Two Models For Integrating Persistence and Lazy Functional Languages

David J. McNally
Antony J. T. Davie

Department of Computational Science
University of St Andrews
Scotland

Abstract

A new programming system — STAPLE (Statically Typed Applicative Persistent Language Environment) — which integrates a lazy functional programming language and a persistent store is described. The motivation for introducing orthogonal persistence into a functional setting is given. Two models for achieving this integration are then described together with a discussion of the way laziness interacts with persistence and the benefits resulting from this interaction. In the first model, a system of persistent modules allows the programmer to create persistent values by naming them in a module. In the second model, a combination of stream I/O and a dynamic type allows functional programs to manipulate values already in the persistent store and to allow dynamically created values to become persistent.

1 Introduction

The integration of persistence and lazy functional programming languages promises to reduce the complexity and improve the efficiency of functional programs. In addition, a persistent functional programming system facilitates the construction of database applications. This paper describes two possible models of persistence for lazy functional languages. The first, persistence with modules, relies on static binding to objects in the persistent store (By 'object', we mean the concrete representation of a value). The ObServer object-oriented database system has a similar mechanism for binding[Horn87]. The second, stream persistence, provides a more flexible mechanism which supports dynamic binding to objects in the persistent store. This second approach is found in non-functional languages including Amber[Card83] and Napier88[Morr89]. The Staple functional programming system[Davi90] incorporates both these models within a single programming system.

This section examines the reasons for using functional languages, lazy evaluation and persistence. Section 2 describes the first model of persistence based on modules. Section 3 describes the second model – stream persistence. Examples of both models are given in sections 2 and 3. Section 4 concludes., and a description of the Staple language is included as an Appendix.

1.1 Functional Programming Languages

Functional programming languages have some desirable properties. They are good vehicles for fast prototyping because features such as list comprehension and equational definitions make them near to specification languages [Joos89a]. Work on concurrent implementations [e.g. Darl89] has shown that there is great promise in automatic extraction of parallelism without the programmer getting involved in this notoriously difficult area. Formal reasoning about functional programs is easier than about imperative programs[Turn81]. Lazy evaluation and higher order functions push back the conceptual limits on the way problems can be modularised[Hugh89].

The efficiency, in both space and time, of functional programs is a major consideration for a programming system designer. An increase in the efficiency of functional programs can be achieved by using lazy evaluation semantics in which only values which are actually needed for the result are computed. Persistence extends the benefit of lazy evaluation and also widens the application areas in which functional programming can be used. In particular, database applications in which the data to be processed resides on backing store which may be much larger than the main store of the

machine executing the program. It will be seen later how persistence can be added to lazy functional languages.

1.2 Laziness

A number of functional programming languages (e.g. SASL[Turn79], Miranda[Turn85], Haskell[Huda90]) exploit so called lazy semantics[Frie76,Hend76] which has the attractive features that:

- No value is ever computed unless it needs to be.
- Every value, if computed, is computed once only.
- Some programs will terminate with a result, when evaluated lazily, which would otherwise have gone into a loop.

These features are usually implemented by using a normal order evaluation strategy coupled with graph reduction[Peyt87]. With normal order, calls of functions do not evaluate parameters before entry, and it is only within the body of the function that a parameter may be evaluated *and then only if it is needed*. This reluctance to evaluate values is further extended to make sure that only enough computation is done for present needs. If, for instance, the behaviour of a function differs according to whether its parameter is the null list or not, the parameter will only be evaluated enough to find out if it is the empty list or not and, other considerations aside, no evaluation of the elements of the list will be carried out. This is called evaluation to *head normal form* [Bare84,Peyt87].

Constructor functions, which create new structured values, are no exception to the above rule. The fields of such values are not immediately evaluated. Such lazy data structures may be used to represent infinite values such as the sequence of all positive integers or very large data structures which it would be unreasonable to pre-evaluate such as the tree of all moves in a game. It is only at a later stage, if and when a field of a structure is required, that it is evaluated. Graph reduction, on encountering any parameter or field which actually does need evaluation, replaces it by its evaluated form so that any later uses of such fields or parameters will not involve any further evaluation. Clearly, to be effective, this strategy requires all uses of a parameter to share the same piece of graph so that each use will benefit from the first use's evaluation.

1.3 Persistence

Persistent programming languages help to address the problems of complexity in software engineering. Large programs can become extremely complex and difficult to

maintain. A large part of the code in non-persistent systems is to do with the movement of data from transient to long term storage. The persistence abstraction hides this movement from the programmer thereby reducing the overall complexity of the system. One of our design aims is that persistence should be orthogonal to the other features of a programming language.

1.3.1 Orthogonal Persistence

We give here our perspective of the definition of *orthogonal persistence* given in[Atki83].

In persistent programming languages, programs may manipulate data values independently of their persistence and need not refer to the persistence of the values they create. This is known as the *principle of persistence independence.*

The *principle of data type completeness* states that all data types should be first class without arbitrary restriction on their use. All values (including functions) have full civil rights and can be passed to and returned from functions, named, stored in data structures and given the full range of persistence. Such languages are not only easier to understand, but are easier to learn since there are no arbitrary exceptions to the design goals to explain to a user.

Another important consideration for orthogonal persistence is the question of how to identify persistent data. One technique, for example, is by reachability from some known *root of persistence*. All values reachable (i.e. in the transitive closure of the 'refers to' relation) from the root persist. This process of identification should be orthogonal to the language and in particular, its type system. That is, there should be no special type used for identifying, for example, the root of persistence in a system with persistence by reachability. This is the *principle of persistence identification*. Some systems such as Exodus[Rich89] propose different types for persistent data from those for non-persistent data. For example a persistent integer, known as a database type, has a different type to a non-persistent one. This contradicts both the principle of persistence identification and the principle of persistence independence and is more complicated for the programmer to understand and use whilst yielding no extra power. A consequence of this in systems like Exodus is that they require the programmer to predict which objects will persist. This may not always be known statically.

The application of the above three principles yields orthogonal persistence. Orthogonal persistence ensures that no code need be written to deal with the transfer of data between short and long term storage explicitly. Moreover, *referential integrity*[Morr90] is preserved over persistent store operations – sharing and topology are unaltered. This

is particularly important in a functional language setting from an efficiency viewpoint (semantically it should not matter). Languages such as Amber[Card83] and a proposal for persistence in ML do not preserve referential integrity in that objects accessed more than once will be duplicated rather than shared. Others such as E[Rich89] may result in dangling pointers if a persistent data structure is created which contains non-database types.

1.3.2 Persistence in Functional Programming Systems

Functional languages have so far provided little support for storage of long term data. Many attempts do not have the property of orthogonal persistence. In Staple, however, orthogonal persistence is achieved.

Most systems which provide anything at all, provide just a simple mechanism for writing a user generated list of characters to a file. This mechanism is not sufficient for orthogonal persistence for a number of reasons. It is difficult to maintain referential integrity in such a system – the programmer must write code to flatten data structures on output and unflatten them again on re-input, reconstructing the topology, so that the sharing of substructures is not lost. More seriously, partially evaluated data structures and functions *cannot* be flattened since the user cannot see the internal representation of these objects nor can they be reconstructed. The Miranda[Turn85] system has a function which will flatten a data structure (after forcing its total evaluation) to a list of characters, but this is subject to the same criticisms.

Some systems (e.g. Poly [Matt88]) attempt to allow the storage of structured data by saving the current environment by dumping a core image to the file system and restarting from the previous saved state. This method allows very little user control over values, and all values which exist will also persist. In addition, the size of the persistent store is limited to the size of the run-time heap.

When an uncomputed value is required, the run-time representation of the value (called a *suspension*) is overwritten in place with the result of the computation (Figure 1). In this way, shared values are not computed more than once. The result of the computation denotes the same value as the suspension (referential transparency is preserved).

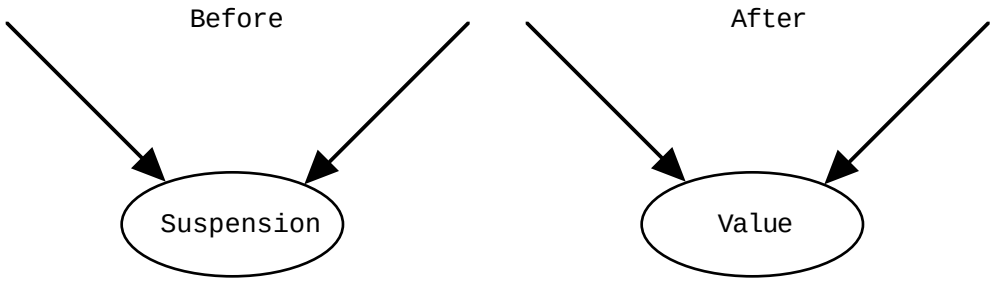

Figure 1

A primary efficiency consideration requires that this sharing of sub-graphs must last for as long as possible, regardless of where the graph is stored. Many functional programming systems preserve the topology of the graph being evaluated while an interactive session lasts. Most functional programming systems, however, until the Staple system reported here, have, at the end of a session, discarded the values which have had so much computation performed on them. At the beginning of subsequent interactive sessions, all values are in their unevaluated state. Some systems even abandon work done *between interactive evaluations* e.g. in Miranda which, when using the facility to access the last value requested, starts to re-evaluate again completely from scratch.

Staple[Davi90] is a purely functional programming system with lazy evaluation semantics which has orthogonal persistence – programs manipulate data independently of their persistence, all objects can be persistent and identification of persistent objects is independent of the type system. The systems described above are not able to cope with database applications where large bodies of persistent data are to be manipulated. Because of the way the persistent store[Brow89] used by Staple only fetches values from backing store as they are needed (lazily) Staple is able to cope with this application area. In addition, the Staple system benefits from an efficiency point of view since no value is ever computed more than once, regardless of its persistence.

1.5 Two Models for Persistence in Lazy Functional Languages

There are many possible approaches to adding persistence to programming languages not all of which yield orthogonal persistence. Two models which do are described below. The first, persistence with modules, is illustrated with two examples – memoising a function and a prototype transportation network system. The second, stream persistence, is illustrated with a database-like example – an index. These examples help to strengthen the claim that persistent functional languages have significant benefits over existing non-persistent ones. The efficiency of the languages is achieved by the preservation of the results of computation, the complexity is

controlled by an integrated system for re-use of programs and data. Applications can be written which can deal with more data than would be possible to access in the main memory of the machine.

2 Persistence with Modules

Our first approach to adding persistence to a lazy functional languages is to use a module system as the means of identifying persistent objects. The aims of this approach are firstly, to add persistence in a transparent way – a user need not modify existing code – and secondly, to preserve referential transparency. The first of these is achieved by representing modules in the persistent store with the values defined in the module being persistent by reachability. The second is achieved by binding to values in the persistent store at compile time.

In order to facilitate the development of persistent functional programs, the Staple system provides an integrated environment in which objects, grouped together into modules, are stored. The recursive nature of functional languages, particularly with a lazy evaluation strategy allowing infinite data values, leads to many such objects being partially evaluated. Objects may be completely unevaluated if not yet needed, evaluated enough to determine which pattern they match in some definition, evaluated enough to print the first 30 elements ... right through a spectrum leading eventually (in the case of finite objects) to complete evaluation in a normal form. Because of the insistence on functional purity, referential transparency is maintained. That is, no object changes its value, except in the above sense of being partially reduced on the road to normal form. The state of evaluation of partially or wholly evaluated objects should be made to persist so that evaluation of an object will not have to be repeated.

The Staple persistent store contains a set of modules each of which can contain a number of definitions. This set of modules is organised into a directory structure within the persistent store and modules can be accessed via their name. A module constitutes the unit of compilation. Any module can be compiled in an environment created by importing the definitions exported by other previously compiled modules. An interactive session can also run in such an environment.

The persistent store is utilised by the use of two commands at the operating system level:

- A command to create a module in a persistent store
- A command to allow expressions to be entered and evaluated interactively using values in the persistent store.

The implementation of the Staple module system is described in detail in [McNa90].

2.1 Creating a New Module in the Persistent Store

The command

```
mkmodule mname [ othermodulenames ]
```

places a module named *mname* into the persistent store by compiling the source file called *mname*. This module imports all definitions from *othermodulenames* which must already be in the persistent store. A module exports all names defined at its own top level of definitions. If *mname* already exists, this command will overwrite it and any objects defined in the old version will become candidates for garbage collection unless they have been imported by another module in the persistent store. Note, however, that this sytem does not provide an automatic re-binding mechanism. This is totally under user control. If there is a clash of names in *othermodulenames* and/or *mname*, these are resolved by considering each module as a block in which the previous ones appear as sub-blocks and letting lexical scoping resolve the conflicts.

In the current implementation of Staple, this rule means that modules cannot be mutually recursive. All recursion must be expressed within modules where the defined objects can be mutually recursive. In addition, modules do not export any names that they import from elsewhere. In the particularly simple set-up described here, the names declared at the top level of a module (and only those names) are exported by that module. Other models for naming in modules can be used to avoid these problems and further work will be carried out on this.

Values in a module may be shared by many other modules as shown in Figure 2.

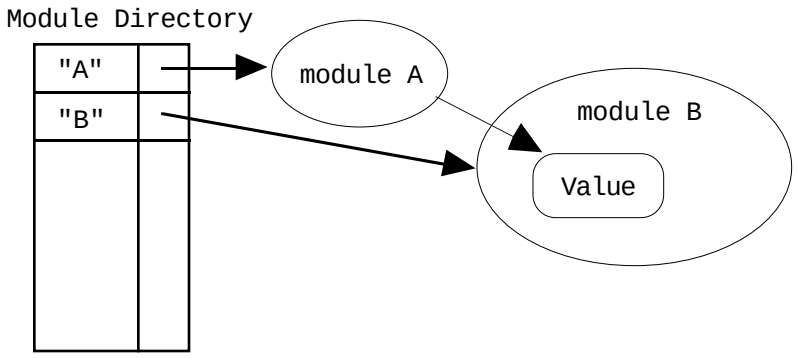

Figure 2

Figure 3 shows the result of recompiling module B. The association in the module directory is replaced with a reference to the new module. The values in the old module which are not referenced may then be garbage collected. Values shared by other modules, however, are not lost.

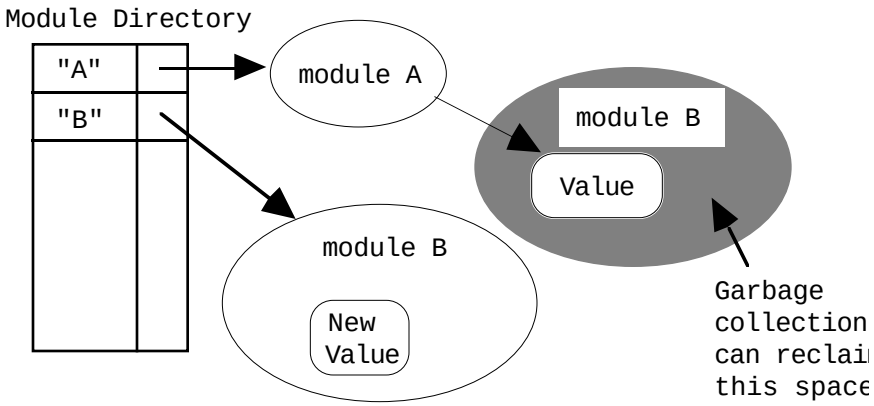

Figure 3

2.2 Interacting with the Persistent Store

The command

```
staple [ modulenames ]
```

allows the user to evaluate expressions interactively, in an environment defined by *modulenames* which must be in the persistent store. Expressions evaluated during the interactive session may cause the partial or total evaluation of objects defined in the environment being used. This evaluation will be reflected in a permanent (referentially transparent) change in state of the persistent store.

Partially computed values are represented as suspensions – a suspension consists of some code and an environment such that the code when evaluated in the given environment yields a value in head normal form. When the value of such an object is required, the code is executed and the resulting value overwrites the suspension in place. All values which shared the suspension will then see the result of this evaluation. This overwriting in place is not an update in the imperative sense since the value is not being replaced by another different value, but is being replaced by a different representation of the same value which is closer to normal form (being completely evaluated).

2.3 Two Examples of functional programming with Modules

Two examples of modular persistent functional programming will now be described. The first is a small one to illustrate the principles involved and the second describes how a prototype of a large application was built.

2.3.1 Prime Numbers

The Sieve of Eratosthenes method of calculating prime numbers can be extended, see for instance [Turn82], to allow the specification of an infinite list of primes to be defined in a very expressive way:

```
primes = sieve(from 2)
sieve (p:x) = p:sieve [n||n <- x , n mod p > 0]
```

Here the `from` function is in the 'standard prelude' module and specifies an infinite list of the integers starting from `2`. The `sieve` function removes all multiples of the first element of a list from the rest of the list. Although this is an attractive definition because of its conciseness, it does require work to calculate any given prime, but these are evaluated lazily and on demand only. To compile this module (named `prime` say), the command

```
mkmodule prime prelude
```

is used. The argument `prelude` is needed to enable the `from` function from the precompiled `prelude` module to be used. Subsequent uses of the `primes` list can then be made either interactively by

```
staple prime ... other modules to be used ...
```

or in another module by

```
mkmodule mymodule prime ... other modules ...
```

Any evaluation of sections of the `primes` list using either of these methods will be retained in the persistent store and subsequent users of the module will benefit. Some evaluation can be performed on the list of primes by asking, for example, for the 25th prime number as follows

```
primes!25 ?
```

The `?` indicates the end of an expression to be evaluated. The evaluation required to evaluate the above expression involves 1050 function applications. If, at some later time, the 20th element in the list is requested

```
primes!20 ?
```

only 42 function applications (associated with the list indexing operator `!`) are needed. However, if the 30th prime number was required

```
primes!30 ?
```

some further work would be required (445 more applications in fact) whereas it would have taken 1443 applications if started from a completely unevaluated state. The use of an array rather than a list would give further significant benefit.

Before any evaluation takes place, the value associated with primes is a suspension as shown in Figure 4.

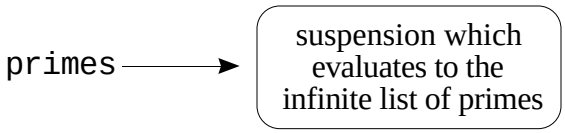

Figure 4

Figure 5 shows how after some of the list has been evaluated, primes is associated with the partially evaluated list of primes which terminates in a suspension.

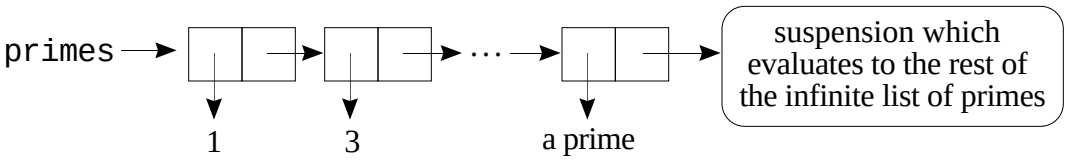

Figure 5

As long as `primes` is reachable from some module, previous evaluation will be preserved in perpetuity and subsequent users of prime numbers will both gain a benefit of work already carried out and confer a similar benefit on later users.

2.3.2 The Frankfurt travel network

The Staple module system has been used to make a number of prototypes, among them a model of the Frankfurt travel network (Frankfurter Verkehrs und Tarifverbund — FVV)[Joos89b] including subways, trains, buses and pedestrians. It covers 110 stations including many which lie on more than one line.

The model uses 7 interrelated modules which contain between them over 80 definitions of objects. They deal with such aspects as

- a graph containing a basic description of the raw data of the network including a description of each transport line in the system and details of each station on it.
- functions for doing basic graph manipulation

- functions for doing graph operations specific to this problem
- functions for doing matrix operations

and so on.

The raw data graph is transformed and manipulated in a number of ways and one of the ultimate results is a matrix giving routes of minimum cost between each pair of stations. In a previous, non-persistent, attempt to prototype this problem using Miranda, it was found that the only way that information about these minimum cost routes could be used in subsequent runs was to pre-calculate the whole of this large matrix, output it and re-input at the beginning of later executions. Apart from the fact that referential integrity is not preserved with this technique, and enormous amounts of data which was originally shared is now duplicated many times, this approach does not make use of the laziness aspect in the way that was intended and the whole of the matrix has to be evaluated at once. The main points here are

- referential integrity is lost
- the whole structure must be evaluated at once

both of which lead to a loss of efficiency.

The Staple approach suffers from neither of these drawbacks as no intermediate values need to be output at all.

- Referential integrity is preserved so shared structures are not computed more than once.
- The data structure is evaluated lazily so only values which are needed to compute the result are evaluated.

3 Stream Persistence

Functional programs should be able to interact with external agents such as the file system, keyboard, screen and persistent store. In this section, a mechanism for providing access to the persistent store which allows the user to make dynamically created objects persist, and to access objects in the persistent store is described .

Staple provides these mechanisms by a combination of stream based I/O and a dynamic type similar to that found in Amber[Card83] and Napier88[Morr89], but which preserves referential integrity and is orthogonal. These mechanisms introduce update into the semantics of the language, but this update has exactly the same semantic

consequences as any other kind of non-referentially transparent I/O operation in functional languages.

3.1 Stream I/O

A Staple program can be considered as a black box which produces a stream of requests and consumes a stream of responses. The run-time system itself is a consumer of requests and producer of responses. This is similar to the Haskell [Huda90] interactive environment. Requests are typically things like “display the string ‘hello world’ on the standard output” or “read the contents of the file ‘datafile’”. The responses produced by the run-time system are things like “OK I’ve done it” or “the contents are ‘Some text’” or in some cases “Failure: no such file” etc. Figure 6 shows a conventional stream I/O system.

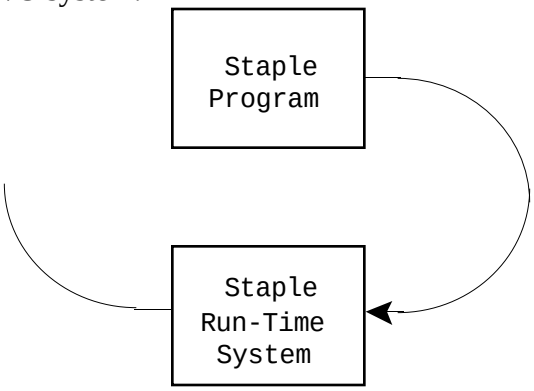

Figure 6

Staple requests and responses are defined as an algebraic data type which is part of the standard environment.

3.2 Dynamic Typing

If arbitrary values are to be stored in the persistent store by requests at run-time and later retrieved, there needs to be a check at retrieval time that the values have appropriate types. This is achieved by injection into a universal type and later projection out of it. This universal type is an orthogonal concept which happens to be particularly suited to this application.

Staple is a statically typed language which incorporates a dynamic universal type `Any`. `Any` is the type of the set of all values. In implementation terms, objects of type `Any` are represented as pairs consisting of a value and a type representation being the type of the value. Objects of type `Any` can be created using the injection function `mkany`. For example

```
mkany 4
```

creates an object of type `Any` which contains the value 4 and a type representation `Int`. Values can be projected out of the union type using the projection operation `coerce` as in

```
coerce expression :: Int
```

where `expression` evaluates an object of type `Any`. This operation performs a dynamic type instance check which ensures type safety. The type declared as the result of the projection must be an instance of the type associated with the dynamic object.

The type to be associated with a value at injection time is inferred using a standard Hindley-Milner style type inference algorithm[Miln78]. Staple adopts the restriction in [Abad89] due to Mycroft which states that only closed types (those involving no unquantified type variables) may be associated with a value by the injection operation. The reason a restriction is necessary is that the type rules can be broken without one. While this restriction is sufficient to preserve type safety, it may be over-restrictive and further research needs to be carried out.

3.3 Streams in Staple

All Staple objects are created in the persistent store. Objects only persist, however, if they are reachable from the Staple root of persistence. Staple programs can make objects reachable from the root by producing a request “make the object ... reachable and give it the name ...” and can access such an object by producing a request “get me the object in the persistent store with name ...”. An object can be deleted by the request “delete object with name ...”. The clauses of the algebraic data type definition for requests is extended with these requests as follows :-

```
data Request = ... |
               Insert Name Any |
               Lookup Name |
               Delete Name | ...
```

where `Name` is a string and `Any` is the built in dynamic type. The responses to these requests will indicate success or failure and the `Lookup` request may produce a response containing the object required. The response data type is extended as follows

```
data Response = ... | Db Any | ...
```

Because objects in the persistent store can be bound to dynamically, type safety must be preserved by assuring that objects brought from the persistent store are of the required type. This is guaranteed by the fact that only objects of type `Any` can be associated in the persistent store and by the type check performed dynamically by a `coerce`.

The `Delete` request simply removes the association between the string and the value, not the value itself which may be shared by some other object. If it is not referenced by any other persistent object, it will be garbage collected.

3.4 Initiating Evaluation

A Staple expression entered interactively which is inferred to have the type `[Response] -> [Request]` is treated specially by the system. A function of this type is called a *dialogue*.

To evaluate a dialogue function, the Staple system transforms the expression into an application of the dialogue function to a system generated list of responses. This list of responses is completed lazily as a consequence of the evaluation of the list of requests which is now the result of the transformed expression. The list of requests is interpreted by the system rather than just displayed as is the case with ordinary expressions. The laziness deals with the synchronisation of requests and responses[Thom86].

3.5 A Functional Index

To demonstrate how Staple can be used to implement database style applications, this section describes an example of the use of Staple to implement a simple database of employee records. Staple is being used,here, as a database programming language.

```
data Tuple = Employee [Char] Int [Char] Int
```

This defines the type used to represent records in the database. Now we define an database which is empty – lists of records are used to represent the database.

```
emptydb :: [Tuple]
emptydb = []
```

The empty database is the empty list. Next a function is defined which will initialise an empty database in the persistent store.

```
dbname = "employee database"
initdb resps = [Insert dbname (mkany emptydb)]
```

This can be executed by typing

```
initdb ?
```

to the staple interactive session prompt. Since this is a dialogue function, it is treated specially and applied to a list of responses. The responses are generated according to the list on the right hand side of this function definition. In this case, the single response (which is not accessed) is an indication of success. Next a function which will insert a new tuple into the database is defined.

```
addtuple n a ni s resps = [Lookup dbname,Insert dbname newdb]
                          where
                          newdb = mkany ( newtuple : db)
                          newtuple = Employee n a ni s
                          db = coerce stored :: [Tuple]
                          stored = getval (resps!0)
                          getval (Db val) = val
                          getval (Failure mess) = error mess
```

The `addtuple` function defined above produces a dialogue if it is applied to four arguments. This dialogue results in two requests - one to lookup the database in the persistent store and one to insert the new updated database into the persistent store. The subsidiary definitions extract the old database from the dynamic object returned as the first response and construct a new database by appending a new tuple onto the old database.

The contents of the database can be displayed using the following function

```
displaydb resps = [Lookup dbname,
                    AppendChan "stdout" (showdb db)]
                  where
                  db = coerce stored :: [Tuple]
                  stored = getval (resps!0)
                  getval (Db val) = val
                  getval (Failure mess) = error mess
```

The definition of the `showdb` function is shown below. It simply creates a string which represents the database in a standard format.

```
showdb ts = "Employee database'n" ++
            "Name              Age     NI number  Salary'n" ++
            showtups ts
```

The ++ operator appends two lists together.

```
showtups [] = []
showtups (h:t) = showtup h ++ showtups t

showtup (Employee n a ni s) = ljustify 20 n ++
                              rjustify 4 (intTOstring age) ++
                              rjustify 14 ni ++
                              rjustify 8 (intTOstring s)++"'n"
```

A function to delete a tuple from the database can be specified similarly to the `addtuple` function above.

```
deltuple n resps = [Lookup dbname,Insert dbname newdb]
                   where
                   newdb = mkany (remtuple name db)
                   db = coerce stored :: [Tuple]
                   stored = getval (resps!0)
                   getval (Db val) = val
                   getval (Failure mess) = error mess
                   remtuple name [] = []
                   remtuple name (h:t) = t, match h name
                                       = h : remtuple name t
                   match (Employee n a b c) name = (n = name)
```

Notice the similarity between the add and delete operations. We can abstract over these to get a general action function incorporating all changes to the database.

```
dbaction f resps = [Lookup dbname,Insert dbname newdb]
                   where
                   newdb = mkany (f db)
                   db = coerce stored :: [Tuple]
                   stored = getval (resps!0)
                   getval (Db val) = val
                   getval (Failure mess) = error mess
```

We can now define the add operation as follows

```
addtuple n a ni s resps = dbaction addit
                          where
                          addit db = Employee n a ni s : db
```

and similarly for delete and any other database operations we might require.

4 Conclusions

Laziness and persistence are both ways of improving efficiency in programming systems, laziness by avoiding the need to recompute values and persistence by reducing complexity and reducing code size. When laziness and persistence are integrated, new benefits are realised. Values are never computed more than once, even between interactive sessions. That is, the results of computation are preserved and are available for use by other programs. The ability to manage the construction of large prototypes, such as the Frankfurt Network, is enhanced.

We have described two models for the integration of persistence and lazy functional languages. A module system which preserves referential transparency, and stream persistence which allows programs to access objects in the persistent store and to place dynamically created objects into the persistent store for use at a later time.

The persistence in Staple allows programs to be written which manipulate bodies of data whose size is greater than the main store of the machine in a transparent way. The programmer need not be concerned with the persistence of the being manipulated. The transfer of data between main store and backing store is managed by the underlying persistent storage system.

The two models we have described reap the benefits of having orthogonal persistence. Referential integrity is maintained in the Staple system. This is essential from an efficiency point of view. Maximising sharing is a necessity in functional language implementations and in Staple, the sharing lasts for the lifetime of the objects involved.

5 Acknowledgements

We are indebted to Fred Brown for providing the persistent storage system[Brow89] which underlies Staple and to our colleagues at St Andrews for their valuable comments.

References

[Abad89] Abadi, M., Cardelli, L., Pierce, B. & Plotkin, G.D., *Dynamic Typing in a Statically Typed Language*, DEC Systems Research Center Report 47, 1989

[Atki83] Atkinson, M., Bailey, P., Chisholm, K.J, Cockshott, W.P. & Morrison, R., *An Approach to Persistent Programming*, The Computer Journal, Vol. 26, No. 4, 1983, pp.360-365.

[Bare84] Barendregt, H., *The Lambda Calculus: Its Syntax and Semantics* (Revised Edn.), North Holland, 1984

[Brow89] Brown, A.L., *Persistent Object Stores*, Ph.D. Thesis, University of St Andrews; and University of St. Andrews Dept. Comp. Sci. and University of Glasgow Dept. of Computing, Persistent Programming Research Report PPRR-71-89, 1989

[Card83] Cardelli, L., *Amber*, AT&T Bell Labs Report, 1983

[Darl89] Darlington, J. et al., *A Functional Programming Environment Supporting Execution, Partial Execution and Transformation*, Proc. PARLE '89 Conference, Eindhoven, Lecture Notes in Computer Science 365, 1989, pp. 286-305

[Davi90] Davie, A.J.T. and McNally, D.J., *Statically Typed Applicative Persistent Language Environment (STAPLE) Reference Manual*, University of St Andrews, Dept. of Math. and Comp. Sci., Research Report CS/90/14, 1990

[Frie76] Friedman, D.P and Wise, D.S., *CONS Should not Evaluate its Arguments*, In *Automata, Languages and Programming* Eds. Michaelson & Milner, Edinburgh Univ. Press, 1976, pp257-284

[Hend76] Henderson, P. and Morris, J.H., *A Lazy Evaluator*, Proc. 3rd Annual ACM Symposium on Principles of Programming Languages, Atlanta, 1976, pp95-103

[Horn87] Hornick, M.F. and Zdonik, S.B., *A Shared Segmented Memory System for an Object Oriented Database*, ACM Transactions on Office Information Systems, Vol. 5, No. 1, 1987, pp. 70-95

[Huda90] Hudak, P. and Wadler, P. (Eds.), *Report on the Programming Language Haskell*, Glasgow and Yale Universities, 1990

[Hugh89] Hughes, J., Why Functional Programming Matters, The Computer Journal, Vol. 32, No. 2, April 1989

[Joos89a] Joosten, S. *The Use of Functional Programming in Software Development*, Ph.D. Thesis, University of Twente, The Netherlands, 1989

[Joos89b] Joosten, S. and Ruppert, W., *Public transport in Frankfurt – an experiment in Functional Programming*, STAPLE Project (ESPRIT 891) Six monthly review, September 1989

[Matt88] Matthews, D.C.J., *An Overview of The Poly Programming Language*, in *Data Types and Persistence*, Eds. Atkinson, M.P., Buneman, P. and Morrison R., Springer-Velag, 1988

[McNa90] McNally, D.J., Joosten, S. and Davie, A.J.T., *Persistent Functional Programming*, in proc. Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Mass, 22-27 Sept 1990, pp59-70

[Miln78] Milner, R., *A Theory of Type Polymorphism in Programming*, J. Comp. and Sys. Sciences, 17, No 3, 1978, pp. 348-375

[Morr89] Morrison, R., Brown, A., Connor, R.C.H., Dearle, A., *The Napier88 Reference Manual*, University of St. Andrews Dept. Comp. Sci. and University of Glasgow Dept. of Computing, Persistent Programming Research Report PPRR-77-89, 1989

[Morr90] Morrison, R. and Atkinson, M.P., *Persistent Languages and Architectures*, in *Security and Persistence*, Bremen Workshop, pp9-28, Springer-Verlag, 1990

[Peyt87] Peyton Jones, S. *Implementation of Functional Programming Languages*, Prentice Hall, 1987

[Rich89] Richardson, J.E. and Carey, M.J., *Persistence in the E Language: Issues and Implementation*, Software Practice and Experience, Vol. 19, 1989, pp1115-1150

[Thom86] Thompson, S., *Writing Interactive Programs in Miranda*, University of Kent at Canterbury Computing Lab. Report 40, 1986

[Turn79] Turner, D. A., *SASL Language Manual*, University of St. Andrews Dept. of Comp. Sci., Report CS/79/3, 1979

[Turn81] Turner, D. A., *The Semantic Elegance of Applicative Languages*, Proc. FPCA Conference, Portsmouth, New Hampshire, ACM Publications, 1981, pp. 85-92

[Turn82] Turner, D. A., *Recursion Equations as a Programming Language*, in *Functional Programming and its Applications* Edited by Darlington, J., Henderson, P. & Turner, D.A., CUP 1982, pp. 1-28

[Turn85] Turner, D. A., *Miranda: A Non-Strict Functional Language with Polymorhic Types*, Proc. FPCA Conference, Nancy, Lecture Notes in Computer Science 201, 1985, pp. 1-16

Appendix I – The Staple Language

Expressions and their Types

The types of Staple objects are, in the main, inferred by a Hindley-Milner style algorithm [Miln78]. There is a language of types so that users, if they wish, may specify the types they expect objects to have and so that they may define new types.

Base Types

The common base types of object are Bool, `Int`, `Real` and `Char`.

Objects may be combined using a number of built-in prefix and infix operators and using a conditional construct. Here are some examples:

```
-a + b * (c - d)              -- integer operators
-.x +. y /. z *. float(a)     -- real operators
a > b or a < 2*b and a > b-1  -- relations and Boolean
f x' y'                       -- function application (no
                              -- brackets required
if a > b then a+b else a-b
```

Lists

Square brackets are used to enumerate objects in a list, the `:` and `++` operators are used for extending concatenating lists and some other list operators are available.

```
[ 1 , 1, 2, 3, 5, 8 ]        -- enumerated lists
head : tail                   -- constructed lists
lefts ++ rights               -- concatenated lists
1 .. 10                       -- lists of integers. This =
                              -- [1,2,3,4,5,6,7,8,9,10]
```

```
x ! 5                          -- selecting an element from a
                               -- list
```

Lists of characters (strings) may be expressed in a special simple way using " as a shorthand list delimiter:

```
"Hello"                        -- = [ 'H' , 'e' ,'l' ,'l' ,'o' ]
```

Comprehensions

Lists may be generated using so called list comprehensions (sometimes called set expressions or Zermelo-Frankel set abstractions)[Turn82].

```
[ f x || x <- s ]              -- the list of all f x such that
                               -- x is taken from list s. So
[ x*x || x <- 1..5 ]           -- [1,4,9,16,25]
[ x || x <- 1..100 , p x ]     -- the list of all those x
                               -- between 1 and 100 for which
                               -- p x is true
[ (a,b,c) || a <- 3..100, b <- (a+1)..100, hcf a b = 1,
                  c <- (b+1)..100,a*a+b*b=c*c ]
                               -- the Pythagorean triples with
                               -- elements <= 100
```

If `t` is the type of a list element, then `[t]` is the type of the list itself.

Tuples

Inhomogeneous groupings may be made using tuples represented by gathering a group of objects together in round brackets:

```
( "Bert" , 16 , Male , "Room 251" )
```

If t_1, ... t_n are the types of the elements, then $(t_1,...t_n)$ is the type of the tuple.

A special category of functions called constructor functions may also be used to make inhomogeneous objects. These are described below

Functions

Many new functions are declared using a definition but they can, of course, be the result of some operation

```
(if a > b then f else g) a
(operations ! i) x             -- applying a fn from a list
(f x).g.(h y)                  -- the composition of 3 fns
```

If a function maps type `s` to type `t` we say its type is `s->t`.

Declarations

Users may define new types, new objects or specify the types of objects.

Algebraic Data Type Declarations

Constructor functions for new types can be declared. Here are some examples

```
data Expr = Literal Int | Add Expr Expr | Sub Expr Expr
```

This definition, which is meant to model simple algebraic expressions, defines three constructor functions, `Literal` of type `Int -> Expr` and `Add` and `Sub` both of type `Expr -> Expr -> Expr`.

Enumerations can also be defined using this mechanism. e.g.

```
data Weekday = Sun | Mon | Tue | Wed | Thur | Fri | Sat
```

This definition defines seven nullary constructors which are all of type `Weekday`.

```
data List = Nil | Cons Int List
```

This definition defines a nullary constructor, `Nil`, of type `List` and a constructor `Cons` of type `Int -> List -> List`.

```
data List a = Nil | Cons a List
```

This definition is a parameterised version of the previous one and allows the definition of lists of any type, not just `Int`. This type is entirely isomorphic to the built-in type of list which uses `[]` and the infix `:` as its constructors.

Object Definitions

Objects are defined using equations which may be regarded as axioms for the benefit of any theorems which are proved about a program. A set of definitions can occur at the top level when defining a module (see below) or in a purely local context when appearing in a `where` clause. Definitions usually involve binding names found in patterns. A name, by itself is the simplest kind of pattern, but constructor functions are also allowed to appear in them. We will not go into details of all the possibilities but merely give some examples:

```
pi = 3.14159                              -- defines a real
(x,y) = complexlog (-pi,pi)               -- defines 2 reals
[x,y,z] = [3.14,2.187,0.0]                -- defines 3 reals
first:rest = list                         -- decomposing a list
Tree(left,value,right) = a'tree           -- selecting from a
                                          -- user defined type
filter p x = [y || y <- x , p y]          -- a function
```

```
-- but patterns more commonly appear in function definitions

first 0 list          = []                -- a fn defined with
first n (h:t)         = h:first(n-1)t     -- two clauses and
                                          -- pattern matching

-- another defn of filter showing a guard being used. The 2nd
-- equation only holds if p x is true
filter p []       = []
filter p (x:xs)   = x : filter p xs    ,p x

-- and a third defn exhibiting local definitions
filter :: (t -> Bool) -> [t] -> [t]       -- optional type spec
filter p              = filter'
                      where
                      filter' []       = []
                      filter' (x:xs)   = x : filter' xs    ,p x
                                       = filter' xs
```