Abstract

A language construct called *environment* is introduced. Environments are collections of bindings and have first class data rights. As such, they provide the programmer with a type secure mechanism to control bindings in the system. It is shown that environments may be combined to provide a naming graph that subsumes the functions of file systems in traditional operating systems. Such a mechanism provides a conceptually simple framework for manipulating bindings which permits the control of complexity and system evolution from within a unified language framework.

1. Introduction

A persistent store is a conceptually unbounded space populated by objects. In such a space, the naming of objects becomes a problem: the use of names is highly restricted by the names that have been previously used. This may be observed in programming systems which adopt a simple, flat object naming strategy, such as Smalltalk-80 [13]. These systems have a single name space in which names are introduced, resulting in the use of names being 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 objects in different contexts. Contexts are used in everyday life to overcome naming problems; for example, when I say that 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. As another example, to someone who knows a beardless person called Ron, the sentence could be untrue. To an non-English speaking person, 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 the program fragment shown in Example 1. The bindings contained in this static piece of program may be represented by the tree shown in Figure 1.

```
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 structured contexts.

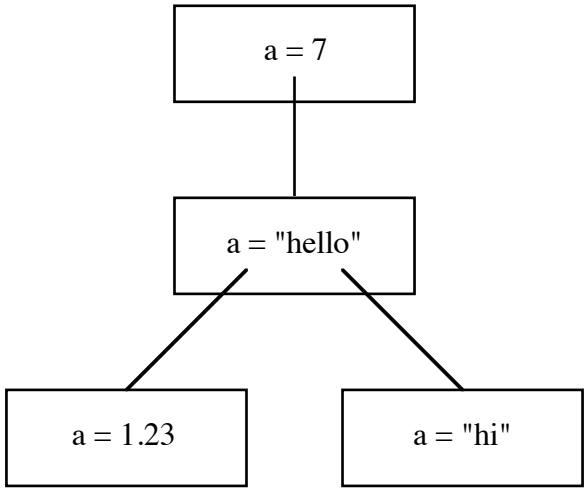

Figure 1. A graph of block structured contexts.

Notice that each instance of the clause "**write** a", displays the value of different objects, due to the imposition of the context tree. The tree shown in Figure 1, derived from the block structure of Example 1, is static in nature. That is, the contextual structure is assembled by the compiler. 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, for example, in Lisp [16].

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 context tree shown in Figure 1 is a special case of a more general structure, that of a graph. In general, the persistent object space comprises a graph of objects. The environments discussed in this paper provide extensible contextual naming on this graph.

The controlled evolution of data, including program, is of paramount importance in large systems. In particular, persistent systems of data and program must be partially reconstructible and thus incrementally enhanceable. Therefore, persistent systems need to provide binding mechanisms to support incremental development. The understanding of these mechanisms and their significance in the persistent environment is an important consideration in the provision of contextual naming strategies. Binding mechanisms are therefore discussed in the next section as background to this paper.

In the following section, a historical perspective is given, showing the various mechanisms researchers have used in an attempt to provide contextual naming in programming languages and databases. Included in this survey are the languages Clear [6], Pebble [7], Galileo [1] and an earlier proposal aimed at providing an extensible naming mechanism for Napier [4].

The main part of this paper introduces a language construct called ‘environment’ and written **env**. Objects of type **env** are collections of bindings and have first class data rights. As such, they provide the programmer with a type secure mechanism to control bindings in the system. It is shown that environments may be combined to provide a naming graph that subsumes the functions of file systems in traditional operating systems. Next, the paper shows that such a mechanism provides a conceptually simple framework for manipulating bindings which permits the control of complexity and system evolution from within a unified language framework. Finally, some conclusions and a description of further work are presented.

The work presented in this paper is part of a larger project – that of providing a persistent programming environment. This research effort is more fully documented in the author's Ph.D. thesis [11].

2. Background

2.1 A model of binding

Traditionally, a binding consists of a name-value pair [20]. That is, a value is bound to a name for some period during the evaluation of a program. This has been extended by Burstall & Lampson to include a type [7] and further by Atkinson & Morrison to add constancy [4]. Bindings can therefore be regarded as comprised of a four-tuple [18]:

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

We bindings will described as follows:

{name, value, type, constant}

Thus, the PS-algol constant binding (denoted by '=')

let a = 7

may be written:

{a, 7, int, true}

and the variable binding (denoted by the ':=' symbol)

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

These properties 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 [20]. 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 [21] 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 contain only static binding.

However, in order for a system to evolve, a measure of dynamic binding must exist. 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.

In determining the appropriate binding mechanisms for a particular system, the designer is faced with the problem of balancing safety against flexibility. The safety in the system is derived from being able to say (even prove) something about the program before it runs (i.e. statically) in order to improve confidence that it is correct. This explains the wish by most language designers to employ static type checking as one of the devices for static checking.

A second aspect of static checking is that the programs so checked are usually more efficient. By performing the checking statically, the need for dynamic checking is removed making the run time representation of the program execute faster and in less space.

Finally, an aspect of static checking that is often overlooked in programming systems is that of the source code on program documentation. If a compiler can statically check a program, then so can another user. Thus statically checked programs have better documentation properties and consequently better cost properties throughout their life cycles.

Therefore, a delicate balance exists 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.

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 since types are known, 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.

2.2 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 changes 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.

3. Previous Language Approaches

The environments described in Section 4 of this paper support programming in the large by providing incremental program construction mechanisms and contextual naming facilities. This problem has also been tackled by other researchers. Some of the approaches taken, both in the field of databases and in programming languages, are discussed in this section.

3.1. Clear and Pebble

In the specification language Clear [6], Burstall and Goguen identify the three main operations on environments (used in the Scott-Strachey sense of the word) as:

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

In the later language, Pebble [7,8], bindings are treated as first class data objects, where a binding is defined as either a name bound to a value or a pair of bindings. Unlike Clear, there does not appear to be any mechanism for creating an empty binding. However, bindings containing a single name-value pair may be created as follows,

LET E:~[x~3]

where 'E' has as its value the binding "x bound to 3". In this example, 'E' has type (x:int). In the notation used earlier, the value of 'E' is

{x, 3, int, true}

Since Pebble is a purely applicative language, extending an binding produces a completely new binding. For example, a new binding called 'EE' may be produced as follows,

LET EE:~[E,b~7]

and 'EE' has the following value,

{{x, 3, int, true},{b, 7, int, true}}

In Pebble, 'EE' has the following type,

(x:int)×(b:bool)

Pebble allows values associated with a given names to be extracted. For example, if the programmer wished to extract the value bound to 'x' from the binding 'E', the following could be written:

LET i:~E **IN LET** i **IN** x

which has the value 3.

In the papers on Pebble, Burstall states that programming in the large will become merely typed functional (applicative) programming. However, it is not obvious how applicative languages may help with incremental system construction since the must, by their very nature, be statically bound. In fact, the bindings of Pebble are also statically type checked so are more like the structure classes provided in many languages than the environments provided by Napier. For these reasons, there will be no further discussion of purely applicative (static) languages.

3.2. Galileo

The language Galileo [1] 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}. The expression a + b may then be evaluated in this environment.

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. Galileo 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, the following dialogue may be carried out in Galileo:

```
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 schema 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 for Clear.

3.3. Name Spaces

In [4], Atkinson and Morrison introduce name spaces. Name spaces were proposed to form part of the language Napier, then in its infancy. However, they were never implemented and were eventually replaced by the environment mechanism described in this paper. Name spaces were designed to 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 the names available in a name space; and
5. the safe exchange of arbitrary data between parts of a system.

A name space is created by the following expression:

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

This is best illustrated with an example, such as that shown in Example 2.

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

Example 2. Name space instantiation.

Here ‘new’ has as its value the following set of bindings, using the notation introduced earlier:

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

Notice that, unlike the environment expressions of Galileo, name spaces are first class data objects. Note also that the name space does not contain the binding

{aa, 49, int, false}

This is because only the bindings denoted by names in the identifier list after ‘**ns**’ are exported from the block into the name space.

In order to accommodate change, name spaces provide a mechanism to add new bindings and to remove old bindings. A new binding may be added to a name space using following construct:

extend <name space 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** <name space expression>

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

using <name space expression> **with** <signature> **compile**
    <sequence>
end compile

This notation is used to denote dynamic binding to a name space. The name space expression yields a value of type name space. The type of the name space is checked dynamically to ensure that it satisfies (i.e. includes) 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 name space. Thus, the signature specifies a formal store and the name space expression provides an actual store each time the statement is executed. This is similar to the situation with regard to parameters in procedure definition and application, in which the formal store is specified by the formal parameters and the actual store is provided by values at application time. The dynamic binding is thus localised to the scope of the **using** .. **compile** construct.

Name spaces may also be used statically, the notation for which is:

using <name space> **compile**
    <sequence>
end compile

Here the sequence is statically bound to the name space. Notice that in this case the name space must be statically determinable, which is indicated by the fact that we must specify a name space rather than permitting the use of a name space expression.. However, despite the static nature of the binding, the name space 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 name space using drop. Indeed, an error condition or exception may arise at run time due to a required binding not being present in the name space. This is shown in Example 3.

```
let new = ns a from          ! Define name space containing
        let a = 7            !   one binding {a,7,int,true}.
end from

let useNew = proc()          ! Define a procedure which
begin                        !   uses 'new' by binding to it
        using new compile    !   statically and writes out
                write a      !   the value of 'a'.
        end compile
end

drop a from new              ! Drop the binding from the
useNew()                     !   name space; then call the
                             !   procedure will cause an
                             !   exception when 'a' is
                             !   checked.
```

Example 3. Using a name space.

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 binding is made to the name space itself and not to the bindings contained in the name space.

4. The datatype environment in Napier

The programming language Napier [19] 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 provide the three main operations defined in Clear, with the addition of one new operation – the dropping of bindings. The operations on Napier environments are therefore:

1. create an empty environment;
2. extend an environment with a
   (name, value, type, constancy) 4-tuple;
3. extract the location associated with a name; and
4. remove a binding from an environment.

4.1 Creating an empty 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, i.e. 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 the program shown in Example 4.

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

Example 4. An illustration of traditional block structure.

4.2 Extending an environment

Bindings are also introduced into 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 the code shown in Example 5.

```
let e = environment()    ! Create a new environment ‘e’.
in e let a = 7           ! Add the binding {a, 7, int, true}
                         !   to ‘e’.
```

Example 5. Static use of environments.

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}

Note that the introduction of the binding {a, 7, int, true} to ‘e’ does not affect the static environment. An exception is generated if a name is added that has already been used to identify another binding in the environment. The example shown in Example 2 above would be written in Napier as shown in Example 6.

```
let locala = 7                    ! Declare 'locala' in local env.
let new = environment()           ! Declare 'new' in local env.
in new let a = locala             ! Declare 'a' in 'new'.
in new let b := proc( -> int )    ! Declare 'b' in 'new'.
                begin
                    locala * locala
                end
```

Example 6. Instantiation of environments.

4.3 Projecting out of an environment

Bindings may be extracted (projected) from environments 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 value associated with 'a', the programmer would write the program shown in Example 7.

```
use new as a : int in
begin
    writeint( a )
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 and return the value associated with the name 'a' from the environment, the following could be written,

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

This is equivalent to the following piece of code written using traditional language constructs,

```
let a = 7
let valueOfa = a
```

4.4 Dropping bindings from environments

Bindings may be removed from an environment using the **drop** construct. This has both the same syntax and semantics as in the case of name spaces, namely,

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

This expression removes the binding containing the identifier from the environment specified in the environment clause. The effect of dropping a binding is illustrated in Example 8. Note that the binding is not deleted, merely removed from the environment. This distinction, which is very important, is also shown in Example 8.

```
let new = environment()
! 'new' is an empty environment.
in new let a = 7
! 'new' now contains the binding {a, 7, int, true}.
use new as a : int in begin
                          in new let aa := a * a
                      end
! 'new' now contains the bindings {a,7,int,true} and
!   {aa,49,int,true}.
drop a from new
! 'new' now contains only the binding {aa,49,int,true}.
```

Example 8. Dropping values from environments.

4.5 Binding to environments

Recall that a difficulty arose in Example 3 concerning the use of **using** and **drop** with name spaces. The problem with name spaces was that the binding was always to the name space and not to the bindings stored in the name space. Example 9 is semantically equivalent to Example 3; as with Example 3, it will cause an exception to occur on the last line.

```
let new = environment()       ! Define an env containing
in new let a = 7              !   one binding {a,7,int,true}.

let useNew =  proc() ;
              begin
                    use new as a : int in
                          writeint( a )
              end

drop a from new         ! Drop the binding from the env.
useNew()                ! 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 shown in Example 10. This example will write out the value 7, rather than raising an exception.

```
let new = environment()         ! Define an env containing
in new let a = 7                !   a binding {a,7,int,true}.

let useNew =  use new as a : int in  ! Define a proc
        proc() ; writeint( a )       !   which writes out the
                                     !   value of a.

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

Example 10. Example 3 with the desired semantics.

The difference between the two examples is in the difference between the lines

```
let useNew =  proc() ;
              begin
                    use new as a : int in
                          writeint( a )
              end
```

in Example 9, and

```
let useNew =  use new as a : int in  ! Define a proc
        proc() ; writeint( a )       !   which writes out the
                                     !   value of a.
```

in Example 10.

In Example 9, 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 Example 10, 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 that environments exhibit the same semantics as block structure. That is, the bindings made by a **use** clause are direct L value bindings.

4.6 Type checking

The type **env** is the infinite union of all labelled cross products. That is, all environments have the same type, namely **env**, and members of this type may contain zero or many {name, value, type, constancy} tuples. Furthermore, there are an infinite number of members of this set whose tuples are drawn from the infinite number of types, values and names which may be constructed from the type, value and name space, respectively.

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 allows 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 type env permits 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 types 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 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 [14], 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 and 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 [10,12]. 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 [9], the user could interactively construct programs to operate against data held in the information space. In such a programming 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. Research is currently being undertaken to blurr this distincion further in order to provide a more integrated support environment for persistent programming [15].

4.7 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 the program shown in Example 11.

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

Example 11. Scope and block structure in algol-like languages.

Similarly, in Napier, the programmer may write the code shown in Example 12.

```
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
        writeint( a )          ! Writes out 6.
        a := 4
        writeint( a )          ! Writes out 4.
    end
    writeint( a )              ! Writes out 7.
    a := 32
    writeint( 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 block structured paradigm is widely believed to be a good one, the only problem with block structure is that the totally static composition it provides restricts software evolution. The environment construct is designed to permit the same program structuring facilities as block structure whilst permittiing change.

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 binding and constant value bindings are R-value bindings. The bindings stored in an environment exhibit the same behaviour – all variable bindings are to locations and all constant bindings are to values.

4.8 Binding to the persistent store

In Napier, the root of the persistent store is called 'ps' and is of type env. Any data that is reachable from 'ps' is persistent. In persistent systems, the root of persistence must always be an extensible data structure of some kind to permit change in the system. This explains why both Galileo and Napier use similar mechanisms.

Napier uses reachability as its persistence criterion and making any data structure persistent is simply a matter of binding that data structure to something reachable from 'ps'. For example, suppose that a binary tree, for simplicity over integers, has been defined in a program and that an instance of such a tree is to be made persistent. This may be performed as shown in Example 13. The final line of this program binds data structure bound to 'atree' in the local environment, to be bound to 'savetree' in the persistent environment 'ps'. In order to use this data structure in another program, the user may write the code shown in Example 14.

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

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

in ps let savetree = atree
```

Example 13. Binding to the persistent store using environments.

```
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 is a node with value : " )
        writeiint( savetree'node( val ) )
end
else writes( "It is a tip" )
```

Example 14. Accessing 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. Note that the type information must be stored in the environment so that the system may check that the object in the environment is of the expected type. In order to perform this check, the type must also be declared in the program which uses the environment. This check is performed in the use statement.

In general, the persistent store will form a graph consisting of environments and data bound to those environments. An example of a graph of persistent objects is shown in Figure 2. The environment at the root of persistence called 'ps' contains a few objects 'a','b' and 'c' and two environments called 'trig' and 'al'. The environment 'trig' contains a few objects which we will assume to be the normal trigonometric functions. This environment is also reachable from the environment called 'al'.

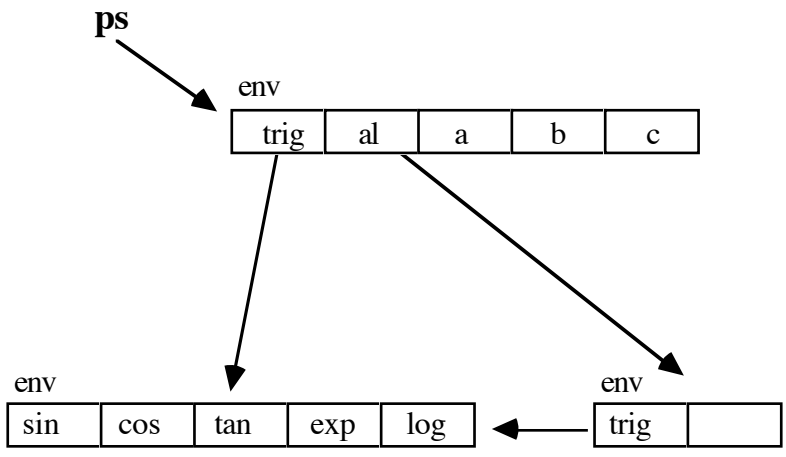

Figure 2. The persistent store as a graph of environments.

This graph of environments subsumes the function of traditional file hierarchies, 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 [5] to be one of the most important features required of new generation operating systems.

4.9 Supporting incremental construction

The way in which environments may be used to support incremental system construction is now examined. 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. After the completion of the dialogue, the persistent store will be arranged as shown in Figure 3. The effect of the interaction is to place a new function called ‘square' in the environment called ‘al’. Notice that the procedure called ‘square’ has the location of the procedure ‘exp’ bound into it. Thus, if the programmer assigned another value to the location ‘exp’, the function ‘square’ would also change.

```
! Only ‘ps’ is in scope at the beginning of the session.
! First introduce the environment ‘trig’ into the local env.
let trig = use ps as trig: env in trig
! Next, declare ‘square’ in the local environment,
! which uses ‘exp’ from the environment ‘trig’.
let square = use trig as exp:proc( int,int -> int ) in
    proc( a : int -> int ) ; exp( a,2 )
writeint( square( 7 ) )       ! Test out ‘square’
49
! The system writes out 49; we are satisfied and so save
!  ‘square’ in the environment called ‘al’.
use ps as al : env in
begin
    in al let square := square
end
```

Example 15. The incremental construction of a program.

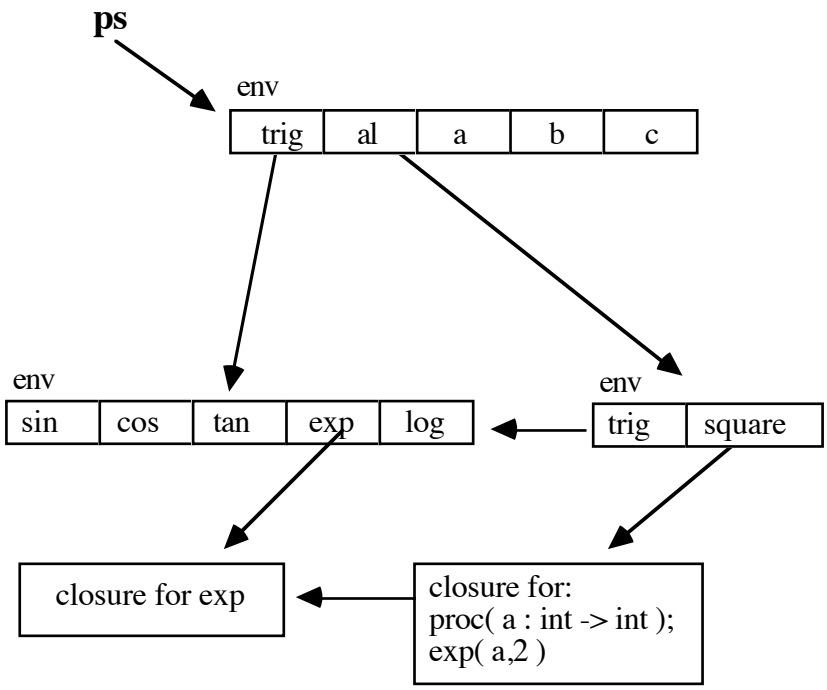

Figure 3. The persistent store after the interaction in Example 15.

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 situation, Example 15 could be rewritten as shown in Example 16.

```
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
begin
      in al let square := square
end
```

Example 16. Alternative version of Example 15.

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 Figure 3 not only exhibits graph structure in terms of data structures and environments, but also with respect to 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 [17] 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 interaction shown in Example 17.

```
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.

5. Conclusions

The problems of building large systems have been known for many years. If these problems are to be overcome, mechanisms to control names and control system evolution must be provided in programming systems. These two areas have been observed as being of particular importance in the PISA project [2]. For example, during the development of the persistent object browser [10], 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.

Control over binding mechanisms is extremely important in large persistent information spaces, as has been foreseen by myself and others [1,4,5,6]. In response to these problems, the datatype environment was introduced in order to provide a mechanism that would allow incremental construction and change within a large system.

The environment mechanism provides a contextual naming scheme that can be composed dynamically. The **use** clause may be nested and environments involved calculated dynamically or statically, thus permitting the name bindings to be constructed dynamically. This does not yield full dynamic scoping in the Lisp sense, since all objects in the individual environments are statically bound. The technique complements the block structure in the language and completes the context mechanisms required for persistent information spaces.

Environments therefore provide the programmer with a number of binding mechanisms and styles with which to bind objects. Used with care, this mechanism provides the power needed to evolve large flexible persistent object stores containing valuable information.

Thus, 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.

6. Further work

Problems still remain in this area, in particular, how functions like 'ls' in Unix may be expressed in a strongly typed system [3]. The problem here is that functions like 'ls' operate in an untyped environment. It is difficult to see what type a function like 'ls' would have in a strongly typed environment. Another difficulty in this area is that the names stored in environments are names and not strings. If functions like 'ls' are to be written over environments some capability is necessary to turn names into strings and vice versa.

The implementation of environments in the Napier system is ongoing and is expected to be complete soon. The Napier language also features first class procedures, objects with universal and existential types, graphical objects and, most importantly, orthogonal persistence.

6. Acknowledgements

I would like to thank Ron Morrison for his help and advice during this research and the preparation this paper. I would also like to thank the referees, for their many useful comments, and Chris Marlin for his advice and helpful criticism.

References

[1] Albano A., Cardelli L. & Orsini R. Galilieo: A strongly typed, interactive conceptual language. *ACM TODS*, **10**, 2 (1985), pp. 230-260.

[2] Atkinson M.P., Lucking J., Morrsion R. & Pratten G. Persistent Information Space Architecture club rules. Persistent Programming Research Report 47, Universities of Glasgow and St Andrews, Scotland (1986).

[3] Atkinson M.P. & Morrison R. Polymorphism, Type checking and magic in a type secure persistent object store. In *Proceedings of the Second International Conference on Persistent Object Systems*, Appin, Scotland {Persistent Programming Research Report 44, Universities of Glasgow and St Andrews, Scotland, (1987)}, pp. 1-12.

[4] Atkinson M.P. & Morrison R. Types, bindings and parameters in a persistent environment. In *Data Types and Persistence*, Atkinson M.P., Buneman P. & Morrison R. (Eds.), pp. 3-20 (Springer-Verlag, Berlin, 1988).

[5] Baltzer R. Living in the next generation operating systems. In *Information Processing 86* {Proceedings of IFIP 10th World Computer Congress, Dublin, Ireland}, Kugler H.J. (Ed), pp. 283-291 (North-Holland, Amsterdam, 1986).

[6] Burstall & Goguen J.A. The semantics of Clear, a specification language. Lecture Notes in Computer Science, **86**, pp. 292-332 (Springer-Verlag, Berlin, 1984).

[7] Burstall R. & Lampson B. A kernal language for abstract data types and modules. In *Proceedings of the International Symposium on Semantics of Datatypes* (Springer-Verlag, Berlin, 1984).

[8] Burstall R. Programming with modules as typed functional programming. In *Proceedings of the International Conference on 5th Generation Computer Systems*, Tokyo, Japan (1984).

[9] Dearle A. Constructing compilers in a persistent environment. In *Proceedings of the Second International Conference on Persistent Object Systems*, Appin, Scotland {Persistent Programming Research Report 44, Universities of Glasgow and St Andrews, Scotland, (1987)}, pp 443-455.

[10] Dearle A. & Brown A.L. Safe browsing in a strongly typed persistent environment. *The Computer Journal* (Dec. 1988), to appear.

[11] Dearle A. On the Construction of Persistent Programming Environments. Persistent Programming Research Report 65 {Ph.D. Thesis}, Universities of Glasgow and St Andrews, Scotland (1987).

[12] Dearle A., Cutts Q. & Kirby G. Browsing, grazing and nibbling persistent data structures. In *Proceedings of the Third International Conference on Persistent Object Systems*, Newcastle, Australia (January 1989).

[13] Goldberg A. & Robson D. *Smalltalk-80: The language and its implementation* (Addison Wesley, Reading 1983).

[14] Harper R., MacQueen D. & Milner R. Standard ML. Technical Report ECS-LFCS-86-2, University of Edinburgh, Edinburgh, Scotland (1986).

[15] Marlin C., Dearle A. & Morrison R. Integrated persistent programming environments. Persistent Programming Research Report, Universities of Glasgow and St Andrews, Scotland, *to appear*.

[16] McCarthy J. et al. *Lisp programmers manual* (MIT Press, Cambridge, Mass. 1962).

[17] Morrison R., Brown A.L., Connor R. & Dearle A. Polymorphism, persistence and software reuse in a strongly typed object-oriented environment. *Software Engineering Journal* (Dec 1987).

[18] Morrison R., Dearle A. & Atkinson M.P. Flexible incremental bindings in a persistent object store. *ACM SIGPLAN Notices*, **23**,5 (1988).

[19] Morrison R., Brown A.L., Carrick R., Connor R. & Dearle A. The Napier Reference manual. University of St Andrews, St.Andrews, Scotland (1988).

[20] Strachey C. *Fundamental concepts in programming languages*,.(Oxford University Press, Oxford, 1967).

[21] Turner, D.A. SASL language manual. Technical Report CS/79/3, University of St.Andrews, St.Andrews, Scotland (1979).