedings of the IFIP 10th World Computer Congress, pp 703-714 (1986)

[bou78] Bourne S.R. An Introduction to the Unix Shell. Bell Laboratories, (1978).

[brk86] Brooks F.P. No Silver Bullet - Essence and Accidents of Sorftware Engineering. Proceedings of the IFIP 10th World Computer Congress, pp 1069-1076 (1986).

[bro86] Brown A.L. & Dearle A. Implementation issues in Persistent Graphics. University Computing 8,2 (1986).

[bro88] Brown A.L. Ph.D. Thesis - Persistent Object Stores. University of St Andrews (1988)

[bur84a] Burstall and Goguen J.A. The Semantics of Clear, A specification language. Springer-Verlag 86, pp 292-332 (1984).

[bur84b] Burstall R. & Lampson B. A Kernal Language for Abstract Data Types and Modules. Proc. International Symposium on Semantics of Datatypes. Springer-Verlag (1984).

[bur84c] Burstall R. Programming with Modules as Typed Functional Programming. Proc. International Conference on 5th Generation Computer Systems. Tokyo. (1984).

[buh87] Buhr P.A. & Zarnke C.R. Persistence in an Environment for a Statically-Typed Programming language. Proc. International Conference on Persistent Object Systems. Universities of Glasgow & St Andrews PPRR 44, pp 317-336 (1987).

[bux69] Buxton J. & Randell B. (eds.) Software Engineering Techniques. Proc. Nata Conference. Rome (1969).

[cal77] Calderbank V.J & Prior A.J. The Ghost graphical output system. Culham Laboratory Report (1977).

[car85] Cardelli L. & Wegner P. On Understanding Types, Data Anstraction and Polymorphism. Technical Report CS-85-14, Brown University (1985).

[car87] Carrick R. & Munro D. Execution Strategies in Persistent Systems. Proc Workshop on Persistent Object Systems: There Design Implementation and Use, Appin Scotland, (1987).

[cod70] Codd E.F. A relational model for large shared databases. Comm. ACM 13,6 pp 377-387 (1970).

[coop87] Cooper R.L.,Atkinson M.P., Dearle A. & Abderrahamane A. Constructing Database Systems in a Persistent Environment. Proc VLDB 1987, Brighton England, (1987).

[cut87] Cutts Q. & Kirby G. An Event Driven Software Architecture. Universities of Glasgow and St Andrews PPRR 48 (1987).

[dev81] Davie A.J.T & Morrsion R. Recursive Descent Compiling. Ellis Horwood (1981).

[dea85] Dearle A. A new abstract machine for S-algol. University of St.Andrews cs/85/1 (1985).

[dea87] Dearle A. A Persistent Architecture Intermediate Language. Universities of Glasgow and St.Andrews PPRR-35 (1987).

[dea88a] Dearle A. (ed). The Persistent Abstract Machine. University of St.Andrews, In preparation.

[dea88b] Dearle A., Brown A.L. Safe Browsing in a Strongly Typed Persistent Environment. to appear in The Computer Journal (1988).

[dav81] Davie A.J.T. & Morrison R. Recursive Descent Compiling. Ellis Horwood, (1981).

[dod83] KIT/KITIA CAIS Working Group for the Ada Joint Program Office. Common APSE Interface Set. Version 1.1, Ada Joint Program Office (1983).

[gog82] Goguen J.A. Rapid Prototyping in the OBJ Executable Specification Language. Proc. Rapid Prototyping Workshop, Colombia, Maryland (1982).

[gold83] Goldberg A. & Robson D. Smalltalk-80: The language and its Implementation. Addison Wesley (1983).

[gks82] Information Processing - Graphical Kernel System - Function Description: GKS version 7.2. ISO/TC97/SC5/WG2 N 163. (1982).

[har86] Harper R., MacQueen D. and Milner R. Standard ML. Edinburgh University Technical Report ECS-LFCS-86-2 University of Edinburgh (1986).

[hoa74] Hoare C.A.R. Monitors : an operating system structuring concept. Comm. ACM 17,10 pp 549-557 (1974).

[hoa78] Hoare C.A.R. Communicating Sequential Processes. Comm ACM 21,8 (1978).

[ibm78] IBM Report on the contents of a sample of programs surveyed. IBM Research Centre San Jose, California (1978).

[icl83] Introduction to PERQ. International Computers Ltd. RP10103 (1983)

[joe83] Jones T.C. Demographic and Technical Trends in the Computing Industry. Software Productivity Research Inc. (1983).

[joy80] Joy W., "An Introduction to the C Shell", University of California, Berkeley, (1980).

[ker78] Kernighan B.W. & Ritchie D.M., "The C programming language", Prentice-Hall, (1978).

[ker79] Kerighan B.W. & Marshey J.R., The Unix programming environment. Software Practice and experience. 9,1 (1979).

[kra85] Krablin G.L. Building Flexible Multilevel Transactions in a Distributed Persistent Environment. Proceedings Appin Workshop August 1985. PPRR 16 Universities of Glasgow and St.Andrews.

[kre80] Kreig-Bruckner B.& Luckham D.C. Anna: Towards a language for annotating Ada programs. ACM Sigplan Notices 15,11 pp 128-138 (1980).

[lan66] Landin P.J. The next 700 programming languages. Comm ACM 9,3 pp157-403 (1966).

[lis74] Liskov B.H. & Zilles S.N., "Programming with abstract data types", ACM SIGPLAN Notices 9,4 (1974).

[lis77] Listkov B.H. et al. Absraction Mechanisms in CLU. Comm. ACM 20,8 pp 564-576 (1977).

[lis83] Liskov B.H. et al. The Argus manual. Techical Report Memo 39 (1983) M.I.T.

[liv87] Livingstone J. Graphical Manipulation in Programming Languages: Some Experiments. M.Sc. Thesis. University of Glasgow (1987).

[lob87] Loboz Z. PS-algol Abstract Machine Monitoring Universities of Glasgow and St Andrews PPRR 37 (1987).

[loc78] Lochovsky F.H. & Tsichritizis. Hierarchical database management systems. ACM Computer Surveys 8,1 pp 105-123 (1978).

[mac86] Inside Macintosh. Apple Computer Inc. Addison Wesley, (1986).

[mat85] Matthews D. The Poly Manual. Technical Report 63. University of Cambridge Computer Laboratory (1985).

[mcc62] McCarthy J. et al. Lisp Programmers manual. MIT press, Cambridge Mass (1962).

[mcn87] McNally D.J. Implementation in the Staple Project. University of St.Andrews cs/87/2 (1987).

[mit85] Mitchell J.C. & Plotkin G.D. Abstract Types have Existential type. Proceedings POPL (1985).

[mor73] Morris J.H. Protection in programming languages. Comm. ACM 16,1 pp 15-21 (1973).

[mor78] Morris F.L. A time and space efficient garbage collection algorithm. CACM 21,8 pp 662-665 (1978).

[mor79] Morrison R. Ph.D. Thesis - On the development of algol, University of St Andrews (1979)

[mor82a] Morrison R.,"S-algol: a simple algol", Computer Bulletin II/31 (1982).

[mor82b] Morrison, R. Low cost computer graphics for micro computers. Software, Practice & Experience 12, 8 (1982), pp 767-776.

[mor85] Morrison R., Bailey P.J., Brown A.L., Dearle A. & Atkinson M.P. The Persistent Store as an enabling technology for Integrated Support Environments, Proc. 8th International Conference on Software Engineering. pp 166-172 (1985).

[mor86a] Morrison R., Dearle A. & Atkinson M.P., Flexible Incremental Bindings in a Persistent Object Store, Universities of Glasgow and St Andrews PPRR 38 (1986).

[mor86b] Morrison R., Brown A.L., Dearle A. & Atkinson M.P. An Integrated Graphics Programming Environment. Computer Graphics Forum 5,2 (1986).

[mor86c] 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 14,3 (1986).

[mor87a] Morrison R., Brown A.L., Connor R. & Dearle A. Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment. Universities of Glasgow & St Andrews PPRR 32, (1987)

[mor87b] Morrison R., Brown A.L., Carrick R., Connor R. & Dearle A. Polymorphic Persistent Processes. Universities of Glasgow and St Andrews PPRR 39 (1987).

[mor88a] Morrsion R., Process Implementation in Napier. Private Communication, Feb 1988.

[mor88b] Morrison R., Brown A.L., Carrick R., Connor R. & Dearle A., "The Napier Reference Manual", Universities of St.Andrews, In preparation.

[mor88c] Morrison R., Brown A.L., Carrick R., Connor R. & Dearle A. On the integration of Object-Oriented and Process-Oriented compuitation in Persistent Environments. Universities of Glasgow and St Andrews PPRR 57 (1988).

[naur63] Naur P. et al. Revised report on the algorithmic language Algol 60. Comm. ACM 6,1 pp1-17 (1963).

[nee74] Needham R.M. & Walker R.D., "Protection and Process Management in the CAP Computer", International Workshop on Protection in Operating Systems, INRIA, Rocquencourt, (1974).

[nyg70] Nygaard, K., The Simula-67 Common Base Language. Norwegan Computer Centre, Oslo. S-22, (1970).

[org73] Organick E.I., Computer System Organisation: The B5700/B6700 Series, Academic Press, New York (1973).

[par79] Parnas D.L. Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering. March, 1979 pp 128-137 (1979)

[per87] Perry N. "Hope+", Imperial College Internal Report IC/FPR/LANG/2.5.1/7 (1987).

[ps85] PS-algol Abstract Machine Manual. University of Glasgow and St Andrews PPRR11-85 (1985).

[ps87] "The PS-algol Reference Manual fourth edition", Universities of Glasgow and St.Andrews PPRR-12 (1987).

[rey83] Reynolds J. Types abstraction and Polymorphism. Information Processing. North Holland (1983).

[sch85] Schaffert C.,Cooper T. and Wilpolt C. Trellis Object Based Environment. DEC TR-372, Digital Eastern Research Lab (1985).

[shi81] Shipman D.W. The functional Data Model and the data language DAPLEX. ACM TODS 2,3 pp 247-261 (1981).

[smi71] Smith, D.N. GPL/1 - A PL/1 extension for computer graphics. AFIPS SJCC (1971), pp 511-528.

[stra67] Strachey C. Fundamental concepts in programming languages. Oxford University Press, Oxford (1967).

[sun86] Sunview Programmers Guide. Sun microsystems Inc. (1986).

[tay76] Taylor R.C. & Frank R.L. CODASYL database management systems. ACM Computing Surveys 8,1 pp 67-103 (1976).

[tei81] Teitelbaum T. & Reps T. The Cornell Program Synthesiser: A Syntax Directed Programming Environment. Comm ACM 24,9 pp 563 - 573 (1981).

[ten77] Tennant R.D. Language Design Methods based on semantic principles. Acta Informatica 8 pp 97-112 (1977).

[thi86a] Lightspeed C Users Manual. Think Technology (1986).

[thi86b] Lightspeed Pascal Users Manual. Think Technology (1986).

[tur79] Turner, D.A. SASL language manual. University of St.Andrews CS/79/3 (1979).

[vw69] van Wijngarden, A. et al. Report on the algorithmic language Algol 68. Numerische Mathematik 14,1 pp 79-218 (1969).

[weg84] Wegner P. Capital Intensive Software Technology. IEEE Software 1,3 pp 7 - 46 (1984).

[weg87] Wegner P. Dimensions of object-based language design. Proc. Object-Oriented Programming Systems Languages and Applications..pp 168-182 (1987).

[wir73] Wirth N. The programming language Pascal. Acta Informatica 1,1 pp 35-63 (1973).

[wul74] Wulf W.A. et al., "Hydra: The Kernel of a Multiprocessor Operating System", CACM 17,6 (1974).

[zil73] Zilles S.N. Procedural Encapsulation: a linguistic protection technique. ACM Sigplan Notices 8,9 (1973).