The Persistent Abstract Machine

R.Connor, A.Brown, R.Carrick, A.Dearle, & R.Morrison

Department of Computational Science, University of St Andrews,
North Haugh, St Andrews, Scotland KY16 9SS

Abstract

The Persistent Abstract Machine is an integral part of a layered architecture model to support the Napier language. It interfaces cleanly with a persistent store, and allows persistence to be implemented without difficulty in a high-level language. The heap based storage mechanism of the Persistent Abstract Machine is designed to support the block retention nature of the Napier language. This allows the implementation of first class procedures and modules in programming languages with the minimum of effort. A primitive type system within the machine contains just enough information to allow machine instructions which behave differently according to the dynamic type of their operands. This type system, in conjunction with the block retention architecture, may be used to great effect to provide a fast implementation of polymorphic procedures, abstract data types, inheritance and bounded universal quantification.

1. Introduction

In recent years, research into persistent programming systems has led to the design of sophisticated database programming languages such as Galileo[1], PS-algol[2], and Napier[3]. These languages provide a wide range of abstraction facilities such as abstract data types, polymorphism and first class procedures that are integrated within a single persistent store. The development of these systems has required the design of a variety of new implementation techniques. For example, the development of the Napier system necessitated the design of reusable compiler componentry[4], an intermediate language[5], an abstract machine[6] and a persistent object store, all of which are integrated into a highly modular layered architecture[7]. Here we present the objectives and solutions which comprise the design of the Persistent Abstract Machine.

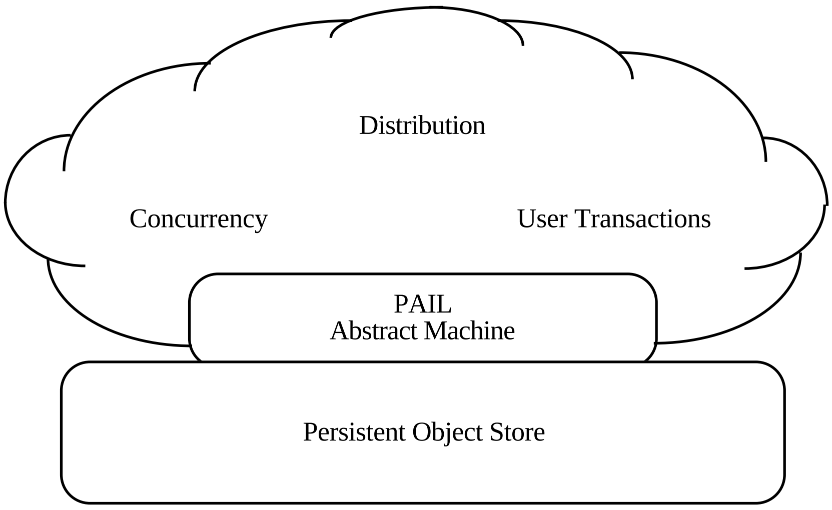

Figure 1. The major architecture components.

The Persistent Abstract Machine is primarily designed to support the Napier programming language. It is closely based on the PS-algol abstract machine[8], which in turn evolved from the S-algol abstract machine[9]. Due to the modularity of its design and implementation, it may be used to support any language with no more than the following features: persistence, polymorphism, subtype inheritance, first class procedures, abstract data types and block structure. Other features, such as object-oriented programming in the Smalltalk style[10] and lazy evaluation, may be modelled at a higher level using the same support mechanisms as first class procedures[11,12]. This covers most algorithmic, object-oriented and applicative programming languages currently in use. The machine can thus be said to be multi-paradigm.

The Napier system is designed so that implementors wishing to use the abstract machine may compile to an intermediate level architecture, consisting of abstract syntax trees. A code generator is available to compile to the abstract machine level. This persistent architecture intermediate language, PAIL[10], supports all of the abstraction listed above, and is sufficiently high-level to ease the burden of compiler writing. Furthermore, it is possible to check at this level that correct (i.e. consistent) PAIL code has been generated, and so output from an untrusted compiler cannot cause a malfunction in the abstract machine. This assures that the persistent store may not be corrupted by the generation of illegal instruction sequences, removing the onus of segmentation protection from the store.

The design of the Persistent Abstract Machine is directly attributable to the Napier language. The major points fall into two sections:

> 1. The machine is an integral part of an entire layered architecture; in particular, it interfaces cleanly with the persistent store, and allows an elegant implementation of persistence in a high-level language. As another consequence of persistence, the machine exists in a single heap-based storage architecture. This architecture directly gives a method of implementing block retention, for almost no extra cost. This allows the implementation of first class procedures and modules in programming languages with the minimum of effort.
>
> 2. A primitive two-level type system within the machine contains enough information to allow machine instructions whose behaviour depends on the dynamic type of their operands. It has a fast and efficient integer encoding. In conjunction with the block retention architecture, the type system is used to great effect to provide a fast implementation of polymorphic procedures, abstract data types, and bounded universal quantification.

2. A Heap Based Storage Architecture

One of the most notable features of the abstract machine is that it is built entirely upon a heap-based storage architecture. Although the machine was primarily designed to support a block-structured language, for which a stack implementation might be the obvious choice, the heap-based architecture was considered advantageous for the following reasons:

1. Only one storage mechanism is required, easing implementation and system evolution.

2. There is only one possible way of exhausting the store. In a persistent system this is an essential requirement, since applications should only run out of store when the persistent store is exhausted, and not merely when one of the storage mechanisms runs out. Although this could be modelled in an environment with more than one storage mechanism, it would be expensive in terms of implementation and evolution.

3. The Napier language supports first-class procedures with free variables. To achieve the desired semantics, the locations of these variables may have to be preserved after their names are out of scope, which would not happen conveniently in a conventional stack-based system.

Stacks are still used conceptually, and each stack frame is modelled as an individual data object. Stack frames represent the piece of stack required to implement each block or procedure execution of the source language. The size of each frame can be determined statically, which leads to an efficient use of the available working space.

There is a trade-off with this implementation of stacks. As the stack frames are heap objects, the persistent heap has a right to treat them like any other object. This means that all addresses used in machine instructions must be two-part addresses, consisting of a frame address and an offset within the frame. Thus the abstract machine may not assume an absolute address for any stack location, with a consequent loss of speed. This is considered to be a relatively minor disadvantage when contrasted with the gains of such a flexible architecture.

The rest of this section describes the format used for all heap objects both used and created by the machine. In particular, the layout of a stack frame object is shown, and the procedure entry and exit mechanisms explained. An example is given of how these frames are used to model block retention.

2.1. Uniform representation of heap objects

The persistent heap upon which the abstract machine is built is designed to use a single object format, no matter the purpose of the object. All objects have the following format:

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

where the header contains the following:

bits 8-31	the number of pointer fields in the object
bit 7	special purpose for use by abstract machine
bits 0-6	reserved for implementation experiments

This is very different from the PS-algol abstract machine, in which different types of object had information placed in their headers to allow the utility programs such as garbage collectors and object managers to discover the layout of the object. This coupled together the abstract machine, the compiler, the garbage collector, and the persistent object manager, making it impossible to change one without the others. The beauty of the new implementation is that the persistent store and its utility programs may be constructed completely independently from each other. This makes maintenance and change very much simpler, and allows for much freer experimentation with separate modules.

The format allows an object manager to know how to find the pointers in any object without requiring any knowledge about what the object may be used for. This allows it to follow store addresses at its own discretion. There are many reasons why this is desirable, not least that the store may now be responsible for its own garbage collection. It may, for example, be able to do this incrementally while there are no programs running. Other possibilities include distribution management, coherent object cacheing, and clustering.

This strict format calls for a small amount of extra work in the implementation and operation of the abstract machine, due to the constraint that all pointers in an object are contained in a contiguous block at the start of the object. This is not always a natural arrangement, and may cause some complication in the machine. The advantages of the arrangement, however, greatly outweigh the disadvantages.

2.2. Stack frames

Stack frames, along with all other objects used by the machine, are thus restricted to the format described above. To achieve this, they are laid out as follows:

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
word m+1..n	the main stack
word n+1	the lexical level
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)

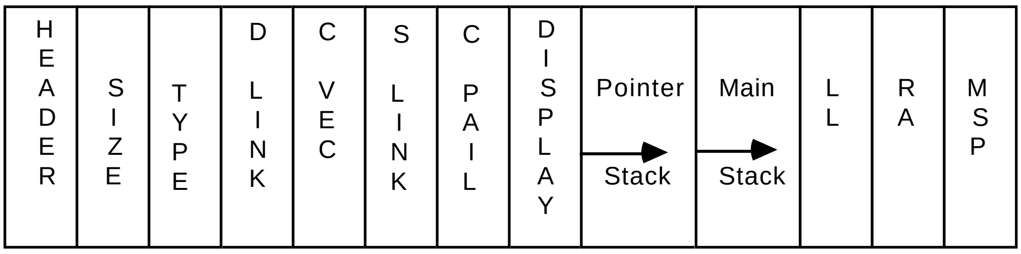

Figure 2. Stack frame layout.

Notice that the frame has a separate scalar (main) stack and pointer stack. This was a decision that was first made in the S-algol abstract machine, and continued into the PS-algol machine. Since a garbage collection could strike at any time in these machines, it was necessary to be able to find the pointer roots which were still required. The pointer stack naturally contains such pointers. Garbage collection using this format is the most efficient possible without resorting to a tagged architecture.

It is interesting that these separate stacks would in any case be enforced by the universal object format. This is because the object format itself is essentially designed by the same guiding principles as the previous machines' stack layouts.

2.3. Procedure entry and exit

Procedure entry consists of the following stages:

1. Load the closure
2. Evaluate the parameters
3. Apply the procedure

As a new context is entered, the code pointer and main stack top registers must first be saved. These are saved in the current frame, in the return address and main stack pointer locations. The pointer stack top does not need to be saved, as this may be calculated from the number of pointer fields in the frame's header on return.

The closure of the new procedure consists of the new static link and the new code vector. The locations of these items on the current pointer stack are known statically. A new object is created to act as the new procedure frame, the required size being found from the code vector. The local main stack pointer register is also set according to information found here.

The pointer stack may now be evaluated to the start of the new procedure's working stack. The new dynamic link is the old frame's address, the code vector and static link have already been accessed, and the PAIL pointer is found in the code vector. The old display is then copied to the new frame, and updated by pushing the new static link on top of it. The old lexical level is incremented and copied to the appropriate place.

The evaluated parameters may now be copied from the old stack frame to the top of the new stacks. The saved main stack pointer, and the number of pointers in the old frame, are adjusted to remove the parameters and the closure of the procedure being applied from the old frame's stacks. Now all that is necessary is to set the local frame base register to point to the new frame, and set the code pointer to the start of the abstract machine code in the current code vector, and the new procedure may start to execute.

The inclusion of both the dynamic link, which allows the dynamic call chain to be followed, and a pointer to the PAIL code from which the current code vector was generated, allows a great deal of debugging support, with a very small penalty in terms of execution speed.

2.4. The Block Retention Mechanism

The architecture described so far using individual stack frame objects and two-level addressing provides all the support necessary for a block retention scheme. This is required to support first class procedures which are allowed to have free variables, which may be shared between more than one procedure. Many languages have module constructs with essentially the same semantics. A simple example of this in Napier is:

```
let counter =
begin
        let a := 0

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

The location "counter" is associated with the value of the following block, which is the procedure literal of type **proc**(-> int). The code of this procedure first increments the location "a", and then returns "a" as the value of the block. The important point to notice is that the procedure literal value is in scope for longer than the free variable which it encapsulates.

In a traditional stack-based implementation of a block-structured language, the location used for the variable "a" would be free for re-use after the end of the block which contains the procedure. This does not work in this context, as the location may be subsequently used. Note too that it is the location itself, rather than a copy of its contained value, which is important. It is possible for two different procedures to use the same free variable, and so they must access the same location for the correct semantics to be preserved.

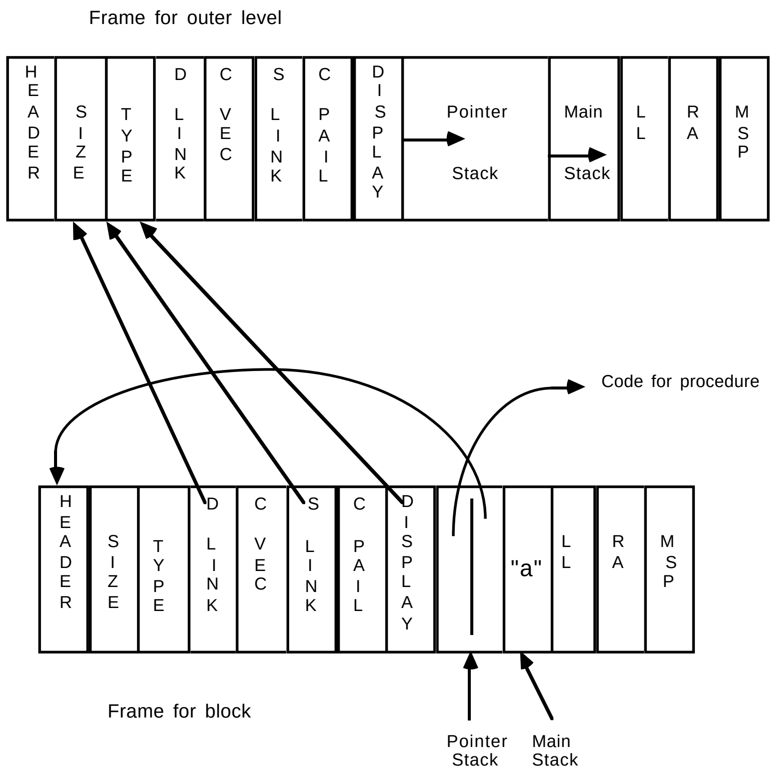

Figure 3. Just before the block exit.

The mechanism described above already performs block retention correctly with no further adjustment. When the main block is entered during execution, a new stack frame is created. The appropriate location in this frame is initialised with the integer value zero. A procedure is then pushed onto the pointer stack, consisting of the correct code vector and the environment, which points to the local frame. The situation is now as depicted in Figure 3.

The procedure value of this block is returned to the calling frame in the usual way. This consists of copying the procedure closure from the pointer stack of the block's frame onto the frame of the enclosing scope. The pointer stack of the block's frame is then retracted to remove the procedure value. This leads to the desired situation in Figure 4., where the variable "a" has been encapsulated in the value of the closure. The compiled code for the procedure contains the offset of "a" within the environment, and the location has been correctly preserved.

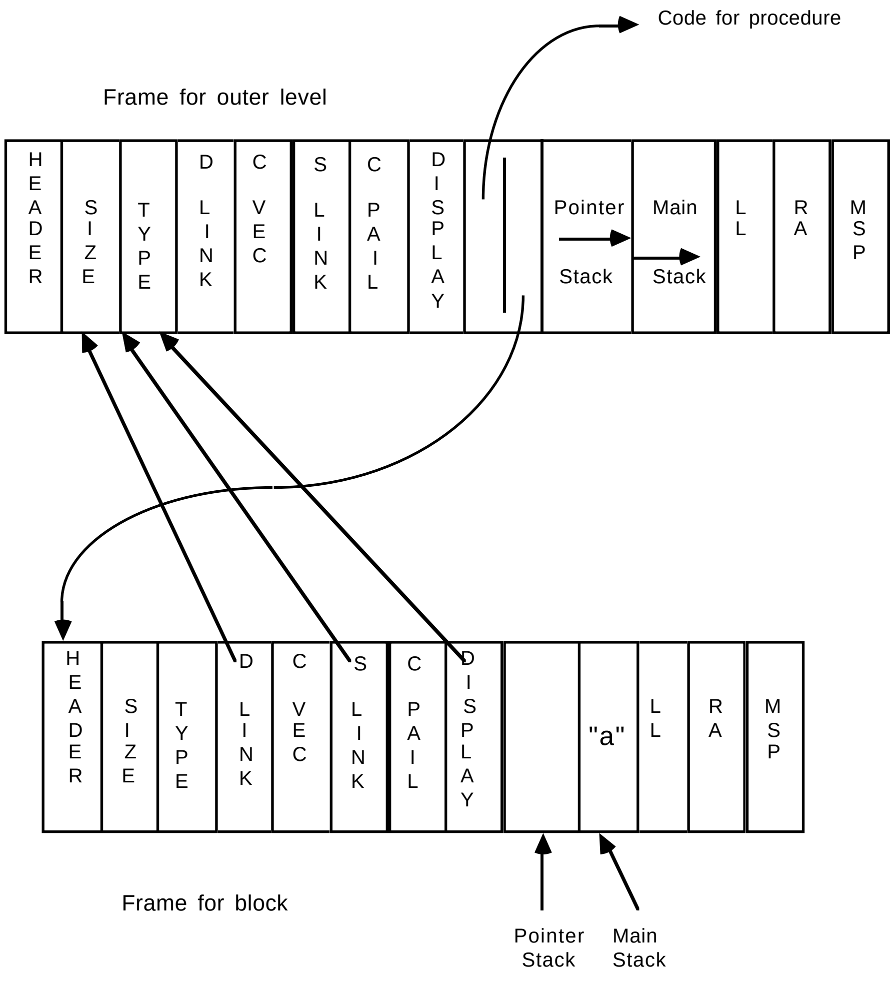

Figure 4. After the block exit.

At compilation time when the code for the procedure is produced there is enough information to know the offset of the location "a" within its frame, and this is planted statically in the procedure's code. If the location is within an outer block, the address of the relevant frame can not be known. What is known, however, is the location within the environment frame which contains the address of the desired frame. This is one of the locations in the display, depending on the lexical level at which the free variable is to be found.

As frames are heap objects just like any other, and not distinguished by the object management system, frames which do not contain any encapsulated locations may be garbage collected in the usual way. This will not happen to frames which contain required locations, as they are pointed to by the encapsulating procedure.

A potential hazard of this system is that other values in the kept frame will also not be subject to garbage collection, as it is not known which of the values within a frame are required. In fact, a great deal of optimisation is possible here and it is unusual for objects which are not required to be kept.

3. A low-level type system

A major design decision was made to have non-uniform representation of different types of objects in the machine. Some systems, particularly those which support polymorphism and other type abstractions, have a uniform representation in which every object is wrapped in a pointer to a heap object[13]. This allows type abstraction to be implemented easily, but has a drawback in efficiency. The Napier system has a number of different representations for objects on the machine's stacks, including some which have part of their value on either stack. This causes problems with stack balancing and object addressing, solutions to which are presented below.

The abstract machine has its own type system, albeit a very low-level and unenforced system. It is a two-level system, one level describing object layout and the other including some semantic knowledge of the object. Most of the abstract machine instructions are typed, although no attempt is made to ensure that the operand is of the correct type ; the type acts in effect as a parameter to the instruction.

Many language constructs involve operations where the type of the operands is not known statically. As equality is defined over all types in Napier, but is defined differently according to the type of its operands, it is necessary to perform a dynamic type lookup wherever the operand type is not known statically. This happens in the case of variants, polymorphic quantified types, and witnesses to abstract data types. A further need occurs when statically unknown types are assigned into or dereferenced from other data objects, when it is necessary to find the dynamic type to calculate the correct size and addressing information.

Both levels of the type system are finite, and contain only a small number of different types. The first level of the type system contains information as to the location and size of the instruction's operand. The machine supports six different types of objects, which are:

Operand shape	Instruction prefix
single word, main stack	w
double word, main stack	dw
single word, pointer stack	p
double word, pointer stack	dp
one word on each stack	wp
two words on each stack	dwdp

The instructions which are typed in this manner are those instructions which need to know only the shape of the object upon which they operate. These are instructions such as stack load, assignment, duplicate, and retract. The machine operations need to know nothing about the semantic nature of the objects in these locations.

The other, slightly higher level, system is required when the operation does depend on the semantics of the operand. These are all the operations which involve comparison of two objects, in which case the shape of the object is not sufficient. The machine supports different types with the same object formats. An example of this is structures and strings: they both consist of a single pointer, but equality is defined by identity on structures and by character equality on strings. The types supported are:

High-level type(s)	Instruction suffix	Equality semantics
integer,boolean, pixel,file	.ib	word equality
real	.r	double word equality
string	.s	heap object equality
structure,vector,image, abstract data type	.p	identity (pointer equality)
procedure	.pr	identity (code and closure)
variant	.var	described later
polymorphic	.poly	described later

These are the seven different classes of equivalence defined by the abstract machine.

It would be possible to do a certain amount of static checking on abstract machine code to ensure that this type system is not broken, which could perhaps be useful if an untrusted compiler was producing abstract machine code. However, incorrect store instructions, such as addressing off the end of an object, could not be statically checked from abstract machine code. This is the main danger in allowing untrusted compilers to access the abstract machine at this level. As there is only one persistent store for all users, it is essential that untrusted machine code is not used.

3.1. Machine type representations

As these types are used dynamically by the abstract machine, it is necessary to represent them in the most speed-efficient manner possible, so that the machine may procure the required information in the shortest possible time. At various points during the running of the abstract machine, these dynamic type representations are interrogated for the following information:

Total size of object
Size of object on pointer stack
Size of object on main stack

For this reason, integer representations for the types are chosen in such a way that this information may be calculated by fast arithmetic calculations from the representation. This is done by taking a binary representation of an integer, and splitting it into three fields:

Most significant bits:	Pointer stack size
Middle bits:	Further information
Least significant bits:	Main stack size

This means that the main stack size can be found by masking, and the pointer stack size can be found by shifting right. The total object size is calculated by adding these together. An alternative strategy would be to keep the total size, and calculate either the main or pointer stack size when required. However, the total size is required only for structure and vector creation, whereas the individual stack sizes are required for procedure application, which is expected to be the more common operation. The further information bits are needed to differentiate between objects of the same physical size with different semantics, like strings and other pointers.

There are no objects with more than two words on either stack, and only one further bit is necessary to differentiate between strings and the other single pointer types. This means that the types may be encoded in five bits thus:

Machine type	Binary representation	Decimal representation
Single word	00001	1
Double word	00010	2
Single pointer	01000	8
Double pointer	10000	16
String	01100	12
Variant	01001	9

3.2 Implementation of variants

Variants are language-level objects which have the semantics of a single labeled location which can be used, according to the label, to store an object of one of a number of different types. An example in Napier is:

```
rec type intList is variant( node : structure( hd : int ; tl : intList ) ;
                             tip : null   )
```

This describes the type of a location which, if it is a list node will be of the described structure type, and if it is the end of the list will be of unit type. These are the semantics of variants as described in [14].

The information needed to represent a variant location in the Napier machine is: the value currently in the location; a representation of the label associated with the branch of the variant which the value is in; and a representation of the abstract machine type of the value. As the machine type is constant for each branch of the variant, it is possible to encode the branch label and the type in a single integer value. This is known as the variant tag. A variant may now be implemented by a compound object consisting of one pointer and one scalar word. The pointer represents the value, and the scalar word is the tag.

The value is represented in different ways depending on its type. If it is a single pointer type, that is structure, vector, string, or abstract data type, then the value itself is used as the pointer value. If it is any other type, then it is wrapped in a structure. The reason for this is that most values used in variant locations are expected to be single pointer types, and this non-general solution is an optimisation based on this. It is because of the different equality semantics that the object's machine type is required to be stored with the tag; projection requires only a test for the correct label representation, and if this does not fail then the type of the value is known statically.

The scalar tag consists of an integer which contains both the label and the type of the object. The labels are encoded in the first twenty-four bits of the word, and the type in the last eight bits. For each variant type, the type-checker allocates integers for each branch of the variant in a consistent manner, so that the label will match the same in other structurally equivalent types. Projection is done by integer equality, as the type will always match if the label does.

The equality tests are performed by first doing an integer equality on the tag. If this fails, then the equality fails immediately. If it succeeds, then the appropriate bits are read from the tag to get the machine type of the value. This is enough information to know how to find the value, and what equality test to do on it.

3.3. Implementation of type abstraction

The support of language constructs which allow the manipulation of objects whose type is not known statically causes new problems with implementation. By static analysis, it is no longer possible to tell:

1. Which stack, and how many words, such an object needs.
2. Where to locate a field of a structure type object.
3. How to perform an equality operation on two such objects.

The Persistent Abstract Machine has polymorphic variants of all instructions whose operation depends upon which of the types is being manipulated. Stack balancing is performed by allocating two words to each stack whenever an object of statically unknown type is pushed onto a stack, thus making sure that there is enough room for it whatever its dynamic type is. This allows stack addressing to be performed statically even when the sizes of objects are non-uniform. If this strategy is used, then many instructions have polymorphic forms which do not need to know the dynamic type, as they simply operate on the relevant two words of each stack. This gives the idea that polymorphic object manipulation is a closed system, with the dynamic type only being needed when an object is entering or leaving the system.

For the operations which do need to know the dynamic type of the object, the information must be in a place where it may be dynamically found and interpreted. This is arranged by placing the machine type representation on the main stack of an artificially constructed frame on the static chain.

This is implementationally equivalent to either a quantified procedure being wrapped in a generating procedure, or an abstract **use** clause being wrapped in an extra block. The new outer level blocks have the necessary integer declarations - one for each type parameter. The compiler has enough information to know what or where the correct integer values are. It may be known statically, when a universally quantified procedure is specialised by a concrete type; its address is known if a quantified procedure is specialised by another parameter type; and it is information which is held as part of the structure of an abstract data type. The compiler now knows statically an address where the information necessary for the rest of the polymorphic machine instructions may be found.

For example, the polymorphic Napier procedure

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

would be compiled as if it were

```
let id = proc( t : int -> proc( ? -> ? ) )
begin
        proc( x : ? -> ? ) ; x
end
```

where the question marks may stand for any type, and the value of the integer parameter depends on the type. As this information is planted by the compiler it may be done safely ; all the type checking is still done statically and safely.

When the quantified procedure is specialised, the compiler plants code to call this generator procedure.

```
id[ int ]
```

will be effectively compiled to

```
id( 00001 )
```

and so cause a call of the procedure, with the result being the value

```
proc( x : int -> int ) ; x
```

and the machine type tag for integer planted in a known place in the static chain. When the compiler needs to know the machine type of an object for a polymorphic operation, it knows where it may be found.

For the implementation of abstract types, the same technique is used. The structure which contains the fields of an abstract type has extra fields which contain the concrete machine type corresponding to each witness type. This information can be planted on creation of the object. When an object of an abstract type is used, the compiler plants code to take this information out and place it in the outer block which is created.

3.4. Implementation of subtype inheritance

The same implementation technique may be used to implement bounded universal quantification when subtype inheritance is used over structure types[15]. For example, in the procedure

```
let noOfDoors = proc[ c <: car ]( x : c -> int )
        x( doors )

! ( this syntax means that the procedure is
!   universally quantified over any subtype of car )
```

it is not known statically where the doors field will be found in the structure x. However, the information which is missing may be planted statically by the compiler in a suitable place, on the procedure call. A generator procedure will be created in the same way as for a universally quantified procedure, only the parameters to this procedure when it is called, at the time of specialisation, will be the offsets of each of the fields which belong to the known supertype over which the procedure is declared. So if

```
type car is structure( doors : int ; fuel : string )
```

then the noOfDoors procedure will compile to something equivalent to

```
let noOfDoors = proc( doorsOffset,fuelOffset : int -> proc( <: car -> int ) )
begin
        proc( x <: car -> int ) ; x( doors )
end
```

where the compiler has enough information to know where to find the offset.

When the function is specialised, as in

```
let fordDoors = noOfDoors[ Ford ]
```

the compiler plants the offsets necessary to access the fields "doors" and "fuel" in an object of type Ford.

4. Conclusions

The major aspects of the Persistent Abstract Machine design have been described. The abstract machine is an integral part of an entire layered architecture, which may be used from any level downwards by a user. It has been shown how the abstract machine interfaces cleanly with the persistent store, and allows persistence to be straightforwardly implemented in a high-level language .

The heap-based storage architecture of the machine supports a method of block retention for almost no extra cost. This allows the implementation of first class procedures and modules in programming languages with the minimum of effort.

A primitive type system contains just enough information to allow machine instructions which behave differently according to the dynamic type of the operands. This type system has a fast and efficient integer encoding.

Finally, this type system in conjunction with the block retention architecture may be used to great effect to provide a fast implementation of polymorphic procedures, abstract data types, and bounded universal quantification.

5. References

1. A. Albano, L.Cardelli and R. Orsini
   Galileo : a strongly typed interactive conceptual language.
   ACM Transactions on Database Systems 10(2), 230-260 (1985).

2. PS-algol Reference Manual, 4th Edition.
   Universities of Glasgow and St Andrews PPRR-12-87, 1987

3. R. Morrison, A. Brown, R. Carrick, R. Connor & A. Dearle
   The Napier Language Reference Manual
   University of St Andrews, 1988

4. A. Dearle
   Constructing Compilers in a Persistent Environment
   2nd International Workshop on Persistent Object Stores, Appin, August 1987

5. A. Dearle
   A Persistent Architecture Intermediate Language
   Universities of Glasgow & St Andrews PPRR-35-87, 1987

6. A. Brown, R. Carrick, R. Connor, A. Dearle & R. Morrison
   The Persistent Abstract Machine
   Universities of Glasgow & St Andrews PPRR-59-88, 1988

7. A. Brown
   A Distributed Stable Store
   2nd International Workshop on Persistent Object Stores, Appin, August 1987

8. PS-algol Abstract Machine Manual
Universities of Glasgow & St Andrews PPRR-11-85, 1985

9. P. Bailey, P. Maritz & R. Morrison
The S-algol Abstract Machine
University of ST Andrews CS-80-2, 1980

10. A. Goldberg & D. Robson
Smalltalk-80. The Language and its Implementation
Addison-Wesley, 1983

11. M. Atkinson & R. Morrison
Procedures as Persistent Data Objects
ACM TOPLAS 7(4) October 1985 539-559

12. D. McNally, A. Davie & A. Dearle
A Scheme for Compiling Lazy Functional Languages
University of St Andrews, Staple/StA/88/4, 1988

13. L. Cardelli
Compiling a Functional Language
Proc. 1984 LISP and Functional Programming Conference
Austin, Texas August 1984

14. L. Cardelli
A Semantics of Multiple Inheritence
Proc. International Symposium on the Semantics of Data Types,
Sophia-Antipolis, France, June 1984

15. L. Cardelli & P. Wegner
On understanding types, data abstraction and polymorphism
ACM Computing Surveys 17, 4 (December 1985), 471-523.