Nondeterminism with Referential Transparency in Functional Programming Languages*

F. W. BURTON
Department of Computer Science, 3160 MEB, University of Utah, Salt Lake City, Utah 84112, U.S.A.

Procedural programmers have found nondeterminism to be unavoidable in many parallel programs (e.g. operating systems). If function programming languages are to be usable as general-purpose parallel-programming languages then provision of some nondeterministic language construct appears to be necessary.

Existing constructs for nondeterminism are not functional and are not compatible with the mathematical foundations of functional programming, which require that the value of a function be uniquely determined by the values of its arguments.

We propose to solve this problem by placing the nondeterminism in pseudo-data. A program is passed an infinite tree of two-valued decisions, along with its input. These decisions may be fixed at run time, thereby permitting nondeterminism. Once fixed, a decision remains unchanged, so equivalent expressions must always have the same value.

The approach generalizes so that a program can make use of other run-time information such as the current time or the current amounts of available storage.

Received June 1986

1. INTRODUCTION

In conventional languages for parallel programming, such as Ada, nondeterminism has been found to be unavoidable, particularly in real-time applications (e.g. operating systems and embedded systems). If functional languages are to be used in similar applications, it seems likely that some form of nondeterminism will be required.

Various nondeterministic constructs for functional languages have been proposed. For example, the operator *amb* may return either of its two arguments subject to the restriction that it will return a defined result if either of its arguments is defined. Of course, *amb* is not a function, since it may not always return the same result given the same arguments.

There are two types of problem for which the use of nondeterminism is frequently advocated.

(1) Nondeterminism is useful in real-time applications where the behaviour of a program should depend on the order in which external events occur. For example, it might be desirable to merge the elements of two lists in the order in which elements become available. This type of nondeterminism appears to be necessary if functional programs are to be used for real-time computing.

(2) Nondeterminism may be used to implement nonsequential functions. For example, we can define a parallel form of the logical *and* by the equations:

\[
\begin{align*}
x \text{pand} \text{false} &= \text{false} \\
\text{false pand} \text{y} &= \text{false} \\
\text{true pand} \text{true} &= \text{true}
\end{align*}
\]

In this case, *a pand b* will yield a result if either *a* or *b* is *false*, even if the other is undefined (e.g. results in a nonterminating computation). Using *amb*, we can define

\[
a \text{pand} \text{b} = \text{amb} \text{(if } a \text{ then } b \text{ else false, if } b \text{ then } a \text{ else false)}
\]

Notice that *pand* is a deterministic function, even though its implementation requires the use of a nondeterministic operator or some other nonsequential function. In Section 4 we will argue that nonsequential functions are not necessary, and that the cost of supporting the bottom-avoiding nondeterminism, required to implement nonsequential functions, is excessive.

Having considered the case for nondeterminism, let us now consider the case against it. Referential transparency, the property that an expression always has the same value in the same environment, is central to the mathematical foundation for functional programs. This is lost if any of the above nondeterministic primitives are included in a language.

For example, consider the function \( f(x) < = (x = x) \). It would seem that \( f(a) \) should be true for any well-defined expression \( a \) for which a test for equality is permitted. If parameter passing is done by value, then \( f(a) \) always will be true. However, if parameter passing is done by name then \( f(\text{amb} \text{(2, 3)}) \) may be either \text{true} or \text{false}, depending on whether \text{amb} picks the same value both times it is evaluated. Clearly we have lost the Church–Rosser property, which says the value of an expression is independent of the order of evaluation provided the evaluation terminates. Similarly, algebraically equivalent expressions need not yield the same value.

The key problem with nondeterminism is in distinguishing independent choices, and ensuring that independent choices are not textually, or even algebraically, equivalent.

An interesting discussion of the complications caused by the introduction of a nondeterministic construct into a functional language may be found in Ref 4.

Our problem is to allow nondeterminism in a functional language while requiring all functions to be deterministic. The only way to do this appears to be to put the nondeterminism in data.

We proposed to supply each program with an extra argument consisting of an infinite (lazy) binary tree of values. (We choose a tree rather than a list, since any
number of subtrees may be extracted from an infinite tree.) In practice these values will be determined at run time (when used as arguments to a special function, choice), but once fixed will never change.

If \( v \) is a value from this tree, then choice \((v, a, b)\) may yield either \( a \) or \( b \) as its value. However, once a choice is made the value of \( v \) is fixed. The expression choice \((v, c, d)\) will yield \( c \) if and only if choice \((v, a, b)\) produced the value of \( a \) (assuming \( c \neq d \) and \( a \neq b \)). If \( a \) and \( b \) are equivalent to \( c \) and \( d \) respectively, then choice \((v, a, b)\) and choice \((v, c, d)\) are equivalent. If an independent choice is desired, a value from the tree other than \( v \) should be used.

While this approach can be extended to provide nondeterminism to solve both of the problem types described above, we will argue that nondeterminism of the second type (to avoid nonterminating computations) is not necessary and is very expensive to implement.

2. PRELIMINARIES

We will present our ideas in the context of a language which passes parameters by value. The list constructor, \( \text{cons} \), will be lazy. That is, when the expression \( \text{cons}(a, b) \) is passed by value, neither \( a \) nor \( b \) will be evaluated. For example, the function from, defined by from\((n) = \text{cons}(n, \text{from}(n+1)) \), will produce the infinite list of consecutive integers starting with \( n \). Since \( \text{cons} \) is lazy, evaluation of from\((n) \) will not result in a nonterminating computation.

If the expression from \((3+1) \) is an argument to another expression, then it will evaluate to cons \((4, \text{from}(4+1)) \). Notice that \( 3+1 \) evaluates to 4 since parameters are passed by value and \( 3+1 \) is an argument of from. On the other hand, \( 4+1 \) is not evaluated since it occurs within an expression which is an argument to the lazy list constructor cons. If \( x \) is the “value” of from \((3+1) \), head\((x) \) will evaluate to 4 and tail\((x) \) will evaluate via from \((4+1) \) to cons \((5, \text{from}(5+1)) \).

We will also assume a type tree with a lazy constructor, node, and selectors, left, right and contents. The first two selectors will select the left and right subtrees and contents will return the value in the root.

The ideas presented here also work well in a language where a programmer has explicit control over evaluation order, or with languages using various other evaluation strategies. We have chosen to present our ideas in terms of a simple language rather than the best possible functional language.

We will present our ideas using a simple and self-explanatory functional notation which does not exactly match any existing language.

3. NONDETERMINISTIC PRIMITIVES

We propose to extend a functional language to include a new type, decision, and a new function, choice. Every functional program will have an additional argument consisting of an infinite lazy tree of decisions.

A decision can have either of two values, one and two. The only primitive functions which can take a decision as an argument are the function choice and polymorphic functions which can take arguments of any type. (For example, an if statement may select a decision or cons may be used to construct a list of decisions.) No primitive function can construct a decision, so all decisions must originate from the tree passed to the program or the constants one and two. (Of course, selectors can select decisions previously inserted into data structures.)

The function choice is polymorphic, like the if statement. Given any type alpha:

\[
\text{choice} : \text{decision} \times \alpha \times \alpha \rightarrow \alpha
\]

The axioms defining choice are simple

\[
\begin{align*}
\text{choice} (\text{one}, a, b) &= a, \\
\text{choice} (\text{two}, a, b) &= b, \\
\text{choice} (\text{undefined}, a, b) &= \text{undefined},
\end{align*}
\]

where undefined (sometimes called bottom) represents any undefined or nonterminating computation. The reader should note that the above axioms do not guarantee bottom-avoiding nondeterminism. This is considered in the next section.

A correct program may make no assumption about the values of the decisions in the tree of decisions passed to a program as an argument, except that all values are well defined. (For example, if \( d \) is a decision in the tree of decisions, then \( d = \text{one} \) or \( d = \text{two} \) may be assumed.)

An implementation is free to set all values in the tree to whatever it pleases. In particular, in a simple, correct, but useless implementation, all values in the tree could be one. We are using nondeterminism to give a computer system the freedom to do things more efficiently. We do not require a system to take advantage of this freedom. This is essentially what happens in many procedural languages which permit nondeterminism.

A useful theorem for reasoning about functions using decisions is the following.

**Theorem 1.** (Case 1)\( d, a, b \) = if choice\((d, \text{true}, \text{false}) \) then \( a \) else \( b \)

Proof: consider the three cases

Case 1: \( d = \text{one} \).

\[
\begin{align*}
\text{choice}(\text{one}, a, b) &= a \quad \text{if true then a else b} \\
&= \text{if choice}(\text{one}, \text{true}, \text{false})\text{then a else b}
\end{align*}
\]

Case 2: \( d = \text{two} \). Similar to case 1.

Case 3: \( d = \text{undefined} \).

\[
\begin{align*}
\text{choice}(\text{undefined}, a, b) &= \text{undefined} = \text{if undefined then a else b} \\
&= \text{if choice}(\text{undefined}, \text{true}, \text{false})\text{then a else b}
\end{align*}
\]

Theorem 1 allows properties of the if statement to be used in reasoning about choice applications. Using the transformation in Theorem 1 may change the value a program actually produces, but it does not change what we can conclude about a program.

**Theorem 2.** (Case 2)\( d, f(a), f(b) \) = \( f(\text{choice}(d, a, b)) \)

Proof: similar to the proof of Theorem 1.

In general, it is desirable to do as little work as possible within a nondeterministic choice, since work on the unselected alternative will be discarded. Theorem 2 may be used to justify moving work outside a choice.

Readers should note that Theorems 1 and 2, suitably modified, do not apply to amb and similar constructs. For example, suppose \( f(1) = 3 \) and \( f(2) = \text{undefined} \). Then \( \text{amb}(f(1), f(2)) = 3 \) but \( f(\text{amb}(1, 2)) \) may be undefined.

If the value of a decision has not been used in a choice, then we will refer to it as a free decision. Once a decision has been used in a choice, it will be termed a fixed decision. (If relational tests are defined for decisions, then a decision becomes fixed once it is used in a test. The same is true of any polymorphic functions which must
know the value of a decision. We would prefer that such functions, including the test for equality, be prohibited, provided this is compatible with the type structure of a language.)

In practice, if a free decision, d, is used in a choice, the system is free to pick whichever value it would prefer for d. Once a decision is fixed, a choice statement is very similar to an if statement.

We note that the transformations in Theorems 1 and 2 may influence the value of a free decision. A programmer may wish to exploit this characteristic and make a choice at the most appropriate time.

4. BOTTOM AVOIDANCE

Some of the nondeterministic constructs proposed for functional languages are bottom-avoiding. For example, 

\[ \text{amb}(\text{undefined}, a) = \text{amb}(a, \text{undefined}) = a. \]

We could stipulate that if d is a free decision, then 

\[ \text{choice}(d, a, \text{undefined}) = a \]

and 

\[ \text{choice}(d, \text{undefined}, a) = a \]

noting that d may be a free decision in only one choice application, avoiding the conclusion that 

\[ \text{choice}(d, \text{undefined}, a) = \text{choice}(d, a, \text{undefined}) \]

which can be true only if \( a = \text{undefined} \).

There are three reasons why making choice bottom-avoiding should be avoided: (i) it would complicate the semantics; (ii) it would be expensive to implement; (iii) it is not necessary. We will give brief and informal arguments in support of each of these points.

4.1 Semantics

The problem with bottom-avoiding nondeterminism as described above is that the property of a decision being free is an operational rather than functional one. For example, if the arguments of strict operators such as + can be evaluated in either order or in parallel, then the expression choice(d, 1, 2) + choice(d, 3, undefined) will yield 4 if choice(d, 3, undefined) is evaluated first, but may yield undefined otherwise.

An alternative approach to bottom-avoiding nondeterminism which probably has simpler semantics, similar to those proposed for nondeterministic call-by-need in Ref. 4, is as follows. If any set of values for decisions can result in termination of the overall computation, then one such set of values must be chosen. Now the order in which choice applications are performed is no longer a factor, and the above problem vanishes. However, with \( n \) decisions, up to \( 2^n \) cases must be considered.

With either form of bottom-avoiding nondeterminism, Theorems 1 and 2 no longer hold, making program transformations more difficult to justify.

4.2 Implementation

There are two problems with implementing bottom-avoiding nondeterminism.

The most widely recognized problem with bottom-avoiding nondeterminism is that it is expensive in time to implement. With recursively nested nondeterministic constructs, a collection of processes will be produced. It is necessary for a correct implementation to ensure that each process progresses within a finite amount of time. This is often implemented using a circular list of processes. The processor will work on each one for a fixed amount of time and then move on to the next. The number of alternatives which must be considered may grow exponentially with the depth of the tree of nondeterministic choices. Much time is spent working on alternatives which are not selected. In addition, sometimes is required to kill off work once a nondeterministic choice is made.

A less widely recognized problem with bottom-avoiding nondeterminism is that storage costs may be very high. With recursively nested nondeterministic choices, a large decision tree may result. With bottom-avoiding nondeterminism, all paths through the tree must be followed in parallel. The number of nodes in a tree may grow exponentially with depth. Without bottom-avoiding nondeterminism, a depth-first search may be used. Hence, in this case, bottom avoidance results in an exponential growth in storage requirements. (This problem can be avoided by making repeated depth-first searches to depth 1, 2, ... However, this would increase the time requirements even more.)

4.3 Alternatives

All computable functions can be expressed in any reasonable deterministic functional language. Hence, bottom-avoiding nondeterminism does not add to the computational power of a language.

Some combinatorial search problems can be solved more simply by using a bottom-avoiding nondeterminist construct. However, these problems can also be solved by deterministic programs which stimulate bottom-avoiding nondeterminism by repeatedly doing some work on each active subproblem.

It has been argued that nondeterminism facilitates parallelism in a functional language. However, a deterministic construct for speculative computation in a functional language has been proposed. This incurs much of the cost of bottom-avoiding nondeterminism, but has deterministic semantics.

On balance, we believe the costs of bottom avoidance outweigh the advantages.

5. IMPLEMENTATION

There are several issues we wish to consider briefly in this section. Our objective is to convince the reader that a reasonably efficient implementation is possible. We will ignore a number of details.

We will start by considering the evaluation of an application of choice with a free decision. We will then consider the problem of ensuring that each decision is free in at most one application of choice.

To evaluate choice(d, a, b) where d is a free decision, we can create processes to evaluate a and b. In a parallel system with at least two free processors we would expect both computations to start executing. With one free processor we would expect one of the alternatives to start executing, and if that alternative could not progress we
would expect the other alternative to start executing. Once the evaluation of one alternative has terminated, the evaluation of the other should be killed. It is possible that the computation to be killed will have produced a tree of computations, either due to parallelism (e.g. evaluating both arguments of +, $\cdot$, etc. in parallel) or nested choices, or both. The problem of killing trees of processes is considered by Grim and Page.\textsuperscript{6}

It is useful to distinguish between mandatory and speculative computation. The evaluations of the two alternatives in a choice with a free decision will be called speculative. Other computation will be termed mandatory. (We are assuming that any computation which is initiated must be completed unless it is part of the evaluation of an alternative of a free choice.)

It is desirable to separate speculative processes and mandatory processes, and run speculative processes only when there are not enough mandatory processes for all processors. In this way, excessive speculative computation will not slow the progress of mandatory computation. In the absence of any nonterminating computation (mandatory or speculative), all speculative processes will eventually be executed or killed.

The parallelism that results from speculative computation is similar to OR-parallelism in Prolog.

The virtual tree machine approach\textsuperscript{1} may be used to keep storage requirements from becoming excessive. With this approach, a depth-first evaluation of an expression tree is favoured (as with sequential recursive computations), except that work which has been stacked for later evaluation by one processor can be performed by another processor which would otherwise be without work.

Thus far, the implementation strategy we have outlined could be used with almost any nondeterministic construct, provided bottom avoidance is not required. Let us now consider the implementation of the tree of free decisions.

Clearly, each decision should be fixed by a single application of choice. This requires mutual exclusion. This mutual exclusion can be enforced using conventional mechanisms such as semaphores. Each decision $d$ will require an associate semaphore, $d.s$, and a value, $d.val$. The value can be one, two or undetermined, with the value of a free decision being undetermined.

A process to evaluate choice($d, a, b$) will start with wait($d.s$). If $d.val$ is found to be one or two, then the process can immediately signal($d.s$) and proceed with the evaluation of the appropriate alternative. If $d.val$ is undetermined, the evaluation of both alternatives should be initiated and exclusive access to $d$ should be retained. As soon as the evaluation of one of the alternatives is finished, $d.val$ may be set, followed by (i) execution of signal($d.s$); (ii) initiation of the killing of the evaluation of the other alternative; and (iii) return of the value of the selected alternative.

If a choice process is killed, it is important that signal($d.s$) be executed if the process has exclusive access to a decision $d$. If a decision has been fixed by a process that is later killed, the decision should remain fixed in case it has been used in another choice.

Finally, we note that a free decision should never be copied, although pointers to a free decision can be copied. Once a decision is fixed it may be copied if necessary (e.g. to a processor's private local memory).

6. EXAMPLES

6.1 Referential Transparency

Consider the function

\[ f(x) \leq (x = x) \]

which was discussed in the introduction. Now $f(choice(d, a, b))$ will always be true (unless it fails to terminate because $a$ or $b$ fails to terminate). Even if choice($d, a, b$) is evaluated twice, it is constrained to produce the same result, since $d$ will be fixed after the first evaluation. Furthermore, any expression algebraically equivalent to choice($d, a, b$) must yield the same value.

6.2 Nondeterministic merge

In Ref. 8, Henderson proposes the function interleave to nondeterministically merge two lists as elements become available.

We can easily define interleave, but will need to give it an extra argument consisting of a tree of decisions. We will use the tree as a list by ignoring the left branches. We would expect the tree to be one of the infinitely many subtrees of the original argument to the program. Recall that the list constructor, cons, is lazy. We can now define interleave by

\[ \text{interleave}(t, x, y) \leq \text{choice} \left( \text{contents}(t), \begin{array}{l}
\text{if } x = \text{nil} \text{ then } y \\
\text{else } \text{cons}(a, \text{interleave}(\text{right}(t), \text{tail}(x), y) \\
\text{where } a \leq \text{head}(x), \\
\text{if } y = \text{nil} \text{ then } x \\
\text{else } \text{cons}(b, \text{interleave}(\text{right}(t), x, \text{tail}(y)) \\
\text{where } b \leq \text{head}(y) \end{array} \right) \]

We assume that

\[ \text{where } a \leq \text{head}(x) \]

forces the evaluation of $a$ and cannot terminate until the first element of $x$ is available. Since cons is lazy, the recursive application of interleave is deferred until the tail of the result of the original application of interleave is computed.

7. TIME STAMP AND SIMILAR

The use of pseudo-data, such as the tree of decisions, can be used to supply a program with run-time information about its environment.

For example, we could supply a program with a tree of timestamps and primitive operations stamp and compare having the following types:

\[ \text{stamp}: \text{timestamp} \rightarrow \text{timestamp} \]

\[ \text{compare}: \text{timestamp} \times \text{timestamp} \times \text{integer} \rightarrow \text{boolean} \]

The function stamp is the identity function, except that it has the side effects of setting its argument to the current time if the value of its arguments is still undetermined. The value of compare $(t_1, t_2, \text{diff})$ is $t_1 < t_2 + \text{diff}$, with times $t_1$ and $t_2$ being interpreted as integers. The evaluation of compare cannot terminate until at least one of its first two arguments has been stamped (perhaps by some other parallel computation). However, compare can use the fact that any undetermined
timestamp must have a value at least equal to the current
time. Let us consider how compare could be used to
implement a time-out test. Assume that stamp is to be
applied to t1 by some parallel computation and that t2 is
a timestamp that has not been previously used. We want
to write an expression which will return the value a once
t1 has been stamped, unless t1 is not stamped within limit
time units of the current time, in which case b is to be
returned. This is accomplished by the expression:

if compare(t1, stamp(t2), limit) then a else b.

Other run-time information could also be supplied in
the form of pseudo-data. For example, a spacetime
might give the amount of storage available at the time a
particular function is first applied to it. This could be
used where time–space trade-offs are possible.

To reach useful conclusions about programs using
timestamps (or spacetimes, etc.) the approach we took
to the semantics of decisions is probably not sufficient.
We can write programs which will force one timestamp to
be stamped before another. The correctness of a program
may depend on such facts. Semantics which include this
type of information are likely to be complicated. On the
other hand, reasoning about real-time programs in
procedural languages is also difficult.

8. CONCLUSION
We have seen that by placing nondeterminism in the data
(or at least pseudo-data) rather than within functions, it
is possible to provide nondeterminism within a functional
language while preserving referential transparency.

The approach can be generalised to provide a program
with any type of run-time information it might need. For
example, in a real-time language an infinite tree of
timestamps could be made available to the program.
Similarly, a program could be passed an infinite tree with
values which, when accessed for the first time, would
indicate how much memory was available. (This could be
useful if a program could make decisions concerning
time–space trade-offs at run time.)

Acknowledgement
The author would like to thank Hamilton Richards, of
the Burroughs Austin Research Center, for his helpful
comments on an earlier draft of this paper.

REFERENCES
1. F. W. Burton and M. M. Huntbach, Virtual tree machines.
   IEEE Transactions on Computers C-33 (3) 278–280
2. F. W. Burton, Annotations to control parallelism and
   reduction order in the distributed evaluation of functional
   programs. ACM Transactions on Programming Languages
3. F. W. Burton, Speculative computation, parallelism, and
   functional programming. IEEE Transactions on Computers
4. W. Clinger, Nondeterministic call by name is neither lazy
   nor by name. Proceedings of the 1982 ACM Symposium on
   LISP and Functional Programming, Pittsburgh, Penn.,
5. D. P. Friedman and D. S. Wise, Cons should not evaluate
   its arguments. 3rd International Colloquium on Automata
   Languages and Programming, Edinburgh, Scotland, 1976,
   257–284.
   expression-oriented multiprocessor system. ACM Trans-
   actions on Programming Languages and Systems 3 (1)
7. P. Henderson, Functional Programming: Application and
   Implementation. Prentice-Hall, Englewood Cliffs, NJ,
   (1980).
8. P. Henderson, Purely functional operating systems. In
   Functional Programming and its Applications, edited J.
   In Computer Programming and Formal Systems, edited P.
   Braffort and D. Hirschberg, pp. 33–70. North-Holland,
   Amsterdam (1963).