Aladin: An Abstract Machine for Integrating Functional and Procedural Programming[*]

Tom Axford[†] and Mike Joy[‡]

16th August 1995

Abstract

Pure functional languages have previously been difficult to integrate with the more common procedural or state-transition style of programming. This paper proposes a simple abstract machine, Aladin, that has pure functional language semantics, with normal-order (lazy) evaluation. Aladin has no built-in primitive functions, however, but operates with primitive functions implemented in any convenient programming language, functional or procedural. The strictness of arguments and results of all primitive functions is specified as part of the definition of the primitive function, allowing very detailed control of the evaluation mechanism. In addition, the Aladin machine is embedded within a state-transition environment that provides for real-time operation and ordered input/output.

Keywords: abstract machine, Aladin, functional language, streams, strictness.

1 Introduction

Non-strict functional languages such as Miranda [1] and Haskell [2] have considerable attractions, as has been argued before [3, 4, 5]. However, they are still relatively little used in practice. There are various reasons for this including (i) implementation inefficiencies for some types of functional programs, (ii) cumbersome ways of handling interactive and real-time input/output, and (iii) the radically different and unfamiliar style of programming needed for functional languages. Functional programming would be more widely used if it was possible to write parts of a program in a functional style while the rest used the traditional imperative style of programming. This has not been easy to do because of fundamental incompatibilities between the two approaches. Although some Haskell implementations, for example, do provide a mechanism for interfacing to routines written in procedural languages such as C, this is a special system-dependent feature and not part of the standard Haskell language. Such mechanisms are not easy to use.

[*]To appear in the Journal of Programming Languages, © COPYRIGHT JOURNAL OF PROGRAMMING LANGUAGES, 1996

[†]School of Computer Science, University of Birmingham, Birmingham, B15 2TT, U.K.
Email: t.h.axford@bham.ac.uk

[‡]Department of Computer Science, University of Warwick, Coventry, CV4 7AL, U.K.
Email: m.s.joy@dcs.warwick.ac.uk

In this paper we suggest an approach that attempts to improve upon this situation and allow an easier and more extensive mixing of the functional and imperative styles of programming. For this to be successful, the benefits of each style must be largely retained: there is clearly no point in mixing the two styles if the result is the worst of both worlds. This is a serious problem, particularly with functional programming where the purity of the language is one of the major attractions of this approach. Pure functional languages have simple mathematical properties, such as referential transparency, which make them well-suited to rigorous program analysis and program transformation techniques. It takes only one small change to the language to lose this purity and consequently lose the advantages associated with it.

The approach we suggest essentially modularises a program in such a way that it is possible to have some modules in a pure lazy functional language, while others are in a conventional procedural language. This requires something different from the usual modularisation in languages such as C++, Modula-2 or Ada. The imperative style makes use of state in the form of variables and data structures whose values change during execution of the program. Very often these state variables are passed from module to module in a conventional modularisation of an imperative program. There is no state in a functional program, so state variables must not be used.

The approach to the design of a functional programming system proposed in this paper differs from the mainstream of lazy functional language development (as exemplified by Haskell) in other ways also. Recent developments in functional programming languages have been mainly towards larger and higher-level languages with sophisticated type systems and easy-to-use syntax; e.g. Haskell, Miranda, Orwell [6] and Ponder [7]). Our approach, on the other hand, focuses on developing a suitable abstract functional language machine which accepts programs in a very low-level language, no more convenient for human programming than is the machine code of any ordinary computer in use today.

This enables us to concentrate on the fundamental evaluation mechanisms and ignore all those language features which can be handled completely by the compiler. What remains is arguably the essence of functional programming, and if this can be conveniently and effectively integrated with the procedural programming model, then it should be relatively straightforward to extend this to the integration of high-level languages in the two styles.

The Aladin abstract functional language machine that we propose also incorporates more user control over strictness than is usual. Functional languages are classed as strict or non-strict. Strict languages such as Lisp (and its many derivatives) and ML [8] evaluate the arguments of a function before applying that function to those arguments. Non-strict languages such as Miranda and Haskell follow the normal-order evaluation rules of the lambda calculus and do not evaluate arguments before applying a function. This can be very inefficient in circumstances where it is not necessary, i.e. where the function is known to be strict. A great deal of research effort has been expended in recent years in devising efficient methods for strictness analysis to enable compilers to determine when it is safe to evaluate arguments before using them. Nevertheless, strictness analysis remains of limited effectiveness, and this is one significant source of inefficiency in non-strict functional languages.

The Aladin machine is fundamentally non-strict, but it allows the user full control over the strictness of all the primitive functions used. The way in which this is done is described in more detail later in this paper.

2 Interfacing Functional to Imperative Programming

Aladin is a programming system which provides a framework for integrating pure functional programming with imperative and real-time programming, while hopefully retaining most of the advantages of both. The Aladin abstract machine uses a pure non-strict functional model of computation based on combinatory logic and the lambda calculus [9]. The abstract machine language of this machine is essentially combinatory logic; i.e. it contains no explicit lambda abstractions (these must be translated into combinator expressions beforehand by the compiler). The primitive functions of this machine are any pure functions (i.e. functions without side effects) implemented in C or any other convenient language (declarative or imperative).

The Aladin programs can also be defined to run in a real-time environment. A single process may be defined to produce not just a single value as its result, but a stream of values which are produced as and when required. This enables fully interactive and real-time systems to be constructed.

Thus the Aladin machine provides two distinct ways in which the imperative or state-transition approach to program design can be incorporated into an Aladin program. The first is by the use of streams, which is described in the next section. The second is by the inclusion of 'primitive' functions which may be implemented using any convenient programming language (such as C). This is described in Section 2.2.

2.1 Streams

Computer hardware designers have for many years used the Mealy machine as a model of a completely general sequential machine [10]. The Mealy machine model is particularly suitable to many aspects of hardware design, but there is no reason why it cannot be applied to software design also. It models any sequential machine as two combinational circuits and a delay (memory) which saves the current state of the machine just long enough to use it in computing the next state of the machine. The Mealy machine is usually expressed as:

$$s' = f(s,x)$$
$$y = g(s,x)$$

where s is the current state of the machine, s' is the next state of the machine, x is the current input, and y is the next output. Of course, each of these may be a vector of values, rather than just a single value. The functions f and g can be represented as combinational circuits in hardware or as functions in a pure functional language (i.e. they have no side effects and depend upon nothing other than their arguments).

For our purposes, it is convenient to ignore the inputs (they can be effectively incorporated into the functions f and g and need not be made explicit[1]):

$$s' = f(s)$$
$$y = g(s)$$

[1]To incorporate the inputs into the functions f and g, the inputs must be regarded as constants (the functions would not be pure functions if they depended on any variables other than their arguments). The inputs are easily treated as constants by considering the whole sequence of input values through time as a single object (an infinite list), even though not all the values in this list can be known at any given point in time. Non-strict functional languages can easily handle such infinite lazy lists.

This model nicely captures the essence of the imperative or state-transition style of programming. Each new value of the state is defined as a pure function, *f*, of the current state, and each new output value is defined as a pure function, *g*, of the current state.

We use this model to define a pure non-strict functional program which can produce a stream of values through time, each value being computed only when required. A stream program has the form:

Transition f g s

where *Transition* is an impure built-in function with the following meaning. When this program is evaluated, the value of *(g s)* is computed and returned, but the evaluation has an important side effect: the program itself is preserved after being re-written as:

Transition f g (f s)

Thus, the second time the program is evaluated it will return a new value and rewrite itself again. The stream of values computed if the program is called repeatedly is:

g s, g(f s), g(f(f s)), g(f(f(f s))), ...

Each value in the stream is supplied only when asked for, rather like a lazy list. The essential difference from a lazy list, however, is that the previous output values are not retained. Only the current state is kept, and hence there is no way of returning to output values computed earlier. They are lost completely. The advantage of a stream over a lazy list is that the stream uses no memory space in saving old values; whereas the lazy list normally keeps all items in the list, so the more items that are computed, the greater the space required. A stream program can produce any type of object as its output, including functions.

See Section 5 for a simple example.

2.2 Primitive Functions

The main interface between functional programs in Aladin and imperative programs in C or other languages is by the use of primitive functions. The Aladin machine language allows new primitive functions to be incorporated into a program at will. The only constraints on a primitive function are:

1. It must be a pure function. That is, it gives a result which varies only with the arguments supplied to the function, and, apart from supplying its result, it has no other effect. This implies that the result of the function does not depend in any way on such things as the past history of the use of this function, the time or circumstances in which it is used, etc. Every time it is called with the same argument values it will give the same result.

2. It must receive its arguments in a specified way and return its result in a specified way. This interface for passing arguments and results will be implementation dependent, but normally will be easy to implement in the commonly used programming languages of the particular computing platform in use (e.g. in C on UNIX systems).

3. It must be available in a specified executable form. Again this is system dependent, but will normally be compatible with the standard linkable binary format of the computing platform being used (e.g. suitable for linking with the *ld* command in UNIX).

4. The number of arguments required must be specified, together with the strictness requirements of each argument and the strictness of the result (see Section 3.2 for a definition of what we mean here by 'strictness', which differs subtley from the usual definition).

5. Provided the function complies with the above requirements, the source code from which it is compiled may have been written in any programming language.

There is no limit to the size or complexity of a primitive function in Aladin. The term 'primitive' is used purely to indicate that as far as the Aladin machine is concerned, it is a single indivisible operation with no internal structure known to the Aladin machine. In fact, primitives may include simple arithmetic operators such as addition and multiplication, higher order functions such as *map* and *reduce*, and even large programs such as compilers or utilities like *grep* in UNIX.

Primitive functions may take any number of arguments, including no arguments at all. Functions with no arguments are just constant data objects, which may represent numbers, characters or any other data (including large data structures).

3 The Aladin Abstract Machine

Aladin is defined as a low-level abstract machine rather than a high-level programming language. The Aladin machine is designed to be a simple and powerful mechanism for computing programs in a pure non-strict functional language. The language is relatively low-level, however, and is not intended to be convenient for human programmers. It contains no syntactic 'sugar' to make it more palatable. As with any machine language, it is not a convenient language in which to write or read programs, and it is assumed that programs would normally be written in a suitable high-level language and then compiled into the Aladin machine language before being run. Suitable high-level languages are not discussed further in this paper.

3.1 Aladin Machine Language

We describe the Aladin machine language in abstract terms. The actual representation of an Aladin program in the computer is as a graph with nodes stored in memory and pointers as the arcs of the graph. The precise details of the concrete representation are implementation dependent and are not described in this paper.

In abstract terms, a program consists of an expression constructed in one of the following ways:

- As a single primitive object

- As the application of a function to a single argument, where both function and argument are themselves any program expressions. (We adopt the usual functional language convention that all functions take a single argument, so functions which we expect to take two or more arguments must be curried [9].)

It is convenient to have a written form for this language, and we choose the usual convention in functional programming of simple juxtaposition to mean function application. So *f x* denotes the application of function *f* to argument *x*.

Function application is not associative in general, so we need to use brackets to distinguish *f (g x)* from *(f g) x*. To avoid the unnecessary proliferation of brackets, we allow brackets to be omitted where implied by the rule *x y z = (x y) z*. Primitive functions and other primitive objects will be denoted by whatever symbols are convenient and familiar (e.g. the symbol '+' or the name *'plus'* could be used to denote a primitive function for addition).

3.2 Aladin Semantics

The Aladin abstract machine (AAM) is a mechanism solely for function definition and function application. The primitive functions and data objects used in Aladin programs are not defined as part of the abstract machine or its language. In other words, any desired set of primitives may be used, and they may be extended at will by the programmer. Each primitive must obey the rules for primitive functions described earlier.

The AAM uses normal order evaluation, modified by local applications of applicative order on arguments specified to be strict.

The denotational semantics of the AAM can be defined by means of an evaluation meta-function $\mathbf{Eval}[\![P]\!]$ which maps programs to programs. It operates on the form of expression of its argument, unlike an ordinary function which operates on the value of its argument. The result of evaluating a complete and correct program should be a program that denotes a known data object (either a simple data object or a complex data structure). More commonly, $\mathbf{Eval}[\![\ ]\!]$ is defined as a function which maps programs to a new domain of 'values'. In the case of Aladin, we do not introduce any domain of 'values' because the primitive functions of the Aladin machine are not defined. The AAM simply defines the way these primitives are combined to form a complete program. Any set of primitives may be used, operating on any set of 'values'.

The evaluation rules for the AAM are as follows, where B^m denotes a primitive function that requires *m* arguments, each of known strictness. The X_i denote any expressions:

$$\mathbf{Eval}[\![B^m]\!] = B^m$$

$$\mathbf{Eval}[\![(X_0 \dots X_n)]\!] = \mathbf{Eval}[\![X_0 \dots X_n]\!]$$

$$\mathbf{Eval}[\![(X_0\ X_1)\ X_2 \dots X_n]\!] = \mathbf{Eval}[\![X_0\ X_1\ X_2 \dots X_n]\!]$$

$$\mathbf{Eval}[\![B^m\ X_1 \dots X_n]\!] = B^m\ Y_1 \dots Y_n, \qquad \text{if } n<m$$

$$\mathbf{Eval}[\![B^m\ X_1 \dots X_m]\!] = \mathbf{Eval}[\![B^m @(Y_1, \dots, Y_m)]\!], \qquad \text{if } B^m \text{ gives a lazy result}$$

$$\mathbf{Eval}[\![B^m\ X_1 \dots X_m]\!] = B^m @(Y_1, \dots, Y_m), \qquad \text{if } B^m \text{ gives a strict result}$$

$$\mathbf{Eval}[\![B^m\ X_1 \dots X_n]\!] = \mathbf{Eval}[\![\mathbf{Eval}[\![B^m\ X_1 \dots X_m]\!]\ X_{m+1} \dots X_n]\!], \qquad \text{if } n>m$$

where:

$$Y_i = X_i, \qquad \text{if } B^m \text{ takes a lazy i-th argument}$$

$$Y_i = \mathbf{Eval}[\![X_i]\!], \qquad \text{if } B^m \text{ takes a strict i-th argument}$$

The metasymbol @ is used to denote primitive function application, so

$$B^m @ (X_1,...,X_m)$$

denotes the result of executing the program for the primitive function B^m with the arguments $X_1,...,X_m$. Every primitive function must have its own evaluation rules to define the result of function application. Any arguments specified to be lazy will be supplied in the form of Aladin programs, while arguments specified to be strict will be evaluated first and the results of the evaluation supplied. The program for B^m must, of course, know what types of arguments to expect and handle them appropriately. The AAM does not, however, know anything at all about data types, and does no type checking. If type checking is required it must be done beforehand by the compiler for whatever high-level language is in use. In this respect the AAM behaves like any ordinary computer programmed in machine code.

The above evaluation rules evaluate an expression to head normal form only, i.e. to a data object or to another expression in which the first (leftmost) function is primitive and requires more arguments than are available. Head normal forms cannot be evaluated further until more arguments are supplied, i.e. by evaluating another program which applies this function.

A program which has been evaluated is always in head normal form:

$$B^m\ X_1\ ...\ X_n$$

where $0 \leq n < m$ or $m=0$. This clearly implements a function which requires $(m–n)$ arguments of the same strictness as the last $(m–n)$ arguments of B^m. Thus it is always possible to determine the number and strictness of arguments required for a program that has been evaluated to head normal form.

3.3 Primitive Functions

All primitive functions are implemented in C or another convenient language which can be compiled into the linkable library format of the host system. Any functions can be implemented as primitive functions. Functions which are obvious candidates for a basic library of primitive functions include the Turner combinators, basic arithmetic operators (addition, subtraction, multiplication, division, etc.); logical and relational operators; trigonometric and other mathematical functions (sin, cos, log, etc.). Of course, some of these may be implemented for more than one type of data representation. Most computers provide arithmetic and relational operators for both integer and real numbers, and similarly, Aladin primitives can be provided for both these representations, and others if needed (e.g. multiple precision arithmetic).

Examples

1. A primitive function for the combinator *S* takes 3 arguments, all lazy. With the lambda calculus definition of strictness, the first argument of S is strict, while the remaining two are non-strict. Our definition of strictness is not exactly the same, however. All three arguments of S need not be evaluated before S is applied (i.e. it is capable to producing a result without evaluating any of its arguments). The result is, of course, a program itself and needs further evaluation, so the result is 'lazy.' The evaluation rule for *S* is:

$\textbf{Eval}[\![S\ X\ Y\ Z]\!] = X\ Z\ (Y\ Z)$

From this definition it is clear that the arguments *X, Y* and *Z* are lazy (need not be evaluated first), and the result is lazy (the resultant expression *X Z (Y Z)* must be evaluated further). We write the strictness of arguments and results using the following notation:

$S :: l \times l \times l \rightarrow l$

2. Primitive functions for the combinators *K* and *I* have the following definitions:

   $K :: l \times l \rightarrow l$
   $\textbf{Eval}[\![K\ X\ Y]\!] = X$

   $I :: l \rightarrow l$
   $\textbf{Eval}[\![I\ X]\!] = X$

3. The program *(S K)* evaluates to itself because *S* requires 3 lazy arguments and only 1 is available. This program is a function that requires 2 lazy arguments (to complete the arguments of *S*) and gives a lazy result (because S gives a lazy result). *S* and *K* are assumed to identify primitives which implement the usual combinators *S* and *K*.

4. Primitive functions for arithmetic operators, such as addition, subtraction, multiplication and division, normally take two strict arguments and give a strict result. They may be implemented to use the standard integer representation on the machine in use, the standard real number representation, or any other desired number representation. The Aladin machine does no type checking, so it is the responsibility of the programmer to ensure that the correct type of arguments are supplied to any primitive function.

5. A primitive function for a conditional operator can be defined as:

   $cond :: s \times l \times l \rightarrow l$
   $\textbf{Eval}[\![cond\ X\ Y\ Z]\!] = \textbf{\textit{if}}\ X\ \textbf{\textit{then}}\ Y\ \textbf{\textit{else}}\ Z\ \textbf{\textit{fi}}$

   Alternatively, an equivalent function can be defined as:

   $cond2 :: s \rightarrow l$
   $\textbf{Eval}[\![cond2\ X]\!] = \textbf{\textit{if}}\ X\ \textbf{\textit{then}}\ K\ \textbf{\textit{else}}\ K\ I\ \textbf{\textit{fi}}$

4 Strictness and Data Structures

The ability to specify the strictness of all arguments and the result of any primitive function makes it possible to create primitives which would otherwise require special modifications to the basic language evaluation mechanism. It is now easy to define primitives which control the order of evaluation, and to define primitives for data structures which are either strict or lazy, or have both strict and lazy features.

4.1 Evaluation Order

The Aladin machine operates using normal-order evaluation as its basic evaluation mechanism, but the following primitives can be defined to force a different order of evaluation when required.

$strict :: l \times s \rightarrow l$
$\mathbf{Eval}[\![strict\ f\ x]\!] = f\ x$

The second argument of *strict* is strict, so forcing its evaluation before *f x* is evaluated. Thus *f* is treated as a strict function, whether it is or not.

$seq :: s \times l \rightarrow l$
$\mathbf{Eval}[\![seq\ x\ y]\!] = y$

The first argument of *seq* is strict, thus forcing its evaluation before *seq* is applied, at which point it simply discards *x* (now evaluated) and returns *y*. The primitive *seq* is thus simply a means of forcing evaluation of any expression before it is needed.

4.2 Lazy Data Structures

The easiest way to create new data structures is to program the primitives in terms of 'markers'. A marker is just an object which has no meaningful value, but is unique and can be distinguished from every other marker. As far as the Aladin machine is concerned, a marker is a function which takes no arguments; it is a constant which evaluates to itself. Markers can be used to distinguish different possible representations of data structures.

The traditional lazy lists that form the basic data structures of lazy functional languages such as Miranda and Haskell are easily implemented in Aladin. The primitive functions required are *cons, head, tail* and isnil, together with two markers (denoted *Nil* and *Cons*) to denote the empty list and a non-empty list. The following definitions provide a simple representation of lazy lists:

$Nil :: \rightarrow s$

$Cons :: \rightarrow s$

$cons :: l \times l \rightarrow s$
$\mathbf{Eval}[\![cons\ x\ y]\!] = Cons\ x\ y$

$head :: s \rightarrow l$
$\mathbf{Eval}[\![head\ (Cons\ x\ y)]\!] = x$

$tail :: s \rightarrow l$
$\mathbf{Eval}[\![tail\ (Cons\ x\ y)]\!] = y$

$isnil :: s \rightarrow s$
$\mathbf{Eval}[\![isnil\ Nil]\!] = true$
$\mathbf{Eval}[\![isnil\ (Cons\ x\ y)]\!] = false$

Notice that the *cons* operator does essentially nothing: it simply returns the expression in which it is applied, with the *cons* replaced by *Cons*. As *Cons* is a marker, it cannot be applied to any arguments and so no further evaluation of this expression is possible. The list is held as a lazy list because both head and tail (*x* and *y* above) are unevaluated expressions.

It is possible to create more efficient implementations for the lazy list primitives by getting *cons* to return a data object which contains pointers to the head and tail of the list packed in a more efficient way. The primitives *head* and *tail* would then need to unpack this box to retrieve these components. However, the advantage of the implementation specified above is that the data structure is represented purely as an Aladin program and no new representation (or memory management scheme) need be defined. All the primitives can be defined in terms of simple pattern matching on Aladin programs.

Lazy data structures contain components which are evaluated only when needed. Nevertheless, it is possible to force the complete evaluation of a lazy data structure defined with markers by using a primitive *force*, defined as follows.

$$\begin{aligned}&\mathit{force} :: s \rightarrow l\\ &\mathbf{Eval}[\![\mathit{force}\ (f\,x)]\!] = (\mathbf{Eval}[\![\mathit{force}\,f]\!])\ (\mathbf{Eval}[\![\mathit{force}\ x]\!])\\ &\mathbf{Eval}[\![\mathit{force}\ x]\!] = x\end{aligned}$$

force takes a single strict argument. If, after evaluation, that argument remains as an expression of the form $BX_1...X_n$, where B is a primitive and the X_i are any expressions, then it returns that expression after all the X_i are evaluated as *(force* X_i*)* (whether or not the definition of B specifies strict or lazy arguments and irrespective of the number of arguments that B takes). If the argument consists of a single primitive, it is returned unchanged.

5 Implementation

The Aladin machine is implemented as a conventional graph reduction machine. The primitive functions are coded in C as pure functions which operate either on data values (e.g. arithmetic operators) or on the graph structure itself (e.g. combinators). The usual Turner combinators are provided as primitive functions coded in C, together with the usual arithmetic and logical operators, list primitives, etc. The program graph is made up of four node types: (i) application nodes (which contain pointers to the function and its argument), (ii) data nodes (e.g. an integer), (iii) primitive function nodes (which contain a pointer to the code for the function, together with strictness details of the arguments), and (iv) marker nodes (e.g. *Nil* and *Cons*).

Example

A very simple example of a real-time program is one which reads characters input to the program and immediately prints the same character to the output, except that a newline is inserted after every 80 characters. Of course, in most imperative languages, this would be regarded as an ordinary program, not a 'real-time' program, but in the functional programming world it counts as 'real-time' in that the input and output must be sychronised, whereas in most pure functional languages this is not specified and the output may appear at any time (except that it cannot precede any input on which it depends).

In Aladin, basic character output may be obtained by creating a stream program with the name *puts*, whose result is a string (treated as a list of characters). The Aladin run-time system will repeatedly call a program with this name and copy the character strings to the output device, in the order in which they are produced.

Thus the form of an Aladin program to output each character immediately as it is input, but inserting a newline after every eightieth character, is as follows. In this program, the 'state' (third argument of *Transition*) is a pair consisting of an integer (the count of the number of characters since the last newline was output) and the input stream. The identifier *input* denotes the input stream as a potentially infinite list of characters. If the program tries to access the next character before it has been input, the Aladin run-time system will cause it to wait until that character is available.

```
puts = Transition f g (1, input)
        where
        f (n,s) = (if n==80 then 0 else n+1 fi, tail s)
        g (n,s) = if n==0 then head s : [] else head s : newline : [] fi
```

Note that *f* returns a pair, being the new value of the 'state'. This program is written in a (functional) pseudocode which should be easily understood. The language processed by the Aladin machine is actually much lower-level than this. Nevertheless, the program given above is a good representation of the semantics of the lower-level Aladin program.

If, for example, the input is the string *"hello"*, the program graph initially represents:

puts = Transition f g (1, "hello")

This is immediately evaluated, and as soon as the first character of the input is available, is rewritten to:

puts = Transition f g (2, "ello")

and the string *"h"* is output. The evaluation is then repeated immediately, and as soon as the second character of the input is available, the program is rewritten to:

puts = Transition f g (3, "llo")

and the string *"e"* is output. The process continues as long as more input is available.

More interesting real-time programs that accept several input streams, and respond to whichever inputs occur first, require the use of non-deterministic or other special operators which will be described in a future paper.

In principle, it should not be difficult to compile high-level lazy functional languages such as Haskell or Miranda into Aladin (a compiler for the language Ginger [11] has already been implemented), but this constricts much of the extra flexibility of Aladin: the ability to easily extend the set of primitive functions, specify the strictness of function arguments, etc. (although some of these things could be incorporated by means of 'annotations' in the high-level language).

6 Comparison with Other Work

There are many functional programming languages, both strict and non-strict, and Aladin builds on the heritage of these languages. The important features of Aladin which differ from most other languages are (i) the use of data streams for output, (ii) the ability to specify the strictness of primitives and thereby control the evaluation mechanism, and (iii) the exclusion of primitive functions from the language definition. Each of these is considered in turn.

6.1 Streams

Data streams are not new, but have been used as a fundamental concept in the functional data-flow language Lucid [12] and its derivatives pLucid [13] and GLU [14]. While all of these languages are declarative, they are not pure functional languages in the sense that functions are not first-class objects that can be manipulated as easily as if they were data objects. The Lucid family of languages permit first-order functions only (i.e. functions cannot generate other functions) and streams are always streams of data items, never streams of functions. Nevertheless, GLU allows the integration of the declarative style of programming supported by Lucid with the imperative style of C. Like Aladin, GLU provides a declarative framework in which to compose pure, strict functions written in C. Aladin differs fundamentally from GLU in that Aladin allows higher-order functions, and permits the use of both strict and non-strict functions written in C. Internally, Aladin is essentially an ordinary functional language; streams exist purely to allow Aladin programs to fit into a state-transition environment. With Lucid and GLU, however, data streams are more central to the language, with primitives for manipulating data streams effectively replacing conventional list primitives. Hence, the Lucid or GLU programmer designs almost all programs in terms of data streams as the primary data structures.

Another approach to integrating the imperative state-transition approach with non-strict functional programming is exemplified by recent work at Glasgow [15, 16]. This work shows how to encapsulate a computation defined in terms of a sequence of state transitions, in such a way that it is a pure function that can be used safely and cleanly in a non-strict functional language, and without destroying important features of functional languages such as referential transparency. This provides a convenient means of writing programs in a pure functional language that have been designed in the sequential state-transition style. Streams are, in a sense, the converse of this approach. Instead of encapsulating state-transition computations in a pure functional program, they encapsulate pure functional programs in a state-transition environment.

6.2 Strictness

The ability to freely specify the strictness of arguments and results of primitive functions appears to be unique to Aladin. We are not aware of any other language that offers complete control of strictness in the same way, particularly the ability to specify the 'strictness' of the result of a primitive function. Of course, almost all functional languages have both strict and non-strict primitives (within their built-in or standard-prelude primitives), and some allow the user to add new primitives implemented in C (or another imperative language). Haskell includes this feature (the *ccall* construct), but it is regarded as an advanced facility that is intended primarily for 'systems programmers' rather than 'ordinary' users. Furthermore, such user-defined primitives are always strict in Haskell; no mechanism is provided for adding non-strict primitives.

6.3 A Language Without Primitives

The other major difference between Aladin and other functional languages is that Aladin is defined totally independently of any particular set of primitive functions. The purpose of this approach is to emphasise the independence of the language evaluation mechanism and the choice of primitive functions. Typically, functional languages do not attempt to clearly separate these issues. Previous work has shown, for example, that the traditional list primitives (*head, tail, cons*) are not the ideal choice for parallel implementation [17]. Yet most functional languages stick to the traditional list primitives despite claims that functional languages are intrinsically better than state-transition languages for parallel computation.

By separating completely the definition of the language (with its evaluation mechanism) from the definition of the primitive functions, the user is given much greater freedom to experiment with different sets of primitives, and different methods of implementing those primitives. Recent work on categorical data types is particularly relevant and offers new insights into the choice of suitable primitives for handling data structures in a functional language context [18, 19]. Although some non-strict functional languages, such as Haskell, provide hooks for calling primitive functions programmed in C, usually these mechanisms are seen as escape routes for use in exceptional circumstances, not mainstream features that are routinely used by most programs. Because Aladin includes no primitives in its definition, no one set of primitives is intrinsically more favoured than any other. The user is completely free to change or extend the set of primitives that he uses (although, obviously, considerable care and experience is needed to design and implement a useful and correct set of primitives as pure functions in C).

7 Conclusions

The Aladin abstract functional language machine has been described, together with its mechanisms for interfacing to procedural language programs. A prototype implementation of Aladin has been implemented in C, along with a library of primitive functions. Work is continuing to extend the library and to provide other utilities to make it easier to write Aladin programs (the Aladin machine language is very low-level and programming in it is very tedious and error-prone). Our experience so far indicates that the Aladin approach is both feasible and practical, but much more work is required to investigate the approach in greater depth and in a variety of application areas.

The Aladin machine provides a basic low-level functional language evaluation mechanism in a minimal form. Normally functional programming languages include many other features: (i) a library of primitive functions and operators, (ii) sophisticated polymorphic typing systems, (iii) sophisticated high-level syntax (e.g. pattern matching, infix and circumfix operators), (iv) a variety of data structures. These features are usually built into the language in such an intimate way that they cannot be altered effectively except by changing the language definition and modifying the compiler or interpreter accordingly. Although all the features listed above fit very conveniently into functional languages, none of them is an essential part of functional programming. Our aim in developing the Aladin machine has been to create a much more basic functional programming mechanism, leaving all the non-essential and high-level features open to variation and experimentation. Instead of putting more and more into the language, we have tried to put in the minimum possible while retaining the pure lambda-calculus-based semantics. In addition, very general interfaces to the procedural programming world have been provided by allowing all primitive functions to be programmed in C, and also by allowing any pure functional program to be evaluated in a state-transition environment in which it produces a ‘stream’ of output values in real time.

It is hoped that the Aladin machine will provide a convenient way to experiment with new approaches to functional programming that do not fit within the constraints of normal functional programming languages, and would otherwise require completely new languages to be created each time. Particular areas of research that may be able to benefit from this are (i) the investigation of new data structures for functional languages, (ii) parallel constructs for functional languages, and (iii) the interfacing of functional languages and operating systems. Investigation into some of these areas is proceeding.

References

[1] D. A. Turner. Miranda: A Non-Strict Functional Language With Polymorphic Types, in *Proceedings of the 1985 IFIP International Conference on Functional Programming and Computer Architecture, Lecture Notes in Computer Science,* **201** (Springer-Verlag, Berlin, 1985)

[2] P. Hudak et al. Report on the Programming Language Haskell - A Non-Strict, Purely Functional Language - Version 1.2. *ACM SIGPLAN Notices*, **27** (1992) No.5.

[3] J. Hughes. Why Functional Programming Matters. *Computer Journal*, **32** (1989), 98–107.

[4] J. W. Lloyd. *Practical Advantages of Declarative Programming* (Dept. of Computer Science, University of Bristol, Bristol, 1994) `ftp://ftp.cs.bris.ac.uk/goedel/jwl_papers/advantages.ps.Z`

[5] S. C. Wray and J. Fairbairn. Non-Strict Languages – Programming and Implementation. *Computer Journal,* **32** (1989) 142-151.

[6] P. L. Wadler. *An Introduction to Orwell* (Computing Laboratory, University of Oxford, Oxford, 1985)

[7] J. Fairbairn. *Ponder and its Type System, Technical Report No.31* (Computer Laboratory, University of Cambridge, 1982)

[8] Å. Wikström. *Functional Programming Using Standard ML* (Prentice-Hall, London, 1987)

[9] C. Hankin. *Lambda Calculi – A Guide for Computer Scientists* (Oxford University Press, Oxford, 1994)

[10] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, **34** (1955) 1045-1079.

[11] M. S. Joy. *Ginger - A Simple Functional Language, Research Report 235* (Department of Computer Science, University of Warwick, Coventry, 1992) `http://www.dcs.warwick.ac.uk/dcs/reports/rr/235/`

[12] E. A. Ashcroft and W. Wadge. *Lucid: the Dataflow Programming Language* (Addison-Wesley, Reading, 1985)

[13] A. A. Faustini, S. G. Matthews and A. A. Yaghi. *The pLucid Programming Manual* (Dept. of Computer Science, Arizona State University, Tempe, Arizona, 1983)

[14] R. Jagannathan and C. Dodd. *GLU Programmer's Guide, SRI-CSL-94-06* (SRI International, 333 Ravenswood Avenue, Menlo Park, California, 1994)

[15] J. Launchbury and S. L. Peyton Jones. Lazy Functional State Threads, in *Proceedings of the ACM 1994 Conference on Programming Language Design and Implementation, SIGPLAN Notices,* **29** (1994) No.6, 24-35.

[16] S. L. Peyton Jones and P. L. Wadler. Imperative Functional Programming, in *20th ACM Symposium on Principles of Programming Languages,* (ACM, Ohio, 1993), pp.71-84.

[17] T. Axford and M. Joy. List Processing Primitives for Parallel Computation. *Comput. Lang.* **19** (1993) 1-17.

[18] C. R. Banger and D. B. Skillicorn. *Constructing Categorical Data Types* (Dept. of Computing and Information Science, Queen's University, Kingston, Canada, 1993)

[19] M. Cole. *List Homomorphic Parallel Algorithms for Bracket Matching, CSR-29-93* (Dept. of Computer Science, University of Edinburgh, 1993)