A Lazy Non-Deterministic Functional Language

A. N. Clark, Department of Computing, University of Bradford
Bradford, West Yorkshire, BD7 1DP, UK
e-mail: a.n.clark@comp.brad.ac.uk, tel.: (01274) 385133

1 Introduction

A useful starting point for the development of programs which involve search, such as Knowledge Based Systems, is a *naive program*. Such a program employs non-deterministic choice to explore a search space and pays no attention to the control which is necessary in order to make the best choice at each stage. Such a starting point has the merit that it is *complete*, *i.e.* it describes every possible solution and that it is *computational*, *i.e* it describes (abstract) calculations and can be used as the starting point of program *refinement*. Refinement is used to introduce clever techniques which control the exploration of the search space and which gradually transform a non-deterministic naive program into a deterministic clever (or Knowledge Based) program.

The techniques for the refinement of naive programs and the properties which must be upheld are described in (**?**) and (**?**). This paper addresses the starting point of the refinement process: the naive program. Although a naive program provides a useful starting point for refinement and is complete with respect to all the possible outcomes it usually exhibits exponential computational complexity which prohibits using the initial naive system as a prototype.

The computational complexity of naive programs arises due to the simple and obvious operational semantics for non-deterministic choice: at the point in the calculation which the choice is made, the calculation splits into two calculations which continue independently. The result of performing a single program is selected at random from the results of all the calculations which the program gives rise to.

If this operational semantics is implemented as a programming language for prototyping naive systems then the resources required are beyond the capabilities of most computer systems for all but the simplest naive programs. The operational semantics for naive programs must be made more sophisticated in order to use them as prototypes.

One possibility is that non-deterministic choice is deferred until it is necessary, in the hope that the choice will never have to be made. When a non-deterministic choice is called for, a single value is returned (called an *nd-value*) which contains both alternatives. Computation need not be duplicated since the calculation never splits into two. Unfortunately, this simple scheme causes problems when nd-values are copied within a calculation. Since an nd-value represents one of the choices it contains, when an nd-value is copied so is the ability to choose. This is to be compared with the case where the choice is made eagerly, if the value is then copied the choice has already been made. These two alternatives produce very different outcomes.

This paper describes a system which implements non-deterministic choice using nd-values, but imposes constraints on the format of programs which ensures that they are consistent with eager choice. The constraints take the form of a type system which prevents programs which copy nd-values from being written. Functions which wish to copy nd-values must be identified and use a special function notation: μ-functions which can be performed automatically. Such functions are shown to be syntactic sugar for expressions which *reify* nd-values, manipulate them as d-values and then *install* them back into the calculation as nd-values.

This paper is structured as follows. §2 describes a simple functional language with non-deterministic choice and uses it to construct a simple KBS application for Data Fusion. The application will be used to demonstrate the results of this work. Examples of program execution are given in terms of the number of computational steps. §3 introduces a modified language which implements lazy non-deterministic choice, examples of calculations are given and problems with copying nd-values are identified. §4 describes a type system which prevents nd-values from being copied and §5 shows that the forcing of nd-values can be localized within a program fragment. §6 shows how the techniques can be applied to the Data Fusion application in order to use a naive program as a Data Fusion prototype. Finally, §7 describes related work, analyses the results and outlines future work.

2 A Naive Program

A *naive* program is a computer program which involves choice and which employs no sophisticated techniques to cut down on the number of different choices which must be explored before the correct choice is taken. Such systems are conveniently expressed in terms of a simple functional programming language which contains primitives for non-deterministic choice. The syntax of such a language is given below:

$$E ::= I \mid \lambda I.E \mid EE \mid (E, \ldots, E) \mid \textbf{if } E \textbf{ then } E \textbf{ else } E \mid E \diamond E \mid \textbf{fail}$$

$i \in I$ is a program identifier; $\lambda i.b$ is a function with argument $i \in I$ and body $b \in E$; $e_1 e_2$ is an application with operator $e_1 \in E$ and operand $e_2 \in E$; $(e_1, \ldots, e_n)$ is an n-tuple where each $e_i \in E$ is a component; **if** e_1 **then** e_2 **else** e_3 is a conditional expression where $e_1 \in E$ is the antecedent, $e_2 \in E$ is the consequent and $e_3 \in E$ is the alternative; $e_1 \diamond e_2$ is a non-deterministic choice where $e_1 \in E$ or $e_2 \in E$ will be chosen non-deterministically; **fail** is a suicide expression.

The operational semantics of this simple language is given by a state transition machine. The machine is based upon the SECD machine (**?**) and includes extensions which are necessary for non-deterministic primitives ($\diamond$) and **fail**. The essential difference is that the state transitions hold between single states and sets of states produced by executing non-deterministic operators. An similar machine and application is described in () and an introduction to functional programming is given in (**?**).

Each machine state is a 4-tuple and contains *computational values* which are described as follows: a list is either empty [] or a *cons* $v : l$ of a value v and a list l, $[v_1, v_2, \ldots, v_n]$ is shorthand for $v_1 : (v_2 : (\ldots(v_n : [])\ldots))$; an n-tuple of

$$
\begin{array}{l}
(s, e, i : c, d) \longmapsto \{(e(i) : s, e, c, d)\} \\
(s, e, \lambda i.b : c, d) \longmapsto \{(<i, e, b> : s, e, c, d)\} \\
(s, e, (e_1 e_2) : c, d) \longmapsto \{(s, e, e_1 : e_2 : @ : c, d)\} \\
(v : <i, e', b> : s, e, @ : c, d) \longmapsto \{([], e'(i \mapsto v), [b], (s, e, c, d))\} \\
(v : \_, \_, [], (s, e, c, d)) \longmapsto \{(v : s, e, c, d)\} \\
(s, e, (e_1, \ldots, e_n) : c, d) \longmapsto \{(s, e, e_n : \ldots : e_1 : [n] : c, d)\} \\
(v_1 : \ldots : v_n : s, e, [n] : c, d) \longmapsto \{((v_1, \ldots, v_n) : s, e, c, d)\} \\
(s, e, \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : c, d) \longmapsto \{(s, e, e_1 : [e_2, e_3] : c, d)\} \\
(\mathit{true} : s, e, [e_1, e_2] : c, d) \longmapsto \{(s, e, e_1 : c, d)\} \\
(\mathit{false} : s, e, [e_1, e_2] : c, d) \longmapsto \{(s, e, e_2 : c, d)\} \\
(s, e, e_1 \diamond e_2 : c, d) \longmapsto \{(s, e, e_1 : c, d), (s, e, e_2 : c, d)\} \\
(\_, \_, \mathbf{fail} : \_, \_) \longmapsto \emptyset
\end{array}
$$

Fig. 1. The Semantics of An ND-Functional Language

values $(v_1, \ldots, v_n)$; an *environment* is a function from identifiers to values, given an environment e it is extended with a binding between i and v by $e(i \mapsto v)$; a function *closure* $<i, e, b>$ contains an argument $i \in I$, an environment e and a body $b \in E$; and, booleans *true* and *false*.

Machine instructions are values which occur in the control of the machine: application @; n-tuple construction $[n]$; and boolean choice $[e_1, e_2]$ where $e_1 \in E$ and $e_2 \in E$ are consequent and alternative respectively.

A machine state is a 4-tuple (s, e, c, d) where s is a sequence of values; e is an environment; c is a sequence of program expressions and machine instructions and d is either () or a machine state. The components of the machine state are referred to as the *stack*, *environment*, *control* and *dump* respectively.

The transition function for the machine maps a single machine state to a set of machine states and is defined in figure 1. Each line in the definition corresponds to a set of transitions created by consistently substituting values of the suitable types for the variables. The functional language is very austere, it can be extended by defining *syntactic sugar* which are new syntactic constructs defined from existing ones. The sugaring and desugaring is described (**?**).

The language may also be extended with builtin data types and operators over those data types, for example integers and arithmetic over integers. The application of builtin operators is described by the following evaluation rule:

$$(v : f : s, e, @ : c, d) \longmapsto (f(v) : s, e, c, d)$$

providing that $f(v)$ is defined elsewhere.

The transition function $\longmapsto$ maps from single machine states to sets of machine states. Given a set of machine states S and a powerset operator P, then the type of the state transition function is defined as $\longmapsto : S \rightarrow P(S)$. We can extend $\longmapsto$ to be a function which maps sets of states to sets of states as follows:

$$\longmapsto : P(S) \rightarrow P(S)$$

$$\longmapsto (P) = \bigcup\{\longmapsto (s) \mid s \in P\}$$

The function can be extended further by the postfix operator * which generates the transitive closure of its argument. An evaluation function $X_\rho : E \to P(V)$ translates a program expression $e \in E$ to a set of program outcomes $S \in P(V)$. The environment for freely referenced identifiers in the expression e is represented as ρ. The evaluation function is defined:

$$X_\rho(e) = S \textbf{ iff } \{([], \rho, [e], ())\} \longmapsto^* Q \wedge S = \{v \mid ([v], \_, \_, \_) \in Q\}$$

The evaluation function E defines the gold standard for any other evaluation strategy for naive programs to be measured against. Suppose that another evaluation function $X'_\rho : E \to P(V')$ exists such that there is a translation $\phi : V' \to V$ from outcomes to outcomes. The following diagram must commute for the new evaluation strategy:

$$\begin{tikzcd}
 & P(V) \\
E \arrow[ur, "X_\rho"] \arrow[dr, "X'_\rho"'] & \\
 & P(V') \arrow[uu, "\phi^*"']
\end{tikzcd}$$

Our claim is that the functional language is useful as the basis for specification of programs which involve search and in particular the specification and refinement of Knowledge Based Systems. In order to show that this is the case we will give a simple but representative specification for a Data Fusion application. Data Fusion involves receiving a stream of entity observations which must be *fused* into a global picture. The application causes problems for an algorithmic approach because the observed entities are autonomous and the messages are noisy, may be duplicated and may not occur in the same order as the events which were observed. Data Fusion applications are described in (**?**) (**?**) and (**?**). A model for Data Fusion has been analysed in the MOSES document (**?**).

The model for Data Fusion calculations which we wish to use as a specification is as follows. Each type of entity is described as a non-deterministic finite state machine. At each instant in time, all possible entities perform one of their possible state transitions. Each observation which is received may contain errors and therefore is non-deterministically selected from a set of possible correct observations. When an observation is received, we assume that it is correct and describes an entity in the current global picture. If no matching entity exists, then the global picture is deleted.

A simple implementation of a Data Fusion system is shown in figure 2. The main data values are vehicles, messages and states. A vehicle is (i, x, y) where i is a unique identifier for the vehicle and (x, y) is the current position of the vehicle. A message is (t, f) where t is an integer representing the time at which the message

$$
\begin{array}{l}
\textbf{let } \mathit{move}(i,x,y) = \\
\quad \textbf{let } \mathit{delta} = (1+) \diamond I \diamond (-1) \\
\quad \textbf{in } (i, \mathit{delta}\ x, \mathit{delta}\ y) \\
\textbf{let } \mathit{at}\ \mathit{vehicle}\ \mathit{vehicles} = \textbf{if } \mathit{member}\ \mathit{vehicle}\ \mathit{vehicles} \textbf{ then } \mathit{vehicles} \textbf{ else fail} \\
\textbf{let } \mathit{trans}\ \mathit{state} = \\
\quad \textbf{case } \mathit{state} \textbf{ of} \\
\qquad (\mathit{vs}, t, m\ (t', \_) : \mathit{ms}) \Rightarrow (\mathit{map}\ \mathit{move}\ \mathit{vs}, t+1, m : \mathit{ms}) \textbf{ when } t' > t \\
\qquad (\mathit{vs}, t, (t', f) : \mathit{ms}) \Rightarrow (f\ \ \mathit{vs}, t, \mathit{ms}) \textbf{ when } t = t' \\
\quad \textbf{end}
\end{array}
$$

Fig. 2. A Simple Data Fusion System

was received and f is a *guard function* which is applied to the current global picture when the message is received. A guard function is applied to a value and either acts as if it were the identity function when the value satisfies the guard or acts as if it were **fail** otherwise. A state is (vs, t, ms) where vs is the current global picture which is a list of vehicles, t is the current time and ms is a list of incoming messages.

The function *trans* performs a single state transition for the current Data Fusion state. If the next message is in the future then all the vehicles are moved and if a message is received then it is handled by using the information to check the current global picture. Moving a vehicle is performed using the function *move* which applies a non-deterministic function *delta* to the x and y co-ordinates of the vehicle's current position.

The function *at* constructs a simple guard function when applied to a vehicle. The resulting guard function ensures that the vehicle is present in the current global picture. If it is not then the guard function performs **fail** which causes the current calculation to die and therefore discard the current global picture.

There are a number of simplifying assumptions made about Data Fusion in the system shown in figure 2. These are: messages are always assumed to be correct *i.e.* they are not noisy, messages are always received in time order and all observable entities are known at the outset. These assumptions are revisited at the end of the paper.

The Data Fusion system described in figure 2 uses non-deterministic choice to describe all the possible behaviours of observable entities. This occurs in the *move* operation which non-deterministically selects and applies $(1+)$, I or (-1) to both the x co-ordinate and the y co-ordinate. If the state of the system before performing a state transition is $([(i, 100, 200)], 50, [])$ then the state after the move will be one of the following:

$$([(i,101,201),50,[])$$
$$([(i,101,200),50,[])$$
$$([(i,101,199),50,[])$$
$$([(i,100,201),50,[])$$
$$([(i,101,200),50,[])$$
$$([(i,101,199),50,[])$$
$$([(i,99,201),50,[])$$
$$([(i,99,200),50,[])$$
$$([(i,99,199),50,[])$$

The operational semantics of the functional language performs non-deterministic choice by splitting the machine into two. Each new machine is the same as the original except that the non-deterministic expression at the head of the control has been replaced with one or other of the sub-expressions. Given an interpretation strategy which treats all machines fairly, this will develop all the possible global pictures concurrently and therefore guarantee to contain the picture which corresponds to reality.

Unfortunately, such an interpretation strategy will perform many equivalent computations more than once. Each machine which is created by performing non-deterministic choice in the Data Fusion system differs from the others with respect to a few simple data values (*i.e.* the x and y co-ordinates). Many expressions, such as those which test whether the next message should be handled, will behave identically on all machines.

Another undesirable result of duplicating machines each time non-deterministic choice is performed is that all the possible choices and values which depend upon those choices become separated. For example, if after starting with a state `([(i,100,200)],50,ms)` and performing several transitions, a message arrives which states that the entity with identifier `i` is in an area identified by $x > 120$ and $y > 215$ then many different machines will have performed computation in vain since their proposed positions for the entity do not fall within the specified area.

A solution for the first problem could be to extend the interpretation strategy with *memoizing* features which remember the results of performing program expressions. This would allow machines to take advantage of previously evaluated expressions. A problem with this improvement is that the evaluation of a program expression depends upon the current context, different machines will have different contexts and it may be difficult to compare contexts. Another objection with this approach is that it does not improve the second problem since information is still decentralized.

Before describing a proposed solution to these problems (in §3) we give some performance results for two simple non-deterministic programs. Figure 3 shows a simple non-deterministic program and the output, `in this font`, which is produced when it is executed. The non-determinism arises in the list l where the first element is either 1 or 10 and the second is either 2 or 20. The operator *counter* is used to simulate some arbitrary computation which is performed by the program,

$$
\begin{array}{ll}
\textbf{let} & \mathit{member}\ \_\ [] = \mathit{false} \\
& \mathit{member}\ x\ (x : \_) = \mathit{true} \\
& \mathit{member}\ x\ (\_ : l) = \mathit{member}\ x\ l \\
& \mathit{counter}\ n\ n = n \\
& \mathit{counter}\ n\ m = \mathit{counter}\ (n-1)\ m \\
& l = [1 \diamond 10, 2 \diamond 20] \\
\multicolumn{2}{l}{\quad \textbf{in}\ \mathit{member}\ (\mathit{counter}\ 1002\ 2)\ l}
\end{array}
$$

```
The following values were produced after 80390 transitions:
1: true
2: true
There are incomplete computations, continue? y
The following values were produced after 80422 transitions:
3: false
4: false
No further values were produced.
```

Fig. 3. Results from program no. 1

$$
\begin{array}{ll}
\textbf{let} & \mathit{member} = \mathit{as\ in\ figure\ 3} \\
& \mathit{counter} = \mathit{as\ in\ figure\ 3} \\
& f\ n_1\ n_2\ l = \textbf{if}\ (\mathit{member}\ n_1\ l)\ \&\ (\mathit{member}\ n_2\ l)\ \textbf{then}\ l\ \textbf{else}\ [] \\
& l = [1 \diamond 10, 2 \diamond 20] \\
\multicolumn{2}{l}{\quad \textbf{in}\ f\ (\mathit{counter}\ 1002\ 2)\ (\mathit{counter}\ 1001\ 1)\ l}
\end{array}
$$

```
The following values were produced after 160606 transitions:
1: []
2: []
There are incomplete computations, continue? y
The following values were produced after 160623 transitions:
3: [1,2]
There are incomplete computations, continue? y
The following values were produced after 160670 transitions:
4: []
No further values were produced.
```

Fig. 4. Results from program no. 2

it takes two positive integer arguments n and m and loops, subtracting 1 from n until it is equal to m. The program determines whether or not 2 is a member of the list l. Since l stands for four different lists there are four different outcomes from the program. The first two outcomes correspond to the cases where l is $[1, 2]$ or $[10, 2]$ and the second two outcomes correspond to the cases where l is $[1, 20]$ or $[10, 20]$. As the output shows, the execution took between 80390 and 80422 machine transitions to execute.

Figure 4 shows a slightly different program and its execution. In this case the operator f is applied to two integers n_1 and n_2 and a list l. If both integers are members of l then f returns l otherwise it returns the empty list. The execution shows that there are four outcomes and only one of the possible values for l contains both 1 and 2.

The expression *counter* 1000 0 is performed in 20030 machine transitions. The

$$
\begin{array}{l}
(s, e, i : c, d) \longmapsto \{(e(i) : s, e, c, d)\} \\
(s, e, \lambda i.b : c, d) \longmapsto \{(<i, e, b> : s, e, c, d)\} \\
(s, e, (e_1 e_2) : c, d) \longmapsto \{(s, e, e_1 : e_2 : @ : c, d)\} \\
(v_1 : v_2 : s, e, @ : c, d) \longmapsto \begin{cases} \{([], e'(i \mapsto v), [b], (s, e, c, d))\} & \text{when } v_2 = <i, e', b> \\ \{(v_2 @ v_1 : s, e, c, d)\} & \text{otherwise} \end{cases} \\
(v : \_, \_, [], (s, e, c, d)) \longmapsto \{(v : s, e, c, d)\} \\
(s, e, (e_1, \ldots, e_n) : c, d) \longmapsto \{(s, e, e_n : \ldots : e_1 : [n] : c, d)\} \\
(v_1 : \ldots : v_n : s, e, [n] : c, d) \longmapsto \{((v_1, \ldots, v_n) : s, e, c, d)\} \\
(s, e, \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : c, d) \longmapsto \{(s, e, e_1 : [e_2, e_3] : c, d)\} \\
(\mathit{true} : s, e, [e_1, e_2] : c, d) \longmapsto \{(s, e, e_1 : c, d)\} \\
(\mathit{false} : s, e, [e_1, e_2] : c, d) \longmapsto \{(s, e, e_2 : c, d)\} \\
(s, e, e_1 \diamond e_2 : c, d) \longmapsto \{(<e, e_1> + <e, e_2> : s, e, c, d)\} \\
(\_, \_, \textbf{fail} : \_, \_) \longmapsto \emptyset
\end{array}
$$

Fig. 5. The Semantics of A Lazily ND-Functional Language

four different possibilities for l in figure 3 leads to $4 * 20030 = 80120 \approx 80422$. In figure 4 the count is performed twice, $4 * 2 * 20030 = 160240 \approx 160670$.

3 Delaying Non-Deterministic Selection

Two problems have been identified with the interpretive strategy of the language described in §2: evaluation of expressions is unnecessarily duplicated and useful information becomes decentralized. This section describes a modification to the functional language which addresses these issues. The solution is to allow non-deterministic choice to be *delayed* until it is forced by primitive machine operations. This allows information to be localized and removes the requirement for unnecessary machine duplication. Unfortunately, this solution does not maintain referential transparency.

The syntax of the functional language is unchanged. There are three new types of computational value: thunks, nd-values and delayed applications. A thunk is a delayed expression and is represented as $<e, b>$ where e is an environment and $b \in E$ is a program expression. A thunk contains enough information to evaluate an expression at some later date. An nd-value has the form $v_1 + v_2$ and represents a delayed non-deterministic choice between values v_1 and v_2. A delayed application has the form $v_1 @ v_2$ and represents the delayed application of the operator v_1 to the operand v_2.

The modified operational semantics for the language has two parts: evaluation of program expressions and forcing of delayed values. Program expression evaluation is shown in figure 5 and is the same as the machine defined by figure 1 except for the the following. The rule for non-builtin operator application must be extended to deal with the case when the operator is either an nd-value, a thunk or a delayed application. In this case a delayed application is constructed. The rule for non-deterministic choice is changed so that an nd-value is constructed. The sub-expressions of a non-deterministic choice expression are themselves expressions whose evaluation is delayed by creating an nd-value consisting of two thunks.

$$
\begin{array}{l}
(s, e, <i, e', b> : c, d) \longmapsto \{(s, e, e' : \{i, b\} : c, d)\} \\
(e' : s, e, \{i, b\} : c, d) \longmapsto \{(<i, e', b> : s, e, c, d)\} \\
(s, e, \kappa(v) : c, d) \longmapsto \{(s, e, v : \kappa : c, d)\} \\
(v : s, e, \kappa : c, d) \longmapsto \{(\kappa(v) : s, e, c, d)\} \\
(s, e, (v_1, \ldots, v_n) : c, d) \longmapsto \{(s, e, v_n : \ldots : v_1 : [n] : c, d)\} \\
(s, e, v_1 + v_2 : c, d) \longmapsto \{(v_1 : s, e, c, d), (v_2 : s, e, c, d)\} \\
(s, e, <e', b> : c, d) \longmapsto \{([], e', [b], (s, e, c, d))\} \\
(s, e, v_1 @ v_2 : c, d) \longmapsto \{(v_2 : s, e, v_1 : @ : c, d)\}
\end{array}
$$

Fig. 6. The Semantics of Forcing Data Values

Forcing a value means that the most recently delayed evaluation will be performed. In the case of nd-values, this will cause the machine to split into two. In the case of thunks this will mean that the body of the thunk will be evaluated with respect to the thunk's environment. In the case of a delayed application this will mean that the operator will be forced and re-applied to its operand.

A value is forced by placing it at the head of the control stack. The machine transitions for forcing a value are shown in figure 6. There are two new machine instructions. The first is $\{i, b\}$ where $i \in I$ and $b \in E$ which expects an environment on the head of the stack and replaces the environment with a closure. The second is a data constructor κ which expects a value v at the head of the stack and replaces it with the data value constructed by applying κ to v. This simple rule covers many different algebraic data types such as lists and environments.

The builtin operators must be updated in order to deal with lazy non-determinism. Each operator must be categorized either as *strict* or *non-strict* with respect to non-determinism. If an operator is non-strict then it may be applied to a data value without ensuring that the value is forced. An example of a non-strict operator is : which builds cons pairs. The rule for such a builtin operator is left unchanged.

If an operator is strict with respect to non-determinism then an operand must be forced before it is possible to apply the operator. Examples of strict operators are + and *hd*. The rule for applying such an operator must be changed, for example:

$$
(+ : v_1 : v_2 : s, e, @ : @ : c, d) \longmapsto \begin{cases} (v_1 + v_2 : s, e, c, d) \text{ when } v_1 \in \mathbf{N} \ \&\ v_2 \in \mathbf{N} \\ (s, e, v_2 : v_1 : + : @ : @ : c, d) \text{ otherwise} \end{cases}
$$

which describes the application of the builtin operator for integer addition. If both values are integers then the builtin operator is applied, otherwise they are both forced and the operator is re-applied. In the case of + it may be necessary to force the values many times before they are reduced to a *ground* value. Some operators are *partially strict* in the sense that values do not need to be ground before they can be applied. For example:

$$
(v : hd : s, e, @ : c, d) \longmapsto \begin{cases} (v' : s, e, c, d) & \text{when } v = v' : \_ \\ (hd : s, e, v : @ : c, d) & \text{otherwise} \end{cases}
$$

where a value v is continually forced until a cons pair is produced. The head and tail of the cons pair need not be ground in order to return the head.

The extensions to the state transition machine meet the two objectives since

$$
\begin{array}{l}
\textbf{let}\ \ \textit{member}\ \_\ [\,] = \textit{false} \\
\qquad \textit{member}\ x\ (x : \_) = \textit{true} \\
\qquad \textit{member}\ x\ (\_ : l) = \textit{member}\ x\ l \\
\qquad \textit{counter}\ n\ n = n \\
\qquad \textit{counter}\ n\ m = \textit{counter}\ (n-1)\ m \\
\qquad l = [1 \diamond 10, 2 \diamond 20] \\
\ \ \textbf{in}\ \textit{member}\ (\textit{counter}\ 1002\ 2)\ l
\end{array}
$$

```
The following values were produced after 20186 transitions:
1 : true
2 : true
There are incomplete computations, continue? y
The following values were produced after 20218 transitions:
3 : false
4 : false
No further values were produced.
```

Fig. 7. Results from program no. 1 using nd-values

non-determinism is delayed until it is forced by the underlying machine primitives. The number of repeated evaluations for the same expression is reduced and the information relating to non-deterministic choice is localized. The following example shows how these objectives have been met.

Figure 7 shows the evaluation of the same program as shown in figure 3 except that the non-deterministic choice operator constructs nd-values. There is a significant saving on the amount of computation which is performed since the machines are not duplicated and the loops are performed once.

Although the objectives have been met, the proposed solution has a serious flaw: it does not preserve referential transparency. This property states that equals can always be substituted for equals and that in particular the value of an identifier can always be used instead of a reference to the identifier. For example consider the following program:

$$\textbf{let}\ \textit{twice}(x) = x + x\ \textbf{in}\ \textit{twice}(10 \diamond 20)$$

If the choice is made before the function *twice* is applied then the program is equivalent to

$$\textit{twice}(10) \diamond \textit{twice}(20)$$

in which case the outcome of the program is either 20 or 40. On the other hand, if the choice is made at the point at which $x + x$ is evaluated then the program is equivalent to

$$(10 \diamond 20) + (10 \diamond 20)$$

in which case the outcome of the program is either 20, 30 or 40.

By delaying non-deterministic choice the outcome of the program is counter-intuitive. If the choice is made at the point at which it is expressed then the result of the program seems correct but the efficiency gains which are produced by being lazy are lost.

Figure 8 shows the evaluation of the same program as shown in figure 4 except

$$
\begin{array}{ll}
\textbf{let} & \mathit{member} = \mathit{as\ in\ figure\ 3} \\
 & \mathit{counter} = \mathit{as\ in\ figure\ 3} \\
 & f\ n_1\ n_2\ l = \textbf{if}\ (\mathit{member}\ n_1\ l)\ \&\ (\mathit{member}\ n_2\ l)\ \textbf{then}\ l\ \textbf{else}\ [] \\
 & l = [1 \diamond 10, 2 \diamond 20] \\
\ \ \textbf{in}\ f\ (\mathit{counter}\ 1002\ 2)\ (\mathit{counter}\ 1001\ 1)\ l &
\end{array}
$$

```
The following values were produced after 40312 transitions:
1: []
2: []
There are incomplete computations, continue? y
The following values were produced after 40422 transitions:
3: [1,2]
4: [10,2]
5: [1,20]
6: [10,20]
7: [1,2]
8: [10,2]
9: [1,20]
10: [10,20]
There are incomplete computations, continue? y
The following values were produced after 40560 transitions:
11: []
12: []
13: []
14: []
No further values were produced.
```

Fig. 8. Results from program no. 2 using nd-values

that nd-values are constructed. Although there is a significant computational saving compared to the original, the semantics of evaluation causes the outcomes to be incorrect. This occurs due to the duplication of an nd-value in the body of the operator f. When the antecedent of the **if**-expression in f is *true* the value l is returned; but l is an nd-value which represents all of the alternative lists, even those for which the antecedent will fail. As the outcomes of the program show, all the alternative list values are produced. Values are produced more than once as a result of the way that nd-values are forced on the machine.

If X' is the evaluation function for the modified machine and ϕ forces all outcomes then it is easy to see that the following diagram does not commute:

$$
\begin{tikzcd}
 & P(V) \\
E \arrow[ur, "X_\rho"] \arrow[dr, "X'_\rho"'] & \\
 & P(V') \arrow[uu, "\phi^*"']
\end{tikzcd}
$$

i.e. the nd-evaluation mechanism is not consistent with the evaluation mechanism which is described in §2.

4 Linear Logic and Non-Determinism

We wish to ensure that a program behaves as though non-deterministic choice occurs or appears to occur where it is expressed in the program. This has a major implication: β-reduction cannot be applied to applications where the operand is non-deterministic, unless it can be guaranteed that the operand will never be copied in the process. This allows non-deterministic choice to be delayed without producing counter intuitive results. For example we wish to outlaw the following program:

$$\textbf{let } \mathit{dup}(x) = (x, x) \textbf{ in } \mathit{dup}(1 \diamond 2)$$

whereas the following program is legal (for deterministic x values):

$$\textbf{let } \mathit{filter}(l) = \textbf{if } x \in l \textbf{ then } x \textbf{ else fail in } \mathit{filter}([1,2,3] \diamond [4,5,6])$$

The restrictions force formal parameters in λ-expressions to be referenced at most once in the function body when there is the possibility of an nd-value being supplied as the actual parameter. This is very restrictive since it may not be known in advance whether or not an nd-value will be supplied. The syntax of the functional programming language is extended with μ-expressions:

$$E ::= I \mid \lambda I.E \mid \mu I.E \mid EE \mid (E, \ldots, E) \mid \textbf{if } E \textbf{ then } E \textbf{ else } E \mid E \diamond E \mid \textbf{fail}$$

A μ-expression behaves like a λ-expression except that when it is applied, it ensures that any actual parameter is forced to completion before it is bound to the formal parameter. A μ-closure is a new type of computational value which is represented as $[i, e, b]$ where the components are the same as those for a λ-closure. The operational rules for μ-expression evaluation and application are as follows:

$$(s, e, \mu i.b : c, d) \longmapsto ([i, e, b] : s, e, c, d)$$
$$([i, e', b] : v : s, e, @ : c, d) \longmapsto \begin{cases} ([], e' \oplus (i \mapsto v), [b], (s, e, c, d)) & \text{when } !(v) \\ ([i, e', b] : s, e, v : @ : c, d) & \text{otherwise} \end{cases}$$

where ! is a predicate which is true of all d-values and false otherwise. Using μ-expressions, the function *dup* can be defined:

$$\textbf{let } \mathit{dup} = \mu x.(x, x) \textbf{ in } \mathit{dup}(1 \diamond 2)$$

The rules which define a legal program must show that no λ-function which refers to its formal parameter more than once is applied to an nd-value. In order to do this we employ techniques from *linear logic* (**?**). Firstly we will define some terms.

A program value may be either an *nd-value* when it has been constructed using the $\diamond$ operator or a *d-value* otherwise. The process of transforming an nd-value into a collection of d-values is referred to as making the nd-value *ground.* A function is a data value which can be applied to a value. The meaning of function application is defined by the evaluation X in §2. The process by which functions are actually applied is defined to be *safe* when the outcome is consistent with the evaluation as described in §2 otherwise it is *unsafe*. A function is *nd-safe* when it does not copy its formal parameter, otherwise it is *nd-unsafe*. A function may be an nd-value in which case it is an *nd-function* otherwise it is a *d-function*. For the proposed function language the following table describes application safety:

	nd-value	*d-value*
nd-safe	*safe*	*safe*
nd-unsafe	*unsafe*	*safe*

where columns represent operand types, rows represent operator types and the entries define whether or not the application of the operator to the operand is safe or unsafe.

A program type is described by the following syntax definition:

$$T ::= B \mid \hat{B} \mid T \rightarrow^{L} T \mid T \Rightarrow^{L} T$$

$$B ::= \mathbf{int} \mid \mathbf{bool} \mid \ldots$$

$$L ::= (T, D)$$

$$D ::= t \mid f$$

T is the syntactic category of program types, B are basic types, L are function type labels and D describe whether a function is deterministic or not. A program type denotes a collection of values. Each program expression has a type which describes which collection of values the outcome of performing the expression falls into. The outcome of performing a program expression may be one of the following: a basic value such as an integer or a boolean denoted by $b \in B$; an nd-value which is composed only of basic values denoted $\hat{b}$; or a function which is denoted $t_1 \rightarrow^{l} t_1$ or $t_1 \Rightarrow^{l} t_2$ depending on various properties of the function. In both cases t_1 is the type of the arguments of the function and t_2 is the type of the result of the function.

As noted above, a function may be nd-safe or nd-unsafe. These terms refer to whether or not a function may be safely applied to an nd-value. A function is itself may be an nd-value or a d-value which determines whether or not it may safely be passed as an argument and whether or not the result of the application is an nd-value. The type of an nd-safe function is denoted by $t_1 \Rightarrow^{l} t_2$ and an nd-unsafe function is denoted by $t_1 \rightarrow^{l} t_2$. The label on a function type describes whether the value may be an nd-value or not. A label $(\_, t)$ means that the value of an expression with this type is an nd-function and a label $(\_, f)$ means that the value is a d-function. Finally, an nd-function can be made ground in which case its type is t contained in the label: $l = (t, \_)$; a d-function may also be made ground in which case the type of the resulting value is the first component of the label, but this will be the same as the type *i.e.* $t = t_1 \rightarrow^{(t,d)} t_2$ and $t = t_1 \Rightarrow^{(t,d)} t_2$ for all d-functions.

$$\begin{array}{l}
G(\hat{\alpha}) = \alpha \\
G(_ \rightarrow^{(\alpha,_)} _) = \alpha \\
G(_ \Rightarrow^{(\alpha,_)} _) = \alpha \\
G(\alpha) = (\alpha) \\
\\
D(\hat{_}) = f \\
D(_ \rightarrow^{(_,d)} _) = d \\
D(_ \Rightarrow^{(_,d)} _) = d \\
D(_) = t \\
D(<>) = t \\
D(< _ \mapsto \alpha >) = D(\alpha) \\
D(A_1, A_2) = D(A_1) \,\&\, D(A_2) \\
\\
N(\hat{\alpha}) = \hat{\alpha} \\
N(\alpha_1 \rightarrow^{(\tau,_)} \alpha_2) = \alpha_1 \rightarrow^{(\tau,t)} \alpha_2 \\
N(\alpha_1 \Rightarrow^{(\tau,_)} \alpha_2) = \alpha_1 \Rightarrow^{(\tau,t)} \alpha_2 \\
N(\alpha) = \hat{\alpha}
\end{array}$$

Fig. 9. Typechecking Operators

Figure 9 defines operators which are used to test and transform types. The operators are as follows: G is used to transform a type to a ground type, *i.e.* a type which denotes possibly nd-values to a type which denotes only d-values; D is a predicate determines whether or not a type, or a bag of type assumptions (see below) denotes nd-values; finally, N transforms a type which denotes possibly d-values to a type which denotes definately nd-values.

A relation $\vdash$ is defined which associates a type with all nd-safe programs, *i.e.* all those programs which do not apply an nd-unsafe function to an nd-value. The relation is defined using a collection of rules each of which is of the form:

$$\frac{\begin{array}{l} a_1 \\ a_2 \\ \ldots \\ a_n \end{array}}{c} \; x$$

which defines that statement c holds providing that all the statements a_i hold; x is a label for the rule.

The relation is a set of 3-tuples (A, e, t) and is written $A \vdash e : t$ when $(A, e, t) \in \vdash$. The components of each tuple are: A a collection of type assumptions; e a program expression; and t a type. The type assumptions contain associations between identifiers and types which freely occur in e and an element of the relation $A \vdash e : t$ should be read as stating that program expression e has type t when all the freely referenced identifiers in e have types assigned by A.

A collection of assumptions contains mappings between program identifiers and types, $i \mapsto t$. The value A is a bag of such mappings where $<>$ is the empty bag, $<v>$ is a singleton bag containing the value v and A_1, A_2 is 'bag concatenation'. Type systems for ordinary programming languages use relationships of the form $S \vdash e : t$ where S is a set of identifier/type mappings. In such languages the assumptions which are contained in S may be used as many times as is required, *i.e.* identifiers in S may be referenced an arbitrary number of times. For programs involving nd-values, this is not the case: we wish to control the number of times an identifier is referenced. Bags of type assumptions allow us to do this.

The rules which define the type relation are described below. Type equality is defined by adding in the necessary information to either of the types in order that they are both ground or non-ground. For example **int** and $\hat{\textbf{int}}$ are equal by adding a ˆ to **int**. This rule allows types to be equal when they differ only in the ground-ness of the values which they denote and also ensures that non-ground-ness is preserved when comparing types. Constants k are typed by rule (a):

$$<> \vdash k : b \ (a)$$

where $b \in B$ is the type of constant k. An identifier i is typed by (b):

$$<i \mapsto t> \vdash i : t \ (b)$$

Notice that the type assumptions must contain exactly one mapping which is for the required identifier. Nd-safe functions are typed by (c):

$$\frac{\begin{array}{c} A\backslash i, <i \mapsto t_1> \vdash e : t_2 \\ G(A)\backslash i, <i \mapsto t_1> \vdash e : t_3 \\ t = t_1 \Rightarrow^{(t,t)} t_3 \end{array}}{A \vdash \lambda i.e : t_1 \Rightarrow^{(t,D(A\backslash i))} t_2} \ (c)$$

The rule (c) defines that the type of i need be added at most once to the type assumptions in order to type the body of the function. This means that the value of the identifier i is never copied during the execution of the function. Nd-unsafe functions are typed by rule (d):

$$\frac{\begin{array}{c} A, < i \mapsto t_1 >^{+} \vdash e : t_2 \\ G(A), < i \mapsto t_1 >^{+} \vdash e : t_3 \\ t = t_1 \rightarrow^{(t,t)} t_3 \end{array}}{A \vdash \lambda i.e : t_1 \rightarrow^{(t,D(A))} t_2} \ (d)$$

The rule (d) defines that the type of i needs to be added more than once to the type assumptions, *i.e.* the value of i might be copied during the execution of the body e. The type of μ-functions is given by rule (e):

$$\frac{\begin{array}{c} A, < i \mapsto G(t_1) >^{+} \vdash e : t_2 \\ G(A), < i \mapsto G(t_1) >^{+} \vdash e : t_3 \\ t = t_1 \Rightarrow^{(t,D(A))} t_3 \end{array}}{A \vdash \mu i.e : t_1 \Rightarrow^{(t,D(A))} t_2} \ (e)$$

Rule (e) adds the type of i to the assumptions as many times as is required. The type of i is made ground using G, this is because the application of a μ-function grounds its argument as described above. The type of nd-unsafe function application

$$\begin{array}{ll}\textbf{let} & \mathit{member} = \mathit{as\ in\ figure\ 3} \\ & \mathit{counter} = \mathit{as\ in\ figure\ 3} \\ & f\ n_1\ n_2 = \mu l.\ \textbf{if}\ (\mathit{member}\ n_1\ l)\ \&\ (\mathit{member}\ n_2\ l)\ \textbf{then}\ l\ \textbf{else}\ [] \\ & l = [1 \diamond 10, 2 \diamond 20] \\ \multicolumn{2}{l}{\ \textbf{in}\ f\ (\mathit{counter}\ 1002\ 2)\ (\mathit{counter}\ 1001\ 1)\ l}\end{array}$$

```
The following values were produced after 40469 transitions:
1: []
2: []
There are incomplete computations, continue? y
The following values were produced after 40486 transitions:
3: [1,2]
There are incomplete computations, continue? y
The following values were produced after 40533 transitions:
4: []
No further values were produced.
```

Fig. 10. Results from program no. 2 using nd-values and a μ-function

is described by rule (f):

$$\frac{\begin{array}{c} A_1 \vdash e_1 : t_1 \rightarrow^{(\_,t)} t_2 \\ A_2 \vdash e_2 : t_1 \\ D(t_1) \end{array}}{A_1, A_2 \vdash e_1 e_2 : t_2}\ (f)$$

Rule (f) defines that nd-unsafe function application is well typed when the type of the argument agrees with the type of the function domain and the argument type is deterministic. The type of nd-safe function application is described by rule (g):

$$\frac{\begin{array}{c} A_1 \vdash e_1 : t_1 \Rightarrow^{(\_,t)} t_2 \\ A_2 \vdash e_2 : t_1 \end{array}}{A_1, A_2 \vdash e_1 e_2 : t_2}\ (g)$$

Rule (g) defines that nd-safe function application is well typed when the argument type is the same as the function domain type. There is no requirement that the argument type is deterministic. Finally, rule (h) describes the type of a choice expression:

$$\frac{\begin{array}{c} A_1 \vdash e_1 : t_1 \\ A_2 \vdash e_2 : t_1 \end{array}}{A_1, A_2 \vdash e_1 \diamond e_2 : N(t_1)}\ (h)$$

Rule (h) uses the operator N to transform a possibly ground type to a non-deterministic type.

Figure 10 shows program 2 which has been changed to introduce a μ-function to protect the repeated references to the identifier l in the body of f. The repeated evaluation of program expressions which is necessary due to non-determinism is localized within the body of f. Notice, however that once an nd-value has been forced, the evaluation strategy causes subsequent program expressions to be performed for

$$\begin{array}{ll}
\textbf{let} & \textit{member} = \textit{as in figure 3} \\
& \textit{counter} = \textit{as in figure 3} \\
& f\ n = \mu l.\ \textbf{if}\ (\textit{member}\ n_1\ l)\ \textbf{then}\ \textit{hd}\ l\ \textbf{else fail} \\
& l = [1 \diamond 10, 2 \diamond 20] \\
\textbf{in} & \textit{counter}\ 1001\ (f'\ (\textit{counter}\ 1001\ 1)\ l)
\end{array}$$

```
The following values were produced after 60184 transitions:
1: 1
2: 1
No further values were produced.
```

Fig. 11. Results from program no. 3 using nd-values and a μ-function

each outcome of the force; in other words: forcing a value splits computation and there is no way of joining it back together again.

Suppose that $\gamma : E \to F$ is a translation from naive syntax to nd-syntax which inserts μ-functions in order that the program is well typed. Then we claim that the following diagram commutes:

$$\begin{tikzcd}
E \arrow[r, "X_\rho"] \arrow[d, "\gamma"'] & P(V) \\
F \arrow[r, "X'_\rho"'] & P(V') \arrow[u, "\phi^*"']
\end{tikzcd}$$

which shows that the nd-evaluation mechanism is consistent with the d-evaluation mechanism after nd-unsafe function application has been removed by introducing μ-functions.

5 Desugaring μ-functions

A μ-function can be used to localize the duplication of expression evaluation. Unfortunately, as soon as an nd-value is forced by applying a μ-function to it there is no way of joining the split computation back together again. For example, figure 11 shows a program which involves applying a μ-function to an nd-value. The calculation proceeds as follows. The identifier l is bound to an nd-value, a counter counts up to 1000, l is passed to f causing computation to split into four, two of the lists contain 1 and both of these computations return 1 as the head of the list, since two outcomes are produced from f the expression which counts to 1000 is performed twice. The number of transitions is therefore: $20030 + (2 * 20030) = 60090 \approx 60184$.

Although the nd-value must be forced when f is called, the results from the call of f can be re-composed as an nd-value. If this is done, then the final counting expression will be performed only once. Furthermore, the outcomes of the program shown in figure 11 are both identical. Once an nd-value has been forced, there is no way of knowing whether information is being duplicated.

In order to address these problems we propose two new operators: I and R which are pronounced *install* and *reify* respectively. The operator R is applied to a value which is possibly an nd-value and returns all the alternatives which it represents

$$
\begin{array}{ll}
\textbf{let} & \textit{member} = \textit{as in figure 3} \\
& \textit{counter} = \textit{as in figure 3} \\
& \textit{map}\ \_\ [] = [] \\
& \textit{map}\ f\ (x : l) = (f(x)) : (\textit{map}\ f\ l) \\
& \textit{append}\ []\ l = l \\
& \textit{append}\ (x : l_1)\ l_2 = x : (\textit{append}\ l_1\ l_2) \\
& \textit{flatten}\ [] = [] \\
& \textit{flatten}\ (x : l) = \textit{append}\ x\ (\textit{flatten}\ l) \\
& \textit{remdups}\ [] = [] \\
& \textit{remdups}\ (x : l) = \textit{remdups}\ l\ \textbf{when}\ \textit{member}\ x\ l \\
& \textit{remdups}\ (x : l) = x : (\textit{remdups}\ l) \\
& f\ n\ l = \textbf{if}\ \textit{member}\ n\ l\ \textbf{then}\ [\textit{hd}\ l]\ \textbf{else}\ [] \\
& f'\ n_1\ n_2\ l = I(\textit{remdups}(\textit{map}(f\ n_1\ n_2))(R(l))) \\
& l = [1 \diamond 10, 2 \diamond 20] \\
\multicolumn{2}{l}{\quad \textbf{in}\ \textit{counter}\ 1001\ (f'\ (\textit{counter}\ 1001\ 1)\ l)}
\end{array}
$$

```
The following values were produced after 40527 transitions:
1: 1
No further values were produced.
```

Fig. 12. A desugared μ-function.

as a list of d-values. The operator I is applied to a list of d-values and returns an nd-value which contains the members of the list as alternatives. Using I and R, the μ-function notation may be viewed as sugar:

$$\mu i.e = \lambda i.I(\textit{map}(\lambda i.e)(R(i)))$$

where the value of i is forced using R, each alternative is processed separately using $\lambda i.e$ and the result is packaged up as a single value using I. If more information is known about the μ-function, then more efficient desugarings are possible, for example:

$$\mu l.\textbf{if}\ p(l)\ \textbf{then}\ f(l)\ \textbf{else fail}$$

can be desugared as

$$I(\textit{flatten}(\textit{map}(\lambda l.\textbf{if}\ p(l)\ \textbf{then}\ [f(l)]\ \textbf{else}\ [])(R(l))))$$

where *flatten* is an operator which flattens a list of lists. Furthermore, if *remdup* is an operator which removes duplicates from a list of values then the following is event more efficient:

$$I(\textit{remdup}(\textit{flatten}(\textit{map}(\lambda l.\textbf{if}\ p(l)\ \textbf{then}\ [f(l)]\ \textbf{else}\ [])(R(l)))))$$

These transformations are used in figure 12 which is an equivalent program to that shown in figure 11. The counting expressions are evaluated once only in each case and only one outcome is produced, since duplicate outcomes have been removed.

6 Applying ND Technology to Data Fusion

The Data Fusion example which is defined in §2 uses non-determinism to advance the entities in the current global picture. The non-determinism is required because if no information is received to the contrary we must assume all the possible moves for an entity have taken place. At any future time, messages may be received which rule out many of the possible moves. The model of Data Fusion shows a very simple type of message which reports a definate observation of an entity at a particular position.

The Data Fusion example was executed with the following initial state:

$$([(1,0,0),(2,10,20)],0,[(2,\mathit{at}(1,0,0)),(2,\mathit{at}(2,10,21))])$$

which describes two entities with identifiers 1 and 2 and positions $(0,0)$ and $(10,20)$ respectively, a current time of 0 and two messages. Both messages occur at time 2 and are definate sightings of entities. The first sighting is for entity 1 at position $(0,0)$ and the second sighting is for entity 2 at position $(10,21)$.

When the functional language with the naive operational semantics described in §2 was used for the Data Fusion application, the result was never produced, even for this small example. The combinatorial explosion of machines caused the system to run out of space. When using the technology described in §5 the result was as follows:

```
The following values were produced after 76113 transitions:
1: ([(1,0,0),(2,10,21)],2,[])
```

The reasonable performance is due to the reification and installation of nd-values.

7 Conclusion and Further Work

Naive systems are a useful approach to the development of computer applications which involve search. This includes many branches of Artificial Intelligence software including Knowledge Based Systems. A problem with naive systems is that they are difficult to use as initial prototypes because their obvious operational semantics leads to a combinatorial explosion. This paper has described the problem and identified a potential solution.

The proposed solution is not complete. The following areas of work remain:

1. The typechecking system has not been implemented. As it stands it is not as general as it might be, for example application of nd-functions is not allowed.
2. The introduction of μ-functions has not been automated. This should be relatively easy.
3. The desugaring of μ-functions has not been automated. It should be possible to translate μ-functions to combinations of list, R and I expressions.
4. The Data Fusion example is very simple. There are many other areas of such an application where non-determinism can be used. For example messages may include noise, may be out of time order and new entities may be observed. These areas would be useful application drivers for the technology proposed in this paper.

References

Clark, A. N. 1994 Pattern Recognition of Noisy Sequences of Behavioural Events Using Functional Combinators. *Computer Journal* 37. 5.

Clark, A. N. 1996 A Formal Basis for the Refinement of Rule Based Transition Systems. *The Journal of Functional Programming* to appear March 1996.

Field, A. J. & Harrison, P. G. 1989 *Functional Programming* Addison-Wesley.

GEC CTR MOSES phase III.

GEC CTR MOSES phase IV.

Haugh, J. An Overview of M2, a Data Fusion Expert System. *Admiralty Research Establishment, Technical Memorandum AXT87005.*

Lakin, W. L. & Miles, J. A. H. 1985 IKBS in Multi-Sensor Data Fusion. *in Proc. 1st IEE Int. Conf. Advances in C^3 Systems, Bournemouth.*

Landin, P. J. 1964 The mechanical evaluation of expressions. *The Computer Journal*, 6.

Pecora, V. J. 1984 EXPRS A Prototype Expert System Using Prolog for Data Fusion. *The AI Magazine*, Summer.

Peyton Jones, S. L. 1987 *The Implementation of Functional Programming Languages* Prentice-Hall International Series in Computer Science.

Wadler, P. 1990 Linear types can save the world! *IFIP TC 2 Working Conference om Programming Concepts and Methods.* North Holland.