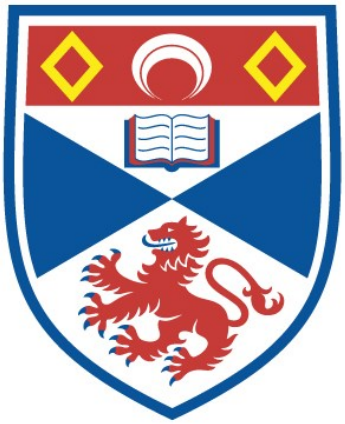
# MODELS FOR PERSISTENCE IN LAZY FUNCTIONAL PROGRAMMING SYSTEMS

David J. McNally

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1993

# Models for Persistence in Lazy Functional Programming Systems

David J. McNally

Department of Mathematical and Computational Sciences
University of St Andrews
Fife
KY16 9SS



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

# Declarations

I, David John McNally, hereby certify that this thesis has been composed by myself, that it is a record of my own work, and that it has not been accepted in partial or complete fulfilment of any other degree or professional qualification.

Signed Date 23/4/93

I was admitted to the Faculty of Science of the University of St Andrews under Ordinance General No. 12 on 1st October 1986 and as a candidate for the degree of Ph.D. on 1st October 1986.

Signed Date 23/4/93

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate to the degree of Ph.D.

Signature of Supervisor Date 23/4/93

# Copyright

# Acknowledgements

It is clear that in any work of this kind, there will be influences from many sources. The research community as a whole is a lively source of ideas and intuitions and it would be impossible to acknowledge all who have contributed to this research though I thank them all. It is appropriate, however, that a number of people be thanked explicitly. Fred Brown, Richard Connor, Quintin Cutts, Al Tony Davie, Al Dearle, Simon Peyton Jones, Stef Joosten, Graham Kirby, Ron Morrison, Dave Munro, Gerald Ostheimer and Norman Paterson have all provided interesting and stimulating discussions leading to ideas presented here.

Ron Morrison has been particularly important in sharpening up many of the ideas which have arisen during the period of this work. Simon Peyton Jones has helped in pointing me in the right direction. Special mention should also go to Tony Davie my supervisor and Richard Connor who both played a significant part in the development of this work.

# Abstract

Research into providing support for long term data in lazy functional programming systems is presented in this thesis. The motivation for this work has been to reap the benefits of integrating lazy functional programming languages and persistence. The benefits are

- the programmer need not write code to support long term data since this is provided as part of the programming system
- persistent data can be used in a type safe way since the programming language type system applies to data with the whole range of persistence
- the benefits of lazy evaluation are extended to the full lifetime of a data value. Whilst data is reachable, any evaluation performed on the data persists. A data value changes monotonically from an unevaluated state towards a completely evaluated state over time.
- interactive data intensive applications such as functional databases can be developed.

These benefits are realised by the development of models for persistence in lazy functional programming systems. Two models are proposed which make persistence available to the functional programmer. The first, persistent modules, allows values named in modules to be stored in persistent storage for later reuse. The second model, stream persistence allows dynamic, interactive access to persistent storage. These models are supported by a system architecture which incorporates a persistent abstract machine, PCASE, integrated with a persistent object store. The resulting persistent lazy functional programming system, Staple, is used in prototyping and functional database modelling experiments.

# Table of Contents

# Chapter 1

# Introduction

Lazy functional programming systems [Turn79a,Bird88,Davi92,Huda92] combine programming by the definition and application of functions with lazy evaluation where subexpressions are only evaluated if their value is required for the program's result. All programs are expressions and execution consists of the evaluation of the expression represented by the program.

Persistence [Atki83,Atki85,Atki87] is the length of time for which data is available for use. Persistent programming systems automatically support data over the whole spectrum of persistence from temporary program variables to long term databases.

Lazy functional programming systems and persistence can be integrated in a way which yields some important benefits. The benefits are

- support for long lived data is provided as part of the programming system
- long term data is strongly typed since the use of persistent data is governed by the programming language type system
- data values can be shared between many program invocations, computation which is performed on shared values is never repeated so the benefit of the evaluation lasts for as long as the data is usable
- the construction of applications in which functional programs access and manipulate persistent data interactively is facilitated.

Two models for the integration of persistence and lazy functional programming described in this thesis are persistent modules and stream persistence. These

models are incorporated in the Staple persistent lazy functional programming system [Davi90].

An introduction to the styles of programming known as functional programming and persistent programming is now given. A brief description of some languages in each style is included and the benefits and costs associated with each paradigm are discussed. An overview of the remainder of the thesis concludes the chapter.

## 1.1 What is Functional Programming?

Functional programming is a style of programming which consists entirely of the definition, and application of functions. Unlike imperative programming, there are no side-effects caused by the evaluation of an expression.

Computation is performed in functional programming by the evaluation of referentially transparent expressions. A programming language is said to be referentially transparent if the value of expressions written in the language depends solely on the value of their well-formed (syntactically correct) sub-expressions [Whit13, Quin60, Turn82, Davi92].

Various claims have been made about the virtues of functional programming languages;

- Functional programs are more amenable to rigour since the formal semantics of a functional language is much simpler than that of other kinds of language primarily because of the absence of an updateable store or state. In addition, unlike imperative languages, equality is reflexive which enables the use of normal mathematical techniques. Because of the recursive nature of many functions, various induction techniques can be used to prove correctness [Burs69]. Since functional programs written in an equational style resemble

mathematics, it is possible to use the function definitions themselves in the mathematical proofs. [Boye84]

- Functional programming languages are well placed to take advantage of modern parallel computer systems [Darl89]. The property of referential transparency guarantees that the value of an expression depends only on the values of its sub-expressions. Thus the order in which the sub-expressions are evaluated does not affect the value of the expression. In particular, the sub-expressions can be evaluated concurrently [Darl81,Peyt87a,Arvi90].

- Joosten [Joos89a] and Henderson [Hend86] have claimed that functional languages are good vehicles for rapid prototyping because features such as list comprehensions and equational definitions make them near to specification languages. Several large scale prototypes have been built to test this hypothesis including a hospital drug management system, a hidden line removal program [Joos89a] and a transportation problem [Joos89b]. The last of these was facilitated using the Staple persistent functional programming system.

- Functional programming languages typically have more succinct syntax than other languages and provide powerful abstractions for data manipulation such as list comprehensions and pattern matching which leads to programs being easier to write, more concise and higher-level. Functional programs resemble mathematics more closely than imperative ones. Functional programs tend to be much shorter than programs written in other languages. Hughes [Hugh89] claims that they are an order of magnitude shorter.

- Program transformation can easily be performed on functional programs to improve their performance in space or time whilst

preserving their semantics. For example, list comprehensions which can be used to model queries in relational algebra are amenable to the same kind of optimising transformations used for database query optimisation [Trin89b]. Other important kinds of transformation can improve the utilisation of parallel architectures [Darl91].

- Functional languages have some properties which make them appealing as query languages [Bune79]. List comprehension are a language mechanism for manipulating lists which allow the expression of relational algebra type queries. Trinder [Trin89b] has shown that list comprehensions can model all queries expressible using relational algebra. Furthermore, Trinder has shown that list comprehensions can be optimised in the same way as relational algebra queries.

These issues are largely subjective and debates on their validity are frequent and lively amongst the programming language community.

### 1.1.1 The Development of Functional Languages

The first functional language (although not originally intended as a programming language) was Church's lambda calculus [Chur41]. Although no serious implementations of the pure lambda calculus have been created, the semantics of the lambda calculus underlie most of the functional languages which were to follow. Indeed, functional languages can generally be considered as pure lambda calculus together with syntactic sugaring and type systems to ease the task of programming. Functional programs can be transformed into the pure lambda calculus and consequently their semantics are often defined in terms of the semantics of the lambda calculus together with transformation rules from the language into the lambda calculus. The following list is not meant to be complete, but should give a flavour of the way functional programming languages have developed and some of the major ideas which have arisen.

LISP [McCa60] was not deliberately based on the lambda calculus [McCa78], but McCarthy intended to incorporate the idea of referential transparency. In addition all data values including functions could be manipulated in the same way. Early implementations, however, did not have static scoping, and names were resolved by referring to the most recent definition that had been made of the name. LISP with dynamic scoping is not referentially transparent since it is not possible to know which value the occurrence of a name has statically - its value depends on the dynamic execution of the program. McCarthy later stated that the dynamic scoping of early LISP implementations was a mistake. In addition, early implementations of LISP included an assignment construct which also precludes referential transparency. Later implementations and variants of LISP (e.g. Scheme [Stee75, Abel85, Rees86] and Common Lisp [Stee82]) have returned to static scoping and no assignment.

ISWIM [Land66] introduced the `where` and `let` clauses which allow the definition of local values. These definitions were potentially mutually recursive. ISWIM also adopted the use of infix notation for operator expressions. LISP had previously insisted on all expressions being written in prefix notation e.g. (+ 3 4) instead of (3 + 4). Landin also introduced the SECD machine [Land64] as an abstract machine ideally suited for the evaluation of referentially transparent expressions. The SECD machine, albeit in a modified form, has been the basis for a number of functional language implementations including the Staple system described in this thesis.

Other approaches such as Iversons's APL (which has a functional subset) [Iver62] and later Backus' FP [Back78] use combining forms (higher order functions) which allow user defined functions to be combined in standard ways. These languages were restrictive in that the programmer could not define new combining forms. They showed that a functional programming style could be achieved without relying on lambda expressions. Backus hinted at some problems which

functional languages would face both in implementation and in particular, the one which concerns this thesis, their lack of support for large bodies of long term data.

The emergence of the importance of strong static typing and polymorphism [Stra67] are typified in the language ML [Gord78,Miln91]. Although ML has assignment and store semantics, it has a largely functional core language and has been popular with the functional programming community. Whether a value is mutable or not is denoted by its type. ML also included user defined algebraic and abstract data types and pattern matching although these innovations were earlier found in Hope [Burs77,Burs80]. Indeed pattern matching originated in the procedural world with SNOBOL [Farb64]. Type checking in ML is done by inference which automatically computes the most general type of each expression and allows the programmer to omit type assertions from the program.

Laziness has become prevalent in modern functional languages [Wads71,Frie76,Hend76]. A revolution in the implementation techniques available for lazy functional languages was begun with a graph reduction technique invented by David Turner [Turn79b]. Evaluation involved only a very small number of fixed transformation rules which were represented as higher order functions. These functions have the property that they have no free variables and are called combinators. Source programs were first transformed to ones involving just combinators and constants.

This new implementation technique for functional languages based on combinators [Turn79b] underlied implementations of the languages SASL [Turn79a], KRC [Turn82] and Miranda [Turn85]. The technique involves combinator based graph reduction and these languages employ lazy evaluation semantics in which values are only computed when needed. In SASL programs, definitions were made using equations which resembled mathematics. KRC introduced ZF expressions [Frae22] – now known more often as list comprehensions. List comprehensions are a syntactic sugaring which incorporate two types of operation which are very

important in functional programming – maps (where a function is applied to all the elements of a list) and filters (where elements of a list are selected to form the result list only if they satisfy some predicate).

One of the most recent languages in this family is Haskell [Huda92]. Haskell, named after the logician Haskell B. Curry, is a lazy functional language defined by an international body of computer scientists which brings together many of the features found in various existing languages. Haskell is an attempt to standardise in a single language, features found in a wide range of popular functional languages. The main technical innovation of Haskell is the ability to automatically infer the type of overloaded operators by grouping types together with operators over the types into classes. An operator belonging to a class has a separate instance for each of the types which are instances of the class. For example, plus (+) has an instance for integers and real numbers. These instances can be user defined, but many are included as part of the language definition.

### 1.1.2 Evaluation Strategy

A consequence of the property of referential transparency is that the language designer has a choice of evaluation strategy. The value of an expression does not depend on the order in which the sub-expressions are evaluated. For example function parameters may be evaluated before the body of the function is entered. This is known as *applicative order evaluation* or *call by value* [Naur63] and is the strategy found in most imperative languages.

A second possibility is to delay the computation of certain values, possibly indefinitely. Uncomputed values are called *suspensions*. This leads to a strategy known as *normal order graph reduction* [Wads71] or, where function calls are involved, *call by need*, because suspension allows a strategy in which no object is ever evaluated until it is needed. This reduction strategy has come to be known as *lazy evaluation*. Some programs which fail to yield a result when evaluated using

an applicative order evaluation strategy will terminate when evaluated using lazy evaluation. This is because the value which causes the program to fail to terminate with applicative order evaluation may not actually be needed for the result. Lazy evaluation would not evaluate such a value.

### 1.1.3 Laziness

Laziness arises from a reduction strategy in the λ-calculus called *normal order reduction*. In its pure form, normal order reduction involves replacing occurrences of a formal parameter of a function with copies of the actual parameter in the right hand side of the function. For example

double x = x + x

when applied to some argument which may take a long time to compute would yield

double *longtime* → *longtime* + *longtime*

which is an expression, with the same value, preserving referential transparency, but whose evaluation involves computing the value of the argument to the function twice. Lazy evaluation on the other hand, as embodied in the technique of graph reduction, is an implementation technique for normal order reduction which optimises expressions so that multiple occurrences of a formal parameter will share a representation of the unevaluated actual parameter (a suspension). When evaluated the representation will be overwritten by its value in place and the sharing of the suspension is replaced by sharing of the computed value (in weak head normal form). When a lazy evaluation strategy is adopted, suspension is made to happen in three situations.

Firstly, constructor functions, which create new structured values, do not immediately evaluate their arguments. 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 move sequences in a game. It is only at a later stage, if and when the value of a field of a structure is required, that it is evaluated.

Secondly, when a function is applied to an argument, the evaluation of the function's body may take a course that does not need the value of the argument. For example the function

```
f( x,y ) = if x>0 then y+1 else 0
```

may not need to evaluate `y`.

Thirdly, when bindings are made directly in definitions, the right hand side of the definition is not evaluated. The formal parameter should be bound to a suspension since its value may never be needed. So for example in the definition

```
fortyTwo = ultimateQuestionOf lifeUniverseAndEverything
```

the right hand side is not computed when the definition is made – thus something which may take a very long time to compute [Adam79] is delayed until its value is really needed if it is needed at all.

As well as being completely suspended, expressions may also be in partially evaluated states. The representation of a value proceeds through a spectrum of states of evaluation (all states of evaluation representing the same value) and may, if needed, finally arrive at a fully evaluated state, a *normal* or *canonical* form. This reluctance to evaluate expressions 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 empty list or not (e.g. a function which finds the length of a list), the parameter will only be evaluated enough to find out if it is the empty list or not and no evaluation of the elements of the list will be carried out. This is called evaluation to *weak head normal form* [Bare84, Peyt87]. An expression is in WHNF if it is either a literal value, the

application of a data constructor or it is a function applied to an insufficient number of arguments for a reduction to proceed.

Lazy evaluation is usually implemented by using a normal order evaluation strategy coupled with *graph reduction* [Turn82, John83, Hugh84, Peyt87b] in which the states of evaluation are represented by graphs. 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. Any graph can be *overwritten* by or reduced to another representing a more evaluated state. It is possible for these graphs to contain shared sub-graphs. When the shared sub-graph is overwritten, all references to it will refer to the new graph.

Sharing is introduced in a functional program by parameter binding. There are two ways in which binding of formal to actual parameters occurs. Firstly, aliases for the same value may be created with different names by giving a definition of the alias whose right hand side is simply the original name of the value. Secondly, formal parameters of a function may be used many times in the body of the function. When the function is applied, each occurrence can share the value of the actual parameter. It would clearly be advantageous if evaluation of any of the aliases implied that the others would not need further evaluation. In any situation in which sharing arises, overwriting a suspension by its value will affect all graphs which share it. If the expressions represented by those graphs are subsequently evaluated, they will have immediate access to the value which was computed for the sub-graph.

A number of functional programming languages (e.g. SASL [Turn79a], Hope [Burs80], LML [Augu84], Miranda [Turn85], Haskell [Huda92]) exploit lazy semantics which has the attractive features that:

- No value is ever computed unless it is needed for computing the result of a program.
- Some programs will terminate with a result, when evaluated lazily, which would otherwise have gone into a loop.
- Every value, if computed, is computed once only.

Consider the following function definition.

```
cond test left right = if test then left else right
```

If this function is applied to some actual parameters, for example

```
cond true 4 (5+6)
```

then the expression `(5+6)` is not evaluated because it is not needed for the result of the expression.

A consequence of this is that if evaluation of the third parameter to the function above fails to terminate, then the expression will return a result when non-lazy systems would not terminate.

### 1.1.4 Advantages of Laziness over Strictness

With lazy evaluation, values are computed at most once and only if they are needed. In addition, it is possible to describe very large and even infinite data values since only enough evaluation is ever done to satisfy present needs.

The efficiency, in both space and time, of programs is a major consideration for a programming system designer. An increase in the efficiency of functional programs may be achieved by using lazy evaluation semantics in which only values which are actually needed for the result are computed.

A programming technique associated with lazy evaluation is the ability to represent and manipulate infinite data objects. For example the following definition

```
from( n ) = n : from( n+1 )
```

allows the infinite list of positive integers to be expressed as `from( 1 )`. The list forming operator is ":". This expression starts life being represented by a graph representing the application of `from` to `1`. Depending on how much of the infinite list is needed later, it can be in any of the states:

```
from( 1 ) = 1 : from( 1+1 )
          = 1 : (1+1) : from( (1+1)+1 )
          = 1 : (1+1) : ((1+1)+1) : from( ((1+1)+1)+1 )
          ...
```

and of course any of the additions may be replaced by the integers that they represent at any time they are needed. All the expressions of the form `(1+1)` will be shared in a graph reduction system as will all those of the form `((1+1)+1)` and so on.

### 1.1.5 Costs of Lazy Functional Languages

Because of the ability to perform assignment, there are some algorithms which can be coded easily and efficiently in an imperative language which are much more difficult to code with the same space and time complexity in a functional language. One example of this is a queue, but the more general case is any algorithm which involves transforming a general graph. The transformed graph may need to be constructed by making a complete copy of the original one whilst maintaining the topology. This operation may be impossible since the program cannot detect where sharing occurs.

There are a number of costs associated with lazy evaluation on conventional machines. There is a significant overhead associated with building and storing representations of uncomputed values (suspensions). All data values must be tagged since it must be possible to distinguish between suspensions and computed values (although the spineless tagless G-machine [Peyt92] uses a different

technique). Every time a value is needed, a check has to be made to determine if the value is suspended. This will usually require a dereference of the object's tag followed by a test and conditional jump.

These costs explain why implementations of lazy functional languages on conventional machines tend to be less efficient than imperative implementations. Some techniques such as strictness analysis [Mycr80,Abra85,Abra87,Hank90], however, can greatly reduce these costs.

## 1.2 Persistent Programming

Persistent programming is a style of programming in which the programmer writes code which treats data uniformly regardless of its lifetime. Persistent programming languages help to address the problems of complexity in software engineering. Large programs can be extremely complex and difficult to maintain. A significant part of the code in data intensive applications is concerned with the movement of data to and from long term storage. The persistence abstraction hides this movement from the programmer, thereby reducing the overall complexity of the system.

### 1.2.1 Persistence

The persistence of data [Atki83] is the length of time for which the data exists and is usable. Thus there is a range of persistence which data can possess which can be grouped into two distinct categories. Those usually provided by a programming language are

1) intermediate values computed during expression evaluation

2) actual parameters in function evaluation.

3) values with global scope and heap items whose extent (lifetime) is different from their scope.

and those provided by the file system or database management system are

4) data which exists between invocations of a program

5) data which exists between versions of a program

6) data which outlives the program

Persistence is an abstraction which deals with the management of long term data in a way which is transparent to the programmer. In a persistent programming system, the programmer is relieved of the task of managing data which is required to outlive the execution time of the program. Traditional programming languages leave the programmer to design and implement specific code for storing and retrieving data from the file system. Persistence allows the programmer to concentrate on solving the problem at hand rather than have to concentrate effort on data transfer.

Database management systems have traditionally been responsible for the storage and manipulation of long term structured data. Access to the data is by the use of query languages which although well suited to simple querying tasks, are not able to perform general purpose computation [Atki87]. Persistent programming languages arise from the integration of general purpose programming languages in place of a query language and some form of persistent object management system (POMS) [Cock84,Brow89] responsible for managing the database of long term data objects.

#### 1.2.1.1 Identifying Persistent Data

The programming system must be given notification of which data values are to persist after the end of the program execution. This is so that the storage manager can arrange for these data values to be kept. There are two main techniques which allow persistent data to be identified [Brow89]. These are:

- Only data which is explicitly marked persists.
- All data which is reachable from some known point(s) will persist.

The first technique involves marking objects explicitly by giving persistent data a type different from those for transient data. In such systems, the user has more control over which data can persist. However, the type system is more complex and code must be written differently depending on the requirement that it be able to manipulate persistent or non-persistent data. Because code has to be written differently, some of the advantages of the abstraction are not realised. More serious objections are given below in Section 6.1.1.

The second technique is used by accessing and updating a data structure called the *root of persistence*. The user indicates that an object is to persist by making it reachable from the root. The underlying storage manager can discover which values must persist by calculating the transitive closure of the reachability relation from the root.

Early attempts at implementing persistence by reachability simply provided a core dump in which the entire machine state is dumped to a file and restored at a later time. Data which exists (e.g. in the heap or stack) is preserved regardless of whether it is available for use or not. The user has little or no control in such a system over which values persist. In addition, it may be impossible for independent developers to share software or data by this method, since their programs inhabit different unconnected universes, without using the file system which breaks the persistence abstraction.

### 1.2.2 Orthogonal Persistence

In addition to the above, persistent systems may have additional desirable properties.

Atkinson and Morrison have given a definition of *orthogonal persistence* [Atki83], defined as follows.

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 persistent data type orthogonality* is an extension of the *principle of data type completeness* which states that all data types should be first class without arbitrary restriction on their use. Data type orthogonality requires that all denotable values have full civil rights and can be passed to and returned from functions, named, stored in data structures and in particular 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 [Morr79,Morr82].

The process of identifying persistent data should be orthogonal to other features of the programming language and in particular, its type system. This is the *principle of persistence identification.* That is, the type of a data value should not depend on its persistence. Failure to adhere to this principle may lead to problems such as those exemplified in E [Rich89a,Rich89b] which proposes different types (called database types) for persistent data from those for non-persistent data. For example the type of a persistent integer is different to the type of 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 E is that they require the programmer to predict which objects will persist. This may not always be known statically. The need to differentiate between persistent and non-persistent data can be avoided by making everything a database type although there may be a loss of efficiency.

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, the sharing and topology of data structures is always preserved. This is the property known as *referential integrity* and is particularly important in a lazy functional language setting which relies on sharing for efficiency. Languages such as Amber [Card83] and a proposal for persistence in ML [MacQ89] do not preserve referential integrity in that objects accessed more than once will be duplicated rather than shared. Others such as E [Rich89a] may result in dangling pointers if a persistent data structure is created which contains non-database types.

In this thesis, the term persistence will be used to mean orthogonal persistence.

### 1.2.3 Advantages of Persistence

Morrison and Atkinson [Morr90] describe four main advantages which are derived from persistence as follows:

First, there is reduced complexity for the programmer. In non-persistent systems, the programmer must manage the mapping between database management system and the programming language and the different ways in which they each model the real world (Figure 1.2).

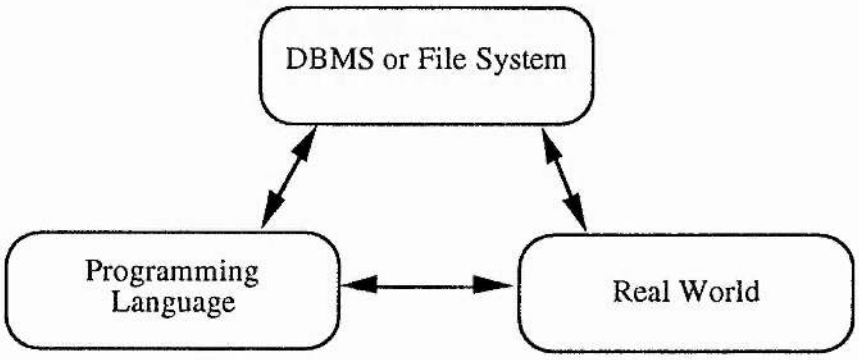



**Figure 1.2 - Mappings between DBMS, PL and Real World**

In persistent systems, however, the programmer need only consider the mapping between the real world and the persistent programming language using a single computational model (Figure 1.3).


**Figure 1.3 - A single mapping in a persistent system**

Secondly, there is a reduction in code size. One experiment [IBM78] found that 30% of the total code in a typical database application deals with the mapping of data between long term and transient storage. This code is not needed in persistent systems since it is exactly this functionality that is provided transparently by the persistent programming system.

Thirdly, persistence provides an opportunity to enforce a single protection mechanism throughout the programming system. A simple way to break the type systems of traditional languages is, for example, to output a string to a file and then read it back in as an integer. By ensuring that persistent load and store operations are strongly typed, such anomalies are removed. The complexity to the programmer of the anomaly is also removed and a uniform view of the type system applies to the entire programming system.

The fourth important benefit with persistence is that the topology of data structures is preserved over store operations. That is, referential integrity is always preserved. Data structures with shared subcomponents do not lose the sharing when they are read in from the persistent store. This is important for lazy evaluation which uses sharing of the representation of expressions to optimise the normal order reduction strategy. It is important if recomputation of values is to be avoided to preserve the sharing introduced during lazy evaluation.

#### 1.2.3.1 Software Reuse

When constructing large and complex software systems, the software engineer is faced with the task of controlling complexity and evolution. The complexity of the system must be managed and the system must be able to evolve over time. Persistent programming systems may be used to ease these tasks. In addition to orthogonal persistence, a flexible incremental binding mechanism can support a number of different methodologies for the binding of program components. Also to maximise the re-use of software, type abstraction allows program segments to be written which do not need to refer to a full specification of the data over which they operate. A polymorphic type system allows generic software components to be constructed in which procedures may be written independently of the type of the data which they manipulate [Stra67].

Languages in which functions are first class values support a style of programming involving higher order functions which provide a means of achieving modularity using first class functions. Such functions can be composed to construct new applications. If persistence is applied orthogonally to the language, it should be possible for these first class functions to be given the full range of persistence. This opens up the possibility of the persistent programming system providing support for users to place functions in persistent storage and for other users to bind to these functions and re-use them in different applications without copies being made [Atki84]

### 1.2.4 Costs of Persistence

Providing persistence in a programming system is not without its costs. The support for persistent data may be provided by a *persistent object management system* (POMS). The POMS manages the storage and retrieval of persistent data between the run-time heap and the persistent store. It is the POMS which decides when objects are transferred between the two. The programmer only sees the

POMS through high-level abstractions which effectively allow a data value to be made persistent or a persistent data value to be accessed. However, these high level abstractions must be supported by the POMS which must maintain internal data structures such as address translation tables.

The persistence abstraction itself may lead to a loss of efficiency in some programs since the programmer does not have full access to the underlying system and is restricted to the functionality provided by the interface to the store. For example, all store access may first have to pass through an address translation mechanism which may cause an object fault which triggers off a disk read.

The persistent systems in existence today are all constructed in software [Cock84,Care86,Brow89]. The activity of the object manager, however, is much more closely related to the memory management functions like paging and virtual memory. A hardware address translation mechanism for persistent objects might improve the performance of persistent systems [Rose85,Rose87].

## 1.3 Combining Persistence with Lazy Functional Programming

By combining persistence with lazy functional programming, it becomes possible to reuse values which have been computed by a previous program invocation or even by a different program. A combined system has advantages with efficiency and the application domain for which the programming system can be used for implementation is extended.

In this thesis, two models for persistence are described which provide automatic management of persistent data. The programmer does not have to write any explicit code for dealing with the transfer of data between long and short term storage. In addition, any data value of any type can be stored and reused later.

Large software systems evolve over time and it is desirable that a persistent lazy functional programming system should support this evolution in a controlled and safe way. The use of persistent values is governed by the programming language type system and it is possible for such a system to be strongly typed. The use of persistent data can be guaranteed to be type safe between different invocations of a program or between different programs.

The state of evaluation of data values can be preserved in persistent storage so that computation performed on persistent values is done only once and all programs which reuse the value will benefit from the evaluation. The persistent store acts as a cache in which values are stored and subsequently shared between many programs over time. During the lifetime of a data value, its state of evaluation changes monotonically towards a fully evaluated state. In a non-lazy persistent system, data values are computed, stored in persistent storage and subsequently reused many times. Analogously in a lazy persistent system a data value is computed, stored and then shared. The data value may have any state of evaluation when it was first stored. Use of the value can cause more evaluation to be performed. Frequently used values will in general be in a more evaluated state than infrequently used ones. There may be a case for replacing data values with the suspensions which yielded them, for example to free some space. This topic is not pursued in this thesis, but may be an interesting avenue of future research.

A combined persistent lazy functional programming system can go some way to tackling the problems of scale and longevity of data. In this thesis, two models are developed which combine the two paradigms. These have allowed a new style of programming – persistent lazy functional programming. Together with persistence, a lazy functional programming system can facilitate the construction of larger scale applications which are difficult to construct in non-persistent functional systems.

Integrating persistence and lazy functional programming can yield benefits in efficiency. In Section 1.3.1 there is a description of common technique in lazy programs, memoisation, which exemplifies the benefits in efficiency which can be achieved with a persistent system.

In addition to the benefits in efficiency, the application domain for which such systems are suitable is extended. Section 1.3.2 discusses some of the issues which arise when constructing a new class of applications in which programs access and manipulate persistent data interactively.

### 1.3.1 Improving Efficiency

Values in functional programs may be in one of any number of states of evaluation. In non-persistent systems, information about the state of evaluation is lost at the end of program execution or the end of an interactive session and there must be some recomputation to return to a similar state. The results of a computation can be stored in the file system, but code must be written to perform the input and output. Furthermore this input and output is inherently unsafe since the type system governing the use of stored values does not match that of the programming language.

In a persistent system, however

- data values computed dynamically can be shared between many program invocations, computation which is performed on shared values is never repeated so the benefit of the evaluation lasts for as long as the data is usable

- support for long lived data is automatic and provided as part of the programming system

- the long term data storage is strongly typed since the use of persistent data is governed by the programming language type system

A data value such as that defined as follows

```
integers = from 1
           where
           from n = n:from (n+1)
```

which is preserved in persistent storage and subsequently reused many times will gradually become more and more evaluated. For example if the third element of the list is requested using the list indexing operator ! as follows

```
integers ! 3
```

then the state of evaluation of the list will be the following list.

`1:2:3:`*a suspended representation of from 4*

Any use of the first, second or third elements of the list will cause no evaluation other than that required to scan down the list to the required point, but if an element beyond the third is requested, for example the ninth

```
integers ! 9
```

more evaluation is performed and the state of evaluation becomes

`1:2:3:4:5:6:7:8:9:`*a suspended representation of from 10*

and so on. The state of evaluation continues to change as further elements beyond the last one computed are requested. This monotonic increase in the state of evaluation is a particularly important feature of persistent lazy functional systems. Any sharing which occurs in the representation of data values is preserved.

### 1.3.2 Facilitating Database Application Building

Trinder [Trin89a] argued that current functional programming systems did not provide enough support to build database applications. Some of his requirements are that an implementation language must have the ability to store data permanently and to retrieve previously stored data. Whilst access to a file store is provided in

many functional languages, there are a number of difficulties with ad-hoc approaches to long term storage (see Section 6.1.1). The operations on long term data (lookup, update, insert, delete) must be efficient. Persistence offers a mechanism which provides efficient storage of long term data with the ability to store values of any type. This allows the database implementor to utilise the richness of the programming language type system when modelling the real world.

A typical functional database application is one which processes a stream of transactions each of which performs an operation on the database. In a functional system, the database is viewed as being transformed and a new database is produced which represents the old database after the transaction has been processed. This process can be modelled in a functional language as follows

```
process (transact:transactions) database
          = process transactions (transact database)
process [] database = database
```

The parameter `transact` is a function which transforms one database into another by performing some operation such as update, insert or delete. The changing state of the database is modelled by passing each transformed database back into the processing cycle at each recursive call. A database system requires that the database itself be stored permanently. This thesis presents a language model which supports this kind of processing.

Query languages are used to interrogate databases [Date77,Ullm80]. The query language FQL [Bune79], based on the functional language FP uses a small set of higher order functions which can be composed to manipulate the database. Lazy evaluation of the list processing can reduce the number of disk accesses required to process a query [Trind89a].

Functional queries can be expressed using the concise notation of list comprehensions. List comprehensions are a clear and powerful syntactic notation

for denoting queries [Nikh87,Breu88]. They provide a more natural notation for users than do the terse parameterless notation of FQL/FP. Query optimisation in the relational world has provided techniques which can significantly improve the performance of relational algebra queries. These optimisations turn out to have analogous optimisations when using list comprehensions to represent queries [Trin89b]. Functional databases have been modelled in lazy functional languages but, because of the lack of persistence, have usually been translated into, at best, a persistent imperative language. The Staple system can be used directly to implement functional databases.

Functional languages seem to offer many of the facilities required for the construction of functional databases. The model for persistence described in Chapter 3 enables programmers to construct interactive database applications in a purely lazy functional language. The implementation of some typical database functions using this model of persistence is developed in Chapter 6.

### 1.3.3 The Staple System

The Staple system incorporates the two models for persistence described in this thesis. These models have been integrated with a lazy functional language with adherence to the principles for orthogonal persistence and whilst retaining referential transparency.

The first model utilises a module system in which a module is a set of definitions of data values and data types. All data values exported by a module are retained in a persistent storage system. This module is essentially static in nature and does not allow functional programs to manipulate the content of the persistent store.

The second model allows more interactive access to the persistent store. The property of referential transparency is preserved by using stream processing to encapsulate commands to update the store and for receiving results back. The persistent store processes requests generated interactively by user programs and

returns responses to the requests in a stream which may be evaluated by the program at some later time.

The implementation of these two models relies on a stable storage system developed by Fred Brown [Brow89,Brow90]. This storage system underlies implementations of other persistent languages including Napier88 [Morr89] and Galileo [Alba85].

## 1.4 Related Work on Persistent Languages

In this section, a number of persistent languages with features related to the work presented in this thesis are discussed. Where appropriate, similarities and differences between these languages and the Staple system are noted.

### 1.4.1 Amber and CAML

Amber [Card85] and CAML [Lero91] are non-lazy applicative languages which support persistence by providing functions to export and import values to the file system. In common with Staple, the values to be made persistent must be of a special dynamic type. When a value of the dynamic type is to be used, a run-time type check is performed to enforce the strong typing in the language. The dynamic type itself is an orthogonal feature of these languages.

CAML the implementation of ML developed at INRIA provides the two functions for creating and accessing persistent data:

```
extern : extern_channel × dyn -> unit
intern : intern_channel -> dyn
```

The expression

```
extern outChannel (dynamic 1)
```

writes a value of the dynamic type out to the file specified by `outChannel`. The expression `(dynamic 1)` creates a value of the dynamic type. To read this in at a later time, the expression

```
intern inChannel
```

is used where `inChannel` is the channel obtained by opening the appropriate file.

The `extern` function is a function which is executed for its side-effect of updating the contents of a file. Because this function is imperative in nature, there is a loss of referential transparency. It would therefore be an unsuitable method for adding persistence to a purely functional language.

These systems have an external file system representation of data which can be accessed by opening and reading the file contents. In this thesis, however, all data regardless of its persistence is treated and stored uniformly and is strongly typed.

The CAML implementation also exhibits a loss of referential integrity over store operations. If a value is read in from persistent storage more than once, then the resulting values are not shared, but are duplicates. It is possible to detect when a value which has already been read in is requested a second time, but the file contents may have changed in between the two requests. As mentioned above, Amber and CAML are non-lazy languages and a loss of sharing may not have serious consequences other than requiring more storage space in the run-time heap. In lazy languages, however, referential integrity over store operations is important for the efficiency of lazy evaluation. If sharing is lost, then additional unnecessary evaluation may be required.

The complexity of the management of persistent values required to ensure referential integrity is preserved is subsumed by using a persistent object management system (POMS) such as the one described in Section 4.4. Such a POMS ensures the referential integrity of persistent values. In Staple, a POMS is

used to ensure referential integrity. Because sharing is always preserved, lazy evaluation provides its full benefits over the full lifetime of a data value.

### 1.4.2 Agna

Agna [Heyt91] is a LISP like language which defines databases as a top-level persistent environment of bindings that associate names with persistent objects. User defined object types may be introduced and if these type definitions are qualified with the extent keyword, then a structure containing all values created with the defined type is maintained automatically by the system. For example, a type for representing lecturers would be defined as

```
(type LECTURER (extent)
((name <=> STRING)
(salary => NUMBER)
(room   => NUMBER)
(coursesTaught *<=>*) COURSE)))
```

Each time a lecturer is created, it is added to the persistent data structure. This method of providing persistence breaks the principle of persistence identification, because persistence of data does depend on its type. Agna, therefore, does not have orthogonal persistence.

In Staple, persistent data can have exactly the same types as transient data. This leads to much less complexity for the programmer to grasp. When a data value is created, there is no side-effect of a persistent data structure being updated with the new value. It may be that the programmer requires this value to be transient. To do so in Agna, a different non-persistent type would have to be used.

### 1.4.3 Poly

Poly [Matt88] is an interactive programming system in which the user enters expressions and definitions which are executed immediately. When a session ends, a complete copy of the state of the system is saved. This is essentially a core

dumping technique for preserving persistent data. At the beginning of the subsequent session, the system state is restored and the user continues as though there had been no interruption. As an optimisation to this technique, Poly uses a persistent storage system which brings values into the run-time core image on demand.

Poly groups definitions together into environments which Matthews likens to directories in a file system. When an environment is selected (the current environment) all new definitions are added to that environment. Thus the environment is extended. Declarations can only be added or removed by the interactive environment. It is therefore not possible to manipulate the environments from within an interactive program. In Staple, stream persistence allows executing programs to issue requests to update persistent data whilst preserving referential transparency.

### 1.4.4 PSASL

PSASL [Dear85] is a persistent version of Turner's SASL language which allows the top level environment to be named, stored and retrieved. For example, a session in which the definitions

```
def a = 3
    b = 4
?
```

have been made defines an environment in which a and b are bound. This environment can be stored with

```
dump envName
```

In a subsequent session, the environment can be restored with

```
load envName
```

persistent modules extend the environment model with a strong type system and the ability to combine modules together.

As with Poly, Agna, CAML and Amber, programs themselves cannot manipulate the contents of persistent storage during execution.

## 1.5 Thesis Structure

This thesis concerns the integration of the two styles of persistent and functional programming into a single programming system. Chapters 2 and 3 describe two possible language models for persistence in lazy functional languages – persistent modules and stream persistence. Persistent modules provide a static model for persistence in functional programs. In order to access persistent values in an interactive way during the execution of a program, a different model is required. Chapter 3 describes Stream Persistence, a model for persistence in which persistent values can be accessed dynamically. Chapter 4 describes the PCASE abstract machine which underlies the implementation of persistent modules and stream persistence. Chapter 5 details the underlying architecture for the implementation of a functional persistent programming language and describes the techniques used to implement persistent modules and stream persistence. Chapter 6 describes two applications which benefit from using the two models for persistence. Chapter 7 concludes and points to future directions of research.

# Chapter 2

# Persistent Modules

The Staple system allows persistent modules to be created, combined, used and modified. The system also provides a standard interactive programming environment in which expressions can be entered and evaluated. These expressions may refer to names which have been brought into scope by importing persistent modules. The state of evaluation of persistent values changes over time monotonically towards a fully evaluated state. When a persistent module is created, binding to free variables in imported modules are made and type checked. Referentially transparency is ensured because these bindings are made at module creation time before any of the values in the module are used.

## 2.1 Introduction

A functional program typically consists of two parts. The first part is a set of definitions of data types and values (including functions). These definitions are usually grouped together into a *script* or *module*. The module defines an environment in which the second part of the program, an expression, is evaluated. Larger programs can be constructed by combining modules together. These modules provide the foundation on which the first model for orthogonal persistence in a lazy functional programming system was built [McNa90].

Persistent modules are a means of providing persistence in a transparent way. The scripts or modules which functional programmers are accustomed to writing are still used, but as a mechanism for identifying which (named) values or type definitions should persist. The values which are named at the outer scope in a module will persist.

A module is a source program created using a text editor and stored in the file system of the host machine. The source file is then compiled and a module is placed in persistent storage. The values and types in a module are accessed by supplying the name of the module to the compiler. Data values defined within the module are initially suspended and as they are used, the state of evaluation changes monotonically towards a fully evaluated state.

In a non-persistent module system, as in Haskell [Huda92] for example, all data values must be recomputed from a completely unevaluated state each time a program which imports a value from another module is executed. In a persistent module system, values imported from modules may be in any state of evaluation reflecting the previous usage of the value.

The persistent store is used in two ways.

- modules are compiled and placed in the persistent store
- expressions are entered and evaluated interactively possibly referring to values in modules in the persistent store.

These two facilities are provided by the programming environment. An integrated persistent functional programming environment which provides persistence in this way is described in [Davi89a]. The Staple system provides these facilities at the operating system command interface level.

## 2.2 The Command Interface to the Persistent Module System

The Staple persistent environment contains a set of modules each of which can contain a number of definitions. The modules are organised as an associative lookup table within the persistent store and modules are accessed at compile time via their name. The name of a module is in no way part of the programming language, but is used by the programming system to identify which modules are to

be imported. 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 involve the use of such an environment. The session allows a user to enter expressions for evaluation which can refer to any named persistent values currently in scope. The values referred to may come from many different modules.

The following examples illustrate the use of the command interface to the Staple persistent module system. A Staple system will include some pre-loaded modules in the persistent store ready for use. These correspond roughly to standard libraries found in other programming environments. In particular, Staple has a standard prelude which contains a number of useful function definitions (see [Davi90]). It may, in any case, be desirable to partition the standard prelude into logical groups of definitions. Two of the function definitions found in the Staple standard prelude are given in Figure 2.1 and are used in the examples which follow.

```
-- sum calculates the sum of all the elements in a list
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

-- zip2 takes two lists and produces a list of pairs
--      containing corresponding elements of the arg lists
zip2 :: [s] -> [t] -> [(s,t)]
zip2 (x:xs) (y:ys) = (x,y) : zip2 xs ys
zip2 [] ys = []
zip2 xs [] = []
```

**Figure 2.1 - Part of the "prelude" module**

The examples below use lists to model vectors of integers and lists of lists to model matrices. The first example module which a user might write is given in Figure

2.2 and defines two functions – one to calculate the inner product of two vectors and one to calculate the transpose of a matrix.

```
-- a vector is a list of integers
-- a matrix is a list of vectors
-- uses sum and zip2 from the prelude

-- inpr calculates the inner product of two vectors
inpr :: [Int] -> [Int] -> Int
inpr v w = sum [ x*y || (x,y) <- zip2 v w ]

-- transp calculates the transpose of a matrix
transp :: [[Int]] -> [[Int]]
transp (v:vs) = [ x:w || (x,w) <- zip2 v (transp vs) ]
transp [] = [] : transp []
```

**Figure 2.2 - The module "utilities"**

For this example, it will be assumed that the source code is stored as a text file called `utilities` in the file system. The user can create a persistent module by using the `mkmodule` command. This command is given the name of the file containing the source code of the module and the names of any other modules which contain values for the free variables used in the persistent module being created. If a module refers to values in more than one module, then the names of those modules are listed as additional arguments on the command line. So the general form of a mkmodule command line is.

```
mkmodule filename modulename1 ... modulenameN
```

If an identifier is bound in more than one of the imported modules, the one which occurs in the first imported module on the command line is used. Each module in persistent storage is given the name of the file which contained the source code used to create it.

For the present example, the `prelude` module must be specified for the resolution of the two external references to `sum` and `zip2`. The command line is as follows.

```
mkmodule utilities prelude
```

During the compilation of the `utilities` module, the use of free variables is type checked. The types of the free variables is obtained from the imported modules, `prelude` in this case. Thus all type checking is performed during compilation and the system is strongly and statically typed. The bindings to values imported from other modules are also made at compile time.

When the command has completed successfully, there will be a new association between the name `utilities` and a module in the persistent store. The name space for modules in the Staple system is a single flat structure in which each module name refers to a unique module. The name space for modules is completely separate from the name space for programs. The Staple system provides support for organising the store structure by stream persistence described in the next chapter.

The values and data type definitions in a module can be accessed by other modules in the same way that the prelude module was used in the above example. A new module, called `matrix`, which uses the definitions in `utilities` to implement a function to multiply two matrices is shown in Figure 2.3.

```
-- mmul multiplies two matrices
mmul :: [[Int]] -> [[Int]] -> [[Int]]
mmul m1 m2 = [[inpr v1 v2 || v2 <- transp m2] || v1 <- m1]
```

**Figure 2.3 - The module "matrix"**

To load this module into the store, the `mkmodule` command is again used. This time, the command is given the name of the module containing the values of `inpr` and `transp` which were defined above. The command line is as follows.

```
mkmodule matrix utilities
```

The examples above illustrate how the values and types defined in modules can be used by other modules. In the Staple system, the values and types can be used when evaluating expressions interactively. To start an interactive session, the `staple` command is used. This initiates a session in which the user is asked to type in expressions which are immediately evaluated and printed. In addition, the user can specify an environment by naming the modules to be imported. For example, to start a session using the matrix multiplication module, the following command line is used.

```
staple matrix
```

The Staple system informs the user that a binding for the name `mmul` has been made in the current environment. The user can then refer to `mmul` when entering expressions to be evaluated. For example

```
mmul [[1,2],[3,4]] [[5,6],[7,8]]?
```

The interactive environment can be changed by issuing commands to import different modules from the persistent store. Further examples are given in Chapter 6.

### 2.2.1 Changing the Interactive Environment

The interactive environment is determined when a session starts by the command line parameters. For example

```
staple matrix prelude
```

starts an interactive session in which all the names in prelude are defined and all the names in matrix are defined (See section 2.4 for a discussion on what happens with name clashes).

During an interactive session, the user can change the environment by typing `\use` followed by the module name. For example, typing

```
\use miniprelude
```

changes the interactive environment to one which only contains bindings for names in the `miniprelude` module. Other interactive commands are described in the Staple reference manual [Davi90].

## 2.3 Binding

The bindings between free variables and values in imported modules are made during the execution of the `mkmodule` command. The state of evaluation of data values can change, but the name remains bound to a representation of the same value. Even when a module is replaced by a more recent version, bindings to values in older versions remain unchanged. It is left to the programmer to reload any modules which depend on updated ones and in which a binding to the new value is required.

## 2.4 Scoping

Identifiers may be bound in many different modules. When a name is bound more than once in the same scope, a name clash has occurred. Name clashes which occur during module compilation are resolved by lexical scoping. The order in which modules are imported determines which value is used. Each imported module forms a separate lexical level and any use of a name refers to the value in the innermost module. This is as though the definitions in each module were made in block expressions with definitions in outer modules appearing as definitions in outer blocks.

A module exports all values named within it including constructor functions. In addition, the type forming operators of algebraic types are also exported. Values imported are not exported, but the names of types are propagated. If all imported values are implicitly exported, then the interface of a module would include many irrelevant and unnecessary references. This simple model, however, does allow a

large degree of flexibility in system construction. For example, one consequence of exporting only local names is that information hiding can be supported in a simple way (see Section 2.6.2).

## 2.5 System Evolution

As software systems are developed, their components typically evolve through various stages and versions. These changes can be accommodated by the system in a way which preserves the integrity of the system from the viewpoint of each individual developer. In particular, changes made by one user do not change anything except that user's view of the system. If such changes are to be distributed, then other developers must explicitly incorporate them. The user has control over the way changes are propagated through the system rather than having changes side-effect other values in the persistent store. This control cannot be provided within the language itself since referential transparency would immediately be lost. The mechanism which supports change must be at a level above the denotation of the language – in this case, it is provided by the programming system, that is the compiler.

The property of referential transparency must be preserved in the persistent module system because it is this property which yields the benefits discussed in Chapter 1. However, system evolution must be supported so that programs can be developed over time. These appear at first to be conflicting requirements but it will be seen how the conflict can be resolved. Once in the persistent store, no value is mutable. The resolution of free identifiers in a program is performed when the module is aggregated into the store. Values which are imported from other modules must already be present in the persistent store before a module can itself be loaded. Mutually recursive modules require a relaxation of this in which the values imported must be in the persistent store or in another module which is being loaded simultaneously. The Staple system does not currently support mutually recursive modules, however.

The bindings to imported values in the persistent store are made at compile time and these bindings are fixed. Referential transparency is preserved because values are completely specified when loaded into the store. In particular, all the values on which an expression depends (such as imported values) are known at the time a module is loaded. When a module is to be updated, references to values in the old version will be retained satisfying the requirement that values represented in the store are immutable.

With a lazy evaluation strategy many objects will be only partially evaluated. The state of evaluation of partially or wholly evaluated objects can be made to persist by naming the values within a module. Re-computation of a value which has been named in a module is avoided since the reference in the module will always refer to the representation of the object in its current state of evaluation.

### 2.5.1 Replacing a Module

The mapping between a module identifier and the module representation in the persistent store is updated by the `mkmodule` command. Values in the old module are no longer accessible through the module name. They may however, be accessible if they are accessible through some other route – if they have been imported into another unchanged module. References from other modules to values in a previous version of the module are not changed by the update operation. A typical situation where a value in a module is shared by many other values is shown in Figure 2.4.

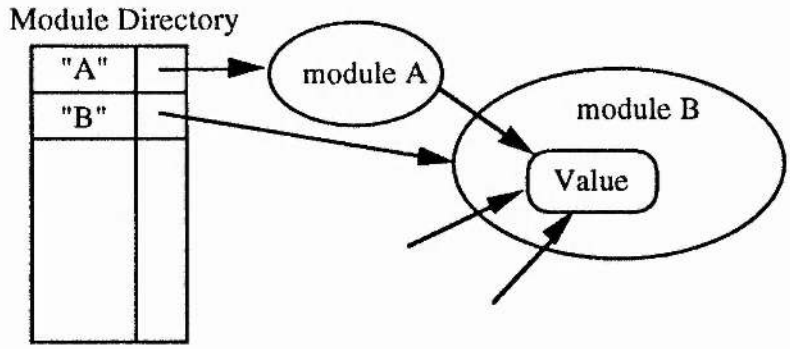



**Figure 2.4 - A shared value in module B**

The value in module B has been imported by module A and referenced in one of the definitions in module A. Figure 2.5 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 are no longer accessible. The value used by module A and shared by other modules, however, is not lost.

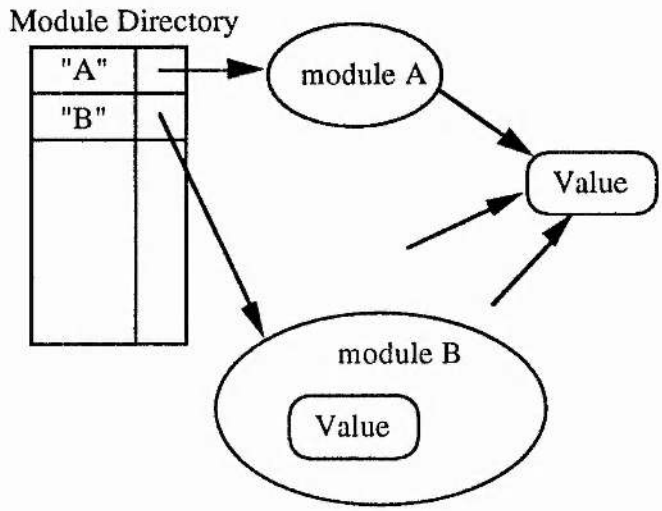



**Figure 2.5 - After updating B**

In some cases, it may be desirable to automate some of the update in the system. Automatic mechanisms such as the "make" utility in UNIX or the system outlined in [Davi89a] could be used in the programming environment to support this kind of

functionality. The user has the choice of automatic or explicit propagation of change in the system.

## 2.6 Examples of Using Persistent Modules

Once modules have been loaded into the store, they can be used interactively. A second mechanism is required which allows the user to evaluate expressions interactively in an environment defined by the import of a number of modules which must already be in the persistent store. Expressions evaluated during the interactive session may require 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 the state of evaluation of the value in the persistent store.

Two examples of modular persistent functional programming will now be described. The first is a small one to illustrate the principles involved, the second shows how abstract data types can be implemented even with the simple model provided here.

### 2.6.1 Memoisation

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 concise way:

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

Here the `from` function is in a '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. The above two definitions comprise a small module which is loaded into the persistent store. An indication

that a prelude module containing a definition for the function `from` is to be imported must also be given.

Before any evaluation takes place, the value associated with primes is a suspension as shown in Figure 2.6.

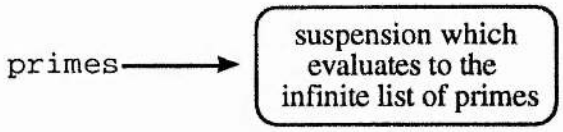



**Figure 2.6 - Initial binding before any evaluation**

Figure 2.7 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.

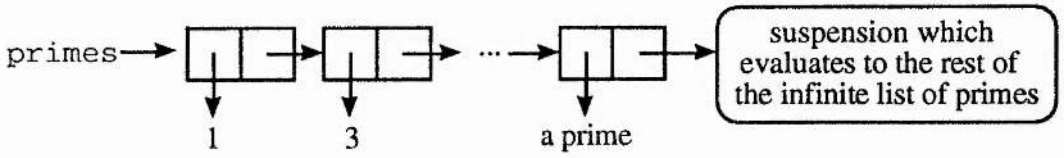



**Figure 2.7 - after evaluating part of the list**

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

Once loaded into the store, subsequent uses of the `primes` list can made in two ways. Firstly, it can be used interactively by invoking the interactive session tool and indicating that the primes module just compiled is to be imported and should form part of the working environment. Second, it can be used by importing it during the compilation of another module in which case the names `primes` and `sieve` will be available.

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 compute the above expression involves 1050 function applications. If, at some later time, the 20th element, say, in the list is requested

```
primes!20 ?
```

only 42 function applications (associated with the list indexing operator `!`) are needed since the majority of the work has been done already. 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.

It may be that over time, the list of evaluated primes will become long after significant evaluation has been performed on it. Some technique to recover space used in this way may be needed. The difficulty is that it is not possible to predict whether a value may be used again at some time in the future (see section 7.2.4).

### 2.6.2 Encapsulation

Encapsulation or information hiding is a commonly used technique to reduce complexity when programming in the large. The technique of hiding the implementation of abstract data types can ensure that values of the implementation type are only manipulated using the interface provided. Constraints on the use can be incorporated into the interface functions. The Staple language does not have

abstract data types since the focus of interest is system construction rather than type systems. Information hiding can be achieved in Staple at the module level as a consequence of the export mechanism.

The method used for type equivalence checking does not require the detailed structural definition of an algebraic type to be known. All that is required is the name of the type constructor and its type argument. This is discussed further in section 3.5.

The module system allows abstract data types to be modelled by a two stage process. Consider the following module which defines a `Stack` datatype.

```
data Stack t = EmptyStack | Push t (Stack t)
```

An example application might require that stacks be created and manipulated not using the data constructors defined in this module, but by using the functions `emptyStack`, `push`, `pop`, `top`. Where these functions are the typical stack operations. A module is required in which these functions and the type forming operator `Stack` are all that are exported hiding access to the constructor functions `EmptyStack` and `Push`.

The above module exports the type forming operator `Stack`, the constructor functions `EmptyStack` and `Push`. Since imported values (including constructor functions) are not propagated then if the above module is imported into the following module the result is a model of a stack abstract data type.

```
-- module called stack

emptyStack :: Stack t
emptyStack = EmptyStack

push :: t -> (Stack t) -> (Stack t)
push x s = Push x s

pop :: (Stack t) -> (t,Stack t)
pop (Push x s) = (x,s)
```

```
top :: (Stack t) -> t
top (Push x s) = x
```

**Figure 2.8 - The module "stack"**

The stack module can be imported and used in other modules without reference to the type definition for the `Stack` data type. The following expression is an example of use of the stack module.

```
result where
        result = top s5
        s5 = push (m+n) s4
        (m,s4) = pop s3
        (n,s3) = pop s2
        s2 = push 3 s1
        s1 = push 2 s0
        s0 = emptyStack
```

**Figure 2.9 - Using the values in module "stack"**

It may be desirable to allow access to the constructors from modules which import the definition of an algebraic data type indirectly. This can be achieved by also importing the module containing the definition of the data type since it is that module alone which exports the constructor functions.

In order to hide fully the implementation type `Stack`, the module defining the type must be made inaccessible. This could be achieved by deleting the module which defined the type. Otherwise it would be possible for a user of the stack module to import the type definition and see the implementation type.

## 2.7 Conclusions

The Staple system provides orthogonal persistence with persistent modules. Persistent modules allow programs to take advantage of persistence without the need to learn a new language mechanism. The source of module contains

definitions of values and types. This is compiled and placed into the persistent store. The values and types in the module can be accessed and used in other modules or in expressions entered interactively by giving the name of the module or modules.

- values in a module may become shared by many other modules.
- any evaluation performed benefits all the users of the value.
- evaluation is performed at most once during the lifetime of a persistent value.
- value persists for as long as it is reachable from a named value in a module in the persistent store.
- static type checking is performed during module or expression compilation.
- bindings to imported identifiers are made at compile time.
- referentially transparency is preserved as a consequence of performing compile time binding.

# Chapter 3

# Stream Persistence

## 3.1 Introduction

Stream processing functions [Burg75] have been used in functional programming languages to provide a model for input/output which preserves referential transparency [Jone84,Thom86,Huda88]. Input from the keyboard, for example, can be modelled as an infinite lazily evaluated list of characters. Character based output can be similarly modelled. Such infinite lists are called streams. This idea can be extended to allow interaction with other resources such as the file system and printers by embedding requests into the output stream which is now a list of requests. The requests are processed by the programming system which returns appropriate responses in the input stream of the executing program.

Persistent data can be regarded as a resource which can be made available to functional programs. Extending the idea of stream I/O with requests to access persistent data yields stream persistence.

Stream persistence is a model for persistence which provides functional programs with the ability to access and update persistent data dynamically. This model is based on a suggestion made by Simon Peyton Jones [Peyt91].

A functional program uses stream persistence by producing requests to lookup or update persistent data. The run-time system accesses the persistent store and responds with the persistent data value for a request or some indication of successful completion.

Stream persistence is provided in Staple by a combination of a dynamic type similar to that found in Amber [Card83] and Napier88 [Morr89] and stream based I/O. Referential transparency is preserved over dynamic store operations because

the response stream, through which the state of the external environment and in particular the state of the persistent store is observed, contains components which themselves do not change.

A dynamic type is used to provide a flexibly evolving object store. However, a consequence of having a dynamic type is that type checking can not be performed entirely statically. A dynamic type check is required when a value is projected out of the dynamic type. All data stored by persistent stream processing has the dynamic type.

The persistent store evolves whilst a functional program is executing. In particular, some data values are transformed by having evaluation performed on them. Evaluation performed on persistent data values is retained for the lifetime of the value. In addition, programs can produce results which themselves can outlive the execution time of the program which created them. These data values may be created as the result of interaction with the user. With stream persistence, it is possible to write interactive systems which process large bodies of long term data in a purely functional style.

## 3.2 Streams and Stream I/O

Consider a situation where one function produces a list of values and another processes the values in the same order. Lazy evaluation allows the second function to begin its work without the whole list being evaluated. Such a system can be set up so that the consumer function requests the next value from the producer (Figure 3.1). In this way the elements of the list are only produced when they are needed. The sequence of values passed between functions can therefore be infinite. A function which generates such a sequence of values is called a stream function or stream. Here, stream will be used to mean an infinite lazily produced list whose elements are evaluated sequentially. This is usually implemented using a head-

strict version of the list construction operator cons. Thus there is a relationship between an element's position in the list and the time at which it is evaluated.

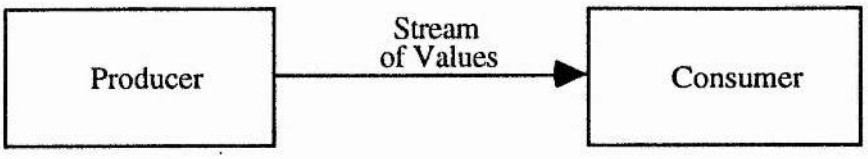



**Figure 3.1 - Producer and Consumer of Streams**

Functions which operate over streams are similar to those over finite lists but they need only consider arguments which are not the empty list. Streams can be used to model the continuous interaction between a functional program and its external environment such as the keyboard, the screen and the file system [Burg75]. Streams are often implemented using the built-in list data type.

### 3.2.1 Character Streams

A basic stream model for interactive functional programs views a program as a consumer of characters on its input stream and as a producer of characters on its output stream. A program is of type:

```
program :: [Char] -> [Char]
```

That is a program takes a stream of characters as its input and produces a stream of characters as its output. When a program is executed, the input list is supplied by the system and the output list is printed causing its gradual evaluation. The print driven nature of the evaluation determines the synchronisation of the input and output. Characters on the output which depend on the input stream will cause the input stream to be evaluated far enough for the output to be computed. To obtain the desired interactive behaviour, the programmer is responsible for writing programs which involve the correct data dependencies between components of the input and output streams. Techniques for performing interactive input/output with character streams are described in [Thom86].

### 3.2.2 Token Streams

A more general mechanism called *token streams* [Stoy85] and used in Haskell [Huda92] provides access to a wider range of resources by embedding requests in the output stream which are subsequently interpreted by the run time system. The input stream consists of a sequence of responses each of which provides an answer to a request to the operating system. Synchronisation is ensured by the programmer adhering to the convention that requests are placed on the output before a response is examined. This is achieved by the same style of programming described above in which the programmer explicitly defines the data dependencies between requests and responses. A program with this model has type:

```
program :: [Response] -> [Request]
```

A program using token streams can be considered as a black box which produces a stream of requests for resources or actions and consumes a stream of responses. The run-time system itself is a consumer of requests and producer of responses. Some examples of requests are "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 typically "OK I've done it" or "the contents are 'Some text'" or in some cases "Failure: no such file" etc. By convention, there is a correspondence between requests and responses. Each request generates a response in the same position in the response stream as the request in the request stream. Figure 3.2 shows a conventional stream I/O system as seen by a single user.

Whilst functional programs are referentially transparent, the external environment is not restricted to being so. The external environment can change as a result of requests from the program or as a result of some other external stimuli. Because the external environment is not directly under the program control, requests for actions may fail and this must be reported back in the form of an appropriate response. Effects such as updating the external environment are performed by the

run-time system, but the program will only access different states of the external environment as distinct immutable values embedded in the response stream which itself is not mutable.

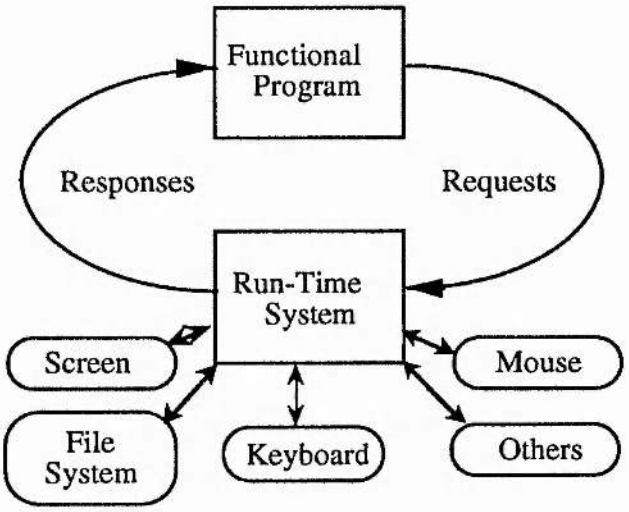



**Figure 3.2 - Token Stream I/O**

In a sense, the response stream traces the history of the changing state of the system because each response is produced according to the state of the external environment at the time the corresponding request is processed. For example in response to requests to read the contents of a file, then to write the string "hello" to the file and finally to read the contents of the file will result in the response stream in Figure 3.3.

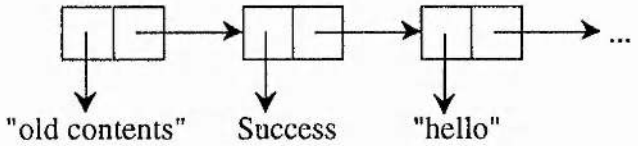



**Figure 3.3 - The response stream**

The first response is a string representing the contents of the file at the time the first request to read the contents is processed. The second response indicates that the request to write to the file succeeded and the third response is the new contents of the file at the time of the second read request. Once created, the cells in the response stream are immutable. In this way, referential transparency is preserved.

A predefined algebraic data type is used to denote the different kinds of request which are supported. This type is available as part of the standard environment in which programs are constructed. The definition of the `Response` and `Request` types in Staple (which are a subset of those in Haskell) is given in Appendix I.

## 3.3 Dialogues

A function type `[Response] -> [Request]` is called a *dialogue*. Such functions are treated specially by the Staple programming system. To evaluate an expression which has the type of a dialogue function, the Staple system transforms the expression into an application of the dialogue function to a system generated list of responses. The list of requests produced by the transformed expression is evaluated with each element of the list being evaluated and processed. Each element is a request for some action to be taken by the system. A response is constructed for each request which either indicates success, returns some requested value or fails with some error message. Data dependency between requests and responses is used to synchronise the I/O.

## 3.4 Streams and Persistence

Token streams provide a way in which functional programs can interact with I/O devices such as the keyboard and with the file system. They can also be used to provide access to values in persistent storage (Figure 3.4) and to make new persistent values.

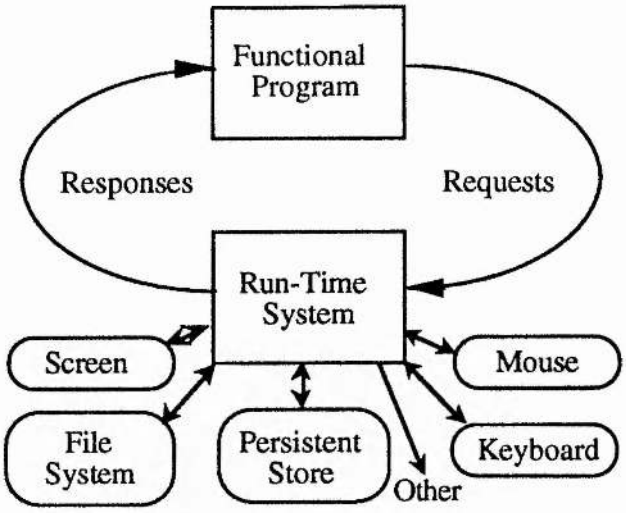



**Figure 3.4 - Stream Access to Persistent Values**

To do this, additional requests are supported which allow a program to manipulate persistent values. A request to look up a persistent value carries with it a method of identifying the particular value. In the Staple system, this is simply a string which represents the name of the persistent value. The persistent store accessible by streams supports an associative look up between strings and values. Figure 3.5 shows how the definition of the `Request` datatype is extended with three new requests for look up, insertion and deletion of associations.

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

**Figure 3.5 - Extension to the Request datatype**

The first parameter of each of the requests is a string (the name of a persistent value). The second parameter of the `Insert` request is a value of the built in type `Any` (the value being bound). Values of type `Any` can contain any denotable value as discussed in Section 3.5.

The run-time system performs a `Lookup` request by finding the value associated with the supplied name. The response which is returned will either be some indication of failure or the persistent value which was found. The `Response` data type is extended with a response which contains the persistent value (Figure 3.6).

```
data Response = ... | Value Any | ...
```

**Figure 3.6 - Extension to the Response datatype**

The other two new requests `Insert` and `Delete` will respond with either `Success` or `Failure`. The `Insert` request causes an association to be made between the name and the value. This association may replace an already existing one. Any future `Lookup` requests will see the latest value associated with the name. The `Delete` request simply removes the association between the name and its value.

## 3.5 Type Checking

In Staple, a dynamic type is used to represent all persistent data values. This allows the persistent store to evolve without the need to specify its contents completely when it is used. This flexibility is particularly important for programs which are prepared independently of the data they manipulate, since the contents of the store are not known when the program is compiled. A value which has a dynamic type may be any denotable value with an arbitrarily complex type. It is only when the value is needed for evaluation that the type need be specified. Since the type of the value is unknown statically, to ensure type safety, a dynamic type check must be performed.

A value of type `Any` is created by using an injection function `mkany`. For example

```
mkany 3
```

denotes a value of type `Any`. Any denotable value can be injected in to the dynamic type. To use such a value, the projection function `coerce` is used. The `coerce`

function requires the value to be projected and the type onto which to project must be specified. For example

```
x = mkany 3
y = coerce x :: Int
```

creates a value, `x`, of type `Any` and a value, `y`, of type `Int`. When the value of `y` is used, a run time type check will be performed to ensure that `x` is indeed an `Int`.

The type checking performed by the Staple system is partly static and partly dynamic. Where typechecking is performed statically types are compared for compatibility. The typechecker ensures that the type of some expression is a suitable one in that context. Where typechecking is performed dynamically, namely when projecting out of the dynamic type, Staple uses a different type checking rule. Two types must be equal and type checking involves a type equivalence check rather than a type compatibility check. The following sections describe the type algebra, static type checking and dynamic type equivalence in Staple.

### 3.5.1 Staple Type Algebra

In Staple, a type is denoted by the name of a type forming operator which may be applied to a number of argument types. The simplest examples of type expressions are those for the base types and consist of a type forming operator with no arguments. They are `Int`, `Real`, `Bool` and `Char`. These type forming operators are built in as are the those for lists and tuples.

The type of a list whose elements are of type `Int`, for example, is denoted by

```
[ Int ]
```

and the type of triples whose first element is an integer, whose second element is a list of characters and whose third element is a Boolean value is denoted by

```
( Int , [Char] , Bool )
```

These types can be thought of as being syntactic sugar for the application of a type forming operator to some type arguments. For example, the syntax might have been defined in such a way that the following type expressions denote the same types as above.

```
List Int
```

```
Triple Int (List Char) Bool
```

for integer lists and the triples respectively.

A further type forming operator is used for function types. The type of a function from integers to reals is denoted by

```
Int -> Real
```

The type of polymorphic functions is denoted using type variables. For example,

```
t -> t
```

denotes the type of a polymorphic function whose result type is the same as its argument type. Implicit in this type denotation is the universal quantification of the type variable t. The type should be read as

```
for all t, the function from t to t
```

New type forming operators can be defined by the programmer. These denote algebraic data types which are defined by enumerating the constructor functions for the type. For example

```
data Tree = Tip | Node Tree Tree
```

defines a new type algebraic type. The type expression `Tree` is used to denote values of this type. Type forming operators can be parameterised as follows:

```
data Tree t = Tip | Node (Tree t) t (Tree t)
```

The type expression

```
Tree Int
```

denotes the type of trees whose nodes contain integers. The right hand side of an algebraic data type definition enumerates the constructors for the type. Thus

```
Node Tip 3 (Node Tip 4 Tip)
```

denotes a value of type `Tree Int`.

Staple allows the definition of type synonyms. These are statically evaluable functions from types to types. For example,

```
type List t = [ t ]
```

defines a type synonym. The type expression `List t` can be used wherever `[ t ]` can and they denote the same type. Synonym definitions, in Staple, are not exported from a module. They are only available for use within a module.

### 3.5.2 Static Type Checking

Static type checking in Staple uses Hindley-Milner [Miln78] type inference to compute a most general type for each expression and for ensuring that the type of operands are compatible with the functions applied to them. That is, given some application

```
f a
```

where the type inferred for `a` is `t` and the type inferred for `f` is `(u->v)` then `u` and `t` must be unifiable.

Since type forming operators may be introduced in multiple modules which themselves may be recompiled over time, they are distinguished from one another by a unique name consisting of three parts.

- The algebraic data type name as used in the definition
- The name of the module in which the algebraic type was defined
- The time at which this instance of the module was created

So for example, the algebraic type

```
data Tree = Tip | Branch Tree Int Tree
```

if defined in a module called tree and instantiated on 23-Nov-1992 at 11:23.27am, the unique reference for this type forming operator is

```
(Tree, tree, 23-Nov-1992 at 11:23.27am)
```

The use of this name is hidden from the programmer, but is part of the symbol table information stored with a type definition in a module. Two algebraic type forming operators are compatible if they derive from the same definition.

### 3.5.3 Dynamic Type Equivalence

A dynamic type check occurs when a value of type `Any` is projected onto some other type. The system designer has a choice of methods for determining the expected type for such a dynamic type check.

- The expected type can be inferred
- The type can be stated explicitly by the user

In either case, the expected type must be checked for compatibility with the actual type of the value.

The type which is inferred for some value by the static type inference algorithm will be referred to as the creation type. If such a value is subsequently injected into type `Any`, it is this type which discriminates the value.

The type which the value has when projected out of the union type will be referred to as the view type.

A dynamic type check, then, involves a check that the creation type is compatible with the view type. There are a number of choices which can be made when deciding on which rule to use to check the compatibility of the two types.

The most general rule would be that the view type is an instance of the creation type. One type, S, is an instance of another, T, if there is a substitution for the type variables in T which results in the type S. This requires a relatively expensive instance check operation at run-time. This option is found in CAML [Lero91]. An alternative to this is to require that the view type is the same as the creation type which can be performed more quickly at run-time. Staple uses this approach. Napier88 [Morr89] uses a similar approach, but also supports abstract data types.

The latter choice requires some definition of type equality in the context of projection from the union type. From the definition of the type algebra above, it can be seen that all types in Staple are the application of some type forming operator to a number of argument types or a variable ranging over types.

Type variables are implicitly quantified at the outermost level in any type denotation. In addition, $\alpha$-conversions are performed so that the order in which type variables first occur is alphabetically from the left. Thus the type

```
t -> [u] -> [s] -> [(u,s,t)]
```

gets transformed to

```
a -> [b] -> [c] -> [(b,c,a)]
```

Types can be represented using the algebraic data type

```
data Type = TypeCons [Char] [Type] | TypeVar [Char]
```

For example

```
TypeCons "->" [TypeCons "Int" [],TypeCons "Int" []]
```

represents the type

```
Int -> Int
```

An equality rule over such types can be defined as follows.

```
eq (TypeCons c ts) (TypeCons d us) = (c = d) and eql ts us
eq (TypeVar c) (TypeVar d) = (c = d)

eql [] [] = True
eql (t:ts) (u:us) = eq t u and eql ts us
eql ts us = False
```

A consequence of this choice of equality is that it is not possible to project polymorphic types directly onto less general instance types. So for example

```
id x = x

anyId = mkany id

intId = coerce anyId :: Int -> Int
```

will fail with an type error at run-time. It is however possible to do this in a two stage process by first projecting onto the exact type and then restricting this with a compile time type assertion. To achieve the desired result for the above example, the programmer must write the definition of `intId` as

```
intId :: Int -> Int
intId = coerce anyId :: t -> t
```

For pragmatic reasons, Staple has adopted the restriction proposed in [Abad89] which only allows values with closed types to be injected into type `Any`.

## 3.6 An Extended Example

To demonstrate how Staple stream persistence can be used this section describes an implementation of a simple relation of employee records.

The first consideration is to define a model of an employee.

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

This defines an algebraic data type used to represent records in the database. The fields are the name, age, national insurance number and salary of the employee. The database will be modelled as a list of employee records. An initial empty database can be defined as follows.

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

This empty database can be associated with a name in persistent storage using the following function.

```
initdb :: [Char] -> [Response] -> [Request]
initdb dbname resps = [Insert dbname (mkany emptydb),
                       AppendChan "stdout" (msg (resps!0))]
                      where
                      msg Success = "OK'n"
                      msg (Failure m) = "Failed'n"
```

`AppendChan` is a request which prompts the output to the output stream identified by its first parameter of its second string parameter. The function can be executed by typing the dialogue expression

```
initdb "payroll"?
```

to the staple interactive session prompt. The system can tell that this is a dialogue function because of its type. As such it is treated specially by the system and applied to a list of responses. The first response generated indicates the result of

performing the `Insert` request. This response will be an indication of success or failure. The second request prints out a message conferring the result status of the previous request.

A function which will insert a new tuple into the database is defined below.

```
addtuple dbname 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 (Value val)   = val
            getval (Failure msg) = error msg
```

The `addtuple` function defined above produces a dialogue if it is applied to five arguments. This dialogue results in two requests - one to lookup the database in the persistent store and one to insert the new updated database back into the persistent store. The subsidiary definitions extract the old database from the dynamic value 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 dbname resps
         = [Lookup dbname, AppendChan "stdout" (showdb db)]
           where
           db                   = coerce stored :: [Tuple]
           stored               = getval (resps!0)
           getval (Value val)   = val
           getval (Failure mess) = error mess
```

Here, `showdb` produces a textual representation of its database argument. A function to delete a tuple from the database can be specified similarly to the `addtuple` function above.

```
deltuple dbname n resps
        = [Lookup dbname, Insert dbname newdb]
          where
          newdb                  = mkany (remtuple name db)
          db                     = coerce stored :: [Tuple]
          stored                 = getval (resps!0)
          getval (Value 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. It is possible to abstract over these to get a general action function incorporating all changes to the database.

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

The add tuple operation can be redefined 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 operations that might be required. Further examples are given in Chapter 6.

## 3.8 Conclusions

Dynamic access to persistent data can be provided in a way which preserves referential transparency by extending a token stream based I/O model with requests to access persistent storage. Referential transparency is preserved because the response stream, through which the state of the external environment and in particular the state of the persistent store is observed, contains components which themselves do not change, but can be viewed as snapshots of the changing state over time. Once inserted into the response stream, the values are not mutable. An explicit request must be generated to examine any new changes in the state.

A dynamic type is used for persistent data values to allow for a flexibly evolving object store. A consequence of a dynamic type, however, is that not all type checking can be performed statically. A run-time type check must be performed to check that the type of a value retrieved from persistent storage is compatible with the expected type.

An example of the kind of application which can be constructed with stream persistence is an interactive database application in which no explicit code need be written to convert between programming language and long term representations of the data. Such a system is more difficult to construct in non-persistent functional programming systems because explicit conversion code must be written to transfer data between short and long term storage. With stream persistence, the transfer of data between main store and backing store is managed by the underlying persistent storage system.

# Chapter 4

# PCASE – An Abstract Machine for Persistent Lazy Functional Programming

The PCASE machine lies at the heart of the Staple persistent lazy functional programming system. It has been designed to execute lazy functional programs using a heap of persistent objects as its only run-time data space. This chapter presents the argument which led to the decision to design the PCASE machine. The architecture of the PCASE machine is then presented and an operational semantics is given for the machine instructions. The PCASE machine uses an already existing persistent object storage system [Brow90,Brow91]. The interface between the store and PCASE is described and the way the interface is used by PCASE machine instructions is exemplified.

## 4.1 Introduction

Why build a persistent abstract machine? In this section, the reasons why the decision to build a new abstract machine are discussed. The system architecture is first described to provide a framework for the discussion which follows.

A persistent programming system architecture consists of four components [Brow89]. These are

- The programming systems user environment
- The compilation system
- The target machine
- The persistent object store

The corresponding components in the Staple system are shown diagrammatically in Figure 4.1. At the heart of the architecture is the PCASE abstract machine. Together with the persistent object store, the PCASE machine provides the evaluation engine for Staple.

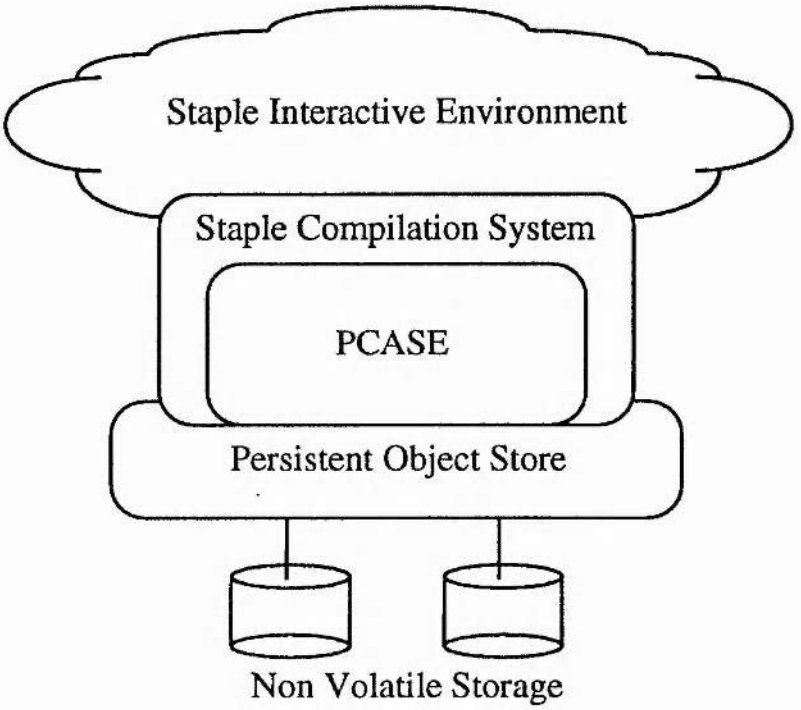



**Figure 4.1 - The Staple Architecture**

The Staple interactive environment has been presented in Chapters 2 and 3. It provides a mechanism whereby users may instantiate modules or evaluate expressions interactively. The compilation system takes source code written in the Staple language and produces PCASE machine code as its result.

PCASE is a persistent abstract machine which

- is capable of evaluating persistent lazy functional programs in an efficient manner
- supports creation, dereference and updating of lazy data values whose persistence may outlast the execution time of the program or interactive

session. Such data values may be in various states of evaluation and can be of any type

- provides automatic garbage collection of a heap which may exist partly in permanent storage and partly in volatile storage

- ensures that failure (software failure and certain kinds of hardware failure known as hard failures e.g. the cpu shuts down) does not destroy the persistent data. The abstract machine should provide a method of checkpointing during execution. In the event of system failure, the machine can be restarted from the last checkpoint.

These features are achieved by the integration of the PCASE abstract machine and a persistent object store.

The PCASE component of the architecture consists of an interpreter for the abstract machine and run-time support for resource management such as access to persistent storage.

### 4.1.1 How to Proceed

In deciding how to implement the above architecture, it would be possible to take the approach of building all the components from scratch. This is in general software engineering terms something to be avoided. A more pragmatic approach taken in this thesis was to reuse available technology. Two possibilities present themselves.

- Take an existing abstract machine and design a persistent object store tailored to the needs of that machine.

- Take an existing persistent object store and build an abstract machine which supports persistent lazy functional programming.

A reusable object store, postore, in the form of C libraries has been developed by researchers at St Andrews [Brow91]. This object store provides ten object management procedures described in section 4.4 These procedures provide the ability to create and manipulate persistent objects easily and manage the use of such objects transparently. Constructing such a system from scratch is non-trivial [Brow89].

Since in a persistent system, all data can persist, all data must therefore be created in persistent storage and manipulated within it. It is essential that all persistent data be in the persistent heap at the end of a program's execution. Functional languages utilise heap representations heavily. However, abstract machines for lazy functional programming typically store their code outwith the run-time heap and code addresses are indexes into a single large code vector. In a persistent language, the code for functions and suspensions may be created at different times and by different programs which have been prepared independently. Since these values may persist, so must their code. For this reason, code must reside in persistent storage.

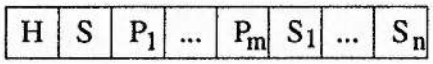
Because the postore library was readily available, it was decided to take the postore library and build an interpreter for an abstract machine suitable for executing persistent lazy functional programs. This is the PCASE machine.

The PCASE machine is an extension to the CASE machine [Davi89b] – an abstract machine designed for the efficient evaluation of functional programs on sequential hardware. By constructing and accessing all run-time objects using the persistent store object management procedures, the abstract machine need not deal with issues such as movement of data between volatile and non-volatile storage, garbage collection and stability. The postore library will be referred to as the stable store.

### 4.1.2 Persistent Object Formats

Implicit in the postore interface used by the PCASE machine is the persistent object format.

Persistent objects contain a number of fields some of which are pointers to other objects, and some of which are scalar values such as integers, machine instructions etc. All objects begin with a header field, H, which indicates the number of pointers in the object and some house keeping information and a size field, S, which contains the total size of the object in words. The general stable store object format is shown diagrammatically as follows:




where $P_i$ are the pointer fields and $S_i$ are the scalar or non-pointer fields.

Staple run-time objects conform to this format. In addition, an extra tag field is used to indicate that the object is evaluated or that it is a suspension. This field is, by convention, the last non-pointer field of the object. A system type indicates which of the object formats described in the following sections is associated with this persistent object. Ideally a single bit should have been used in the header field of the object to indicate that it is suspended, but these bits are reserved for use by the stable store itself – in particular for garbage collection.

## 4.2 A Persistent Abstract Machine

The PCASE persistent abstract machine uses the stable heap as its only run-time data space. The architecture of the machine is influenced by the constraints of the postore library. In particular, for garbage collection, all objects must be reachable from the root of persistence. Consequently, the stacks, code vector and environment are heap objects and the registers can be stored in the heap prior to garbage collection and restored afterwards.

The state of the PCASE machine is represented by seven registers pointing at various storage structures (Figure 4.2). There are two stacks - a main stack for pointer values and a value stack for literal values. The value stack is used for the evaluation of strict arithmetic expressions and for storing return addresses. The stack pointers MS and VS point into the two stacks used for holding working results and for storing previous states. The MS stack holds pointer values whilst the VS stack holds scalar literals during expression evaluation. In Landin's original SECD machine, these functions were performed by the S and D registers.

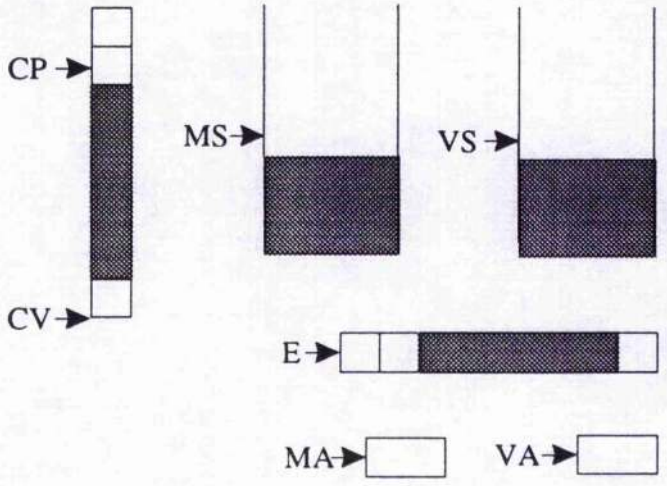



**Figure 4.2 - The PCASE Abstract Machine Architecture**

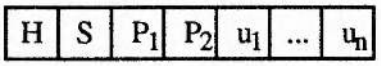
The PCASE machine has an environment pointer E which refers to a flat structure containing values of variables currently needed in the execution. These variables consist of the non-local variables.

There are two argument registers MA (for pointer arguments) and VA (for scalar literal arguments), one of which contains the single argument for the function being executed. Normally this will be in MA unless the compiler can detect that the function can be called by value and the argument is of a non-pointer type in which case the argument is passed in VA. Finally, a code pointer CP points to the currently executing instruction in the code vector pointed to by CV.

The state of the PCASE machine can be represented as a seven-tuple <CV,CP,MA,VA,MS,VS,E>. The PCASE machine uses reverse polish code for operator expression evaluation. Unlike non-persistent machines, PCASE uses stacks, environment and code structures which all reside in persistent storage. The format for an environment object is described in Section 4.4.1.2.

### 4.2.1 Stacks

During garbage collection, space allocated to objects which are not reachable from the root of persistence is collected and made available for reuse. PCASE run-time objects which are on the MS stack must be retained during a garbage collection. This is done by making the MS stack a heap object itself. The MS stack is represented as follows

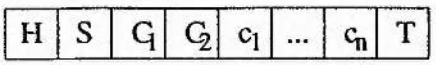



where the $P_i$ are pointers which are active on the stack and the $u_i$ are unused. Each time a pointer is pushed onto the MS stack, it is inserted into the MS object and the number of pointers is adjusted.

The VS stack is also stored in the heap. The reason for this is that should a system failure occur, the machine can be restarted from the state it was in at the last checkpoint.

### 4.2.2 Code Vectors

In the PCASE machine, code is stored in a heap object called a code vector. A code vector object contains executable machine code (PCASE machine code is a byte code) and pointers to other code vectors which are used during the execution of this code vector to construct new closures or suspensions. For example, a code vector object with 2 sub code vector fields and n words of executable code would be represented by




Here $C_1$ and $C_2$ are pointers to the sub code vectors which may be used during the execution of the function body to create other functions or suspensions. The executable code itself is stored in the non-pointer fields $c_1$ ... $c_n$. Each of the $c_i$ are one word in size and contain four bytes of code.

In the PCASE machine, the code vector register CV points to the code vector in which the currently executing instruction is located. The code pointer CP is an offset into the code vector pointed to by CV.

All jumps in the PCASE machine are relative and jump to locations within a single code vector's code. Application and suspension evaluation are achieved by switching code vectors.

When a closure or suspension is constructed, the values which will constitute the non-local environment are loaded onto the stack and an instruction to construct the closure or suspension is executed. This instruction indicates the number of values on the stack to build the environment, and also the offset in the current code vector to the appropriate code vector for the closure or suspension.

### 4.2.3 The Environment

The reasoning behind the structuring of the environment in two parts, namely the single argument and the environment register, is based on a paper by Davie [Davi79] and is as follows. In a conventional stack architecture, objects in an environment are accessed via a *display* and/or a *static chain* [Rand64]. This has the consequence that non-global free variables have an address consisting of two parts, a block level (or, more commonly, a level difference) and an offset. This has the disadvantage of requiring a double dereference to access non-local objects. There is also a high overhead in setting up the chain structures and/or display at function entry and resetting them at exit time.

In the PCASE machine, when a function is created — i.e. when a closure for a lambda expression is formed — that closure is constructed by loading up the values of all its free variables; these represent the objects which it may address non-locally. The construction of the closure takes place only once, and all invocations of the function (there may be several of these and they might be executing simultaneously if several processes could run in parallel on suitable hardware) may use this same closure without any copying. Each separate invocation of the function however will bind its own argument by placing it in the argument register MA.

With this method, the operation of lambda-lifting [Hugh84,John85,John87] is replaced by an operation carried out *at run time* but without extra penalty. The lambda lifting method involves transforming functions to others which are combinators — i.e. those having no free variables. This is achieved by adding extra arguments (by source transformations) through which the values of non-local identifiers are passed. There is therefore a corresponding run time penalty of binding these extra arguments at each recursive invocation when the lambda-lifting method is used. Indeed, if there are n non-locals in the body of the function the penalty is n times the cost of loading an argument into the environment, and this cost is incurred at each function application. The cost of creating the environment for a function in the PCASE machine, however, is n times the cost of loading an argument into the environment, but it is performed only once. Subsequent applications of the function need only load the argument into the argument register.

Consider, as an example, the well-known `map` function,

```
map f []     = []
map f (h:t)  = f h : map f t
```

Here, `[]` denotes the empty list – as a pattern on the left hand side of the equation, and as an expression on the right hand side. List construction is denoted by the colon operator which similarly appears as a pattern and an expression in the second

equation. The function `map` takes two arguments, a function and a list of elements, and returns a list whose elements are obtained by applying the function to the corresponding element of the original list.

Since `f` does not change in the recursive call of `map`, the naïve user might expect that a more efficient solution would be obtained by factoring out `f` to obtain

```
map f = map'
        where map' []     = []
              map' (h:t) = f h : map' f t
```

The `where` clause introduces local definitions. The scope of `map'` is just the expression on the right hand side of the equation for map.

Unfortunately, in the lambda-lifting case, the compiler would notice that `f` was a non-local of `map'` and reintroduce it by program transformation obtaining exactly the first definition for `map`, above.

In this lambda-lifted case, each time the function `map` is recursively applied, an environment is created containing its arguments by loading them onto the stack. There will be one such environment formed for each element of the list to which `map` is applied.

In the PCASE implementation, however, a true gain in efficiency is obtained (and moreover, the optimisation shown above may be made automatically). When map is applied to `f`, a closure for `map'` is calculated (once only) and the environment so formed is bound (together with the code) to the name `map'`. Each invocation of `map'` has access to this new environment and does not need to allocate any further store.

Quantitatively the situation is as follows. In the lambda-lifted case, for a list of length n, 3n locations (for `f`, `h` and `t` at each recursive level) have to be allocated.

In the optimised CASE version, 2n+1 locations are required. One for the storage of `f` in the closure for `map'` and 2 at each recursive level for `h` and `t`.

## 4.3 Operational Semantics

The operational semantics of the PCASE machine are described in the following Sections. The machine instructions are given together with the transformation of the machine state <CV,CP,VA,MA,VS,MS,E>. The code and stacks are denoted using list notation. Thus A:S denotes a stack whose top element is A, the rest being S. Similarly, the code is represented as a list whose head element is the next instruction to be executed rather than an offset into the code vector. The environment is represented using list notation even though it is a contiguous structure which can be accessed by an indexing operation.

In the diagrammatic state transitions, the header fields for the stacks and code vector have been omitted for clarity.

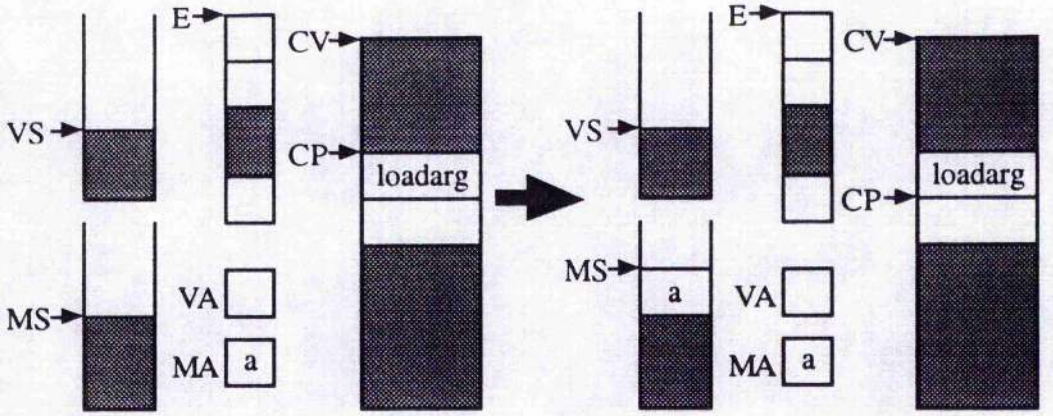
### 4.3.1 Stack Instructions

A number of instructions are available to load objects onto the stacks and to manipulate the stack in a conventional Polish postfix manner.

#### 4.3.1.1 Loading an argument register

<CV,LOADARG:CP,VA,MA,VS,MS,E> → <CV,CP,VA,MA,VS,MA:MS,E>

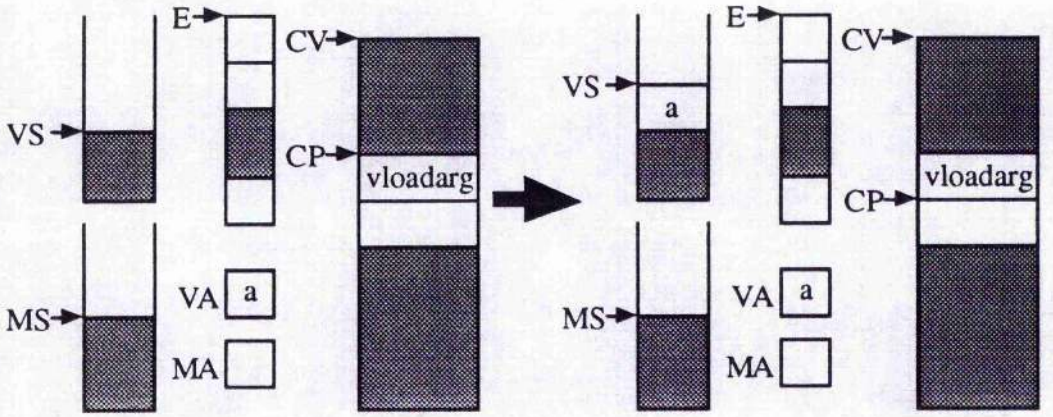
Loads the pointer argument onto the pointer stack.




and

<CV,VLOADARG:CP,VA,MA,VS,MS,E> → <CV,CP,VA,MA,VA:VS,MS,E>

Loads the value argument onto the value stack.



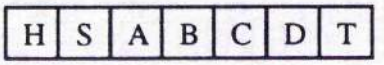

This instruction would be generated for loading the use of a function argument. For example, the function

```
id x = x
```

would cause a `loadarg` instruction to be generated for the reference to `x` in the body.

#### 4.3.1.2 Loading a free variable

Environments are pointed to by closures and suspension and provide the context in which their evaluation takes place. An environment object with 4 components, for example, will have the following format:

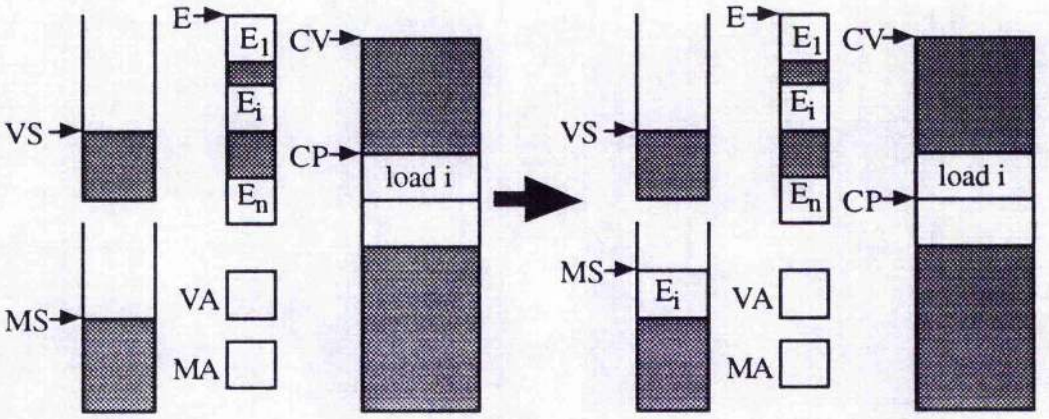



where A,B,C and D are pointers to other objects which are the graphs representing the values of non-local variables used by the closure or suspension and T indicates that this is an environment object. The machine instruction to load an object from an environment simply indexes this object which yields a constant access time for environment lookup [Davi79].

$$\langle CV,LOAD\ i:CP,VA,MA,VS,MS,E\rangle \rightarrow \langle CV,CP,VA,MA,VS,E_i:MS,E\rangle$$

$$\text{where } E = [E_1,\ldots,E_n]$$

Loads the i'th item in the environment onto the main stack.



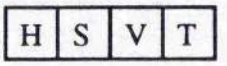

Non-local variables are accessed using this instruction. For example, the code

```
a+b
where
a = 3
b = 4
```

will generate a `load` instruction to load each of the variables `a` and `b` in the body of the where clause.

#### 4.3.1.3 Loading literals

Staple has four literal types: `Int`, `Char`, `Bool` and `Real`. The first three of these are represented by objects of the following kind:

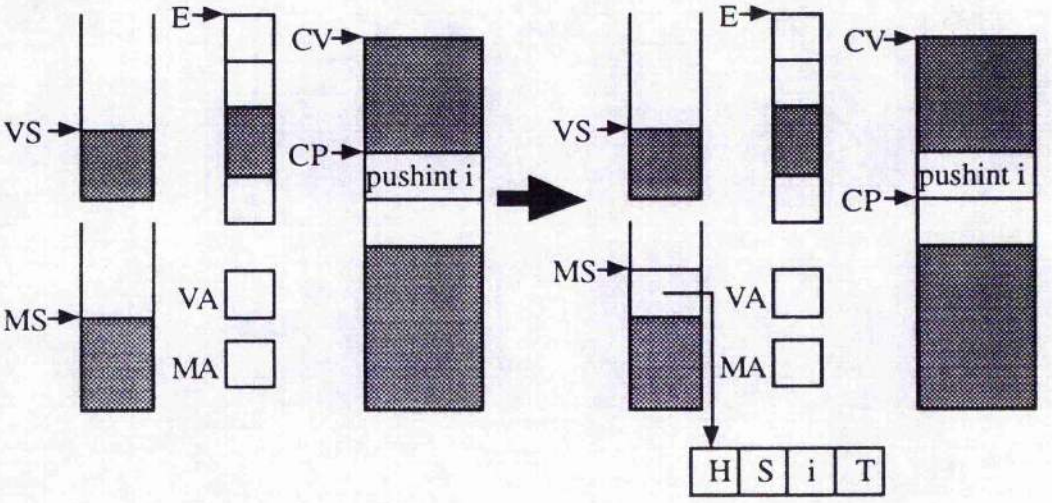



Where H and S are the header and size fields, V contains the value and T is the tag which indicates the system type of the object. Real literal objects are represented similarly but have two words which contain the value.

<CV,PUSHINT i:CP,VA,MA,VS,MS,E> → <CV,CP,VA,MA,VS,&i:MS,E>

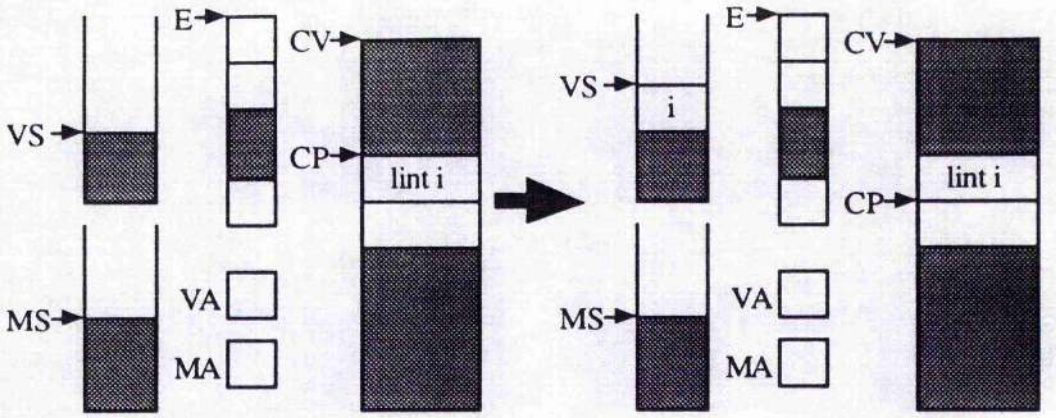
where &i is a pointer to a new heap cell containing i

Creates a new persistent heap object to represent an integer and fills in the value of the object. A pointer to the newly created object is placed on the main stack.




<CV,LINT i:CP,VA,MA,VS,MS,E> → <CV,CP,VA,MA,i:VS,MS,E>

Loads the integer literal i onto the value stack.



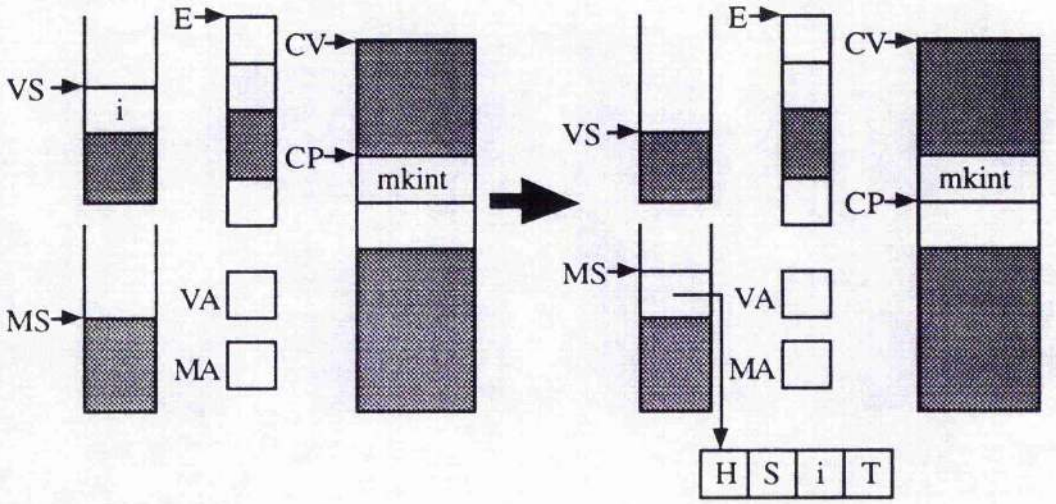

This instruction is used when the literal is used in evaluating an expression. For example, the code

```
3+a
```

would generate the instruction `lint 3` to load the literal 3 which would subsequently be added to the value of `a`.

<CV,MKINT:CP,VA,MA,i:VS,MS,E> → <CV,CP,VA,MA,VS,&i:MS,E>

where &i is a pointer to a new heap cell containing i

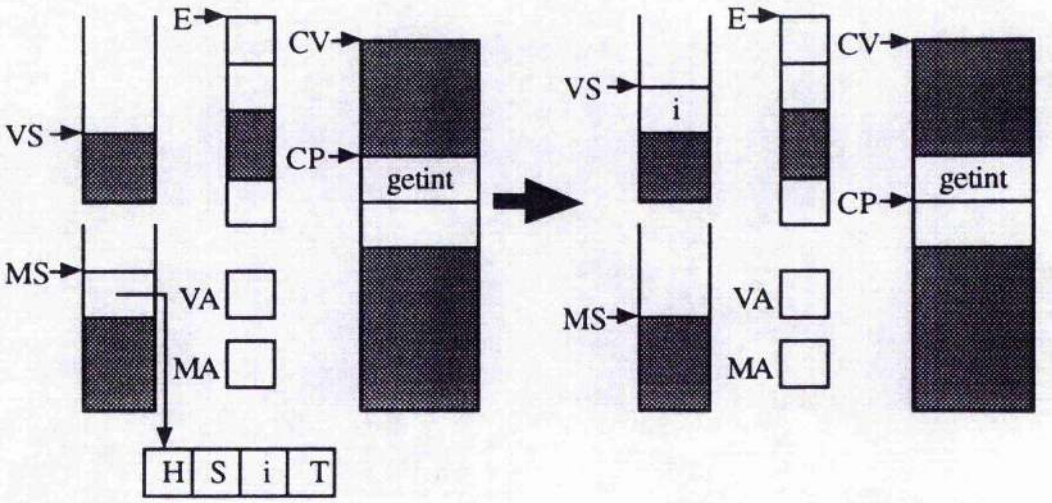



The heap cell is created in persistent storage.

<CV,GETINT:CP,VA,MA,VS,&i:MS,E> → <CV,CP,VA,MA,i:VS,MS,E>

where &i is a pointer to a new heap cell containing i

A pointer to an integer object is on top of the main stack. The object is dereferenced to obtain the integer value and this is loaded onto the value stack.

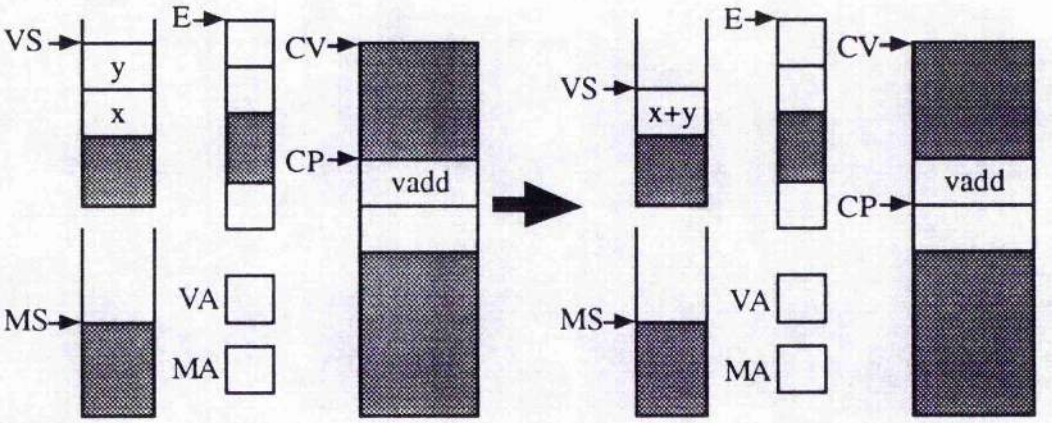



#### 4.3.1.4 Built in operators

Conventional unary and binary operators are available. For example

<CV,VADD:CP,VA,MA,y:x:VS,MS,E> → <CV,CP,VA,MA,(x+y):VS,MS,E>

adds the top two integer values on the value stack, pops both the arguments and puts the sum back onto the stack.




The code

```
a + 3
```

generates the following machine instructions

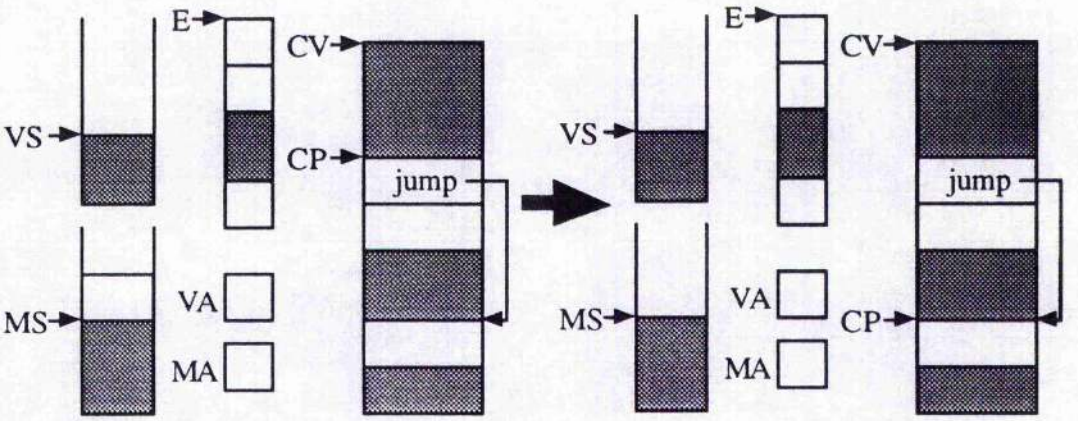
```
load 2          -- assuming a is at offset 2 in E
getint          -- extract the integer value
lint 3
vadd
mkint           -- assuming a boxed value is needed
```

Other instructions of this form are available including subtraction (VSUB), multiplication (VMUL, division (VDIV), remainder (VMOD), relational operators (VLT,VGT,VLE,VGE) and equality (VEQ,VNE).

#### 4.3.1.5 Conditionals and Jumps

<CV,JUMP cp:CP,VA,MA,VS,MS,E> → <CV,cp,VA,MA,VS,MS,E>

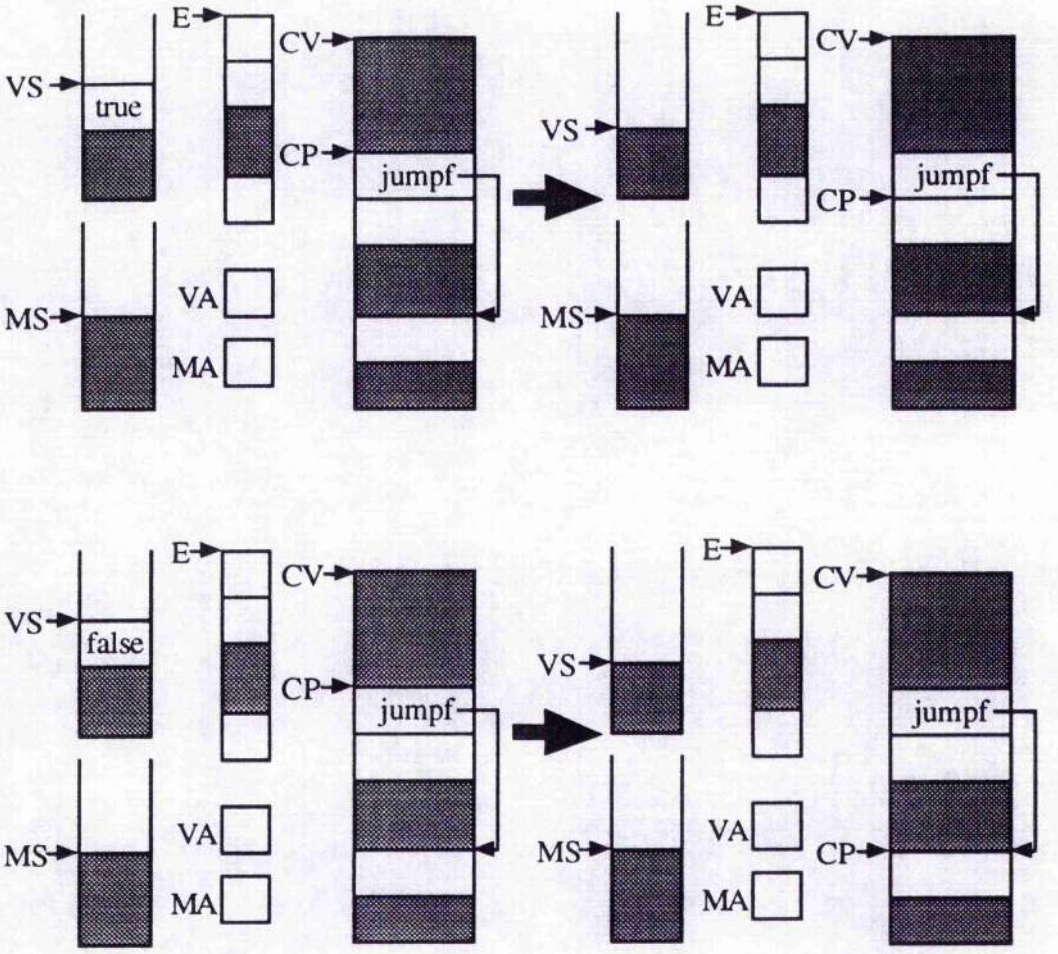
Performs an unconditional jump to a new instruction in the current code vector.




<CV,JUMPF cp:CP,VA,MA,true:VS,MS,E> → <CV,CP,VA,MA,VS,MS,E>

<CV,JUMPF cp:CP,VA,MA,false:VS,MS,E> → <CV,cp,VA,MA,VS,MS,E>

Pops the top item from the value stack and performs a jump if this value is false.




The program fragment

```
if a < 3 then a else b
```

would generate the code

```
        load 0        -- assumes a in location 0 in E
        getint
        lint 3
        vlt
        jumpf L1
        load 0        -- load a
        jump L2
L1:     load 1        -- load b
L2:
```

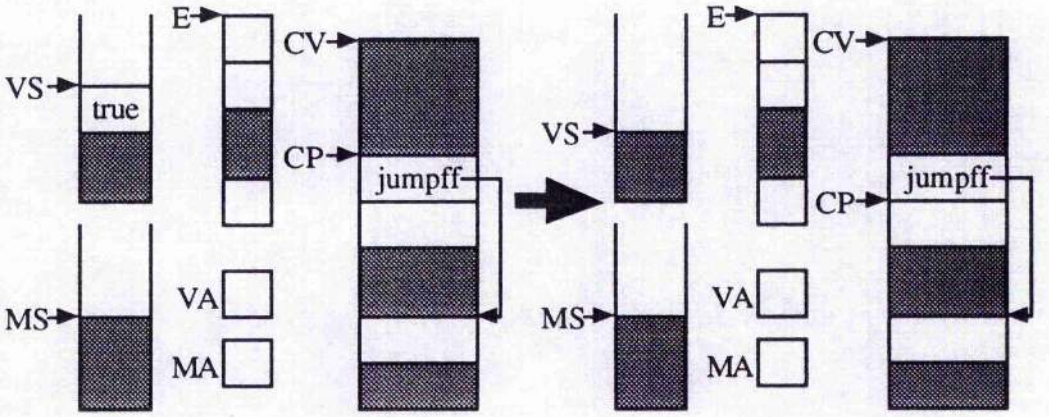
Logical conjunction and disjunction are also implemented using jump instructions. Conjunction is implemented using the JUMPFF instruction [Turn76,Morr79] which examines the top item on the value stack. If it is false, a jump is performed otherwise the top item is popped and the next instruction executed.

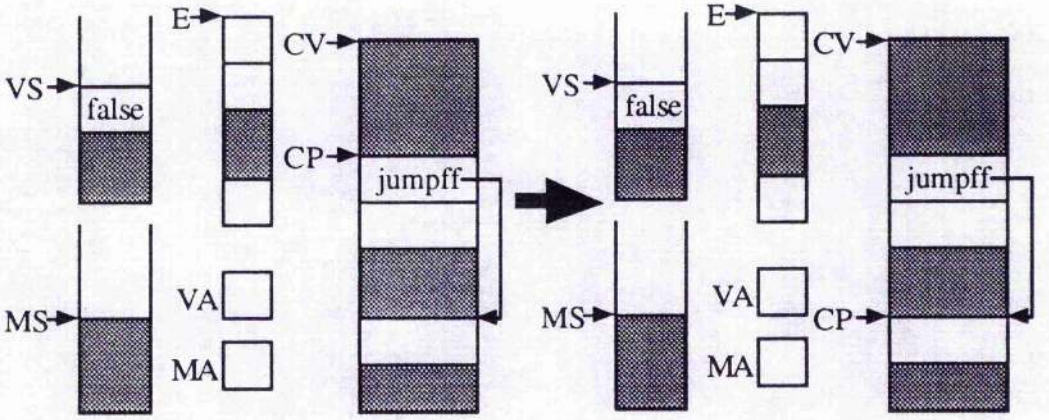
<CV,JUMPFF cp:CP,VA,MA,true:VS,MS,E> → <CV,CP,VA,MA,VS,MS,E>

<CV,JUMPFF cp:CP,VA,MA,false:VS,MS,E> →

<CV,cp,VA,MA,false:VS,MS,E>

The state transition diagrams for the JUMPFF instruction are shown below.




Similarly, disjunction uses the JUMPTT instruction.

<CV,JUMPTT cp:CP,VA,MA,true:VS,MS,E> →

<CV,cp,VA,MA,true:VS,MS,E>

<CV,JUMPTT cp:CP,VA,MA,false:VS,MS,E> → <CV,CP,VA,MA,VS,MS,E>

As an example, the program fragment

```
a and b
```

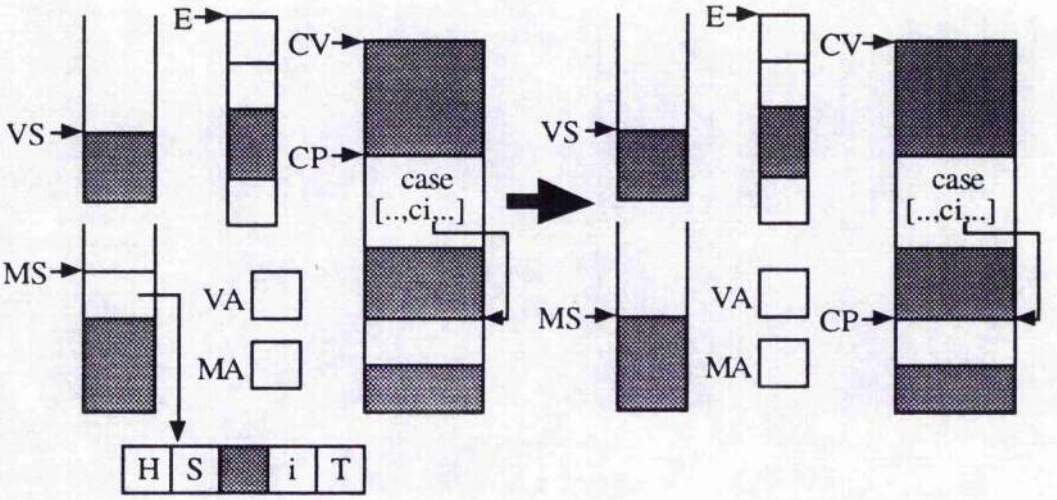
generates the code

```
        load 0      -- load a
        jumpff L1:  -- if its false dont evaluate b
        load 1      -- load b
L1:
```

One further jump style is provided. The CASE instruction is used for pattern matching. The top item on the main stack is a structured object with a variant tag. The tag is examined and a jump made to the appropriate location. If the value of the tag is i, then the location is determined by taking the $i^{th}$ element of the vector of code addresses which is part of the CASE instruction.

<CV,CASE [C1,..,Cn]:CP,VA,MA,VS,structure(i,fields):MS,E>

→ <CV,Ci,VA,MA,VS,MS,E>




The code for the following function

```
isnull [] = true
isnull xs = false
```

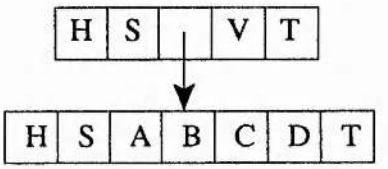
assuming the tag for the null list is 0 and the tag for a cons cell is 1 would be

```
        loadarg
        case [L1,L2]
L1:     pushbool true
        jump L3
L2:     pushbool false
L3:     return
```

#### 4.3.1.6 Data Structure Creation and Dereferencing

Staple provides tuples as standard data structures and allows the user to define algebraic data types. The system also provides lists as a built-in algebraic type.

In order to facilitate ease of overwriting suspension objects with objects of a different kind, compound data structures have been implemented as two level objects. Structured objects contain a single pointer field pointing to an environment object which contains the fields. In objects of algebraic type, a variant tag is stored as a scalar field in addition to the usual system type tag. For instance, the format of a structure with four fields A, B, C and D is as follows.

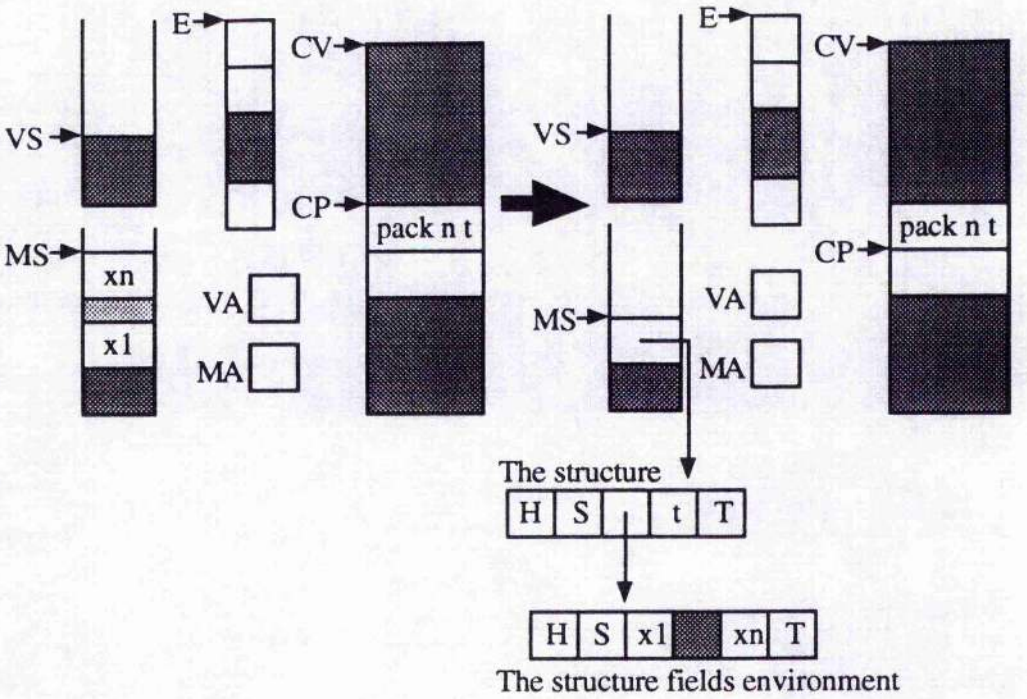



Where V is the variant tag. The variant tag is used in pattern matching to discriminate between cases. It is different to the tag T which is the machine tag used to determine if an object is a suspension.

$$\langle CV,PACK\ n\ t{:}CP,VA,MA,VS,x_n{:}\ldots{:}x_1{:}MS,E\rangle \rightarrow$$

$$\langle CV,CP,VA,MA,VS,\&data{:}MS,E\rangle$$

where &data points to structure($[x_1,\ldots,x_n]$,t)




The list expression

```
[1,2]
```

generates the code

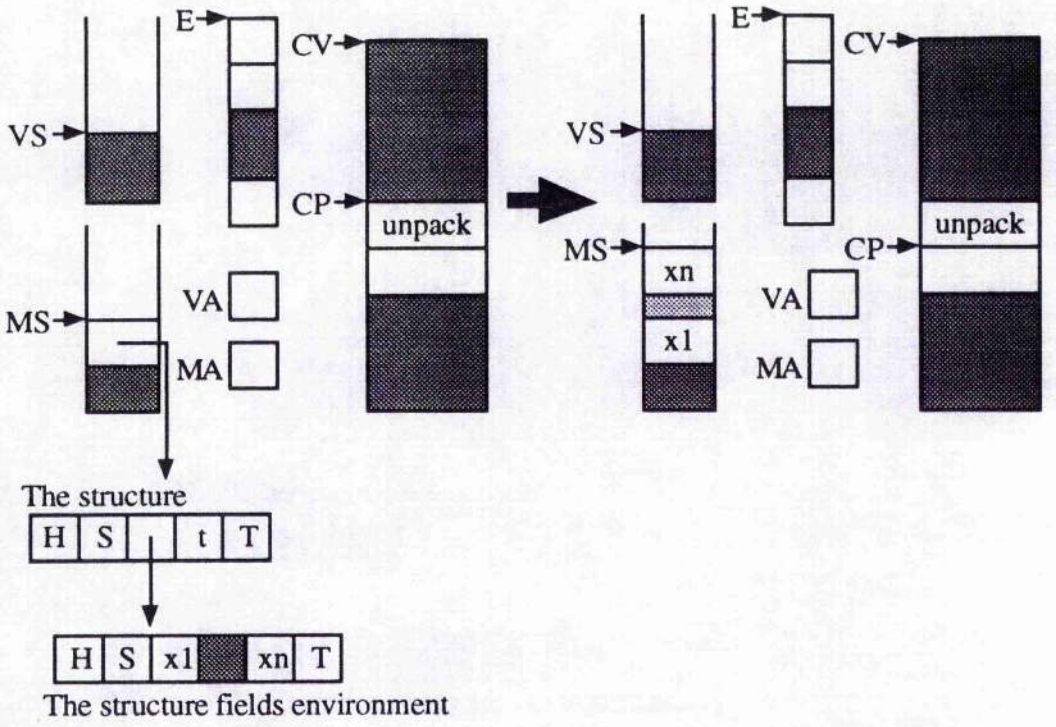
```
pack 0 0     -- the empty list
pushint 2
pack 2 1     -- a cons cell creates the list [2]
pushint 1
pack 2 1     -- another cons cell - creates [1,2]
```

Dereferencing structure values is done by pattern matching. A case instruction is used to determine which right hand side to use. The fields of the structure are given names in a block whose body is the right hand side of the function.

<CV,UNPACK:CP,VA,MA,VS,&data:MS,E> →

<CV,CP,VA,MA,VS,$x_n$:...:$x_1$:MS,E>

where &data points to structure([$x_1$,...,$x_n$],t)




The function definition

```
hd (x:xs) = x
```

produces the code

```
        loadarg
        case [L1,L2]
L1:     error        -- the case hd [] is not defined
        jump L3
L2:     unpack
        block 2      -- see below (creates an environment
        load 0       -- containing x and xs)
        exit         -- finish this simple block (see below)
L3:     return
```

#### 4.3.1.7 Function creation, entry and exit

Function values are represented as closure objects. A closure consists of code for the body of the function, and an environment in which the function is to be executed. The object format is as follows.

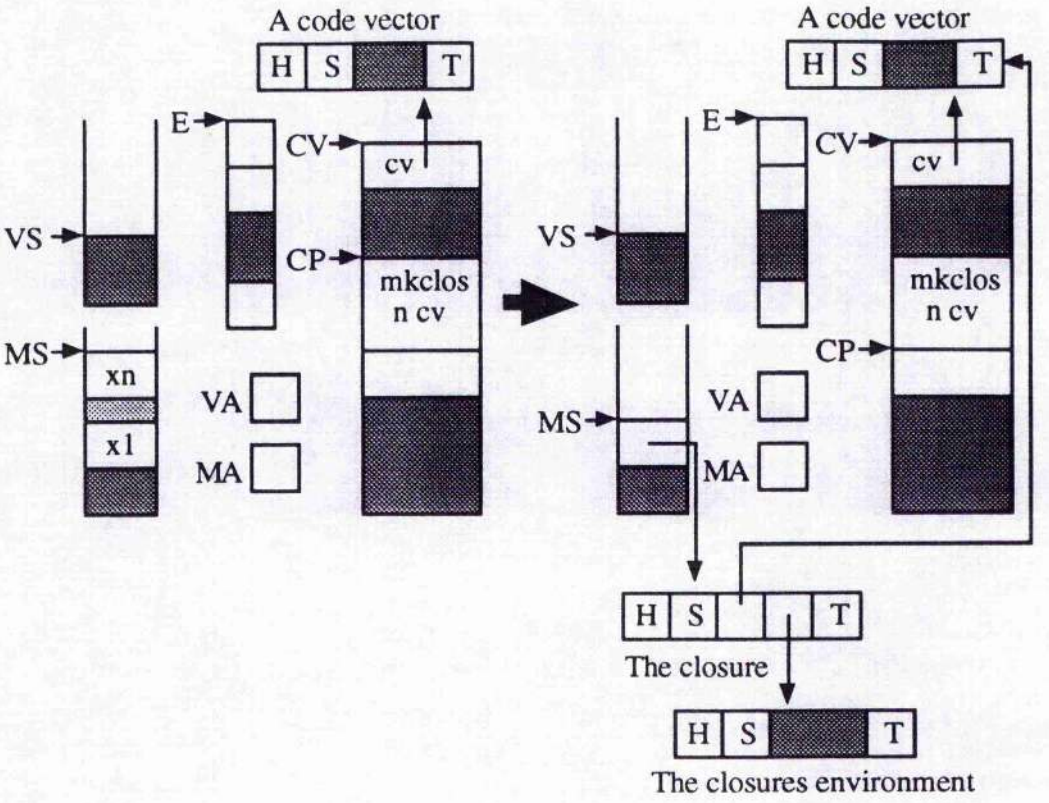

Here C is a pointer to a code vector object and E is a pointer to an environment object. When a function is applied to an object, a new environment is created consisting of the environment from the closure, and the argument of the function which is placed in the argument register. The code from the function is then executed in this environment. The tag indicates the system type (closure) of the object. It does not give any further information about the user type of the object (e.g. its domain or range type) which is factored out at compile time.

A closure is created using the MKCLOS instruction. The top n items on the main stack are used to form the environment for the closure. The curent code vector contains pointers to other code vectors which are referenced from this one. One of these is used as the code for the closure.

$$\langle CV,MKCLOS\ n\ cv{:}CP,VA,MA,VS,x_n{:}\ldots{:}x_1{:}MS,E\rangle \rightarrow$$

$$\langle CV,CP,VA,MA,VS,\&cl{:}MS,E\rangle$$

where &cl points to closure(cv,$[x_1,\ldots,x_n]$)




The program fragment

```
a = 3
f x = x + a
```

will generate the following code to create the function value for f

```
        load 0        -- load the free variable a
        mkclos 1 0    -- assume sub code vector at location 0

C0:     loadarg       -- the sub code vector's code
        getint
        load 0        -- a is at locn 0 in the closures env
        getint
        vadd
        mkint
        return
```

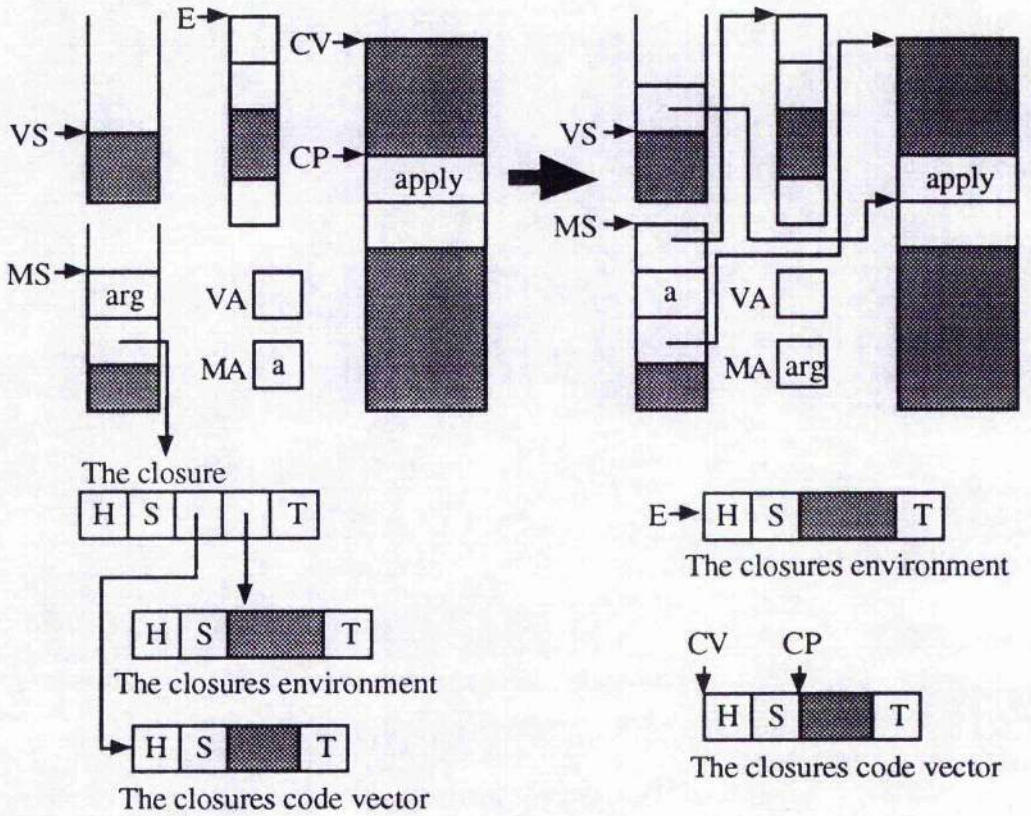
A function call is made using the APPLY instruction. Before this instruction is executed, the function's closure must first be evaluated (see the STRICT and OVERWRITE instructions below).

<CV,APPLY:CP,VA,MA,VS,arg:&cl:MS,E> →

<cv,cp,VA,arg,CP:VS,E:MA:CV:MS,e>

where &cl: points to closure(cv,e) and cp points to the 1st instruction in cv

The main stack initially contains a closure and an argument. The closure contains a code vector for the function and an environment in which to execute that code. The actual parameter is placed in the main argument register and the environment pointer is set to point to the closures environment. The code pointed to by the closure is then executed. The old code, environment and argument are saved on top of the main stack. The old code pointer is saved on the value stack.




This instruction is generated for any application of a function to an argument

```
f x
```

generates

```
load 0        -- assume f is at locn 0 in the env
load 1        -- assume x is at locn 1
apply
```

A similar instruction VAPPLY is used for scalar call-by-value applications.

<CV,VAPPLY:CP,VA,MA,varg:VS,&cl:MS,E> →

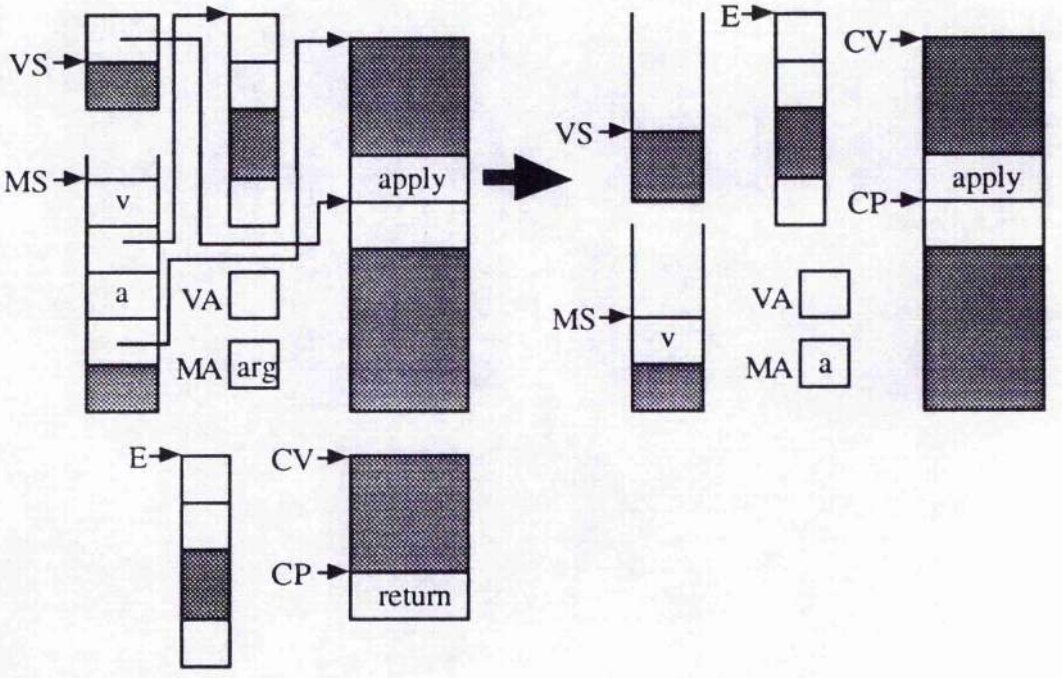
<cv,cp,varg,MA,VA:CP:VS,E:CV:MS,e>

where &cl: points to closure(cv,e) and cp points to the 1st instruction in cv

Function exit is performed by the RETURN instruction. The result of the function application is on top of the main stack together with the saved environment, argument register and code vector (and return address).

<CV,RETURN,VA,MA,cp:VS,v:env:arg:cv:MS,E> →

<cv,cp,VA,arg,VS,v:MS,env>




Call by value applications are returned from when the VRETURN instruction is executed.

<CV,VRETURN,VA,MA,v:va:cp:VS,env:cv:MS,E> →

<cv,cp,va,MA,v:VS,MS,env>

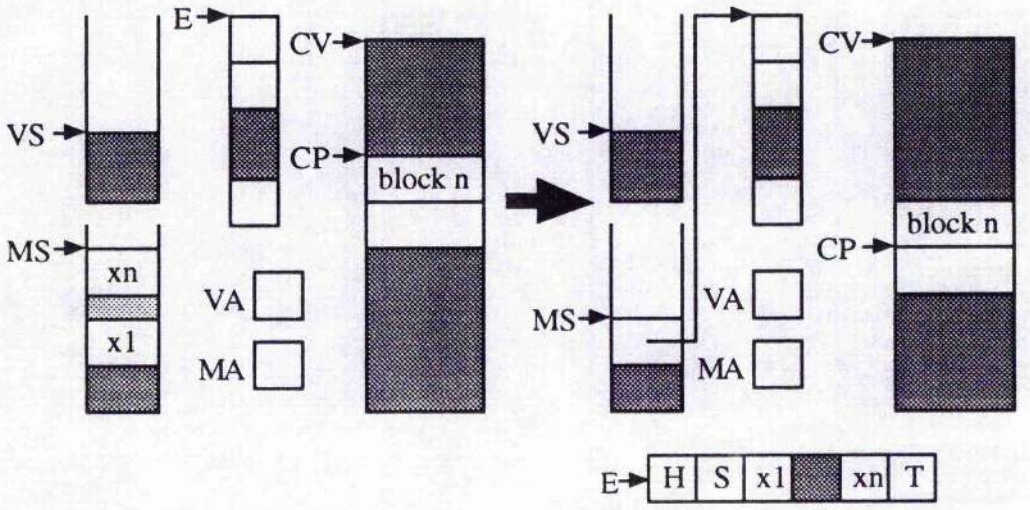
#### 4.3.1.8 Block entry and exit

A block is entered when local definitions are made. Block entry is similar to function application except that the function is formed and entered in one step. In addition, a block does not have an argument, but must retain the one used in its calling function.

<CV,BLOCK n:CP,VA,MA,VS,$x_n$:...:$x_1$:MS,E> →

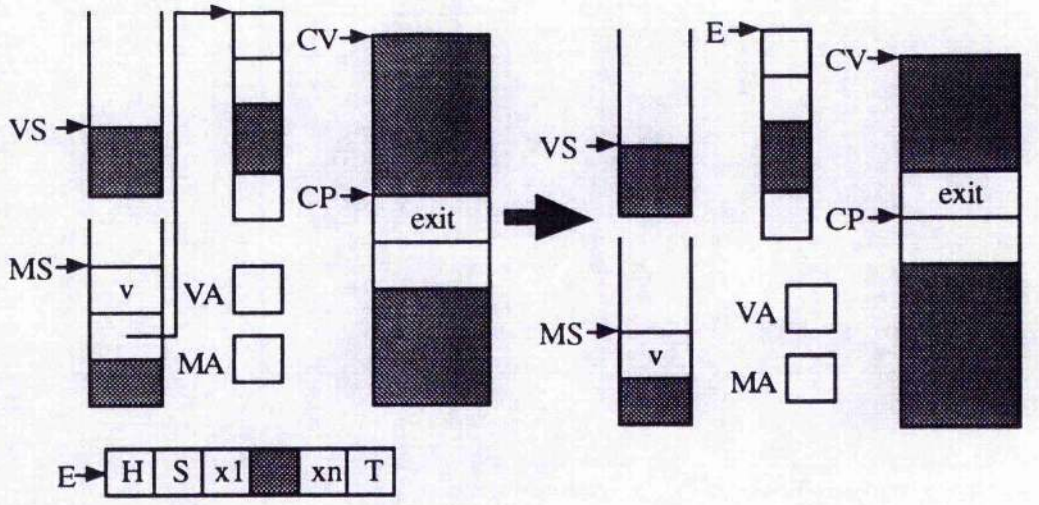
<CV,CP,VA,MA,VS,E:MS,[$x_1$,...,$x_n$]>

The effect of a BLOCK instruction is to take the top n elements off the main stack and create an environment object containing these items. This is then made the current environment, the old environment is saved on the stack.




At block exit time, the value of the block is on top of the stack together with the saved environment. The saved environment is restored and the value replaces it on top of the stack.

<CV,EXIT:CP,VA,MA,VS,v:env:MS,E> → <CV,CP,VA,MA,VS,v:MS,env>




The program fragment

```
a+b
where
a = 3
b = 4
```

generates the code

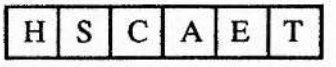
```
pushint 3     -- load a
pushint 4     -- load b
block 2       -- enter block
load 0        -- load a
getint
load 1        -- load b
getint
vadd          -- add them
mkint         -- box the result
exit
```

#### 4.3.1.9 Laziness

Lazy evaluation semantics require the evaluation of some expressions to be delayed (or suspended) until some later time when their value is required. This is implemented by constructing a suspension object consisting of the code for the expression and an environment created by taking the objects needed for evaluation

from the current environment. Later on, if it is needed, this object will be evaluated by executing its code with this environment. If this happens, the value computed will be used to overwrite the suspension.

A suspension object contains a code vector, C, and an environment, E, in which to execute the code. A suspension differs from a closure, however, in that the full environment is present in the suspension. A suspension has an additional field which holds the argument, A, which was in the argument register (in fact the argument as a heap object) when the suspension was created. For details of the use of the argument register see [Davi89]. The format of a suspension object is as follows.




The suspension object's tag is used when the value of the object is required to indicate that the object should first be evaluated. That is, the tag indicates that this object does not represent a value in weak head normal form.

When the value of an object is required strictly, for example when two integers are to be added, the run-time system examines the tag of the object to see if it is a suspension. If the object is not a suspension, it can be dereferenced with no further work. If, however, the object is a suspension, a process similar to a function call is initiated. The code from the suspension is executed in an environment created by loading the argument and environment fields of the suspension into the appropriate machine registers. When the execution of this code returns, an evaluated object is left on top of the stack with the suspension immediately below it. Next, the evaluated object is used to overwrite the suspension object in place so that any other objects which shared the suspension object will, in future, see the newly evaluated object. The space released when a smaller object overwrites the suspension can be garbage collected provided that there is support from the underlying object manager [Brow92].

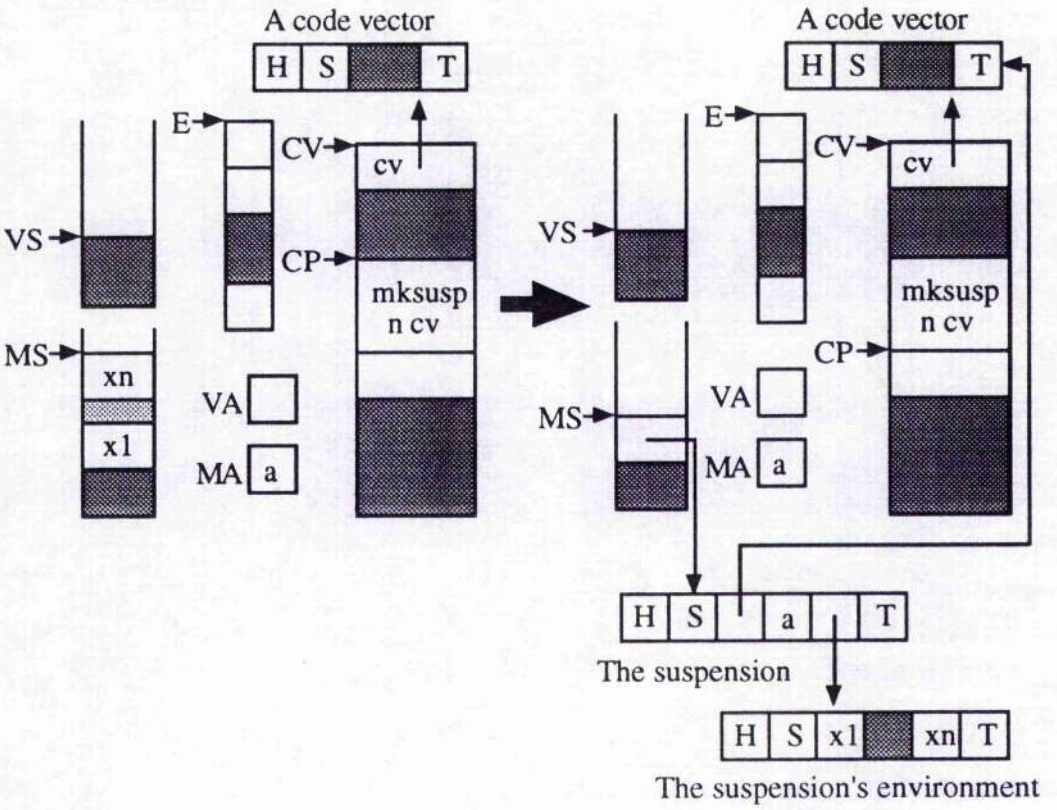
A suspension is created by the MKSUSP instruction.

$$\langle CV,MKSUSP\ n\ cv{:}CP,VA,MA,VS,x_n{:}\ldots{:}x_1{:}MS,E\rangle \rightarrow$$

$$\langle CV,CP,VA,MA,VS,\&sus{:}MS,E\rangle$$

where &sus points to suspension(cv,MA,[$x_1$,...,$x_n$])

This makes the top n items on the stack into an environment. The code for the suspension being created, the argument register and the environment are used to create a suspension object. A pointer to this suspension is placed on the stack.




The code fragment

```
a = 3 + b
```

would generate

```
load 0       -- get b from the current environment
mksusp 1 0   -- sub code vector 0
```

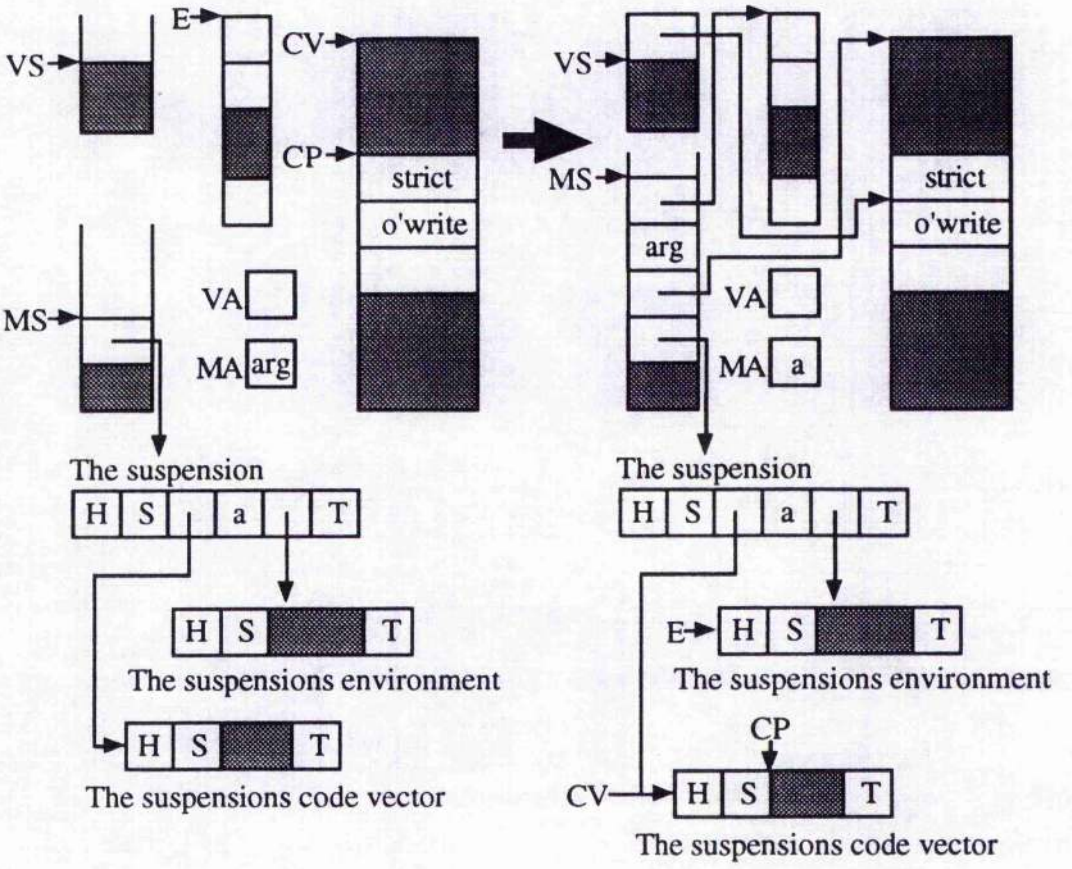
```
C0:   lint 3
      load 0        -- get b from the suspensions environment
      getint
      vadd
      mkint
      return
```

When the value of a suspension is required, the STRICT instruction initiates its evaluation. This operation is performed in two parts. First, the code for the suspension is executed with the argument and environment taken from the suspension. This process is very similar to function application. The second part of the evaluation is performed by an OVERWRITE instruction (this always follows a STRICT instruction). This instruction carries out a graph reduction by overwriting the suspension with the value calculated. To enable the overwriting, a pointer to the suspension is left on the stack before executing its code.

<CV,STRICT:OVERWRITE:CP,VA,MA,VS,&sus:MS,E> →

<cv,cp,VA,a,(OVERWRITE:CP):VS,E:A:CV:&sus:MS,e>

where &sus points to suspension(cv,a,e) and cp points to the 1st instruction in cv



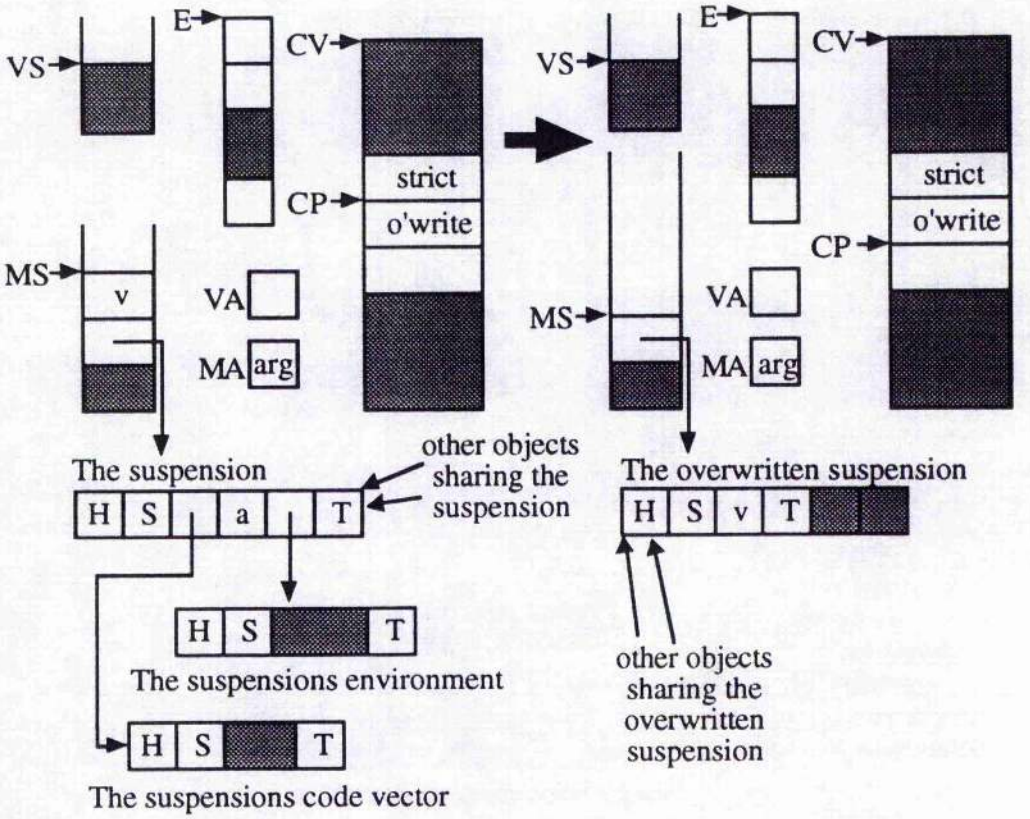

<CV,OVERWRITE:CP,VA,MA,VS,v:&sus:MS,E> →

<CV,CP,VA,MA,VS,V:MS,E>

where &sus points to suspension(cv,a,e)

and V points to the overwritten suspension

The OVERWRITE instruction causes the suspension to be overwritten by the value on top of the stack. Objects sharing the suspension will share the overwritten suspension.




If the suspension is already evaluated, then the OVERWRITE instruction is ignored

<CV,STRICT:OVERWRITE:CP,VA,MA,VS,v:MS,E> →

<CV,CP,VA,MA,VS,v:MS,E>

In the examples above, the strictification has been omitted. The example above

```
a = 3 + b
```

would actually generate

```
load 0       -- get b from the current environment
mksusp 1 0   -- sub code vector 0
```

```
C0:    lint 3
       load 0        -- get b from the suspensions environment
       strict        -- make sure its evaluated
       overwrite
       getint
       vadd
       mkint
       return
```

#### 4.3.1.10 Recursion

Recursive definitions are ones in which the right hand side of the definition involves the use of the name being defined. The notation

```
e(x)
```

is used to indicate an expression involving x. Recursive definitions have the form

```
x = e(x)
```

and it can be seen that x is the fixed point of the function λx.e(x). That is, when the function λx.e(x) is applied to x, the result is x. A fixed point finding operator `fix` is used to find the fixed point of functions such as this. It is therefore possible to transform a recursive definition of the form

```
x = e(x)
```

into a non recursive one

```
x = fix(λy.e(y))        -- alpha conversion of λx.e(x)
```

using the `fix` operator. An alpha conversion has been applied for clarity.

In the Staple system, PCASE abstract machine code is generated which constructs an object (a closure) representing the function

```
λy.e(y)
```

A fix instruction is then generated which finds the fixed point of the function.

The `fix` instruction converts the closure into a suspension object whose argument field points at itself. That is a closure object :-

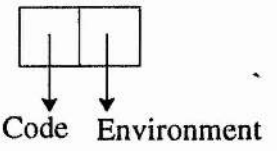



**Figure 4.4 - A Closure**

which represents λy.e(y) is converted to its fixed point fix(λy.e(y)). This can be represented as a suspension representing the application of the function λy.e(y) applied to the fixed point as follows

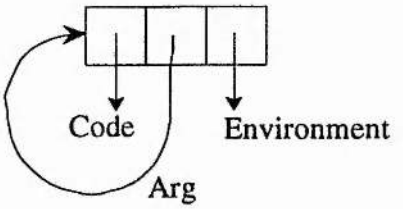


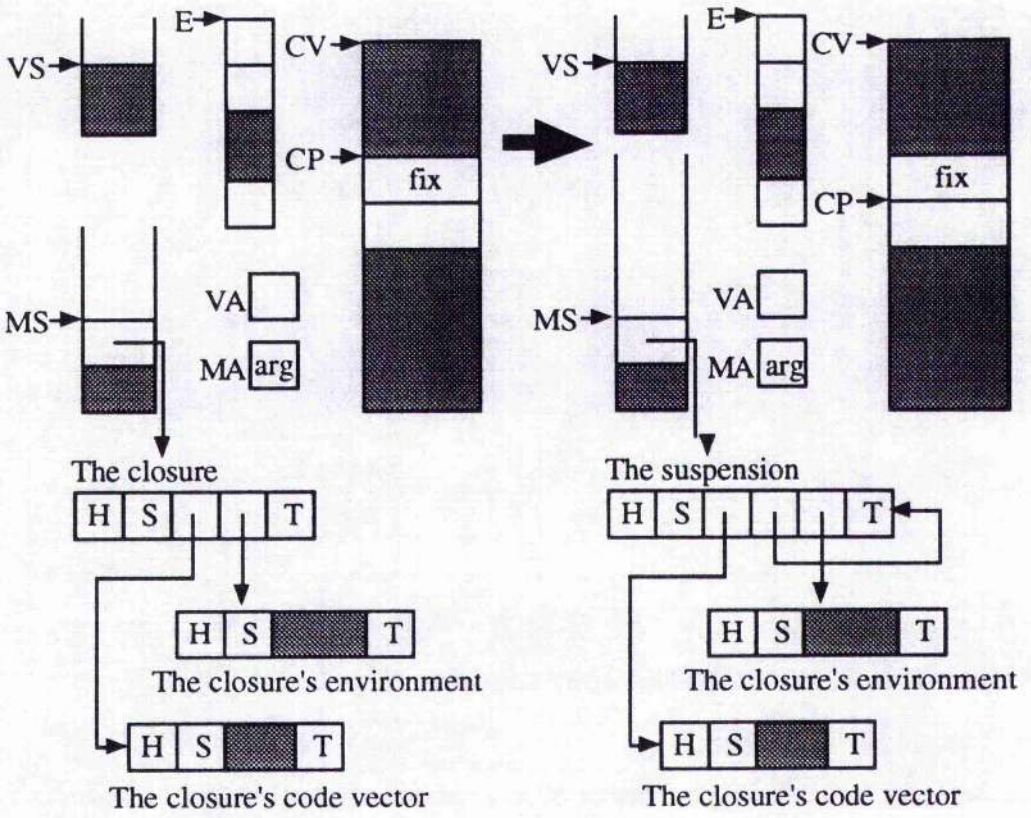

**Figure 4.5 - The Suspension Created by the fix Instruction**

where the code and environment of the suspension are obtained from the closure for λy.e(y). The result of evaluating a suspension is exactly the same as applying a function whose code and environment are the same as in the suspension to an argument which is the same as the argument part of the suspension. So the suspension in Figure 4.5 represents the expression (λy.e(y)) (fix(λy.e(y))) which is exactly the fixed point required. The FIX instruction performs this operation.

<CV,FIX:CP,VA,MA,VS,closure(cv,e):MS,E> →

<CV,CP,VA,MA,VS,&sus:MS,E>

where &sus points to suspension(cv,&sus,e)




The definition

```
x = 1 : x
```

generates the code

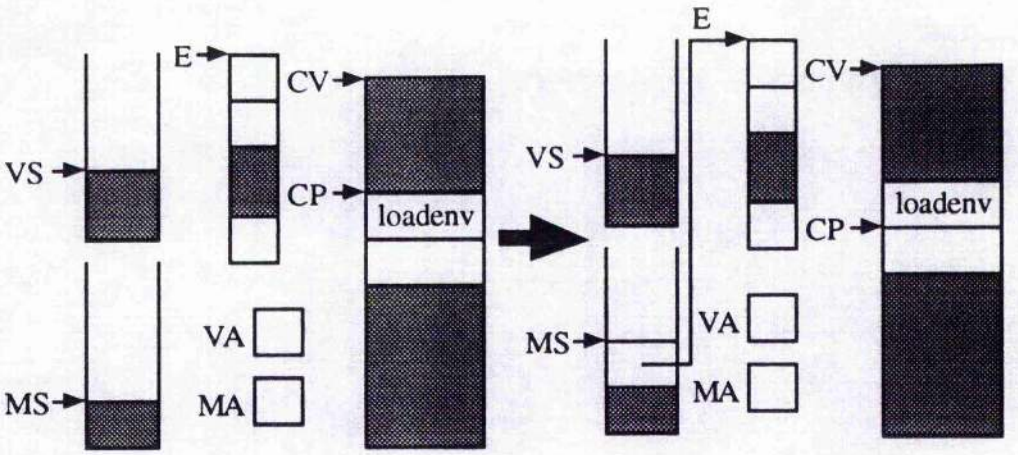
```
        mkclos 0 0          -- no free vars, code vector 0
        fix                 -- on stack is susp for x

C0:     loadarg             -- load x
        pushint 1           -- boxed integer
        pack 2 1            -- construct the list cell
        return
```

#### 4.3.1.11 Module creation

The LOADENV instruction is used to load a pointer to the current environment onto the main stack. This is used when creating a module (see Section 5.3.3).

<CV,LOADENV:CP,VA,MA,VS,MS,E> → <CV,CP,VA,MA,VS,E:MS,E>




This instruction is only generated by the transformation of a module (see section 5.3.3.1).

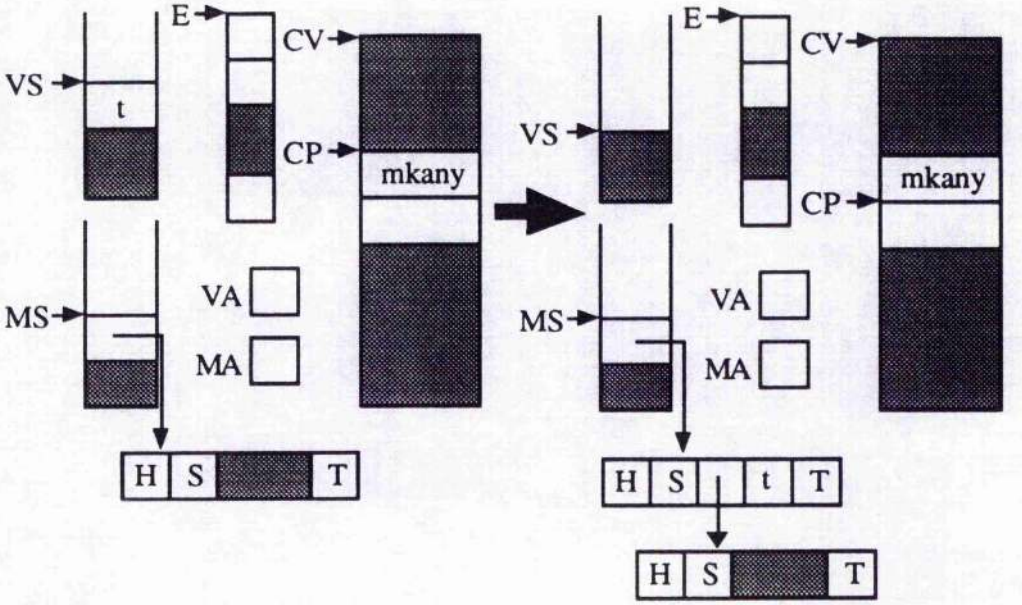
#### 4.3.1.12 Type Any

There is a canonical string representation for every type scheme. In Staple, this information is used to optimise the run-time type check. The canonicalized type is stored as a string in a global associative lookup table. When a type is used (in a mkany or coerce) the lookup table is examined to obtain a unique integer value for that type.

The MKANY instruction takes a value on the main stack and a type number from the value stack and builds an Any object.

<CV,MKANY:CP,VA,MA,t:VS,v:MS,E> → <CV,CP,VA,MA,VS,&any:MS,E>

where &any points to any(v,t)




The expression

```
mkany 3
```

generates

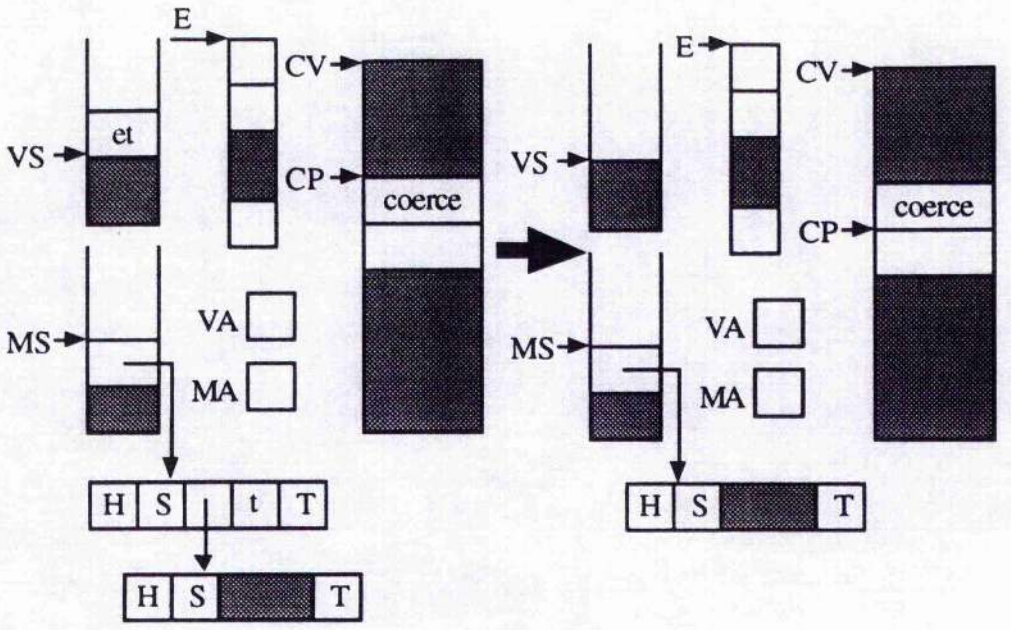
```
pushint 3
lint 7       -- the type tag for Int
mkany
```

The COERCE instruction is used when an Any object is projected. The expected type number is on the value stack and the Any object is on the pointer stack. The expected type and the actual type are compared and if they are equal the value field of the Any object replaces it on the pointer stack. Otherwise a run-time type error is reported and execution ceases.

<CV,COERCE:CP,VA,MA,et:VS,&any:MS,E> →

<CV,CP,VA,MA,VS,v:MS,E>

where &any points to any(v,t) and et = t




The expression

```
coerce x :: Int
```

generates

```
load 2        -- load x
lint 7        -- the type tag for Int
coerce
```

### 4.3.2 Efficiency

The compiler is responsible for optimising the use of the environment register so that it contains only the variables which are required. This will avoid unnecessary block retention and thus avoid any space leakage which might result from retaining unwanted data. In traditional frame based implementations, all the variables in scope in a block or function body can be accessed through the static chain. Values such as functions and suspensions created in such an environment will retain references to all the variables currently in scope even if they are not needed by the function or to compute the values of a suspension.

With the PCASE architecture, only space needed in the closures or suspensions which are reachable are retained. All other space is available for reuse and will be garbage collected. With fewer objects being retained, there will be fewer garbage collections.

The persistent Staple implementation was compared with an earlier non-persistent implementation both of which use an interpreter implementation of the CASE abstract machine and are different only in that the PCASE machine uses the stable heap as its run-time heap whilst the non-persistent CASE machine uses a semi-space heap with direct memory access. The measurements indicate that the persistent implementation is 10% slower than the non-persistent implementation. Persistent values are referred to by an object identifier.

## 4.4 Using The Stable Store

The Staple system and PCASE machine use the stable store interface to create all run-time data values, for example closures, suspensions, data structures and literals as well as modules and support structures. All objects must be represented using the stable store generic object format described above. The object store provides procedures covering ten object management areas:

- initialisation of the object store
- close the object store
- create object
- destroy object
- get the root object (objects reachable from here persist)
- write object
- read object

- checkpoint the object store
- garbage collection
- quality of service

How these are used within the Staple system and the PCASE abstract machine is described in the following sections.

### 4.4.1 Initialising the Object Store

When a Staple interactive session is started or a module is to be compiled, the persistent store is initialised by calling the *init_sheap* procedure. This performs the tasks necessary to initialise a heap of persistent objects.

### 4.4.2 Closing the Object Store

When a Staple interactive session ends, or the compilation of a module (see Chapter 5) is complete, the object store is closed by calling the *close_sheap* procedure.

### 4.4.3 Creating Objects

Every time a data value is created by the PCASE machine, a call is made to the function *create_object*. The size of the object to be created (in words) is passed as a parameter. The function allocates space in the stable heap and returns the key of an object of the appropriate size. For example, the PCASE machine code

```
pushint 42
```

causes the creation of a heap object representing an integer. The interpreter performs this with the following call

*theInteger = create_object( 4 2)*

Four words are required to represent an integer. There are two words for the header and size fields, one word for the value and one word for the system type. Section 4.4.6 describes how the fields of objects are updated.

A call to *create_object* may fail returning 0 as its result. If this happens, the PCASE machine calls the garbage collection procedure (See Section 4.4.9)

All objects are initialised with zero pointer fields. The abstract machine has the responsibility of inserting pointer fields and updating the object's header appropriately.

### 4.4.4 Destroying an Object

The procedure *destroy_object* is used to destroy a persistent object and release the storage allocated to it. This function is not used in the PCASE interpreter.

### 4.4.5 Accessing the Root Object

For objects created by the PCASE machine to persist, they must be made reachable from a distinguished persistent object called the root object or root of persistence. In the Staple system, once the store has been initialised by a call to *init_sheap*, the root object is obtained by calling the interface function *first_object*. The root object is used to provide a hook for making objects persist. The precise contents of the root object are described elsewhere [Brow90]. The first pointer field of the root object is available for use by users of the interface. The first field can be made to point to any persistent object and therefore to any data structure created in persistent storage.

In the Staple system, the stable store root object is made to point to an object which contains the supporting structures for the system. These structures are

- a structure mapping module names to module representations
- a structure mapping strings to values of the dynamic type

- an object in which to store the PCASE abstract machine state which contains pointers to the stacks and stores the values of the machine registers
- a table mapping canonical type representations to integers
- a module containing built in functions
- an object (nil) used in the mapping structures as a leaf node.

These are described in more detail in Chapter 5.

### 4.4.6 Writing to Objects

After an object has been created, its fields can be written to. In the example above, an object was created to represent the integer 42 in the heap. The object returned by create_object must be filled in. To do this, the interpreter calls write_object. This function is parameterised by the object's address, the offset to the first word to be copied (starting at 0), the number of words to be copied and the RAM address of the data to be copied. The call is as follows

*write_object( theInteger, 2, 1, &theArg )*

where *theArg* is a variable containing the integer 42 obtained from the machine code and *&theArg* is its RAM address.

If an object has pointer fields (for example an environment) then the header must also be updated. To write a single pointer value called *theValue* into an environment with only one element called *theEnv* , the following sequence of calls can be made.

*write_object( theEnv, 2, 1, &theValue )*
*numPointers = 1*
*write_object( theEnv, 0, 1, &numPointers )*

Subsequent garbage collections will ensure that the value pointed to by the environment is retained.

### 4.4.7 Dereferencing Objects

Objects can be dereferenced using the procedure *read_object* which has the same parameters as *write_object*. The requested number of words are read from the object into RAM. An example of the use of read_object in the PCASE machine is during the loading of non-local variables. The code

```
load 3
```

loads the element at offset 3 in the current environment. The PCASE interpreter maintains a pointer to the current environment in a variable called *env*. To obtain the value from the environment, the following call is made

*read_object( env, 5, 1, &theValue )*

where *theValue* is a suitable variable. The offset is computed from the code plus two from the header and size words. The value will subsequently be pushed onto the main stack to complete the load instruction.

### 4.4.8 Stability

During the execution of a program, the changes to data in the persistent store can be made permanent by means of a commit or checkpoint operation provided by the stable store *chkpt_sheap*. The PCASE machine incorporates the ability to be restarted in the event of system failure without loss of evaluation from the last checkpoint. In order to facilitate this, the entire state of the machine must be stored when a checkpoint takes place. The stacks, argument registers, code vector and code pointer registers and environment register are stored in part of the root object before a checkpoint operation. When a program successfully completes, the saved machine state is deleted since a new session begins by building the environment for

that session as specified by the user. If the system fails, it can be restarted by loading the abstract machine registers from the stored state and continuing the execution. A checkpoint occurs in Staple when a garbage collection occurs and when a session terminates.

### 4.4.9 Garbage Collection

Should object creation fail (indicated by a zero value returned from *create_object*), the PCASE machine initiates a garbage collection by calling *gc_sheap*. This is done by first saving any cached pointers into the stable heap. Data is retained if it is reachable at the time of the garbage collection. This data will include all persistent data and all data required for the current computation. The abstract machine should ensure that all usable data values are reachable. In fact they are all reachable from the stacks so since the stacks are reachable then so is the data. The PCASE abstract machine maintains its stacks within the persistent heap. The stable store garbage collection retains all data which is reachable from the root of persistence. All the PCASE instructions which create persistent objects may cause a garbage collection.

Unlike other implementations of languages using this object store technology, Napier88 [Morr89], Quest [Matt92] and Galileo [Alba85], PCASE does not maintain a local heap of persistent objects. These implementations have a local heap which requires objects to be copied from the stable heap to and from the local heap during a disk garbage collection. PCASE executes directly inside the stable heap itself. The advantage of a local heap is the ability to perform a local garbage collection which may be quicker than a full disk garbage collection. However, there is an overhead in copying data between the stable heap and the local heap. In addition, since both use main memory, there will be less space available for a local heap so more garbage collections will be required. Direct comparison between these implementation methods would conclude the debate, but have not been carried out by any of the language implementors as yet.

### 4.4.10 Quality of Service

Quality of service information is provided to allow implementations to bypass the stable store interface when greater efficiency is required. This is permitted provided strict rules are adhered to. In particular, an implementation must not maintain RAM addresses of objects over garbage collection since the garbage collector may move objects.

The persistent heap interface provides functions which create and manipulate persistent data values which are referenced by their persistent object identifier or key. It is not possible using these functions to directly access the memory in which the persistent data value is stored directly. However, a function is provided, *SH_keytoaddress*, which allows an implementation using the stable store to obtain the RAM address of a persistent object. This address can then be used to dereference the object directly thus bypassing the interface.

An additional stable store procedure provides a user with quality of service information [Brow92]. In particular this information includes details of how to map keys to addresses without the need to go through the interface function. Also included in the quality of service information is an indication of whether this implementation of the stable store interface supports objects being shrunk in size. This is important in PCASE when overwriting suspensions with values represented by smaller objects.

## 4.5 Conclusions

To avoid engineering the Staple system from scratch, it was decided to use existing object store technology and to build an abstract machine suitable for the evaluation of persistent lazy functional programs. The resulting abstract machine, PCASE, provides support for

- the execution of persistent lazy functional programs which use a persistent heap as their only run-time data space.

- Creation, dereference and update of persistent lazy functional objects which conform to the heap format requirements of the stable store

- Automatic garbage collection by ensuring that the stacks and registers are reachable from the root of persistence during execution

- Stability by maintaining the machine state in the heap

- Efficiency by using an environment structure which retains only references to values which may be needed during the evaluation of a particular expression thus reducing the number of objects retained and consequently the amount of garbage collections.

- Code in the heap by storing code in short chunks in code vector objects. The machine registers CV and CP point to the current code vector and the current instruction respectively.

These features are provided in part by the postore library. The PCASE machine takes advantage of the postore functionality by mapping all object representations onto the generic stable store object format. The PCASE abstract machine instructions utilise the stable store interface to create and dereference run-time objects. Persistent modules and stream persistence build on the PCASE machine to provide persistence to user programs.

# Chapter 5

# Implementing Persistent Modules and Stream Persistence

The Staple system architecture supports two models for persistence: persistent modules and stream persistence. The system consists of

- A programming environment
- A compiler for the Staple language
- A run time system which drives the evaluation of functional programs
- An interpreter for the PCASE abstract machine
- A persistent object store

These components are shown diagrammatically in Figure 5.1. Use of the programming environment was described in Chapter 2. Chapter 4 described in detail the architecture and operational semantics of the PCASE abstract machine and the usage of the object store library. The present Chapter describes the implementation of the other components of the Staple system with a particular emphasis on support for persistent modules and stream persistence.

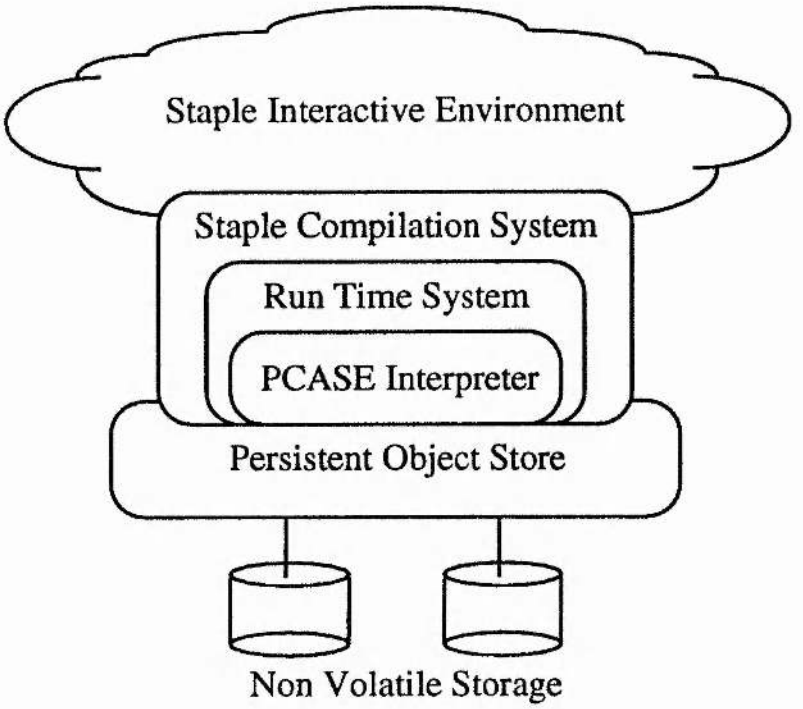



**Figure 5.1 - The Staple System Architecture**

## 5.1 The Staple Compiler

The Staple language is a lazy functional programming language whose syntax resembles Orwell [Wadl85] and Miranda [Turn85]. The language has a rich type system which includes algebraic data types and a dynamic type. Function definition is by pattern matching equational definitions. The compiler structure is shown in Figure 5.2

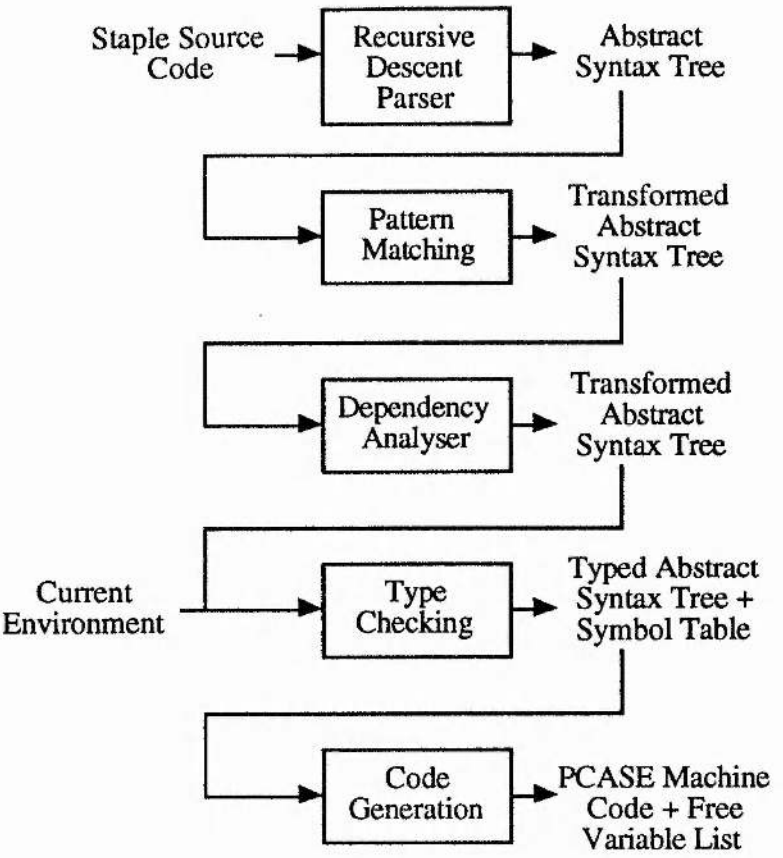



**Figure 5.2 - The Staple Compiler Structure**

Source code is parsed using a recursive descent parser [Davi81]. The parser produces a data structure representing an abstract syntax tree which is transformed in later passes to perform pattern matching removal, dependency analysis, type checking and code generation.

The pattern matching pass transforms functions defined as a set of clauses involving patterns into ones involving only case analysis. Pattern matching removal is described in detail in [Peyt87b]. Generating code for functions which involve pattern matching is made much simpler after performing this transformation. The PCASE machine instruction CASE is used to determine which variant of the algebraic data type a particular value is and selects which code to evaluate as the function result.

The dependency analyser transforms definitions in a block into many nested blocks each containing definitions which depend only on definitions in outer blocks or are mutually recursive with definitions in the same block. Dependency analysis for this kind of functional language is described in detail in [Peyt87b]. Dependency analysis is required to ensure definitions are attributed a most general type by a Hindley-Milner type checking algorithm. Without dependency analysis, some expressions may receive less general types than they should have. This is also detailed in [Peyt87b].

The type checker assigns a most general type for each expression eventually arriving at a most general type for each definition. The inferred type must be more general than a declared type (if given) for the definiens and this is checked. The declared type is always used if one has been given. The type checker carries out its work in an environment which assigns types to imported identifiers (i.e. identifiers brought into scope by being imported from a module). The type checker is also used to construct a symbol table for a module. The symbol table assigns a type to the identifiers exported by a module.

The code generator produces PCASE abstract machine code. Examples of code generation were given in Chapter 4. The resulting code for a given program consists of a code vector object which may contain sub code vectors forming a tree structured code. In addition to the code, the code generator computes the list of free variables in a given program (this will not necessarily be the same as the list of imported variables). This list of variables is used to construct an environment in which the code will be executed.

## 5.2 The Run Time System

The run time system drives the evaluation of persistent lazy functional programs. It also manages the module directory mapping, the mapping between strings and values of dynamic type and interprets stream persistence requests.

### 5.2.1 Driving Evaluation

Evaluation in the staple system is print driven. A compiled program consisting of a code vector object and an environment object is passed to the run time system which in turn calls the PCASE abstract machine interpreter to evaluate the program. The PCASE interpreter provides a function

```
eval_and_print( code, environment, arg, type ) = ...
```

for performing evaluation. The initial argument is a null pointer. The machine state is initialised with the code vector, environment and argument as shown in Figure 5.3 The code pointer points to the first instruction in the code vector and the stack pointers point to the first location on the stacks.

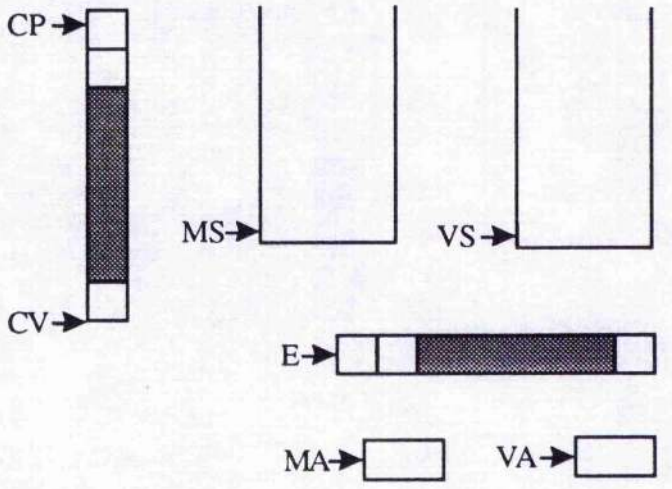



**Figure 5.3 - The PCASE Initial Machine State**

At this point, control is passed to the interpreter's instruction execution loop. The interpreter stops when the result in weak head normal form is on top of the main stack.

If the resulting value is a scalar value (determined by its type), then it is printed and program evaluation terminates. If it is a data structure, then the components of the structure are evaluated and printed. The evaluation of components may involve additional calls to the PCASE interpreter to evaluate suspensions. When evaluating

a suspension, the code, environment and argument passed to the interpreter are obtained from the suspension object. When execution of the suspension code is completed, the suspension is overwritten by the value on the stack and this value is then printed. This process is repeated until evaluation completes or is interrupted by the user. The evaluation function is used in two places. It is used for evaluating interactively entered expressions and it is used during module compilation for building an environment containing the values of the identifiers exported by the module.

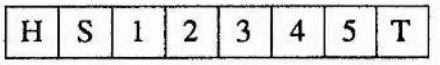
## 5.3 Persistent Modules

The Staple persistent module system provides a mapping between module names and module representations. The modules are stored in persistent storage in the form of an environment object. The environment contains one entry for each of the identifiers exported from the module. The Staple system maintains an associative lookup table in the form of a binary tree in persistent storage which allows new associations between names and modules to be created and modules to be looked up. In addition to the environment, other information for type checking and linking is stored as part of the module. The lookup table is made reachable from the root of persistence which results in all values in the lookup table being persistent.

To facilitate this system, a persistent store representation of a module is defined. The module representation must contain more information than just the values defined. There must be information about how the names for these values in the module map to the representations of the values. The types of these names must be known for static checking and any algebraic data type definitions must be included in the information in persistent storage. The stored information forms a complete interface specification for the module (see section 5.3.3.4).

### 5.3.1 Staple Modules

A compiled module is capable of being stored in the persistent store. It conforms to the object format described in Chapter 4. A module descriptor is represented as a stable store object with five pointer fields –




1) points to a string containing a *textual description* of the module. (A user supplied comment).

2) points to the *source* of the module.

3) points to a *symbol table* for the module, this associates each name defined in the module with its type and its address stored as an offset into the environment object pointed to by field 5.

4) points to a *types table* which contains definitions of all algebraic data-types and type synonyms defined in the module. Fields 3 and 4 together form the module's interface.

5) this field points to the *value* of the module which is an environment object with one entry for each value exported by the module.

Each entry in the environment initially points to a *suspension*. This suspension represents the value of an identifier defined in the module and exported from it.

### 5.3.2 The Staple Universe

As mentioned above, the Staple system maintains a flat name-space mapping names to modules. This mapping is implemented using a string indexed associative lookup table organised as a sorted binary tree.

All objects which are reachable from the stable root object (a special object in the stable store) will persist provided a stabilise operation has taken place. Our universe consists of a Staple root object which contains several objects particular to the Staple system. One of these is the module name-space. The second is a global nil object (for efficiency), the third is a special module which contains functions for Staple's built in operators and finally there is a machine state object which is used to preserve the state of execution of the entire machine over a garbage collection or stabilise operation. Figure 5.4 shows the organisation of the Staple stable store.

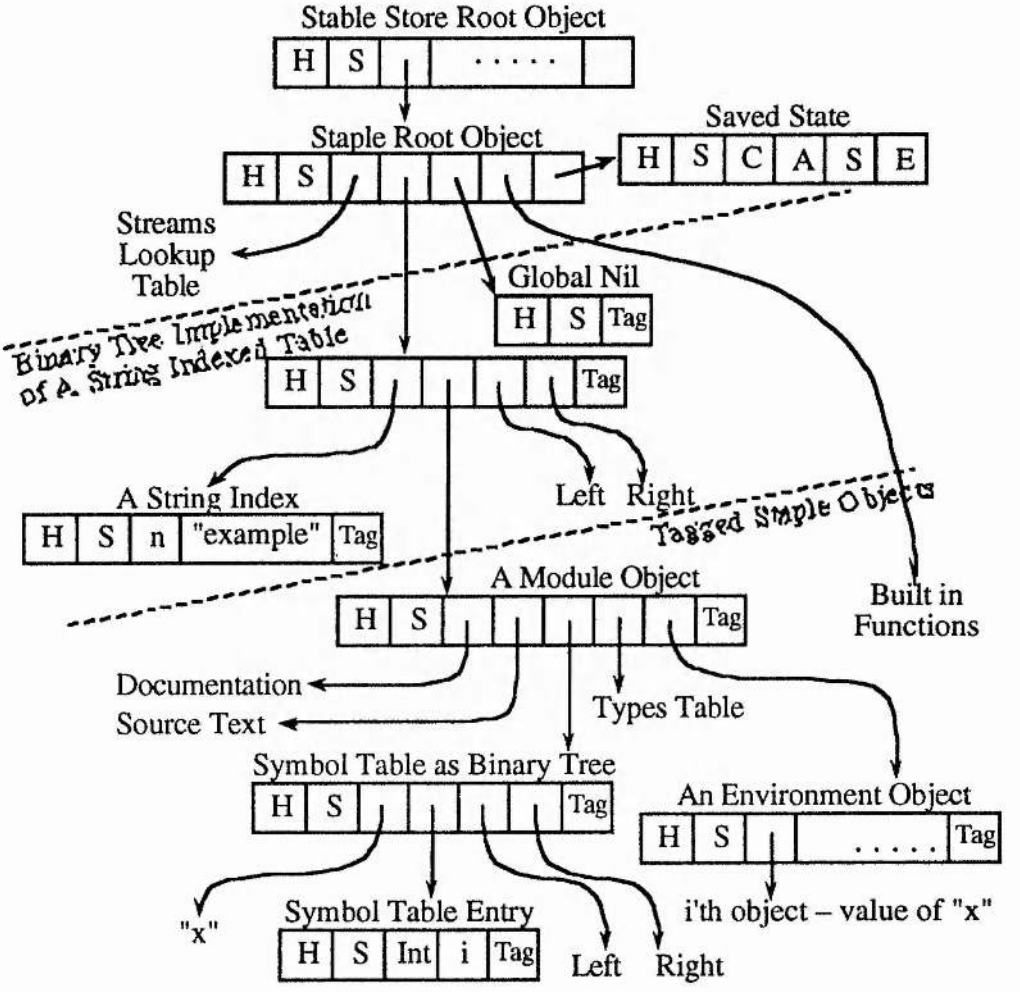



**Figure 5.4 – The Staple Stable Store Organisation**

### 5.3.3 Module Translation

An executable program is produced by compilation of the source code followed by a linking phase where external references are resolved. This process is completed by the compiler or in some cases by a compiler and a linker. There is normally no execution of the program during this transformation.

The process required to create a persistent module in Staple involves both the compiler and the run-time system. There is an initial transformation of the source which is then compiled. External references are then resolved and code executed to construct an environment for the module which is then associated with the name of the module in the persistent store. In addition, a symbol table is stored with each module for use when the module is imported into some environment. Loading a module involves the following five steps:

- Parsing and Transformation
- Code Generation
- Linking
- Execution
- Binding

An example module in which only a few simple values are defined will be used to illustrate each of the above five steps. The module given in Figure 5.5 defines two functions over integers. The first takes two arguments and computes their sum using an imported function `addPair` and the second returns the successor of its argument. The type signatures of the functions are given just before the equations which define their value.

```
-- A module containing some simple integer functions
add :: Int -> Int -> Int
add x y = addPair (x,y)
succ :: Int -> Int
succ = add 1
```

**Figure 5.5 - A Simple Module**

The name of the module is supplied to the module compilation system together with the names of any modules which are to be imported (in this case, there must be at least one containing `addPair`).

#### 5.3.3.1 Parsing and Transformation

The first step in the process is to parse the source code to produce an abstract syntax tree. The tree is transformed as described in Section 5.1 to the stage before code is generated. A further transformation is then performed on the abstract syntax tree. The result is an abstract syntax tree which represents an expression whose value is an environment object containing the values defined in the module. The environment is represented in the same way as other environment objects in the PCASE machine. The data type definitions in a module will be obtained from the symbol table during the compilation step. The module source given in Figure 5.5 is transformed into the expression in Figure 5.6

```
let
   -- A module containing some simple integer functions
   add :: Int -> Int -> Int
   add x y = x + y
   succ :: Int -> Int
   succ x = add 1
in
   loadenv
```

**Figure 5.6 - After Transformation by the Compiler**

#### 5.3.3.2 Code Generation

Code is generated for the expression in Figure 5.6 as though it had been entered interactively during a Staple session. The result of code generation is a PCASE code vector object. This may contain sub-code vectors (see Sections 4.3.1.7 and 4.3.1.9) in which case the resulting code is a tree structure.

The expression `loadenv` is not available to the programmer, but is only recognised by the module compilation system. It has the effect, at run-time, of loading the current environment onto the stack. An environment object is not otherwise found on the stack during execution of programs, but in this case it allows us simply to execute the above program, and fill in the fifth field of the module with the result found on top of the stack.

The symbol table of the compiler is retained and re-used in the last stage – it contains the information about the types of the values defined in the module. Information about what algebraic data types have been defined within the module is also retained. This information allows typechecking to be performed completely at this stage.

#### 5.3.3.3 Linking

A Staple module may refer to identifiers bound in other modules. The names of these modules are supplied when the present module is to be compiled. These names identify the modules in persistent storage in which the values for free variables in the present module can be located. The references are resolved by creating bindings to the values in imported modules. The mechanism used to do this is for the compiler to produce a list of free variables in a module and to allocate addresses for these free variables inside an environment in which the module program (Figure 5.3) will be executed. The Staple system obtains values for these free variables by looking them up in the persistent store during this stage of processing of the module and placing references to the values into a new

environment object. The execution of the program begins by initialising the PCASE abstract machine with the environment containing values for the free variables and the code generated by the transformed module program. The initial state of the PCASE machine is shown in Figure 5.7

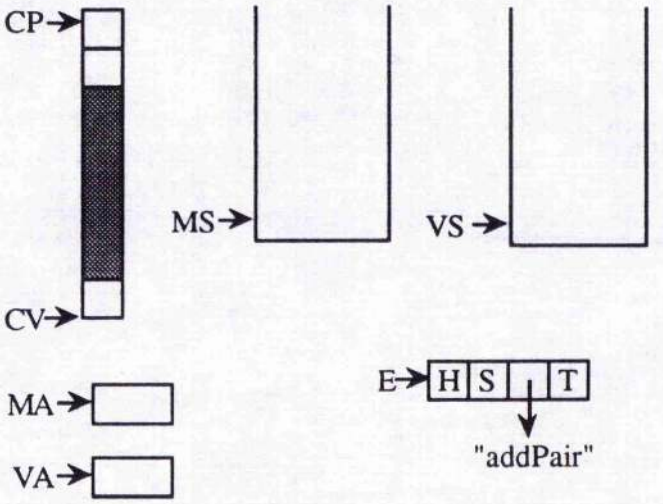



**Figure 5.7 - Initial State with Environment**

#### 5.3.3.4 Execution

The program in Figure 5.5 is evaluated in the environment provided by the linking stage. The result of the execution will be an environment containing references to the named values at the top level of the module. It is important that no evaluation of the components of this environment take place at this stage. By doing so, the execution is guaranteed to terminate. The environment value which is created will contain references in closures and suspensions to values imported from other modules. The state of the PCASE machine after execution is shown in Figure 5.8

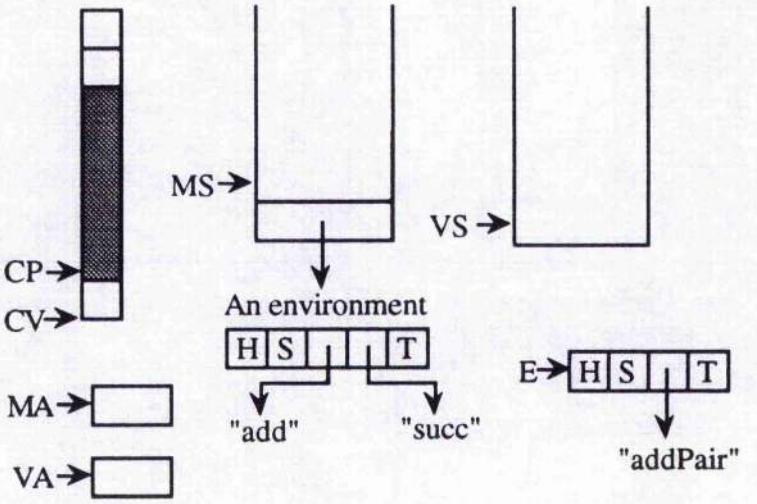



**Figure 5.8 - After Execution**

#### 5.3.3.5 Binding to the Module Name

The result of evaluating the program is an environment object which will be on top of the main stack. A module object is created and the environment is obtained from the top of the stack. The symbol table and types table are obtained from the compiler. Finally, a binding between the module name and the module is created in the persistent store.

## 5.4 Stream Persistence

Stream persistence is implemented in the Staple system by providing

- an associative lookup table in persistent storage between strings and values of type `Any`,
- a method of performing stream I/O and
- support for persistence requests.

Implementation of stream I/O must ensure synchronisation between requests and responses such that there is a one to one mapping between the two. In addition, it must be possible for the programmer to ensure that no response is examined before

a request for that response has been fully generated. The mechanism used to implement stream I/O is a head strict list constructor function [Thom86].

### 5.4.1 Stream Persistence Resource Mapping

To support the persistent stream requests, Staple has in addition to the module system mapping from names (strings) to module representations, another mapping between names (strings again) to values of type `Any`. This mapping is implemented in the same way as the lookup table for modules and the same operations for lookup, and update are provided. The operations on this structure are performed during processing of stream persistence requests.

### 5.4.2 Stream I/O on Files

Stream I/O uses the model of computation described in Chapter 3 in which the programmer writes dialogue functions which when entered interactively are supplied automatically with an additional argument – the list of responses to the requests which the function produces. If the type of an interactively entered expression is a dialogue

```
[Response] -> [Request]
```

then this special behaviour is triggered. A function of this type is applied to a value representing the list of responses. The value is initialised with an error value – if this value is requested, then evaluation stops and an error is reported. Otherwise, each request which is evaluated is examined by the evaluator and acted upon. A response is then generated according to the status of the action initiated by the request. This response is added to the end of the responses list as a new cons cell. Subsequent evaluation may now refer to the response value. For example, consider the following dialogue:

```
f
where
f resps = [ReadFile "somefile",
           AppendChan "stdout" "Hello World'n"]
```

This program is transformed to the following

```
f getresps
where
f resps = [ReadFile "somefile",
           AppendChan "stdout" "Hello World'n"]
```

where the identifier `getresps` generates a special PCASE machine instruction which obtains a pointer to an initial list of responses. The responses list is initialised to the empty list so when the function f is entered, the PCASE machine state is as shown in Figure 5.9

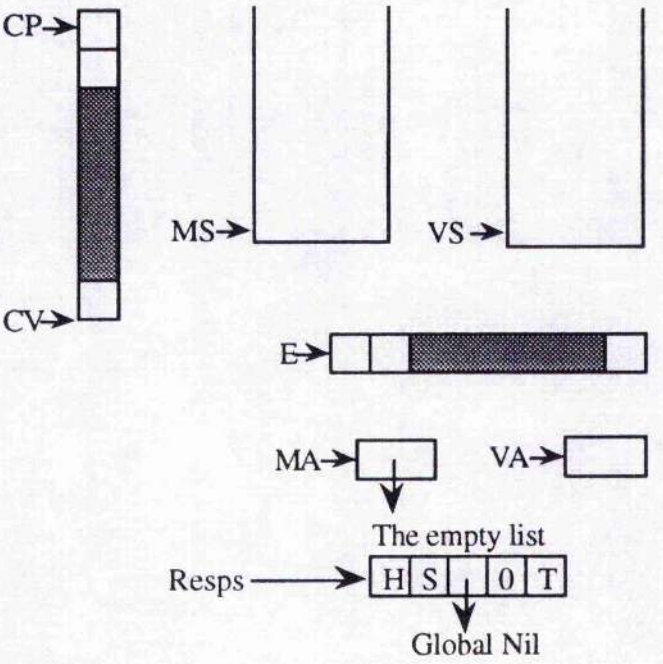



**Figure 5.9 - After Entering the Dialog Function**

The interpreter maintains a pointer to the last element in the response list Resps. This is used to fill in the responses as requests are processed. A transformed dialog function is executed like any other program and execution terminates when

the result is in weak head normal form. In the case of a dialog, this will be a cons cell whose head is a request and whose tail evaluates to the remaining requests.

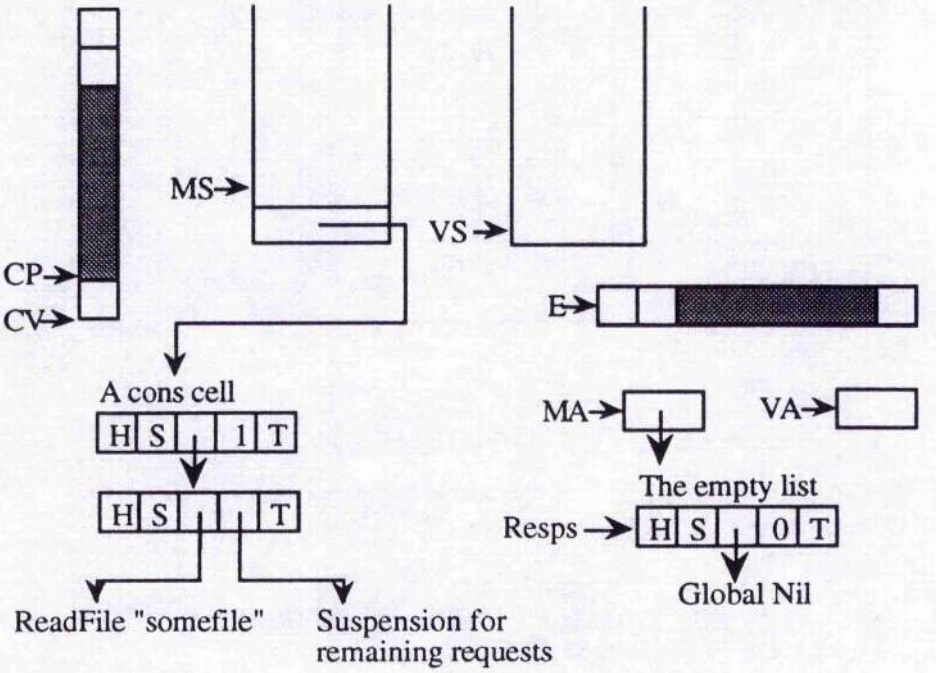



**Figure 5.10 - After Execution Terminates**

The result is popped from the stack and the tail of the list pushed back onto the stack for later processing. The head of the list is examined to check that it is not a suspension - if it is, a call to the interpreter is made to evaluate the suspension. The resulting value represents a request structure.

The kind of request is determined by examining the structure's variant tag. In the present example, the request is one to read the contents of a file. This request is processed by reading the file and creating a string (list of characters) which represents the contents of the file. This string is used to construct a response which is added to the response list.

Each time a response is generated, the current response list's last component (an empty list) is overwritten with a cons cell whose head points to the newly created response and whose tail points to a new empty list. The state after processing the read file request is shown in Figure 5.11.

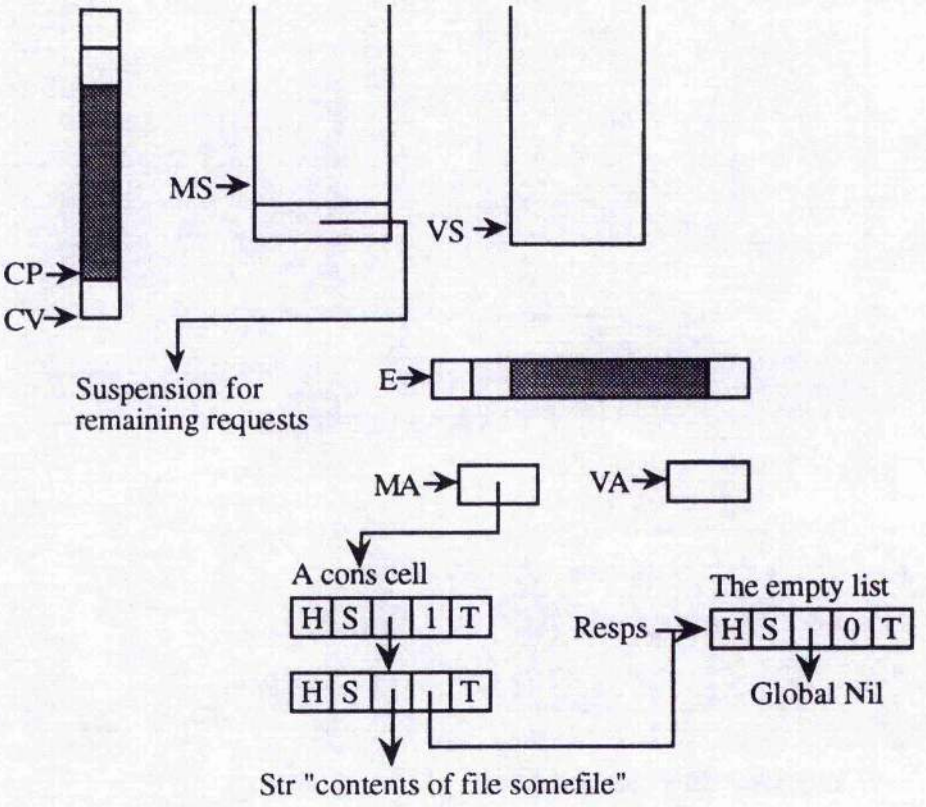



**Figure 5.11 - After the Response has been added**

The value on top of the stack at this point is evaluated to weak head normal form by a call to the interpreter. In the present example, the result is a cons cell whose head is an `AppendChan` request. This request is processed by writing the string "Hello World'n" to the channel "stdout" (standard output). The response to this request is an indication of success or failure. A successful response results in the responses list shown in Figure 5.12.

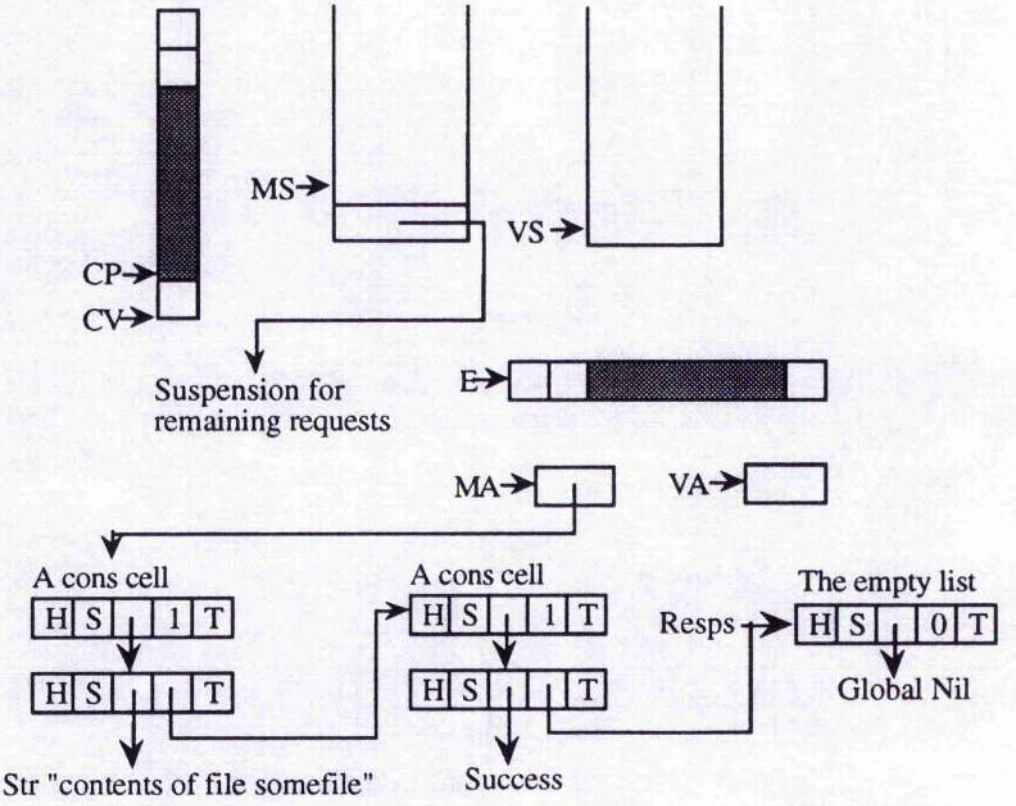



**Figure 5.12 - After the Success Response has been added**

### 5.4.3 Persistent Stream Requests

If the first request is one to lookup the name "fVVnet" then the state of the machine when execution stops is as shown in Figure 5.13

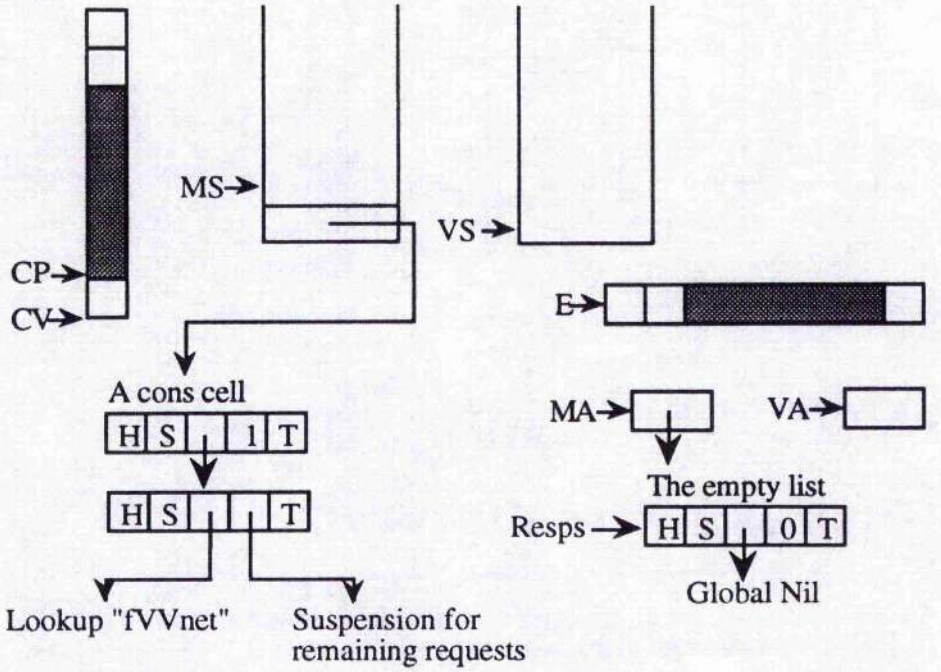



**Figure 5.13 - After Execution Terminates**

The request is processed by performing a lookup of the persistent store mapping between strings and values of the dynamic type `Any`. If a value is found, a response value is created and placed in the response list. After adding the response, the machine state is as in Figure 5.14

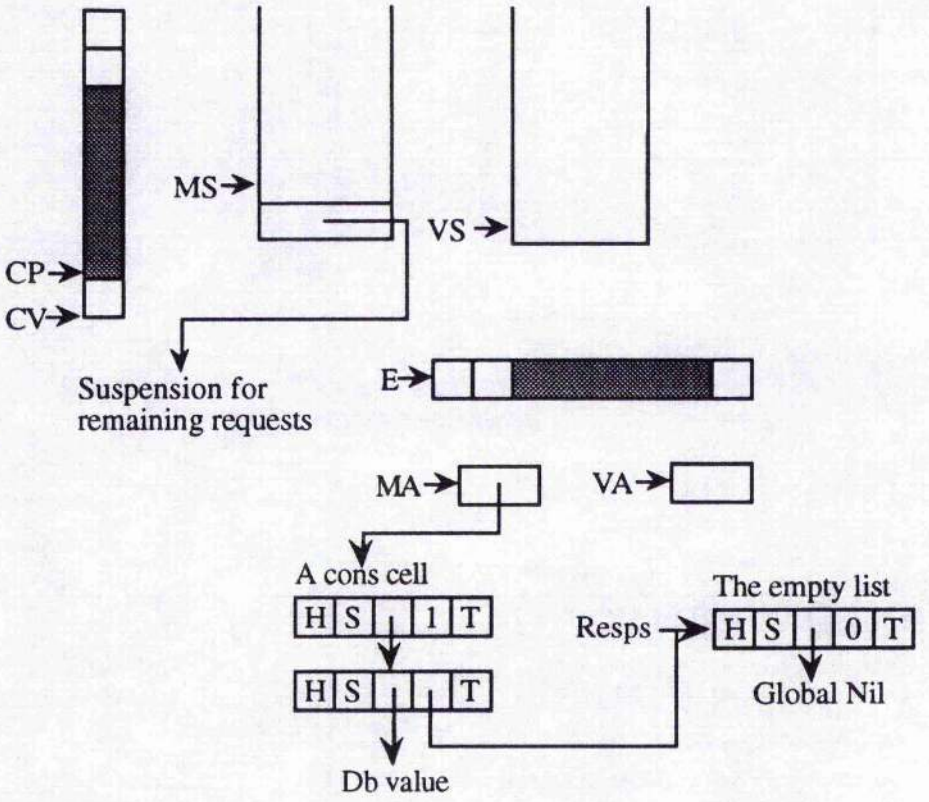



**Figure 5.14 - After the response has been generated**

The response for an `Insert` or `Delete` request is an indication of success or failure.

## 5.5 Conclusions

This completes the description of the Staple system architecture and how it supports two models for persistence: persistent modules and stream persistence. The system consists of

- A programming environment
- A compiler for the Staple language
- A run time system which drives the evaluation of functional programs
- An interpreter for the PCASE abstract machine

- A persistent object store

The compilation system provides the programming language interface to the programming system and compiles the Staple programming language into PCASE abstract machine instructions.

The run-time-system drives the evaluation of compiled programs. It supports persistent modules and stream persistence by

- creating module values whose environment contains exported identifiers in a module
- processing requests when the program is a Dialog function

# Chapter 6

# Persistent Functional Programming

Two applications which benefit from implementation using the Staple system are described in this chapter.

Persistent Modules are used in a prototyping experiment which models the transportation network for Frankfurt. This prototype, implemented originally in Miranda, suffered from the need to recompute a very large data structure each time the prototype was executed. Several ad-hoc attempts were made to alleviate the problem, but by using persistent modules, the data structure is evaluated at most once and can be reused many times.

Stream Persistence is used to model a database of parts and suppliers. Functions for accessing and manipulating the data are described. These example functions are illustrative of typical database programming tasks. An interactive database system can be constructed using the functions defined together with stream persistence and a stream I/O model.

## 6.1 The Frankfurt Transportation Network

Public transport in the city of Frankfurt is organised by the Frankfurt Verkehrs und Tarifverbund (FVV). It consists of a number of modes of transport, subways (U-bahn), trams (Straßenbahn), trains (S-bahn) and buses. The transportation network includes over one hundred stations and stops on tens of routes. A model of the network can be constructed from information about the locations of individual stations and the relationship between stations and lines. The data structure constructed in this way can then be used in a number of different route searching algorithms. For example, one for calculating all the shortest routes in the

network from a given starting point. This prototype was originally constructed in Miranda [Joos89b]

The raw data for the network consists only of the transport routes through the city. Further information such as the distance between nodes is specified algorithmically. This information is needed by the route searching algorithms. However construction of a data structure holding the extra information accounts for a significant proportion of total run time which is incurred every time the prototype is executed. Ideally, this data structure should be calculated only once, incrementally and lazily, since it does not change from one run of the system to the next nor is all of it needed on every run and indeed some of it may never be needed at all.

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. The Miranda implementation of this prototype was found to be spending much of the execution time evaluating the matrix. The network matrix takes a significant amount of time to compute (71815 function applications) if evaluated completely. During any session, of course, only the part of the structure which is needed is evaluated. Any part of the structure which is computed during a particular session has to be recomputed during every other session in which the value is required. By porting the implementation to Staple and using persistent modules, it is possible to avoid the need to recompute any part of the network matrix.

### 6.1.1 A Provisional Expedient

Before looking at the persistent module implementation, an attempt which was made to avoid recomputing the matrix between sessions is described. This highlights the difficulties with non-persistent functional programming systems.

A stopgap solution which was adopted when Miranda was used to develop the Frankfurt network was to evaluate the data structure and print it to a file in the form of a Miranda definition. This file is then compiled and used as a Miranda script. This technique removed the overhead of generating the data structure more than once. The following example shows how this was done:

```
fVVnet = ...        || definition of a huge data structure
output = "fVVnet = " ++ show fVVnet
                    || definition of 'output' - the string
                    || to be stored on file
```

The string `output` contains Miranda source code representing the fully evaluated structure `fVVnet`. During subsequent interactive sessions, having compiled this source code, expressions can use this data-structure without having to re-evaluate its components. This technique does not solve the problem completely, however, since compilation of the new definition generates code to build the network data structure, but this code has to be executed every time the prototype is run. Consequently instructions must be executed to construct the data structures each time. An even more serious problem from an efficiency viewpoint is that sharing is lost. The sharing that is inherent in the Frankfurt network means that the overall size of the data structure when sharing is present is not great. The flattening process, however, causes all sharing to be discarded with source code being generated many times for the same value. This yields a severe penalty in space and time performance of subsequent evaluations.

There is an ad-hoc solution to this problem as well. Provided that the programmer has particular knowledge of the data structure which is being flattened and that the structure is of a suitable kind (see below), it is possible to write a formatting function to generate code which will recreate the shared structure by using auxiliary definitions. Instead of printing the coordinates and other details of stations as:

```
fVVnet = [ ("456231","335210",5,"u1","456231")
         , ("335210","456231",13,"u1","556231")
         ]
```

it is possible to generate

```
fVVnet = [ (s456231,s335210,5,u1,s456231)
         , (s335210,s456231,13,u1,s556231)
         ]
         where
           s335210 = "335210"
           s456231 = "456231"
           s556231 = "556231"
           u1      = "u1"
```

Being applied systematically on a large scale, this technique would allow ad-hoc, user programmed sharing to be preserved, but the cost of doing all this in terms of programming effort and complexity would have increased considerably in the meantime. Writing a function to generate the textual representation of the data-structure is significantly more difficult . This task would have to be repeated from scratch for every kind of topology which was required. In addition, it is a task that has nothing to do with the problem being solved. Users of lazy functional programming languages should not have to expend effort in performing such transformations by hand. Systems exist (e.g. PGraphite [Tarr89]) though not functional ones, where similar transformations have been automated. This kind of problem is common to large scale software development [Parn79]. Many applications need to build large data-structures which should only be constructed once and persist thereafter.

### 6.1.2 A Persistent Implementation

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

- 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

The module called `fVVmain` contains a definition of the Frankfurt network data structure itself

```
fVVnet = reduce [] expose net
```

which defines the network. The functions `reduce` and `expose` are used to transform the raw data (the value `net`) into the required matrix form. The module is loaded into the store using the mkmodule command and a definition of the environment in which to compile the module

```
mkmodule fVVmain  fVVfrankfurt fVVgraph ... prelude
```

The route searching functions are presented initially with an environment in which `fVVnet` is completely unevaluated. After each execution of a function which uses `fVVnet` , more of the structure may become evaluated. Eventually it is possible that all of the structure becomes evaluated, but only if all of it has eventually been required to compute a succession of results.

The network matrix is available for use every time a new route searching algorithm is to be tested without need for any re-evaluation.

The Staple approach has some advantages over the original Miranda version.

- Evaluation performed on parts of the data structure is never repeated even in different sessions. That is the persistence ensures that the evaluated data structure is retained in its evaluated form.

- The data structure is evaluated lazily so only values which are needed to compute the result are evaluated. This is in contrast to the ad-hoc solution of flattening the entire structure.

## 6.2 Functional Database Programming With Stream Persistence

This Section describes an experiment in which Staple was used to implement a number of database style programming tasks. Database programming languages are ones which support the creation and manipulation of large bodies of structured data. Here, an attempt is made to show how a persistent lazy functional programming language, Staple, can be used in database type applications.

### 6.2.1 Introduction

Atkinson and Buneman [Atki87] described four programming tasks which they consider illustrative of the problems in database programming. The tasks are typical of those in a manufacturing company's parts database. These are

#### 6.2.1.1 Describe the Database

This task can be compared with providing the database schema in traditional databases. In functional languages the type system is rich enough to allow certain kinds of database schema to be specified. In addition, there is generally a need to support conditions on the data which are implicit in some data models [Thom85], or are explicit as part of the type system (eg [Reyn83]) or as automatically enforced integrity constraints.

#### 6.2.1.2 Print Details of Expensive Imported Parts

This task is to print the names, costs and mass of all imported parts that cost more than £100

This is the type of task which database query languages are specifically designed to make easy. A good measure of programming languages in general is how easily queries such as this can be expressed.

#### 6.2.1.3 Print Total Cost and Mass of a Composite Part

The structure of a part may be complex in that other parts are used in its construction. Those other parts may also be composite parts. Thus the total cost and mass of such a part depends on the numbers, costs and masses of the component parts. This task requires a recursive traversal of the part structure. Database query languages have traditionally been unable to do transitive closures of this kind. In addition, there is an interesting efficiency problem associated with this task which is to avoid recomputation of the cost and mass of a part used as a component of several other parts.

#### 6.2.1.4 Record a New Manufacturing Step

This task is to record a new manufacturing step in the database, i.e. how a new composite part is manufactured from sub-parts.

This task is meant to show how and where, in the program or type system, integrity constraints are implemented. Recording a new manufacturing step involves inserting new data into the database. This task exemplifies the ability of the programming system to support such updates as atomic transactions. Support for verifying constraints may be automated or left to the programmer who may construct an ADT in which the constraints are encapsulated and are therefore inviolable.

### 6.2.2 The Tasks in Staple

Staple can be used to code all the above tasks. In particular, use is made of the stream model of persistence. It should be noted that the database schema and each of the individual tasks can be implemented as different modules (and indeed were for this experiment).

#### 6.2.2.1 Describing the Database

In general, describing the database schema is a very complex task. Consideration must be given to more than just the structure of the data, but also to constraints on the structure and constraints on values stored in the database. The database in this example can be modelled fairly simply, although no integrity constraints are specified here (see Section 6.2.3). A database will consist of a list of parts and a list of suppliers aggregated as a pair.

```
type Database = ( [Part] , [Supplier] )
```

A part can be either a basic part or a composite part. For this we use an algebraic data type.

```
data Part = Basic
                [Char]      -- the part's name
                [UsedIn]    -- what it's used in
                Int         -- the cost of the part
                Int         -- the mass in grams
                [Supplier] -- who can supply it
          |
          Composite
                [Char]      -- the part's name
                [UsedIn]    -- what it's used in
                Int         -- the cost of assembly
                Int         -- the mass of the glue
                [Uses]      -- what parts are used

type UsedIn = [Char]        -- super-part name
```

```
type Uses = (Int,[Char])    -- number, component part name
```

Suppliers should contain information such as their reliability delivery times etc., but for simplicity this will be simplified to just a name.

```
data Supplier = Supplier [Char] -- etc.
```

It is common for database schemas to include attribute names for record fields. These correspond to selector functions in the model described here. Some example selector functions are given as follows.

```
name :: Part -> [Char]              -- the name of a part
name (Basic n _ _ _ _) = n
name (Composite n _ _ _ _) = n

cost :: Part -> Int                 -- the cost of a part
cost (Basic _ _ c _ _) = c
cost (Composite _ _ c _ _) = c
```

These functions are usually unnecessary with pattern matching languages since the selection can be done implicitly. They may be of use, however, in some situations, hence their inclusion in the description of the database.

It is also common to include some predicate functions which determine which of a set of object types a data value represents. The selector functions for parts can be defined as follows

```
isBasic :: Part -> Bool
isBasic (Basic _ _ _ _ _) = True
isBasic part              = False

isComposite :: Part -> Bool
isComposite (Composite _ _ _ _ _) = True
isComposite part                  = False
```

This completes the description of the database schema. The type systems typically found in functional languages are capable of describing simple database schemas of

this kind. More realistic database schemas would include integrity constraints as well, these are discussed later in Section 6.2.3.

Before task 2 is considered, some functions to create and manipulate a database are required. Unlike non-persistent languages which would require code at the beginning of each program to re-construct the database from some stored representation (usually a file), it is possible to use the streams model of persistence to access the database structure directly. For this, some additional functions are required.

Firstly to create a new empty parts database which will be called "Parts".

```
newDatabase :: [Response] -> [Request]
newDatabase resps = [Insert "Parts" (mkany db)]
                    where
                    db :: Database
                    db = ( [] , [] )
```

The type of `newDatabase` indicates that when evaluation of this procedure is initiated, it is supplied with the responses argument by the interactive system. The type assertion is required so that the correct type is associated with the empty database in the universal object to be stored.

The remaining tasks will require access to the database and for this, a function `getDatabase` can be written.

```
getDatabase :: Response -> Database
getDatabase (Db db) = coerce db :: Database
```

This combined with the request `Lookup "Parts"` will obtain the database from persistent storage.

#### 6.2.2.2 Printing Details of Imported Parts

Print the names, costs and mass of all imported parts that cost more than $100

The tasks which output some information to the user rather than those which perform a transformation of the database can be performed using a generic `doTask` function.

```
doTask :: [Char] -> (Database -> [Char]) ->
                [Response] -> [Request]
doTask dbname task resps
   = [Lookup dbname,AppendChan "stdout" s]
     where
     db = getDatabase (resps!0)
     s = task db
```

The task itself can be coded very succinctly using a list comprehension. The function takes a database and produces a string as its output.

```
task2 :: Database -> [Char]
task2 (parts,_) =
  appendall( layn [ print (name,cost,mass)
                  || part <- parts , cost part > 100,
                     isBasic part ])
```

The function `layn` is defined in the Staple prelude. The function takes a list of strings and produces a list of strings whose elements are the corresponding strings from the argument list with numbers appended to their front. The task uses a function `print` which takes a triple, the name, cost and mass of a part, and produces a suitable string representation of the part.

Note that a more optimal way of writing this query would be to reverse the test for the cost being greater than 100 and the test that the part is a basic part. It is not necessary to perform the cost test on composite parts since they represent parts which are manufactured and not imported. Trinder [Trind89b] describes techniques for generalised query optimisation and shows that as well as their ability to represent relational algebra queries, list comprehensions can be optimised using analogous optimisation techniques to those used for optimising relational algebra queries.

The task itself can be performed by evaluating the following expression.

```
doTask "Parts" task2
```

#### 6.2.2.3 Print cost and mass of composite part

This task requires a scan of the transitive closure of the relation between parts and sub-parts starting at the part being analysed.

```
task3 :: [Char] -> Database -> [Char]
task3 partname (parts,_) =
    = print (cAndM part)
      where
      part = findPart partname parts
      cAndM (Basic name _ cost mass _) = (name,cost,mass)
      cAndM (Composite name _ acost minc uses)
          = (name,cost+acost,mass+minc)
            where
            (cost,mass) = foldr addPair (0,0) components
            components = [ scale (cAndM thePart) n
                         || (n,partname) <- uses]
            thePart = findPart partname parts
            scale (_,cost,mass) n = (n*cost,n*mass)
            addPair (a,c) (b,d) = (a+b,c+d)
```

In a composite part, the total cost and mass of the components is the sum of the list calculated using a list comprehension which calculates the costs and masses of each of the component parts. This task is executed, for example, using the following.

```
doTask "Parts" (task3 "bike")
```

The function `findPart` has not been defined here, but simply locates the part with the given name. The `addPair` function adds two pairs by adding corresponding component elements.

#### 6.2.2.4 Update the Database

Certain kinds of operation simply transform the database into a new form such as adding a new part or changing the cost of a part. This kind of operation can be expressed as a generic `dbAction` function.

```
dbAction :: [Char] -> (Database -> Database) ->
                [Response] -> [Request]
dbAction dbname action resps
    = [Lookup dbname,Insert dbname newdb]
      where
      db = getDatabase (resps!0)
      newdb :: Database
      newdb = action db
```

The fourth task is to record a new manufacturing step in the database. The requires specifying how to add new Basic parts and how a new composite part is manufactured from sub-parts.

```
addPart :: Part -> Database -> Database
addPart part (parts,suppliers) = (part:parts,suppliers)
```

This is all that is required for the simple case of adding a basic part since the generic function `dbAction` defined earlier can be used to perform the update.

```
dbAction "Parts" (addPart (Basic "wheel" [] 23 200 []))
```

However, to add composite parts, the component parts must be updated to record that they are used in this composite part. The `addPart` function can again be used to perform the update.

```
dbAction "Parts" adder
where
adder db@(parts,suppliers) =
    addPart part newdb
    where
    part = Composite "bike" [] 230 1234 [Uses 2 "wheel"]
    newdb = (addUsedIn "bike" "wheel" parts,suppliers)
```

The function `addUsedIn` transforms the database by updating the wheel part so that it contains the information that it is used in the composite part bike.

```
addUsedIn name component (part@(Basic n u c m s):parts)
        = (Basic n (component:u) c m s):parts, n = name
        = part:(addUsedIn name component parts)
addUsedIn name component (part@(Composite n u c m s):parts)
        = (Composite n (component:u) c m s):parts, n = name
        = part:(addUsedIn name component parts)
```

By referring to parts by a string identifier which is independent of the name space of the language, problems with referential integrity can be avoided. The data referred to by a string may change over time, whereas a binding between a name and value is fixed.

If a model had been used where sharing was implicit then complications arise when the data structure needs to be changed. If a part is shared and has to be updated (for example if it is used by a new composite part) then all the references to the shared part must also be updated. General graph transformation is an extremely difficult problem in a purely functional language.

### 6.2.3 Integrity Rules

Integrity refers to the accuracy or correctness of data, usually in a database. It is usually operated by applying some set of rules which are designed to maintain the logical consistency of the database. Integrity constraints are used to describe properties which the data needs to satisfy for that data to be correct. Such constraints are used to prevent non-malicious errors in a system rather than preventing certain users from accessing or modifying the data. A simple example of an integrity constraint might be to ensure that a basic part has at least one supplier. That is someone exists who supplies the part. In addition, integrity constraints can be used to remove duplicate entries if that is required. The errors must somehow be transmitted back to the user and an updating transaction aborted

or some dialogue must be initiated in which the user can override certain constraints.

Certain types of integrity constraint are amenable to implementation in terms of non-free algebras or lawful types. There are potential benefits in locality of application of such constraints in that only the affected parts of the database need be checked.

#### 6.2.3.1 Lawful Types

An interesting approach to providing integrity constraints is to encode the constraints in the functions which construct instances of the database. Early versions of Miranda had lawful types [Thom85] in which the constructor functions could include transforming computations depending on the values of the arguments. These were later removed from the language because of a difficulty with pattern matching semantics, but they can be modelled (excluding pattern matching) by defining ordinary functions which construct values of the algebraic data type. One function is defined for each constructor function. So for example, the following definition will be used for the type of sorted lists

```
data SortedList t = End t | Cons t (SortedList t)
```

This type has no implicit sorting, but if the following two functions are defined to replace the `End` and `Cons` constructors then using only these functions to construct `SortedList`'s will yield sorted lists.

```
end x = End x

cons x xs@(Cons y ys) = Cons y (cons x ys) , x > y
                      = Cons x xs
cons x xs@(End y) = Cons y (End x) , x > y
                  = Cons x xs
```

To use this technique to enforce integrity constraints, the following function might be used to replace the constructor function `Basic`.

```
basic n u cost mass s = error "cost <= 0" , cost <= 0
                      = error "mass <= 0" , mass <= 0
                      = .... , ....
                      = Basic n u cost mass s
```

Many simple integrity constraints can be encoded in this way. There is also the advantage that the constraints are applied locally.

One possible disadvantage is that laziness may delay evaluation of integrity constraints until the modified data is accessed. Whilst the constraints are strongly enforced, they may only be discovered at some arbitrary point in the future when the data is accessed. It may be desirable for integrity constraints to be evaluated strictly.

## 6.3 Conclusion

The Staple persistent module system can be used to avoid the need to recompute data which is to be used by many programs. The Frankfurt Network benefits by not having to recompute the network matrix itself more than once. Each graph traversal program can access the structure and need only evaluate parts which have not been evaluated previously. Such evaluation benefits any program which later requires that part of the network.

Staple can be used to implement the four tasks specified in Atkinson and Buneman's paper. The fourth task, however, can only be said to be partially completed since no account is taken into consideration of concurrency and transactions. Database applications are, however, more easily constructed given the persistence provided by Staple because the programmer need not implement code to handle the long term storage of the database. The fourth task in particular requires such code to be written in other functional programming systems. Lawful types are an approach which allow local integrity constraints to be easily applied. Global constraints may be more difficult to encode. The laziness which is implicit

in the system may cause problems if integrity constraints are to be applied at an early time.

# Chapter 7

# Conclusions and Future Research

Research into providing support for long term data in lazy functional programming systems has been presented in this thesis. The motivation for this work has been to reap the benefits of integrating lazy functional programming languages and persistence. The benefits, realised by the development of models for persistence in lazy functional programming systems, are fourfold.

The programmer need not write code to support long term data since this is provided as part of the programming system. This means that a programmer can concentrate all efforts on the problem to be solved.

Persistent data can be used in a type safe way since the programming language type system applies to data with the whole range of persistence.

The benefits of lazy evaluation are extended to the full lifetime of a data value. Whilst data is reachable, any evaluation performed on the data persists. The evaluation state of values changes over time from completely unevaluated towards completely evaluated through many intermediate stages. Whichever state of evaluation an object possesses, it never becomes less evaluated.

Data intensive applications such as functional databases can be developed.

## 7.1 Models for Persistence Lazy Functional Programming Systems

In this thesis, two models have been developed.

- Persistent Modules
- Stream Persistence

An architecture which is designed to support PLFPS has been described. The implementation of the architecture consisted of

- The design of an abstract machine to support persistent lazy functional programming
- An architecture combining an interactive programming environment, a compilation system, abstract machine interpreter, run time support and persistent object store.

The resulting system (Staple) supports persistent lazy functional programming. Use of persistent modules and stream persistence improve prototyping and functional database applications.

This section summarises the two models for persistence and their realisation in the form of an abstract machine design and system architecture for persistent lazy functional programming. The section ends with a discussion of some applications of the Staple system.

### 7.1.1 Persistent Modules

The first model for persistence described in this thesis is persistent modules. Persistent modules provide persistence in a simple way. Functional programs usually consist of an expression to be evaluated in some given environment. The environment consists of a set of definitions often grouped together into modules. Persistence is provided by allowing environments to be stored for later reuse. The language model itself remains the same as with other lazy functional languages.

Lazy evaluation ensures that shared suspensions are computed at most once. By ensuring that sharing is preserved over store operations, this property is retained. The object store itself guarantees that sharing is preserved.

Persistent modules only permit update by recompilation of a module. The update is part of the programming environment rather than in the language. Because no program can refer to a mutable location, referential transparency is preserved.

### 7.1.2 Stream Persistence

Stream persistence allows programs to be written which interrogate and update a persistent store. Values are made to persist using stream persistence by issuing a request to add an association between a string and a value to the persistent store. A request can be issued to obtain the value associated with a string. These requests are issued within a stream processing model. The programming system is considered as a black box which processes requests generated by the program. The program receives back a list of responses from the programming system (See Figure 7.1)

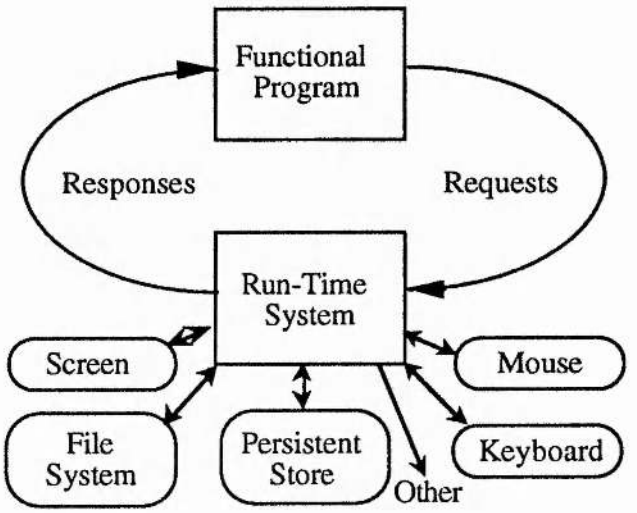



**Figure 7.1 - Stream I/O and Stream Persistence**

The system supports requests to insert, delete and lookup values in persistent storage. The value returned by a lookup request is always of the dynamic type Any. This type is an infinite union. Values may be projected out of the type with an associated dynamic type check which ensures strong typing within the system. Values of type Any are created by an injection operation.

The response list is a lazily created list which contains a single response for each request which has been processed. The values which are obtained with lookup requests are returned as part of the response list. The program can access responses in the list as required. If an association is updated, any value returned as the response to a previous lookup request on the same string key remains unaltered. The responses do not refer to the mutable locations which exist in the store, but to the value itself. In this way, referential transparency is preserved.

Any data value can be injected into the infinite union type and made to persist. The means of identifying persistent objects is by reachability from the values associated with strings in the persistent store. Programs never refer to the persistence of the data values they manipulate. Staple with stream persistence still has orthogonal persistence and referential integrity is maintained.

### 7.1.3 The PCASE Machine

The implementation strategy for the Staple system was to use existing object store technology and to design an abstract machine for functional programming which would use the object store as its only data space. The resulting machine (PCASE) is designed to support persistent lazy functional programming with persistent modules and stream persistence. The machine supports the execution of functional programs with the usual lazy functional language features of pattern matching, algebraic data types and higher order functions. The machine architecture is shown in Figure 7.2.

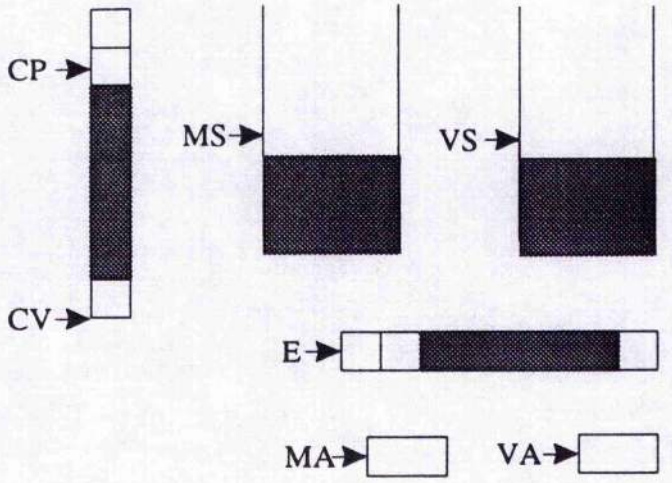



**Figure 7.2 - The PCASE Abstract Machine Architecture**

The PCASE machine incorporates a mechanism for constructing persistent module representations. This is done by a machine instruction which takes values from the machine stack and builds an environment for the module. The environment contains the values exported by the module. The machine also supports use of a dynamic type by providing an instruction for injecting a value with a given type and one for projecting a value onto a specified type. The projection instruction involves a run-time type check which is optimised to an integer equality check. All data values created and manipulated during program execution are created using the persistent object store interface functions.

The environment part of the PCASE machine contains the value of non-local variables. Only those non-locals which may be needed during the evaluation of an expression are present. This results in less objects being retained than with methods using a static chain of stack frames.

### 7.1.4 The System Architecture

The Staple system architecture consists of

- A programming environment

- A compiler for the Staple language
- A run time system which drives the evaluation of functional programs
- An interpreter for the PCASE abstract machine
- A persistent object store

These components are shown diagrammatically in Figure 7.3.

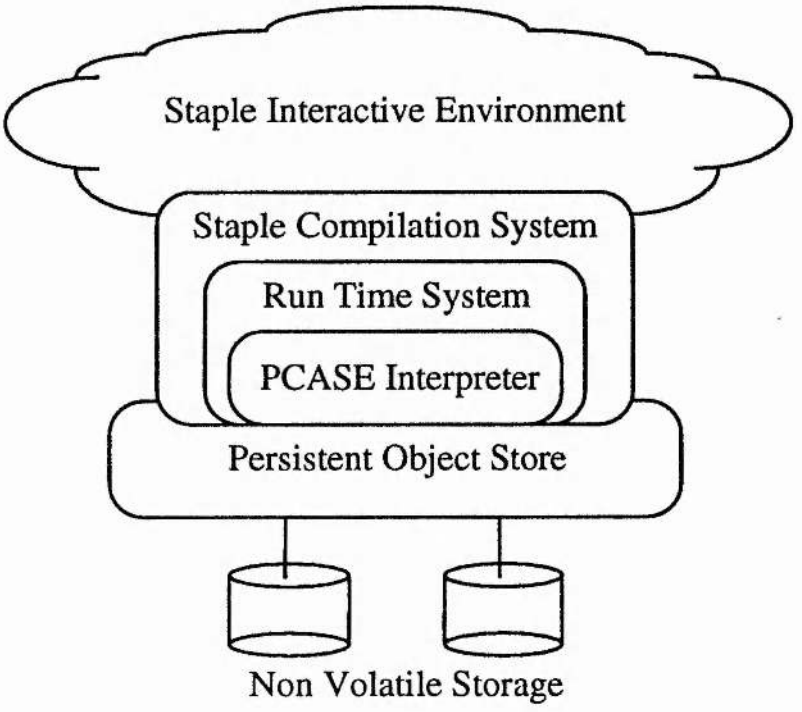



**Figure 7.3 - The Staple System Architecture**

The interactive environment allows expressions to be entered and evaluated. In addition, the interactive environment allows modules to be compiled and loaded into the persistent store.

The compilation system translates Staple expressions and modules into PCASE abstract machine code. This provides the language interface to the programming system. Module translation is novel in that the translated program is an expression whose value is the module's environment. The translated program can simply be executed by a normal call to the evaluation engine and when execution terminates,

the module environment can be found on top of the main stack. The compilation also detects when a dialogue expression is entered and begins stream processing by automatically supplying the response list to the dialogue function before starting execution.

The run-time-system drives the evaluation of Staple programs. Evaluation of expressions is print driven in the usual way. Persistent requests are processed by the run-time-system which is responsible for maintaining the stream persistence mapping between strings and values.

### 7.1.5 Applications

Persistent lazy functional programming has been used in two areas:

- Prototyping
- Towards Functional Database Modelling

The Frankfurt transportation network was represented as a large graph structure and experimental algorithms written to perform shortest path analysis on the graph. With persistent modules, it was possible to store the graph and reuse its value many times. The graph would eventually be completely evaluated as each node in the network was visited. Thus the graph was evaluated lazily, but with the advantage that evaluation was not lost at the end of a session which had been the case with a non-persistent Miranda implementation.

Some illustrative database programming tasks suggested by Atkinson and Buneman [Atki87] were implemented using persistent modules and stream persistence. A database of parts was stored in persistent storage using stream persistence. The tasks included printing the total cost and mass of a composite part which required a recursive traversal of the part structure. This task is not usually possible with relational query languages.

These examples, it is believed, illustrate that the models for persistence described in this thesis can be used for practical benefit in prototyping and functional database programming. There are undoubtedly other application areas which can benefit from Staple's persistence.

### 7.1.6 Concluding Discussion

The persistent modules model for persistence was developed because it presents an extremely simple mechanism which can be easily learned by a programmer. Indeed, no new language mechanism is involved – the composition and construction of programs is essentially the same as a non-persistent system with modules. Whilst more powerful module systems exist, they are more complicated. The major difference is in the evaluation strategy which takes advantage of the evaluation state of persistent values.

Stream persistence was developed because persistent modules (developed first) were not sufficient to write interactive programs, or to allow programs to change the contents of the persistent store during their execution. Streams themselves are a very elegant I/O model [Huda88]. They are simpler to understand than continuation models for functional I/O. A dynamic type is used in other persistent systems to avoid the need to completely specify the store contents and seemed a natural choice to integrate with the stream model of I/O.

The Staple system was constructed using some existing technology - the postore object store. Implementing an object store is a major undertaking and would have detracted from the work presented in this thesis. Staple's abstract machine architecture was heavily influenced by earlier work on the non-persistent CASE machine [Davi89b]. These provided a good springboard from which to develop the models described in this thesis. The PCASE is more efficient in its space usage because it retains the flat environment of the CASE machine which ensures that there are no references to values which will never be needed.

It is intended that this thesis allow another researcher to understand what developing a model for persistence in a functional language entails. In addition, he should gain some insight into how to implement models by studying the approach used for persistent modules and stream persistence. It is hoped that these models can be extended and built upon.

## 7.2 Future Research

There are undoubtedly a large number of directions which could be taken following on from the research presented in this thesis. This section discusses two possibilities. They are

- A method for recovering storage by replacing values with their suspensions
- An extension of the interactive environment to allow interactive creation of modules.

Two other possible areas of interest, are

- An integrated model of persistence to support flexible, incremental binding mechanisms.
- A concurrency model to allow concurrent access by multiple users.

### 7.2.1 Unevaluation

In persistent lazy functional languages, objects which persist and are frequently used will over time gradually become more and more evaluated. In addition, other objects may persist which are used only once, but remain reachable for a long time (potentially for ever). Furthermore, there are situations in which the code generated by compilers leave objects reachable which will never be accessed again (this latter case is known as a space leak). There have been a number of proposals for solving space leaks in non-persistent language implementations. These

techniques can help alleviate the problem of an over-full heap in a persistent system, but can not deal with the problem of space being occupied by reachable and usable data which is accessed infrequently.

Unevaluation is an ad-hoc approach to the problem of space leaks and involves overwriting a value with the suspension which created it. During execution, each time a suspension is evaluated, a copy of the suspension is made and stored together with a pointer to the original which will be overwritten by the value of the suspension. This value may become large over time. The (suspension,value) pairs form a potential unevaluation list. Over time, large data structures may be generated from small amounts of suspension code. The store may be less full with the suspensions than with the data. When available persistent storage space becomes low, it may be pragmatic to replace a value with its suspension. This could be achieved by scanning the potential unevaluation list for suitable candidates and replacing references to the value with references to the suspension. One possible approach would be to overwrite the value, in place, with its suspension.

There appear to be two main difficulties with this approach. Firstly, the potential unevaluation list overhead may outweigh the benefits and secondly, there is the problem of deciding suitable values to overwrite. Techniques similar to those algorithms found in paging such as least recently used could be used to decide which values to replace. In addition, though, the system would need to avoid overwriting values whose representation requires small amounts of space (e.g. an integer) with a suspension requiring large amounts of space.

### 7.2.2 Creating Modules Interactively

Interactive programming environments allow a greater degree of experimentation with prototypes than is the case with batch systems. The user can simply evaluate test expressions without the need to go through a edit-compile-run cycle. Values

computed interactively may be of interest later in the session or even in subsequent sessions in which case some mechanism for preserving such values is needed.

A mechanism which creates a binding between a name and the value which has just been computed could be provided. The binding could then be placed in a module in the persistent store. A user would create a new module interactively by issuing a command to do so to the interactive system. For example, to create a new module named A, the user would type

```
\make A
```

which would create a new module in the store with no values or types in it. To add a value to the module, the user might issue the command

```
\add defn
```

where `defn` is any staple definition of either a type or a value. An additional command could be provided to allow binding to the value which has just been computed.

```
\addvalue name
```

which would add a binding between the identifier `name` and the most recently computed value. The module creation would be completed if a new `make` command is issued, the session ends or the command `end` is given.

## 7.3 Some Parting Words

Functional programming language research is reaching a certain level of maturity both in terms of linguistic design and in programming language implementation. Such languages are still considered unsuitable for certain kinds of application, however. With support for long term data, a wider range of applications can be built and current application areas can also take advantage of the benefits of such a system. It is hoped that orthogonal persistence will find its way into the standard

set of facilities provided by all programming languages and especially lazy functional ones.

# References


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

[Abel85] Abelson, H., Sussman, G.J. and Sussman, J., *Structure and Interpretation of Computer Programs*, MIT Press, 1985

[Abra85] Abramsky, S., *Strictness Analysis and Polymorphic Invariance*, In Lecture Notes in Computer Science Vol. 217, Springer-Verlag, Ed. Harald Ganzinger and Neil Jones – Proc. Workshop on Programs as Data Objects, Copenhagen, October 1985

[Abra87] Abramsky, S. and Hankin, C., Eds., *Abstract Interpretation of Declarative Languages*, Ellis Horwood Ltd., 1987

[Adam79] Adams, D., *The Hitch Hikers Guide to the Galaxy*, Pan Books, 1979

[Alba85] Albano, A., Cardelli, L. and Orsini, R., *Galileo: a Strongly Typed, Interactive Conceptual Language*, ACM ToDS Vol. 10, No. 2, 1985, pp. 230-260

[Arvi90] Arvind and Nikhil, R.S., *Executing a Program on the MIT Tagged-Token Dataflow Architecture,* IEEE Trans on Computers Vol. 39, No. 3, pp. 300-318, March 1990

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

[Atki84] Atkinson, M.P., Morrison, R., *Persistent First Class Functions are Enough*, Foundations of Software Technology and Computer Science, LNCS 181, Springer-Verlag, 1984

[Atki85] Atkinson, M.P., Buneman, P., Morrison, R., Eds., *Persistence and Data Types – Papers for the Appin Workshop*, Universities of St.Andrews and Glasgow Persistent Programming Research Report 16, 1985

[Atki87] Atkinson, M.P. and Buneman, O.P., *Types and Persistence in Database Programming Languages,* ACM Computing Surveys, Vol. 19, No. 2, June 1987, pp. 105-190

[Augu84] Augustsson, L, *A Compiler for Lazy ML,* In Proc. ACM Symposium on LISP and Functional Programming, Austin, Texas, August 1984

[Back78] Backus, J., *Can Programming be Liberated from the von Neumann Style,* A functional style and its algebra of programs, CACM Vol. 21, No. 8, 1978, pp. 613-641

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

[Bird88] Bird, R. and Wadler, P., *Introduction to Functional Programming,* Prentice Hall, 1988

[Boye84] Boyer, R.S. and Moore, J.S., *A Computational Logic,* Academic Press, 1984

[Breu88] Breuer, P.T., *Applicative Query Languages,* Cambridge University Engineering Department, 1988

[Brow89] Brown, A.L., *Persistent Object Stores,* PhD 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

[Brow90] Brown, A., *Persistent Object Store Library,* Internal Document, Department of Computational Science, University of St Andrews, Scotland, April 1990

[Brow91] Brown, A., Connor, R.C.H., Cutts, Q.I., Kirby, G., Morrison, R. and Munro, D., *Postore: a re-usable persistent object store in the form of C-libraries,* Department of Computational Science, University of St Andrews, 1991

[Brow92] Brown, A.L., Mainetto, G., Matthes, F., Mueller, R. and McNally, D.J., *An Open Systems Architecture for a Persistent Object Store,* in Proc. of the Hawaii International Conference on System Sciences, Vol. II, IEE Computer Society Press, 1992, pp. 766-776

[Bune79] Buneman, P. & Frankel, R.E., *FQL – A Functional Query Language*, In Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, 1979

[Burg75] Burge, W.H., *Recursive Programming Techniques*, Addison-Wesley, 1975

[Burs69] Burstall, R.M., *Proving Properties of Programs by Structural Induction*, Computer Journal Vol. 12, No. 1, 1969

[Burs77] Burstall, R.M., *Design Considerations for a Functional Programming Language*, In Infotech State of the Art Conference. The Software Revolution. Infotech, Copenhagen 1977

[Burs80] Burstall, R.M., McQueen, D.B. and Sanella, D.T., *HOPE: An Experimental Applicative Language*, Proc. 1980 LISP Conference, Stanford, CA, August 1980

[Burs84] Burstall, R.M. and Lampson, B., *A Kernel Language for Modules and Abstract Data Types*, DEC SRC Research Report No. 1, September, 1984

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

[Care86] Carey, M.J., DeWitt, D.J., Frank, D., Graefe, G., Richardson, J.E., Shekita, E. and Muralikrishna, M., *The Architecture of the EXODUS Extensible DBMS*, Proc of the International Workshop on Object Oriented Database Systems, Pacific Grove, California, September 1986

[Chur41] Church, A., *The Calculi of Lambda-Conversion*, Annals of Mathematical Studies No. 6, Princeton University Press, 1941

[Cock84] Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. and Morrison, R., *Persistent Object Management System*, Software Practice and Experience, Vol 14, 1984

[Date77] Date, C.J., *An Introduction to Database Systems*, Addison-Wesley, 1977

[Darl81] Darlington, J. and Reeve, M., *Alice, a Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages*, in Proc. FPCA 1981

[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

[Darl91] Darlington, J., Field, A., Harrison, P., Harper, D., Jouret, G., Kelly, P., Sephton, K. and Sharp, D., *Structured Parallel Functional Programming*, Proceedings of the Workshop on the Parallel Implementation of Functional Languages, Ed. H. Glaser, Southampton, 1991, pp31-52

[Davi79] Davie, A.J.T., *Variable Access in Languages in which Procedures are First Class Citizens*, St.Andrews University Department of Computational Science Report, CS/79/2, 1979

[Davi81] Davie, A.J.T. and Morrison, R., *Recursive Descent Compiling*, Ellis-Horwood, 1981

[Davi89a] Davie, A.J.T., McNally, D.J. and Munro, D., *Informal Specification of a User Interface to the STAPLE Persistent Applicative Programming System*, Staple Project Research Report Staple/StA/89/1, Department of Computational Science, University of St Andrews, Scotland, 1989

[Davi89b] Davie, A.J.T. and McNally, D.J., *CASE – A Lazy Version of an SECD Machine with a Flat Environment*, in Proceedings TENCON '89, Bombay, India, 1989, pp. 864-872

[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

[Davi92] Davie, A., *An Introduction to Functional Programming Systems using Haskell*, CUP, Cambridge Computer Science Texts 27, 1992

[Dear85] Dearle, A. and Brown, A.L., *Private Communication*

[Farb64] Farber, D.J., Griswold, R.E. and Polonsky, I.P., *SNOBOL – A String Manipulation Language*, Journal of the ACM, Vol 11 (1), pp 21-30, 1964

[Frae22] Fraenkel, A., *Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre*, Math. Ann., Vol. 86, 1922, pp. 230-232

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

[Gord78] Gordon, M., Milner, R., Morris, L., Newey, M. and Wadsworth, C., *A Metalanguage for Interactive Proofs*, In Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages, ACM, 1978, pp. 119-130

[Hank90] Hankin, C., *Abstract Interpretation of Term Graph Rewriting Systems*, In Functional Programming, Glasgow 1990, pp 54-65, Springer Verlag, London, 1991

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

[Hend82] Henderson, P., *Purely Functional Operating Systems*, in Functional Programming and its Applications, Eds. Darlington, J., Henderson, P. and Turner, D.A., Cambridge University Press, 1982

[Hend86] Henderson, P., *Functional Programming, Formal Specification, and Rapid Prototyping*, IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986, pp. 241-250

[Heyt91] Heytens, M.L., and Nikhil, R.S., *List Comprehensions in Agna, A Parallel Persistent Object System*, In LNCS 523, Springer-Verlag, 1991, pp. 569-591

[Huda88] Hudak, P. and Sundaresh, R., *On the Expressiveness of Purely Functional I/O systems*, Technical Report YALEU/DCS/RR-665, Department of Computer Science, Yale University, 1988

[Huda92] Hudak, P. and Wadler, P. (Eds.), *Report on the Programming Language Haskell*, SIGPLAN Notices, 27,5, also Glasgow & Yale Universities, August 1992

[Hugh84] Hughes, R. J. M., *The Design and Implementation of Programming Languages*, Technical Monograph PRG-40, Oxford University, July 1983, pub. 1984

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

[IBM78] IMS/VS Publications, IBM, White Plains, N.Y., 1978

[Iver62] Iverson, K., *A Programming Language*, Wiley, New York, 1962

[John83] Johnsson, T., *The G-Machine: An Abstract Machine for Graph Reduction*, Proc. Joint SERC/Chalmers Workshop on Declarative Programming, Univ. College London, 1983

[John85] Johnsson, T., *Lambda Lifting: Transforming Programs to Recursive Equations*, In Lecture Notes in Computer Science Vol. 201, Springer-Verlag, Ed. J-P.~Jouannaud --- Proc. Conference on Functional Languages and Computer Architecture, Nancy 1985 --- Also Aspenäs Workshop on Implementation of Functional Languages, Göteborg 1985

[John87] Johnsson, T., *Compiling Lazy Functional Languages*, PhD Thesis, Chalmers University of Technology, 1987

[Jone83] Jones, S.B., *Abstract Machine Support for Purely Functional Operating Systems*, Oxford University Computing Laboratory, Programming Research Group, Monograph PRG-34, August 1983

[Jone84] Jones, S.B., *A Range of Operating Systems written in a Purely Functional Style*, Oxford University Computing Laboratory, Programming Research Group, Monograph PRG-42

[Joos89a] Joosten, S. *The Use of Functional Programming in Software Development*, PhD 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

[Land64] Landin, P.J., *The Mechanical Evaluation of Expressions*, Computer Journal Vol. 6, No. 4, 1964, pp. 308-320

[Land66] Landin, P.J., *The Next 700 Programming Languages*, CACM Vol. 9, No. 3, 1966, pp. 157-166

[Lero91] Leroy, X. and Mauny, M., *Dynamics in ML*, In LNCS 523, Springer-Verlag, 1991, pp.406-426

[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

[Matt92] Matthes, F., Müller, R. and Schmidt, J.W., *Object Stores as Servers in Persistent Programming Environment - The P-Quest Experience*, ESPRIT BRA Project 3070 FIDE Technical Report (1992)

[McCa60] McCarthy, J. et al., *LISP 1 Programmer's Manual*, Cambridge, 1960

[McCa78] McCarthy, J., *The History of LISP*, Proc. SIGPLAN History of Programming Languages Conference, SIGPLAN Vol. 13,No. 8, 1978, pp217-223

[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, pp 59-70

[McNa91] McNally, D.J. and Davie, A.J.T., *Two Models for Persistence in Lazy Functional Programming Systems*, SIGPLAN Notices Vol 26, No 5, May 1991, pp. 43-52

[MacQ89] MacQueen, D., Oral Presentation, 2nd International Workshop on Database Programming Languages, Oregon, 1989

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

[Miln91] Milner, R., Tofte, M., *Commentary on Standard ML*, MIT Press, 1991

[Morr79] Morrison, R., *On the Development of Algol*, PhD Thesis, University of St Andrews, 1979

[Morr82] Morrison, R., *Towards Simpler Programming Languages: S-algol*, in IUCC Bulletin, pp130-133, vol. 4, 1982

[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

[Mycr80] Mycroft, A., *The Theory and Practice of Transforming Call-by-need into Call-by-value*, In Lecture Notes in Computer Science Vol. 83, Springer-Verlag, Proc. Int. Symp. Programming, 1980

[Mycr83] Mycroft, A., *Dynamic Types in ML*, Draft Article, 1983

[Naur63] Naur, P., *Revised Report on the algorithmic language Algol 60*, CACM, Vol 6, No 1, pp 1-20, 1963

[Nikh87] Nikhil, R., *Semantics of Update in a FDBPL*, Proc. of the Workshop on Database Programming Languages, Roscoff, France, September 1987, pp 365-383

[Ohor89] Ohori, A., Buneman, P. and Breazu-Tannen, V., *Database Programming in Machiavelli – a Polymorphic Language with Static Type Inference*, in Proc. ACM SIGMOD, May 1989, pp. 46-57

[Parn79] Parnas, D.L., *Designing Software for Ease of Extension and Contraction*, IEEE Transactions on Software Engineering, March 1979, pp 128-137

[Peyt87a] Peyton Jones, S.L., *GRIP – A High Performance Architecture for Parallel Graph Reduction*, In Lecture Notes in Computer Science Vol. 274, Springer-Verlag, (FPCA Portland 1987), pp98-112

[Peyt87b] Peyton Jones, S.L., *Implementation of Functional Programming Languages*, Prentice Hall, 1987

[Peyt91] Peyton Jones, S.L., *Private Communication*, 1991

[Peyt92] Peyton Jones, S.L., *Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine*, JFP 2,2 pp127-202, 1992

[Quin60] Quine, W.V.O., *Word and Object*, MIT Press, 1960

[Rand64] Randell, B. and Russell, L.J., *Algol 60 Implementation*, Academic Press, 1964

[Rees86] Rees, J. and Clinger, W., *Revised Report on the Algorithmic Language Scheme*, SIGPLAN Vol. 21, No. 12, pp37-43, 1986

[Reyn83] Reynolds, J.C., *Types, abstraction and parametric polymorphism*, IFIP 83 (ed R.E.A.Mason), pp 513-523, 1983

[Rich89a] Richardson, J.E. and Carey, M.J., *Persistence in the E Language: Issues and Implementation,* Software Practice and Experience, Vol. 19, 1989, pp1115-1150

[Rich89b] Richardson, J.E. and Carey, M.J., *Implementing Persistence in E*, In proc. Third International Workshop on Persistent Object Systems, Newcastle, Australia, 1989, pub. Springer-Verlag, 1990, pp302-319

[Rose85] Rosenberg, J. and Abramson, D.A., *MONADS-PC: A Capability Based Workstation to Support Software Engineering*, Proc. 18th Hawaii International Conference on System Sciences, pp. 512-522, 1985

[Rose87] Rosenberg, J. and Keedy, J.L., *Object Management and Addressing in the MONADS Architecture*, Proceedings of the International Workshop on Persistent Object Systems, Appin, Scotland, 1987

[Stee75] Steele, G.L. and Sussman, G.J., *Scheme: An Interpreter for the Extended Lambda Calculus,* Memo 349, MIT Artificial Intelligence Laboratory, 1975

[Stee82] Steele, G.L., *An Overview of Common LISP*, Proceedings of the ACM Symposium on LISP and Functional Programming, pp. 98-107, 1982

[Stoy85] Stoye, W., *The Implementation of Functional Languages Using Custom Hardware*, Technical Report 81, Computer Laboratory, University of Cambridge, December 1985

[Stra67] Strachey, C., *Fundamental Concepts in Programming Languages*, NATO Summer School in Programming, Copenhagen, 1967

[Tarr89] Tarr, P., Wileden, J. and Clarke, L., *Extending and Limiting PGraphite-style Persistence*, in proc. Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Mass, 22-27 Sept 1990, pp 74-86

[Thom85] Thompson, S., *Laws in Miranda,* University of Kent at Canterbury Computing Lab. Report 35, 1985

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

[Trin89a] Trinder, P., *A Functional Database*, D.Phil. Thesis, Oxford University, Technical Report PRG-82, 1989

[Trin89b] Trinder, P. and Wadler, P., *Improving List Comprehension Database Queries*, in Proceedings TENCON '89, Bombay, India, 1989

[Turn76] Turner, D. A. and Morrison, R., *Towards More Portable Compilers*, University of St. Andrews Dept. of Comp. Sci., Technical Report TR/76/5, 1976

[Turn79a] Turner, D. A., *SASL Language Manual*, University of St. Andrews Dept. of Comp. Sci., Report CS/79/3, 1979

[Turn79b] Turner, D. A., *A New Implementation Technique for Applicative Languages*, Software Practice and Experience Vol. 9, 1979, pp 31-49

[Turn82] Turner, D. A., *Recursion Equations as a Programming Language*, in *Functional Programming and its Applications* Edited by Darlington, J., Henderson, P. and 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

[Ullm80] Ullman, J., *Principles of Database Systems*, Addison Wesley, 1980

[Whit13] Whitehead, A.N. and Russel, B., *Principia Mathematica*, Cambridge University Press, 1913 and 1927

[Wadl85] Wadler, P., *An Introduction to Orwell (Draft)*, Internal report, Oxford Programming Research Group, 1985

[Wads71] Wadsworth, C.P., *Semantics and Pragmatics of the Lambda Calculus*, Oxford University D.Phil. Thesis, 1971

# Appendix I

```
data Request  = ReadFile [Char] |              -- filename
                WriteFile [Char] [Char] |      -- filename and
                                               -- new contents
                AppendFile [Char] [Char] |     -- filename and
                                               -- addition
                DeleteFile [Char] |            -- filename
                StatusFile [Char] |            -- filename
                ReadChan [Char] |              -- channel name
                AppendChan [Char] [Char] |     -- channel and
                                               -- output
                StatusChan [Char] |            -- filename
                Echo Bool |                    -- on or off
                GetEnv |
                SetEnv [Char] [Char] |         -- name value
                Lookup [Char] |                -- get named
                                               -- persistent
                                               -- value
                Insert [Char] Any |            -- add
                                               -- persistent
                                               -- value
                Delete [Char]                  -- delete named
                                               -- persistent
                                               -- value

data Response = Success |                  -- request succeeded
                Str [Char] |               -- return value
                Failure IOError |          -- error details
                Db Any                     -- persistent value

data IOError  = ReadError [Char] |         -- cant read file
                WriteError [Char] |        -- cant write file
                SearchError [Char] |       -- cant find file
                OtherError [Char]          -- any other error
```