The Knowledge Engineering Review, Vol 6:3, 1991, 223–235

Functional programming languages for AI problem solving

ELEANOR BRADLEY
Artificial Intelligence Applications Institute, University of Edinburgh, UK

Abstract

Many problem domains exhibit inherent parallelism, and parallel systems which capture and exploit this can be used to look for efficient solutions to AI problems. Functional programming languages are expected to be efficiently realisable on fifth generation hardware. A rational reconstruction of AI programming paradigms is used to investigate the programmability and performance of functional languages in this particular area.

Three languages—Standard ML, Hope+ and Miranda[1]—are used in the rational reconstruction, each language being used to implement three applications. Results indicate that functional programming languages have become much more useable in recent years, and have the potential to become useful tools in AI problem solving. A brief annotated bibliography of texts which covers the introduction to, theory and implementation issues of, functional programming languages, is included.

1 Introduction

As conventional computing methods become less able to deal with the kind of problems that are increasingly being faced by industrial and commercial users, (such as increased volumes of information), attention turns to novel and unconventional computer architectures, in particular *parallel* architectures, in the hope that solutions can be found. In order for any progress to be made in this direction a greater understanding and awareness of such architectures and their practical application has to be achieved. The Parallel Architectures Laboratory (PAL) at the Artificial Intelligence Applications Institute (AIAI), University of Edinburgh, aims to increase the understanding and awareness of these novel and parallel architectures within industrial, commercial and academic communities.

Many problem domains exhibit inherent parallelism, and parallel systems which capture and exploit this can be used to look for efficient solutions to AI problems. A methodology for solving such problems involves mappings from the problem to an abstract representation then onto a concrete representation; a programming language. Programming languages which could provide possible concrete representations include the committed choice non-deterministic family of languages (Trehan, 1988) and functional programming languages. Such languages are expected to be efficiently realisable on fifth generation hardware. In order to investigate the programmability and performance of these languages, a plan of work was undertaken by PAL to rationally reconstruct abstract AI programming paradigms using them. Here we report on this rational reconstruction using functional languages.

2 Background

In 1987 the AI Applications Institute at the University of Edinburgh, took delivery of an ALICE machine (Cripps et al., 1987). ALICE was a prototype machine which had been designed to exploit

[1] Miranda is a Trademark of Research Software Ltd.

the inherent parallelism of applicative languages. AI has been proposed as an area of study that could benefit from the use of parallel processing. Bringing the two proposals together, i.e., marrying the inherent parallelism of applicative languages with the possible benefits to AI problem solving from parallel processing, results in another question to be answered; are applicative languages suitable to AI problem solving?

The main development language available for use on the ALICE was Hope (Burstall et al., 1980) with unification (Uhope) (Prideaux, 1987; Glaser, 1987). The development route involved producing a Uhope application on a machine other than ALICE (a Vax station), then compiling it down to a lower level code before down-loading this to run on the ALICE (Bradley, 1988b). A number of application developments were attempted for execution on ALICE (Bradley, 1988a), but limitations of the development route and the language itself resulted in limited success.

The general feeling obtained from the applications development using Uhope was that this particular language system was lacking in a number of features that were required for sensible development, but had they been present the results may have been quite successful. Another dialect of Hope, Hope+ (Perry, 1988) did appear to address a number of these problems, and it was felt that further study using it, and moving away from the ALICE, would be worthwhile. At the same time, the study should be extended to include other functional languages, so widening the rational reconstruction work programme.

3 Study outline

Our study looked at three functional programming language systems; Miranda (Turner, 1985), Standard ML (Wikström, 1987; Harper, 1986; Harper et al., 1986) and Hope+. A small number of application areas were chosen, and implementations would be produced for each of the languages. We expected this to give a common base for comparing the languages. The applications are described in later sections.

3.1 Language systems

For some functional languages there are a number of implementations available. Here are some implementation details of the systems used in this study.

3.1.1 Miranda

Miranda is a commercially available language system from Research Software Ltd, which has so far been used in the areas of teaching and research. The language was developed by Professor David Turner of the University of Kent. The system, which runs under UNIX, is described as "a self-contained interactive programming environment". The version of Miranda used in this study was Release One.

3.1.2 Standard ML

There are a number of implementations of the functional language Standard ML available. The implementation used in this study was the Standard ML Compiler of New Jersey (Version 0.18, 7 March 1988), from AT&T Bell Laboratories. The status of this compiler is "beta test", but can be described as "fairly complete".

3.1.3 Hope+

There are now a number of dialects of the original Hope (Burstall et al., 1980) language. Previously, a similar applications study was carried out using an implementation of Hope with unification (Uhope) (Bradley, 1988a). This highlighted a number of areas where the language was deficient for use in AI problem solving. Most of these deficiencies have been addressed in Hope+ (Perry, 1988).

There are two systems available that use Hope+, neither of which are commercially supported

products. Both are produced by the Department of Computing Science at Imperial College. The first system considered for this study is the "Hope+ transformation and development environment". This is an interpreter-based interactive environment that supports development of Hope+ programs along with program transformation. The other system considered is the Hope+ compiler (Perry & Sephton, 1988). It was the compiler that was eventually used for the study. As mentioned already, this is not a commercially supported product, but it is a serious development compiler.

4 Functional languages

Functional languages are regarded by their supporters as being more concise, and as being much quicker to write programs in than traditional imperative languages. They are high-level languages with a strong mathematical basis, having been influenced heavily in their development by the *lambda calculus* (Church, 1941). This results in an equational form of expressions, that exhibit referential transparency, and thus make it easier to reason formally about the correctness of a program. Since there is no need for sequencing with functional programming languages, there are obviously implications for parallel processing.

Let us take a better look at some features of functional languages and how the languages chosen for the study handle them.

4.1 Modularity

Often, large programming projects require the code to be split into distinct sections in order to make it feasible to complete the coding. On other occasions, programmers often have to re-implement the same functions in a variety of different applications. For these reasons a good system of programming in modules is attractive. To program in modules sensibly there has to be a well-defined way of interfacing them, and of sharing information between them.

There is no module system as such in Miranda. All Miranda scripts are treated equally, but modularity in an application can be achieved by use of the library mechanism. This provides three directives: **%include**, **%export** and **%free**. In Release One of the system the directive %free is not yet implemented. The presence of a **%include** directive in a script makes available to that script the definitions of the %included script (but not vice versa). Any definitions that have been %included into this second script are not available to the first script. If these are required they must be explicitly **%exported** from the script, where they are defined.

Standard ML provides a well-defined module system. It has three central concepts: *structures*, *signatures* and *functors*. A structure is essentially an environment, consisting of bindings such as type or value bindings. A signature represents the type information associated with a structure, providing an interface specification for a structure. A functor maps structures to structures, it defines how one structure is defined in terms of another. Structures and functors are collectively known as a module. Modules are used to provide the standard library of operations, with each structure handling the operations of one or more of the basic types.

Again in Hope+, modules are collections of declarations, identified by the key-word **module**. In order for any of the declarations in a module to be accessed outside they have to be explicitly *exported*. There are three export declarations; **pubtype** (exports types), **pubconst** (exports data constants and constructors), and **pubfun** (exports functions). An exported item can be imported by another module or a program by means of the **use** declaration, which takes the module name as its argument.

All three methods support separate compilation to some degree, thus making the development of large applications simpler.

4.2. Typing and data abstraction mechanisms

The feature of strong typing is being adopted quite widely in functional programming languages. In a strongly typed language all objects—functions, variables and expressions—are given a specified

```
Miranda    string == [char]
           intpair == (int,int)

SML        type name = [char];
           type intpair = int * int;

Hope+      type string == list(char);
           type intpair == (num # num);
```

Figure 1

type. A function cannot be applied to objects of a different type from that specified; if this is attempted it is signalled as an error. Type checking in this manner helps to identify errors at a very early stage of program development.

The three languages in the study are all strongly typed, but take different approaches to the necessity of providing function type declarations. In Standard ML the type of a function is inferred by the system, and there is no requirement on the user to provide it in the first place. At the other extreme, in Hope+ the function type declaration must be provided by the user before the function definition can be accepted by the system. In between these two extremes lies Miranda, which will infer the type of a function for you, but accepts the type declaration if given by the user, using it in the type checking system.

The three study languages have slightly different sets of base types, and also have slightly different means for allowing user-defined types to be introduced. But generally speaking the three languages take a similar approach to introducing new types. The first approach is used to define new names for existing types and combinations of existing types. This method does not introduce a new type it only creates a new name. Figure 1 contains examples of how each language caters for these *type synonyms*.

The first example for Standard ML is given the type name "name" instead of string, since string is an existing base type in the language.

In order to introduce a new data type, a *concrete* or *algebraic* data type has to be declared. An example is shown for each of the languages in Figure 2. A data type declaration generally results in a new type name and constructor functions. Constructor functions differ from normal functions in that when they are applied it results in the creation of a new data object. When a normal function is applied to its argument some computation takes place, and the result is the value of the application.

In Miranda the name of a constructor must begin with a capital letter to indicate that it is a type constructor. This is not a requirement of the other two languages, but adoption of this rule would probably result in more readable code. Functions can be defined that operate on these new data types.

The final way of introducing user defined types is to use an *abstract type definition*. This enables a new data type to be introduced by abstraction from an existing type. Examples of abstract type definitions are given in Figure 3.

There is no equivalent abstract type definition in Hope+. The same results can be achieved by use of modules and explicit exporting of selected types, constructors and functions. The two examples given for Miranda and Standard ML look similar. The Miranda example consists of two

```
Miranda    tree ::= Null | Tip num | Node tree num tree

SML        datatype tree = null
                         |  tip of int
                         |  node of tree * int * tree;

Hope+      data Tree == null ++
                        tip(num) ++
                        node(tree # num # tree);
```

Figure 2

```
Miranda    abstype stack *
           with empty :: stack *
                push :: * -> stack * -> stack *
                isempty :: stack * -> bool
                top :: stack * -> *
                pop :: stack * -> stack*

           stack * == [*]
           empty = []
           push a x  = a:x
           top (a:x) = a
           pop (a:x) = x

SML        abstype 'a STACK = Stack of 'a list
           with val empty = (Stack nil)
                fun push a (Stack x) = Stack (a::x)
                fun isempty (Stack nil) = true
                  | isempty (Stack _) = false
                fun top (Stack(a::x)) = a
                fun pop (Stack(a::x)) = (Stack x)
           end;
```

Figure 3

parts: the first is a declaration of the interface between the abstract data type and the rest of the script, with the second being the implementation equations. The Miranda type checker handles the implementation equations and the representations as the same type, but to the rest of the script they are completely different, unrelated types. In the Standard ML version the objects of type "STACK" are represented by an existing type, the **representation** type. There are also a number of operations on this new type declared. It is only through these operations that objects of the new abstract data type can be manipulated, the representation of the abstract data type is hidden, it is not exported from the definition.

Abstraction such as this helps to model real concepts more easily. Dependencies on representations can be limited by hiding them in abstract data types. This in turn makes it much easier to change representations without having to alter large pieces of code.

4.3 Higher order functions

Lists are a common data structure in functional programming languages. Many functions can be defined that perform different operations on different types of lists, but they may all have a common pattern of recursion. This pattern of recursion can be expressed using higher order functions (HOFs), thus allowing a collection of functions to be defined dealing with most operations that will be required for working with lists. In other words, HOFs make it possible to define general functions that can then be used in a variety of applications. These HOFs will usually be polymorphic. They can be defined for use with user-defined data types as well as basic system-defined data types. HOFs allow abstraction, and make resulting programs concise and often more readable. A higher order function is one which exhibits one or both of the following properties; it can be given as an argument to another function, or it can be the result of a function application.

Functions only ever take a single argument, although on first reading it may appear that they take more than one. Let us consider some examples (for general examples Miranda syntax will be used). The function plus may be defined as:

```
plus (x, y) = x + y
```

It takes one argument which is a *pair* of numbers. The same operation can be defined with the function:

```
add x y = x + y
```

This appears to take one argument (x) first then the second (y). This is an example of a *partially applicable* function, i.e., a function which when given its first argument returns a more specialized function as a result. The call *add x y* is actually read as *((add x) y)*, but since function application associates to the left, the parentheses are generally omitted. Partially applicable functions are also known as *curried functions*.

A curried function will yield a function as its result, but much of its power comes from its ability to take other functions as arguments. It is this property that enables the common recursion patterns mentioned above, and other patterns of computation, to be captured. The order in which arguments are presented to a curried function is important. Generally, the functional arguments should be given before the non-functional arguments.

If we now move on to consider how each of the three study languages handle currying, we can use probably the best known example of a higher order function *map*. Map is provided as a standard list operator in most modern functional languages. It takes as its arguments a function and a list, and applies the function to each element of the list (see Figure 4 for its representation in each of the three study languages).

```
Miranda   map :: (* -> **) -> * -> **
          map f [] = []
          map f (x:xs) = f x : map f xs

SML       map = fn : ('a -> 'b) -> 'a list -> 'b list
          map f nil = nil;
          map f (x::xs) = f x :: map f xs;

Hope+     dec map : (alpha -> beta) # list(alpha) -> list(beta);
          --- map(f,nil) <= nil;
          --- map(f,x::xs) <= f(x) :: map(f, xs);
```

Figure 4

At this point the differences between the languages are mainly syntactic. Each version has equations for two cases (the empty list and the non-empty list), with the definition for the non-empty list being a recursion over the list. The type of the function has also been shown for each language, though the necessity for type declarations differs between the three (see section on typing and data types). Map can be used to produce a non-recursive version of a function which previously would have been defined recursively, e.g.,

```
inclist [] = []
inclist (x:xs) = x + 1 : inclist xs
```

This is quite a trivial example which takes a list of numbers and increments each element by a value of one. Using the function map it can be rewritten as:

```
inclist (x:xs) = map inc (x:xs)
```

Where *inc* is defined using local definitions or at the same level as inclist, as a function in its own right. The three languages differ in how they deal with local definitions. In Miranda, *inc* can be defined in either of the following ways:

```
inclist (x:xs) = map (+1) (x:xs)
```

or

```
inclist (x:xs) = map inc (x:xs)
                 where inc = (+1)
```

This form of partial application of operators in Miranda produces what are known as *sections*. In the case of Standard ML, use could be made of the function "add" and its partial application;

```
fun inclist (x::xs) = map (add 1) (x::xs)
```

or

```
fun inclist (x::xs) = map inc (x::xs);
                      where inc = (add 1);
```

There is no real need to give a name to the result of a partial application, (remember it is a function), it can be used directly.

The situation is slightly different in Hope+, where use is made of lambda expressions. Although modern functional languages have been heavily influenced by the lambda calculus, Hope+ is the only language in our study that makes direct reference to lambda expressions.

```
--- inclist (x::xs) = map (lambda x => x + 1) (x::xs);
```

or

```
--- inclist (x::xs) = map inc (x::xs)
                     where inc == lambda x => x + 1;
```

In all cases the function **inc** could also be defined at the same level as **inclist**, if it were a more complex operation being performed on each element of the list. The same result can be achieved in each of the languages by slightly different means, with Miranda and Standard ML making use of currying, and Hope+ making use of lambda expressions.

4.4 Input, Output

In a pure functional language there should be no side effects; this causes problems in terms of providing I/O facilities. Not only is any I/O operation regarded as a side effect, but with functional languages synchronization also has to be considered due to the evaluation order. A number of proposals have been made as to how I/O can be achieved in functional languages, some of which ensure that referential transparency is maintained, others which do not. This section considers how the three study languages have approached the subject of I/O.

The Miranda language system is implemented on UNIX, and provides a "UNIX/Miranda system interface". This allows input to and output from UNIX files. These facilities can only be called at the system command level; they are not valid in a Miranda script.This approach ensures that Miranda scripts remain referentially transparent, but provides only limited I/O possibilities.

The Standard ML approach to I/O is to use **streams**. A set of basic I/O primitives is provided by the system, and these can be used within functions. These primitives include two basic types of stream; an input stream and an output stream. There are operators for opening input and output streams, along with a group of operations on input streams and a similar group for output streams. The system also provides two standard streams, one for input and one for output, normally connected to the user's terminal. Functions that include these I/O primitives do not have any referential transparency, and are referred to as **commands**. Including commands such as these in a Standard ML program means that it cannot be reasoned about formally. If the commands were to be placed in a separate module (file) then the rest of the code, as long as it is referentially transparent, can be reasoned about.

Hope+ takes yet another approach to I/O in a functional language; it uses **result continuation** (Perry, 1987). If the operating system/hardware combination of a computer is regarded as the *system*, then a program can be looked on as a function invoked by the system. Therefore, a continuation is the name given to a result value of a program. The continuation itself does not perform any operation, but depending on its value, the system will perform one. A continuation is a compound data value consisting of an operation request and a continuation function. The

operation request is interpreted by the system, and the continuation function is the next program that the system will evaluate after the operation status. This approach allows reasonable I/O while at the same time maintaining the referential transparency of the program.

5 Applications

This rational reconstruction of AI programming paradigms used three application areas, each chosen to demonstrate different requirements of AI programming.

For each of the application areas, data representation—often of a quite complex nature—is required. This should be feasible by making use of the user-defined data types available in the functional languages chosen.

In general, a similar approach to each application was adopted across the three languages. It could be expected that the different languages would allow slightly different approaches to be adopted to a problem area, but the overall approach remained the same.

Each of the subsections below describes the problem and the approach adopted for the application areas.

5.1 Flight adviser

5.1.1 The problem

This application has been used as an example of *rule-based programming* in a course on AI Programming Paradigms at AIAI. It is an interactive application. It should provide the user with two options; display one of a selection of timetables, or find details of a flight based on input from the user. This requires I/O which a *pure* functional language should not deal with, since this is a side-effect.

5.1.2 The approach

Each of the three languages approaches the topic of I/O in a slightly different way. The application requires terminal I/O, but could be expanded later to use file I/O.

The concept of having a top-level function that would be called recursively to provide a continuously running system was used. This top-level function would call another function that displayed information on the screen, then read in instructions from the user, and pass this to another function which would take the relevant action, (i.e., display timetables or find suitable information from the tables), then call the original top-level function before exiting.

5.2 Computer vision

5.2.1 The problem

This application demonstrates an example of a *tree-search* algorithm. This is part of a larger application concerned with the consistent labelling of visual scenes, which was originally written by Sara Hopkins (Hopkins et al., 1989). The use of this application in the study results from a collaboration with a group from the Department of Computer Science at Heriot-Watt University, Edinburgh.

5.2.2 The approach

This application requires a number of user-defined data types representing some quite complex structures. The data type "model" represents a visual scene by way of relationships between "features" (also declared as a data type) of the scene. The application works by comparing a known model (database image) with an unknown model (input image). The input image should be compared with a number of database images to find the one which it most closely resembles. The result of comparing a known and an unknown image is an "association matrix" data object. The tree-search is carried out using the association matrix on the two models.

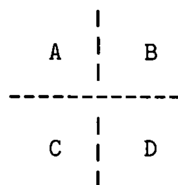

Figure 5

5.3 Map colouring problem

5.3.1 The problem

The Map Colouring Problem (or four colour problem) is an example of the *Generate and Test* paradigm. A solution should provide a map in which the constituent countries have each been assigned one of four colours, such that no two neighbouring countries have the same colour.

A diagram illustrates how countries border with each other (see Figure 5). In this case, country A can be said to share a border with countries B and C, but not with D.

5.3.2 The approach

The general approach to this problem requires user-defined data types to represent maps, countries and colours. The use of the word "map" could cause some problems when using functional languages, because of the function "map" which was discussed earlier.

To represent a country the data type "sector" is defined. This data type holds the following information for a country; its name, the colour that has been assigned to it, and a list of the names of its neighbours. A list of these sectors can then be used to represent a map. To keep the overall solution as simple as possible to begin with, each sector is identified by a number, so no representation for a name needs to be defined. In later versions names could be represented as lists of characters.

The data type "colour" is defined in order to represent the available colours. Five unary constructors are defined for this data type; four colours (red, blue, yellow and green), and one "clear" to indicate that no colour has been assigned yet.

A top level function taking two arguments is defined. The two arguments are a list of "sector" and a list of "colour". Each sector in the list is checked to see if it has had a colour assigned; if not then a list of the colours assigned to its neighbours is compared with the complete list of available colours. The difference between these two lists can then be used to select a colour to be assigned to the sector. The sector is then placed at the end of the list. Since the arguments given to the top level function include a "blank map", and it operates by recursing down the list, once a sector with a colour assigned is found, it can be assumed that all the sectors have had a colour assigned to them and the function terminates.

6 Results

6.1 Completion of applications

In looking at the results of this study we obviously have to consider the relative success and/or failure of each application across the three languages.

6.1.1 The flight adviser

This application provided the most diverse results. One of the reasons for including it in the study was because of its interactive nature. This requirement for I/O resulted in different degrees of success with each language.

In Standard ML the interaction was obtained easily through the use of streams. Streams are not truly functional, so it is probably fair to say that a large portion of this application was non-declarative.

In Hope+ the interaction is achieved using result continuations. This approach allows a similar amount of functionality as in the Standard ML implementation to be obtained, but it is not so simple and easy to program with. The advantage of this bit of extra work is that the application retains its functional nature, and so its referential transparency.

The limitations of the Miranda approach to I/O were demonstrated with this application. If we consider the application as having two parts—display timetables as requested and select possible flights from the timetables dependent on the user input—only the first part was implemented in Miranda. This was done using *magic scripts*, which allow the use of the $- notation within scripts. This is the notation for standard input in command-level expressions. Although the first part of the interaction was implemented using this, it could not be described as an elegant implementation.

6.1.2 Computer vision

The majority of the development work for this application had been done previously, since it was the result of another project (Hopkins et al., 1989). For the Standard ML version the work lay in stripping down the original application to make the *tree-search* algorithm stand alone. This involved ensuring that all the necessary types and functions were included, a task made easier because of the excellent type system.

The Hope+ implementation proved to be relatively successful. A top level expression consisting of a call of the function "expand" compiled without any problems, and the executable code produced was exercised successfully.

Implementing the application using Miranda appears to have been successful. Calls of the function "expand" were exceptionally slow—slow to the point that it never completed, although it did not appear to suspend; it was still not evaluated after an hour of real time processing. Calls made to the underlying functions were all successful but slow, and gave no indication that the top level calls would not terminate if enough time were available.

The size of this application caused problems. In Standard ML the stacksize had to be increased substantially in order for it to execute properly. The heap size in the Miranda system also had to be increased. Unfortunately, an increase in heap size often results in slower responses, as was the case with this application.

6.1.3 Map colouring

This was the most successful application, and all three language versions were executed successfully. Three possible maps were created for it to work on; the simplest containing eight sectors, the next eleven sectors and the third 22 sectors. The possibility of including a function to check if a map is legal in terms of neighbours not having the same colour assigned, first arose when working on the Miranda implementation. This was prompted by the availability of list comprehension. The same facility was incorporated into the other two implementations quite simply, but the Miranda one remains the most concise.

6.2 Ease of use

In discussing the ease of use of the three language systems a number of factors have to be taken into account, including the development environments, support in the form of manuals and user guides, and their performance. We can take these topics in turn and see how each system addresses them.

6.2.1 Development environments

The Miranda system provides a nice working environment. It is simple and easy to use, and makes the development of scripts easy. It takes the form of a command interpreter which can be invoked with a legal Miranda script to give a current environment consisting of the standard environment definitions and the definitions from the current script. There are a number of useful commands in the standard environment which make it quite an interactive system. For development purposes the most useful command is /e. This enables you to edit the current script. The default editor is "vi"

but this can be substituted with another. On quitting the editor the script is recompiled automatically and you are then returned to the interpreter. This editing facility can also be used to edit scripts which are not current. A further command, /f, enables the current script to be changed without having to exit the system and re-invoke it. UNIX shell commands can also be executed from within the interpreter at command level.

The New Jersey implementation of Standard ML used in this study takes the form of a compiler which can run both interactively and in batch mode. Here we have used the interactive mode, which appears similar to the Miranda command interpreter. The most efficient way to develop applications in this mode is to create a file containing the required definitions using an editor, then bring the "use" command into play. This command takes the name of a file, compiling and executing its contents as if they were being typed into the top-level prompt of the system.

Programs written in Hope+ can be compiled to run on UNIX machines by way of the "hc" command. This command acts as a driver program for the various stages of compilation that produce executable code. There are numerous options that can be used to control the compilation of the program (Perry & Sephton, 1988). Developing an application using this system often involves long edit/compile/execute loops, which can become quite tedious. In its favour, this system does allow an application to be split into modules with relative ease. These can be separately compiled, so that if only one module has to be edited, only that module has to be recompiled.

6.2.2 Manuals and help systems

The Miranda system has an exceptionally good on-line manual system. It is menu driven, and even contains a Miranda overview section. This is an on-line version of Turner's (1986) paper. There is also a UNIX manual page describing the "mira" command which invokes the Miranda system.

The support material that accompanies the Standard ML system is not so good. For this version it is still in draft form and incomplete. This is not too much of a drawback, since there is a large amount of published material available on this language, and there is enough information with the implementation to allow reasonable use of the system.

The Hope+ compiler is not a supported product, but the documentation available from Imperial College is of a reasonably high standard. Along with the UNIX manual page for the "hc" command there are a number of documents that explain the compiler, the language and result continuations (Perry & Sephton, 1988; Perry, 1987, 1988). These are necessary for any user new to the system.

6.2.3 Performance

Functional languages have gained a reputation for exhibiting poor performance. Improvements have been made over the years to the point where they are acceptable to work with, but there is still room for improvement. These improvements come from the investigation of compiling and implementation techniques. This is an area where a lot of work is under way.

The three language systems differ widely in the statistics and measurements that can be collected. The Standard ML implementation offers no performance data collection facilities. The Miranda system provides the /count command which returns a set of measurements including the number of reductions, number of cells claimed, number of garbage collections and CPU for each subsequent evaluation. The Hope+ compiler provides an option that results in timing statistics being printed each time the program is executed. These statistics have the labels Real, User and System. Neither system's documentation expand the discussion of these timing/statistics facilities.

Hope+ and Standard ML are both faster in their execution than Miranda. The part which slows down development is the time taken to compile the code in the first place. Once compiled, both languages execute at acceptable speeds. Miranda by comparison is slow, both at compiling the code and executing expressions in the command interpreter.

All three language systems compile their code to some degree. In an attempt to compare their performance, the UNIX command "time" was used to record measurements for the time taken to compile and execute one application in each language. From these crude performance measure-

ments a general trend can be observed; the longer the time spent compiling the code, the less time is taken to execute it. The Hope+ compilation took the longest, but had the fastest execution time. The Miranda compile time was almost equivalent to its execute time. For the application used the difference between the total Standard ML timings and the total Miranda timings was not much, but Standard ML spent more time compiling. As a result, its final execution time was greater. It should be noted that these measurements cannot be used to do a sensible comparison of the three languages systems.

The topic of evaluating the performance of different functional language systems shares many issues with that of Lisp systems, as discussed in Gabriel (1985). A comprehensive study evaluating the performance of functional language systems would benefit from considering many of these issues as a starting point. In collecting the measurements above none of these issues were taken into consideration. To do so would merit a complete study on its own.

7 Conclusions

In terms of the functionality of the applications considered, the three languages were expressive enough. The only area where a language was not sufficient was the I/O handling in Miranda. The expressiveness of the languages provide an excellent media for exploring possible solutions to a problem. The speed at which a piece of code can be written and ready for execution makes it feasible to try a number of approaches to one problem. This makes them ideal as possible prototyping tools. Miranda is already being used for prototyping on a number of sites.

In terms of usable systems, the interactive Miranda system comes out on top. It is very pleasant to work with, allowing easy movement between the command interpreter and editing of the current script or any specified script. Unfortunately, its performance lets it down, but steps are being taken by Research Software Ltd to improve this situation. A close runner up is the interactive mode of the New Jersey Standard ML Compiler. A facility to allow editing of files containing code would improve its usability greatly. The need to exit the interactive system and enter an editor when working on any reasonable sized piece of code is not a great hindrance to application development, but a system similar to Miranda would be an improvement. The least pleasant system to use is the Hope+ compiler. Complete edit/compile/execute cycles take a long time and are very tedious. An interactive system for development work followed by the compilation to excellent executable code would be an ideal system. Such a system would be possible if the "Hope+ transformation and development environment" was used in conjunction with the compiler. At the time of this study it was not possible to do this, since the then current version of the environment was not fully implemented. It is hoped that once the transformation and development environment has been fully implemented it will be a useful tool to use along with the compiler. In terms of producing systems with reasonable final performance and interaction, the Hope+ compiler would appear to be the best choice of development system at this stage.

The conclusion that is reached as a result of this study is similar to one being drawn about the more general use of functional programming languages; namely they are expressive enough to tackle a wide variety of application areas. As more effort is spent on producing efficient implementations and pleasant working environments, their use will become more widespread. A lot of the work done with Standard ML and Hope+, such as compilation techniques and use of result continuations for I/O, indicate that functional languages do have the potential for being used as serious development tools.

Bibliography

There are now a number of good quality texts, covering various aspects of functional programming and languages, available. The texts mentioned in this section are among the most easily accessible ones available. They range from introductions to the concepts of functional programming, and the use of functional languages, to discussions on implementation issues:

Bird, R and Wadler, P, 1988, *Introduction to Functional Programming*, Prentice-Hall.
A tutorial text which uses a notation similar to Miranda; is aimed at undergraduates learning a functional notation as their first programming language.

Field, AJ and Harrison, PG, 1988, *Functional Programming*, Addison-Wesley.
This text is aimed at students of computer science with an existing knowledge of high-level languages, and is of use to researchers and professional programmers. Using the notation of the Hope language it covers theoretical concepts and implementation issues.

Henderson, P, 1980, *Functional Programming: Application and Implementation*, Prentice-Hall.
An early practical introduction to the theory of functional programming.

Peyton Jones, SL, 1987, *The Implementation of Functional Languages*, Prentice-Hall.
This text describes in detail the issues of implementing a high-level functional language using graph reduction.

Wikström, Å, 1987, *Functional Programming Using Standard ML*, Prentice-Hall.
This text is aimed at students of computer science, and introduces the concepts of functional programming using the Standard ML language.

Acknowledgements

This work was funded by a Department of Trade and Industry grant for the "Establishment and Operation of a Parallel Architectures Laboratory" (March 1987) at the Artificial Intelligence Applications Institute at the University of Edinburgh.

References

Burstall, RM, MacQueen, DB and Sanella, DT, 1980, *HOPE: An experimental applicative language. Technical Report CSR-62-80*, Department of Computer Science, University of Edinburgh.

Bradley, E, 1988a, *ALICE progress: Applications study. Technical Report AIAI/PSG123/88*, AIAI, University of Edinburgh.

Bradley, E, 1988b, "An overview of the ALICE system at AIAI" *airing* **5**.

Cripps, MD, Darlington, J, Field, AJ, Harrison, PG and Reeve, MJ, 1987, *The design and implementation of ALICE: A parallel graph reduction machine. Technical report*, ICST.

Church, A, 1941, *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, NJ.

Gabriel, RP, 1985 *Performance and Evaluation of Lisp Systems*, Computer Systems Series, The MIT Press.

Glaser, H, 1987, *ALICE Hope Interpreter User Manual. Technical Report IC/FPR/DOC/4*, ICST.

Harper, R, 1986, *Introduction to Standard ML. Technical Report ECS-LFCS-86-14*, LFCS, University of Edinburgh.

Harper, R, MacQueen, D and Milner, R, 1986, *Standard ML, Technical Report ECS-LFCS-86-2*, LFCS, University of Edinburgh.

Hopkins, S, Michaelson, GJ and Wallace, AM, 1989, "Parallel imperative and functional approaches to visual scene labelling", *Image & Vision Computing*, 7(3) 178–193.

Perry, N, 1987, *Hope+C: A continuation extension for Hope+. Technical Report IC/FPR/LANG2.5.1/21*, ICST.

Perry, N, 1988, *Hope+. Technical Report IC/FPR/LANG/2.5.1/17*, ICST.

Prideaux, T, 1987, *ALICE Hope User Manual. Technical Report IC/FPR/DOC/1*, ICST.

Perry, N and Sephton, KM, 1988, *Hope+ Compiler Release 3.2 (SUN3 UNIX). Technical Report IC/FPR/LANG/2.5.1/14*, ICST.

Trehan, R, 1988, *A Comparison of Committed Choice Non-Deterministic Logic Languages through the Prolog Equation Solving System (PRESS). Technical Report AIAI-PR-13*, AIAI, University of Edinburgh.

Turner, D, 1985, "*Miranda: a non-strict functional language with polymorphic types*" In: *IFIP International Conference on Functional Programming Languages and Computer Architectures.*

Turner, D, 1986, "An overview of Miranda" *SIGPLAN Notices.*

Wikström, Å, 1987, *Functional Programming Using Standard ML*, Prentice-Hall.