This paper should be referenced as:

Dearle, A. & Brown, A.L. “Safe Browsing in a Strongly Typed Persistent Environment”. Computer Journal 31, 6 (1988) pp 540-544.

Safe Browsing in a Strongly Typed Persistent Environment

A.Dearle
A.L.Brown

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

0334 76161 X8100

3500 words
16 pages

Abstract

The need to examine data structures often occurs in programming language and database management systems. In this paper we describe how a browser for a strongly typed programming language (PS-algol) may be written in a type secure manner in a closed persistent environment. This is achieved without resorting to magic or having to break the type rules of the language by exploiting a compiler that is a object with full civil rights in the environment. The use of impact of such an object is discussed.

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 how a complex and highly dynamic system is behaving. 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 a part of the programming language tool set. In a persistent environment the data structures of the programming language and the long term data structures are the same. In such an environment, we have observed browsing tools to be especially useful. In this paper we will report on a method of construction 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 [1,2] with a few exceptions [3] 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.

We will show that in a persistent programming language PS-algol [4] it is possible to write a browsing program which displays the language's own data structures without resorting to programming conventions, having built-in functions or using a lower level of interpretation. The program is however allowed to discover the types of

objects by a mechanism in the language environment. The technique demonstrated utilises a compiler which is a *first class citizen* in the language environment. We anticipate that a first class compiler will be of use in other fields such as artificial intelligence and database management systems.

PS-algol

The language PS-algol is a strongly typed, general purpose persistent programming language. PS-algol is, as its name suggests, a member of the algol family of languages. However, it differs from the other algols in three important ways:

1. Procedures are first class objects in the language.
2. The language supports orthogonal persistence.
3. PS-algol has powerful inbuilt graphics facilities.

Procedures have the same *civil rights* as any other data object in the language. That is they are assignable, may be the result of expressions, other procedures or blocks, and may be elements of structures or vectors. The power of first class procedures have long been known to applicative programmers and are discussed in the context of persistence in [5].

The persistence of a data object is defined as the length of time that data object exists and is usable. In most programming languages data cannot outlive the program activation without the use of an external agency such as a file system or database management system. In a persistent programming language data can outlive the program and is treated in a uniform manner be it long term or short term data. This concept is fully discussed elsewhere [6].

PS-algol has powerful raster and vector graphics facilities which are an integral part of the language [7]. In this paper we will only use one graphics procedure, the *menu* function which we discuss here. 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 ) )
```

In PS-algol a structure class is a tuple of named fields with any number of fields of any type. The structure statement adds to the current environment a binding for the class name and a binding for each field name. When a structure is created it yields an object of type **pntr.** Objects of type **pntr** may be tested by the predicate **is** to determine if the referenced object is of a particular class. The type **pntr** comprises the infinite union of labelled cross products. When we dereference a pointer we are projecting out of a union therefore a run-time check is necessary to ensure that the dereference is legal. Apart from vector bounds checking this is the only run-time checking done in the system with all other type checking being performed at compile time. In this way the PS-algol structure classes provide a mechanism for maintaining a high degree of static type checking whist retaining control over when an object is bound.

PS-algol supports a data structure, implemented in PS-algol, called a table. Tables are indexes from strings to values of type **pntr**. Entries are placed in a table using the procedure *s.enter* which takes an associative key, the table and a value to be stored. The procedure *s.lookup* retrieves a value by supplying it with a key and a table. Note that in PS-algol the dot is part of the name and does not represent a dereference as in many other programming languages.

The Persistent Store

The Persistent Store comprises a conceptually infinite graph of PS-algol data objects. The graph has distinguished points which are the roots of persistence. Any object which is in the transitive closure of these roots become persistent. The persistent store grows as users link new objects into the persistence graph. The number of different classes of objects in the store also grows as users compile programs which intoduce new types into the system. The persistent store is therefore unbounded in variety and magnitude and may be viewed pictorially like this:

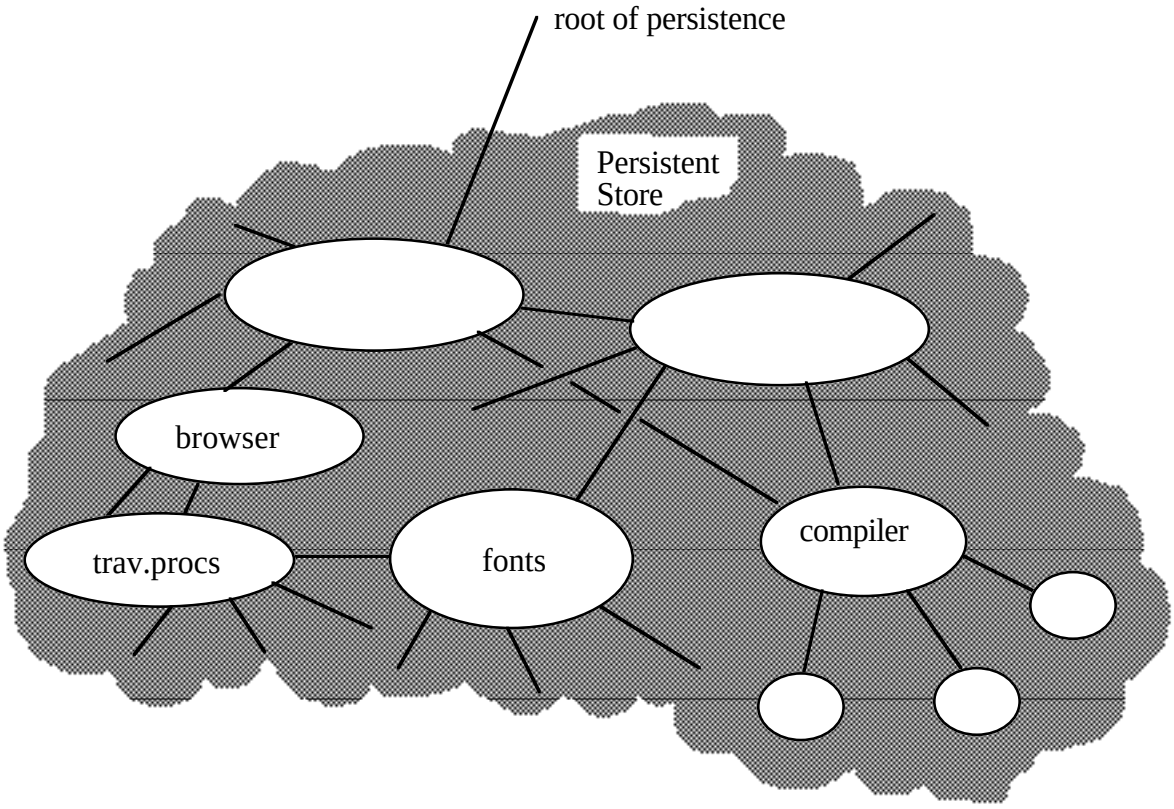

Figure 1

A Simple Browser

Let us now define what we require our browser to do. 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 2 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 previous selection (if any).

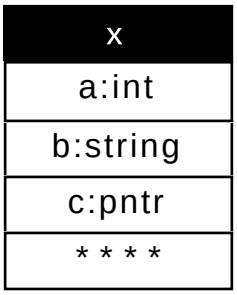

Figure 2

A PS-algol program to draw the menu shown in Figure 2 might look something like this:

```
let traversex = proc( pntr p )
begin
    structure x( int a ; string b ; pntr c )        ! the structure class traversex
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 over 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 procedures
```

```
        if p is x   then this.menu( 20,20 )        ! display menu on the screen at 20,20
                    else Error()                   ! take some error action
end
```

Figure 3

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 we could generalise this procedure 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**,we can never anticipate statically all the structure classes that the procedure may come across. The procedure *Trav* which is called from the menu is faced with this problem since we do not know which member of this infinite union *c* may be pointing at.

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 would operate on different classes their interface would be the same; that is they would all be 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 we ran the following program,

```
structure x( int a ; string b ; pntr c )
let p = x( 7,"abc",nil )
write class.identifier( p )
```

The following would be written out:

```
x( int a
string b
pntr c
)
```

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. 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
```

Figure 4

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 figure 3. Notice that we may add new procedures 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 almost 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.

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 to be able to do this we need another function in our persistent environment - the compiler.

Currently the compiler is of the following form,

let *compile* = **proc**(**string** *s* ; **pntr** *p*-> **pntr**)

The compiler is passed a string containing the 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 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.

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 Figure 3 above
end
```

Figure 5

Therefore we need to compile a procedure of type:

proc(**proc**(**pntr**) -> **proc**(**pntr**))

like that in figure 5 which returns a procedure capable of displaying a structure of a particular class.

We will now show that 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()                           ! """" is the " character
                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
```

Figure 6

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
```

Figure 7

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 so the *mktrav.proc* procedure returns a display generating procedure which is passed *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. This may be viewed pictorially as:

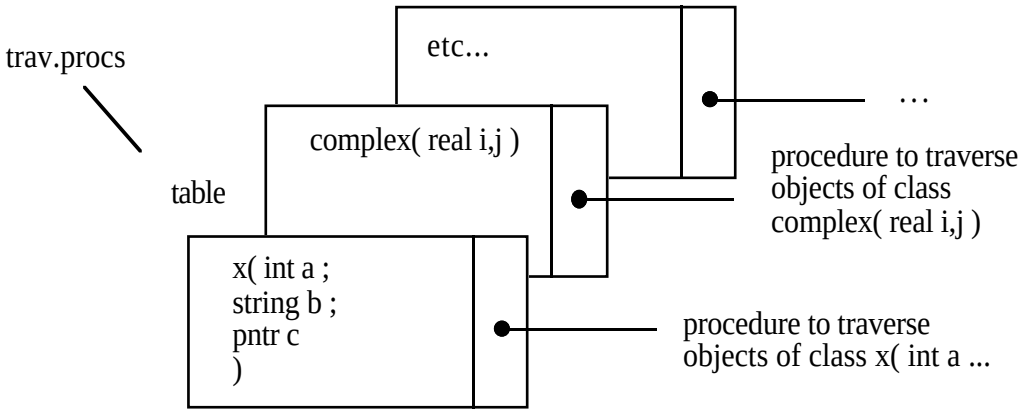

Figure 8

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. This has the effect that 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.

Fire Walls

In the browsing program described we have not broken the type rules of the language. We have, however allowed the types of structure classes to be discovered using the *class.identifier* procedure. The procedure closure has remained sacrosanct and has provided a fire-wall through which this program cannot penetrate. However, the need to see inside a closure or indeed an activation does arise. This happens whenever we wish to construct a symbolic debugger for example. The need to see

inside such objects also arises when a system is in need of repair. We see this 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.

Performance

When looking at the performance of the system described the alternative to such a scheme must also be considered. One alternative 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 this paper is several orders of magnitude faster than this. Another alternative would be to write the browser in a lower level language - a compromise which we are not prepared to make.

The procedure shown in figure 6 to traverse the class,

structure *x*(**int** *a* ; **string** *b* ; **pntr** *c*)

takes the browser 4.5 seconds user time to write,compile,enter into the *trav.procs* table and put the menu on the screen on a SUN 3/260.When the procedure is already in the table looking up the table and putting the menu on the screen takes less than a sixtieth of a second.

Conclusion

We have shown how a browser may be written in a closed strongly typed environment. We have done this 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.

We have allowed the program to discover the type of objects even when the type of an object may have been intended 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.

Acknowledgments

We would like to thank the members of the PISA project in St.Andrews for their help in preparing this paper especially Ron Morrison. We would also like to thank Pete Bailey for his part in the prototyping of the first class compiler.

The work is supported by SERC grant GRC 4324.6 and by a grant from STL Technology Ltd.

References

1 . Goldberg A. and Robson D.
Smalltalk-80: The language and its implementation.
Addison Wesley (1983).

2 . Bobrow D. G. and Stefik M.
The Loops manual.
Tech Rep.KB-VLSI-81-13, Knowledge Systems Area. Xerox Palo Alto Research Centre (1981).

3 . Schaffert C.,Cooper T. and Wilpolt C.
Trellis Object Based Environment.
DEC TR-372, Digital Eastern Research Lab (1985).

4 . PS-algol Reference Manual.
PPRR-12.87, University of Glasgow and University of St.Andrews (1987).

5 . Atkinson M.P.and Morrison R.
Procedures as persistent data objects.
ACM.TOPLAS 7,4 (October 1985),539-559.

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

7 . R. Morrison, A. Dearle, A. Brown and M.P Atkinson.
An Integrated graphics programming system.
Computer Graphics Forum 5,2 (June 1986),147-158.