How to Replace Failure by a List of Successes
A method for exception handling, backtracking, and pattern matching in lazy functional languages

Philip Wadler
Programming Research Group, Oxford University
11 Keble Road, Oxford, OX1 3QD

Abstract

Should special features for exception handling, backtracking, or pattern matching be included in a programming language? This paper presents a method whereby some programs that use these features can be re-written in a functional language with lazy evaluation, without the use of any special features. This method may be of use for practicing functional programmers; in addition, it provides further evidence of the power of lazy evaluation. The method itself is straightforward: each term that may raise an exception or backtrack is replaced by a term that returns a list of values. In the special case of pattern matching without backtracking, the method can be applied even if lazy evaluation is not present. The method should be suitable for applications such as theorem proving using tacticals, as in ML/LCF.

1. Introduction

Exception handling, backtracking, and pattern matching are some of the tools in a programmer's conceptual toolkit. Some programming languages include features that support these tools, others do not. Exception handling is included in ML [Gordon et al 79] and some dialects of Lisp (e.g., `THROW` and `CATCH` in Maclisp). Backtracking is included in Prolog [Kowalski 79], and some artificial intelligence languages such as Planner [Bobrow and Raphael 74]; extending LispKit Lisp to include backtracking is discussed in [Henderson 80, chapter 7]. Pattern matching is included in Poplar [Morris et al 80] and Icon [Griswold 82, 84]. (In this paper, pattern matching is used in the sense of parsing, as opposed to pattern matching against the left-hand side of an equation as in, for example, KRC [Turner 81].)

This paper presents a method whereby many programs that use exception handling, backtracking, or pattern matching can be written in a functional language with lazy evaluation. No special features of the language, other than lazy evaluation, are needed to support the use of these tools. (Lazy evaluation is described in [Henderson and Morris 76, Friedman and Wise 76].) The method works best in a language that includes a "list comprehension" notation, such as the "set expression" notation in KRC [Turner 81]; but the use of list comprehension is not essential.

There are two reasons for presenting this method. First, it may be of use to practicing functional programmers. Second, it provides a further demonstration of the power of lazy evaluation.

Whereas most languages must add special features to support the use of new control structures (such as exception handling, backtracking, and pattern matching), in a lazy functional language these can be supported by simply defining new data structures and functions for manipulating them (or using existing data structures, like lists, in a new way).

The method itself is straightforward. Each term that may raise an exception or backtrack is replaced by a term that returns a list of values. Thus, a term that might either raise an exception (failure) or return the value v (success) is replaced by a term that either returns the empty list [ ] (failure) or the unit list [v] (success). Similarly, a term that might return many values through backtracking is replaced by a term that returns a list of those values.

Two ways of combining terms that may raise exceptions or backtrack are discussed, "or" combination and "and" combination. It is shown that these correspond to well-known functions on lists, namely append and cartesian product.

The method is then applied to pattern matching. Patterns are represented by functions, and higher-order functions are used to combine patterns into new patterns. Higher-order functions corresponding to alternation, sequencing, and repetition are discussed. This method is useful for parsing regular expressions or context free grammars. In the special case of pattern matching without backtracking, the method can be applied even if lazy evaluation is not present. It is also suggested that the method could be used to define "tacticals" like those used for constructing proofs in ML/LCF [Gordon et al 79].

Many different variations of exception handling, backtracking, and pattern matching have been proposed. This paper cannot discuss all of them, and no claim is made that the method I describe applies to all of their various forms. In particular, the method does not apply to forms of exception handling where several different kinds of named exceptions can be raised. These kinds of exceptions can be replaced by other methods, such as the use of type unions or special error values [Black 82].

Some of what is said in this paper is well-known folklore to some functional programmers. For example, Turner's solution to the Eight Queens problem in [Turner 81] uses a similar technique. Despite this, or perhaps because of it, I felt it was worth writing down. Many of the ideas here, although obvious, escaped my notice for many years, and I had a feeling of insight and satisfaction on discovering them. I hope the reader will also receive a little of this satisfaction.

This paper is organized as follows. Section 2 discusses exception handling and backtracking. Section 3 shows how to replace these by operations on lists. Section 4 develops functions for pattern matching. Section 5 presents conclusions.

2. Exception handling and backtracking

This section describes some language features that are traditionally used to support exception handling and backtracking. The next section will show how these features can be replaced by list manipulation in a language with lazy evaluation. As a running example, this section and the next will discuss programs that use association lists.

Exceptions. A routine can raise an exception to indicate that it has encountered an "unusual" situation. (Much of the argument over exception handling revolves around the question of what should or should not be considered "usual".) Evaluation of an expression will either "succeed" and return a value, or else "fail" when an exception is signaled.

For example, consider a routine `assoc` that looks up entries in an association list. That is, given a list of pairs `xys` and a value `x`, the call `assoc xys x` returns a value `y` such that the pair `(x, y)` is in the list `xys`. If there is no such `y`, then the call should raise an exception. Thus,

```
assoc [("a",1),("b",2)] "b"  =  2
assoc [("a",1),("b",2)] "c"  =  FAIL
```

where `FAIL` indicates that evaluation raises an exception. One way to write this routine is

```
(1)     assoc [] x  =  FAIL
        assoc ((x',y):xys) x  =  y              IF x' = x
                                 assoc xys x    OTHERWISE
```

(Here `:` denotes cons. The language used in this paper is similar to KRC [Turner 81], with some differences.)

Roughly speaking, there are two ways in which values can be combined in a language with exceptions. I call these "or" combination and "and" combination.

"Or" combination. "Or" combination is a way of combining two terms so that evaluation of the resulting term succeeds (i.e., does not raise an exception) whenever evaluation of the first term *or* the second term succeeds.

"Or" combination will be written using the `?` operator. (The `?` operator is used in older versions of ML [Gordon et al 79]; other languages have similar operators.) The term `e1 ? e2` is evaluated as follows. First, `e1` is evaluated. If it succeeds, then its value is returned. Otherwise, `e2` is evaluated. If this evaluation succeeds, then its value is returned. Otherwise, an exception is raised. Note that "or" combination is asymmetric, in that it tries its left-hand argument first.

For example,

```
(assoc xys1 x) ? (assoc xys2 x)
```

first looks up x in xys1. If it is there, the associated value is returned. Otherwise, it looks up x in xys2.

"And" combination. "And" combination is a way of combining two terms so that evaluation of the resulting term succeeds whenever evaluation of the first term *and* the second term succeeds.

Whereas "or" combination is explicitly represented by the ? operator, "and" combination is implicit in most operations. Usually, if OP is some operator (say, +) then e1 OP e2 is evaluated using "and" combination. That is, both e1 and e2 are evaluated. If both evaluations succeed, then the two values are combined using OP and the resulting value is returned. If either evaluation raises an exception, then e1 OP e2 raises an exception.

For example,

```
        (assoc xys1 x) + (assoc xys2 x)
```

looks up x in xys1 and looks up x in xys2. If it is present in both lists, the sum of the associated values is returned. Otherwise, an exception is raised.

Backtracking. Backtracking is a generalization of exception handling. In exception handling, each expression may return either no value (i.e., raise an exception) or one value. In backtracking, each expression may return zero, one, or more values.

For example, under backtracking we might wish to arrange that the call

```
        assoc [("a",1),("b",2),("a",3)] "a"
```

returns two values, 1 and 3. (An appropriate definition for this version of assoc is given below.)

"Or" combination generalizes to backtracking in a natural way. Under backtracking, the expression e1 ? e2 first evaluates e1 and returns all values returned by e1, and then evaluates e2 and returns all values returned by e2. For example, under backtracking,

```
        (assoc [("a",1),("b",2)] "a") ? (assoc [("a",3),("c",4)] "a")
```

returns two values 1 and 3, whereas under exception handling it returns just 1.

Note that ? can be used to introduce functions that return more than one value. For example, a backtracking version of assoc might be written as follows:

```
(2)     assoc [] x  =  FAIL
        assoc ((x',y):xys) x  =  y ? assoc xys x   IF x' = x
                              =  assoc xys x       OTHERWISE
```

The only difference between this definition and definition (1) occurs in the second line: rather than simply returning y, the new definition returns y and also returns any values associated with x in the remaining list xys.

"And" combination also generalizes to backtracking. Under backtracking, the expression e1 OP e2 returns one value v1 OP v2 for each value v1 returned by e1 and each value v2 returned by e2. For example, under backtracking,

```
(assoc [("a",1),("a",2)] "a") + (assoc [("a",3),("a",4)] "a")
```

returns four values: 4, 5, 5 (again), and 6 (that is, 1+3, 1+4, 2+3, and 2+4).

In general, if e1 returns m values and e2 returns n values, then e1 ? e2 will return m+n values, and e1 OP e2 will return m × n values. It is the multiplicative behaviour of "and" combination that leads to the exponential running times of many programs involving backtracking.

Backtracking can be implemented by various mechanisms, usually involving stacks or coroutines. (See [Henderson 80] for an example of a stack implementation, and [Griswold 82, 84] for an example of a coroutine implementation.) All of these mechanisms have in common the idea that evaluation of an expression may be "suspended" after it has returned one value, and then may be "resumed" later if more values are needed. This is an important feature of backtracking, because the user can write programs that specify a very large solution space, but only those portions of the space necessary to find an answer are explored.

Note that, as it is treated in this paper, exception handling can be viewed as an approximation to backtracking that is cheaper to implement.

3. Replacing exception handling and backtracking by lists

An alternative to exception handling or backtracking is to rewrite each routine that may fail or backtrack as a function that returns a list of results. Thus, a routine that may raise an exception is rewritten as a function that returns either [] (the empty list, indicating failure) or [v] (a list with one element v, indicating a succesful return of the value v). A routine involving backtracking is rewritten as a function that may return a list of zero, one, or more values.

This section will first show how the method can be applied to backtracking, and then discuss exception handling as a special case.

Backtracking. Using the list method, the backtracking version of assoc behaves as follows:

```
assoc [("a",1),("a",3)] "a"  =  [1,3]
assoc [("a",1),("b",2)] "b"  =  [2]
```

```
        assoc [("a",1),("b",2)] "c"  =  []
```

This version of `assoc` might be written like this:

```
(3)     assoc xys x  =  [y | (x',y) ← xys, x' = x]
```

Here the expression in brackets denotes the list of all `y` such that the pair `(x',y)` is in `xys` and `x' = x`. This notation is called "list comprehension"; it is analogous to "set comprehension" notation in set theory, such as `{y | (x',y) ∈ xys, x' = x}`. (In KRC, list comprehensions are called "set expressions" and the notation is slightly different.)

Definition `(3)` is more compact than definitions `(1)` and `(2)`, and perhaps also more clear (to those familiar with the notation). It also has a simple interpretation: it is the image of `x` through the relation specified by `xys`.

"Or" combination is achieved simply by appending two lists together. Thus, instead of writing `e1 ? e2` one writes `e1 ++ e2`. (Here `++` denotes list append.) The answer list is non-nil (success) if the first list is non-nil *or* the second list is non-nil.

"And" combination is achieved by a simple idiom. Instead of writing `e1 OP e2` one writes

```
        [v1 OP v2 | v1 ← e1, v2 ← e2] .
```

This denotes the list that contains the value `v1 OP v2` for each `v1` in the list `e1` and each `v2` in the list `e2`. The answer list is non-nil if the first list is non-nil *and* the second list is non-nil.

For example, using these methods `(assoc xys1 x) ? (assoc xys2 x)` can be rewritten

```
        assoc xys1 x ++ assoc xys2 x
```

and `(assoc xys1 x) + (assoc xys2 x)` can be rewritten

```
        [y1 + y2 | y1 ← assoc xys1 x, y2 ← assoc xys2 x] .
```

Note that the idiom for "and" combination closely resembles the function for cartesian product:

```
        cp xs ys  =  [(x,y) | x ← xs, y ← ys]
```

For example,

```
        cp [1,2] [3,4]  =  [(1,3),(1,4),(2,3),(2,4)]
```

One could write a function like `cp` to encapsulate "and" combination, but in practice it is usually easier to use the idiom.

As was previously mentioned, implementations of backtracking usually arrange that evaluation of an expression is "suspended" after it returns one value, and may later be "resumed" if more values are needed. Lazy evaluation has exactly the same property: evaluation of a list is "suspended" after the first element is returned, and may later be "resumed" when more elements are needed. Thus, under lazy evaluation the order of evaluation is essentially the same in programs that use backtracking and corresponding programs written using lists.

In fact, lazy evaluation suspends the evaluation of both the head and tail components of a list, not just the tail. In addition, under lazy evaluation no argument of a function is evaluated unless it is needed. Thus, a program using lists under lazy evaluation may perform even fewer calculations than an equivalent program using backtracking.

Exception handling. Exception handling is simply a restriction of backtracking: each expression returns a list of length zero or one, instead of a list of arbitrary length. For most purposes, the method described above will serve just as well when applied to exception handling as when applied to backtracking.

Occasionally, however, backtracking will return additional answers that are not desired. To duplicate the behaviour of exception handling precisely, one may introduce the function cut, which truncates a list to contain at most one element. This function is defined by:

```
cut []        =  []
cut (x:xs)    =  [x]
```

For example, one can define an analogue to definition (1) by

```
(4)     assoc xys x  =  cut [y | (x',y) ← xys, x' = x]
```

(This is just definition (3) with cut added.) Using this definition of assoc, one has that

```
assoc [("a",1),("a",3)] "a"
```

returns [1] instead of [1,3].

Again, the order of evaluation is identical in programs that use exception handling and corresponding programs that use lists under lazy evaluation.

Exception handling in Lisp. It is interesting to note that a representation like this one is often used in Lisp programs, but for a different reason. Frequently Lisp functions are designed to return NIL to indicate failure. Furthermore, sometimes a function will return a unit list [x] to indicate the answer x. This is done so that if x is NIL it will not be confused with a failure value.

The reason NIL is used to represent failure is that in Lisp NIL is also used to represent false. The standard Lisp OR function returns the first non-NIL value in its argument list. Thus, "or" combinations can be conveniently written using the OR function, which seems natural enough. Similarly, some "and" combinations can be written using AND. (The correspondence for AND is not quite as good, since if all arguments are non-NIL, AND simply returns its last argument.)

In fact, I have used just this representation in some of my Lisp programs in the past, without noticing that this corresponded to returning a list of the succesful results. It never occured to me that I could achieve an effect like backtracking by using append and cartesian product in place of OR and AND!

4. Pattern matching

One use of exception handling or backtracking is to facilitate pattern matching. For example, pattern matching was a major motivation behind the introduction of exception handling in Poplar [Morris et al 80] and backtracking in Icon [Griswold 82, 84]. This section discusses a collection of functions for pattern matching, designed using the methods discussed in the last section.

Patterns. A pattern will be represented by a function that takes a string xs as input, and either signals failure or returns a pair (v, xs′), where v is the "value" matched by the pattern (that is, a kind of parse tree), and xs′ is the remaining, unmatched portion of the string. Since failure is represented as before, this means a pattern has the type

```
String → List (Value × String)
```

That is, a list of pairs of the form (v, xs′) is returned, the empty list denoting failure. If a pattern matches a string in more than one way, a list of more than one pair may be returned.

For example, it will be shown later how to define a pattern expr that matches fully-parenthesized arithmetic expressions:

```
expr "not an expr"   =  []
expr "42"            =  [(["4", "2"],  "")]
expr "42 and more"   =  [(["4", "2"],  " and more")]
expr "(42+69)"       =  [(["(", ["4", "2"], "+", ["6", "9"], ")"],  "")]
```

The remainder of this section is organized as follows. First, a few basic patterns are defined: literals, the empty pattern, and the pattern that always fails. Next, three ways of combining patterns are defined: alternation, sequencing and repetition. Next, some useful variations on these functions are described. Finally, some points relating to order of evaluation are discussed, and a generalization of pattern matching is described.

Literals. The literal pattern `lit x` matches a string whose first character is `x`. For example,

```
lit 'a' "apple" = [('a',"pple")]
lit 'a' "banana" = []
```

This pattern is defined by

```
lit x []  =  []
lit x (x':xs)  =  [(x,xs)],   if x = x'
               =  [],   otherwise
```

(Here strings, surrounded by double quotes, are equivalent to lists of characters, surrounded by single quotes, e.g., "abc" = ['a', 'b', 'c'].)

Empty and fail. The pattern `empty v` always succeeds; it matches the empty list and returns value `v`. The pattern `fail` always fails; it never matches anything. These patterns are defined by:

```
empty v xs   =  [(v,xs)]
fail xs      =  []
```

Alternation. Alternation is simply "or" combination. The alternation function `alt p q xs` succeeds if either `p` matches `xs` or `q` matches `xs`. For example,

```
alt (lit 'a') (lit 'b') "banana"    =  [('b',"anana")]
alt (lit 'a') (lit 'b') "cucumber"  =  []
```

Alternation is easy to define, using the method for "or" combination described in section 3:

```
alt p q xs  =  (p xs) ++ (q xs)
```

It is easy to prove that

```
alt p fail  =  alt fail p  =  p
```

so `fail` is an identity for `alt`.

Sequencing. Sequencing is like "and" combination, but with a twist, since the remaining string of the first match is passed on to the second. The sequencing function `seq f p q xs` matches `p` followed by `q`, and uses the function `f` to combine the values returned by `p` and `q`. For example,

```
seq list2 (lit 'b') (lit 'a') "banana"  =  [("ba","nana")]
```

(Here `list2 x y = [x,y]`, so `list2 'b' 'a' = "ba"`.)

Sequencing is defined as follows:

```
    seq f p q xs
        =  [(f v1 v2, xs2) | (v1,xs1) ← p xs, (v2,xs2) ← q xs1]
```

First, `p` is matched against `xs`, returning a value `v1` and a remaining string `xs1`. Next, `q` is matched against `xs1`, returning a value `v2` and a remaining string `xs2`. Finally, the value returned is `f` applied to `v1` and `v2`, and the remaining string returned is `xs2`.

It is easy to prove that

```
    seq f (empty v) p  =  p     if  f v x = x
    seq f p (empty v)  =  p     if  f x v = x
```

so `empty v` is an identity for `seq f` whenever `v` is an identity for `f`. (However, often `f` will be a function like `list2` that has no identity.)

Repetition. The repetition function `rep p xs` repeatedly matches the pattern `p` against `xs` until it fails. If `p` can match `xs` up to `n` times, then `n+1` different matches are returned, one with `n` matches, one with `n-1`, and so on, down to one with `0` matches. For each match, the value returned is a list of the values matched by `p`. For example,

```
    rep (lit 'a') "aardvark"
        =  [("aa","rdvark"), ("a","ardvark"), ("","aardvark")]
```

The longest match is returned first. Note that there will always be at least one match, since the empty match always succeeds.

Repetition can be defined using alternation, sequencing, and recursion:

```
    rep p  =  alt (seq cons p (rep p)) (empty [])
```

(Here `cons x xs = x:xs`.)

It is also useful to have a pattern `rep1 p` that matches one or more repetitions of `p`, instead of zero or more. For example,

```
    rep1 (lit 'a') "aardvark" = [("aa","rdvark"), ("a","ardvark")]
```

This pattern is defined by:

```
    rep1 p  =  seq cons p (rep p)
```

Lists of patterns. The functions alt, seq, and lit are slightly inconvenient. It would be better to have functions alts and seqs that take lists of patterns, and lits that takes a string instead of a single character. To define these, one first defines the familiar "reduce" operator (also sometimes called "accumulate" or "list iteration"):

```
reduce f a []       =  a
reduce f a (x:xs)   =  f x (reduce f a xs)
```

(For example, reduce plus 0 xs sums the elements of a list xs, and reduce cons [] xs is equal to xs.) Then the desired functions are defined by:

```
alts ps  =  reduce alt fail ps
seqs ps  =  reduce (seq cons) (empty []) ps
lits xs  =  seqs [lit x | x ← xs]
```

Using the new functions, one can write, for example,

```
alts [lits "apple", lits "banana", lits "cucumber"] "banana split"
    =  [("banana"," split")]

seqs [lits "ba", rep (lits "na")] "banana"
    =  [(["ba", ["na", "na"]], ""),
        (["ba", ["na"]], "na"),
        (["ba", []], "nana")]
```

As another example, a grammar of fully-parenthesized arithmetic expression can be defined as follows:

```
expr    =  alts [number, seqs [lits "(", expr, op, expr, lits ")"]]
number  =  repl digit
op      =  alts [lits "+", lits "-", lits "*", lits "/"]
digit   =  alts [lits "0", lits "1", ..., lits "9"]
```

Examples of the use of expr were given at the beginning of this section.

Applying functions to values. The application function app f p applies the function f to the value returned by the pattern p. It is defined by:

```
app f p xs  =  [(f v, xs') | (v,xs') ← p xs]
```

For example, let mknumber be a function that converts a list of digits to a number (e.g., mknumber ["4", "2"] returns 42). In the grammar for expr given above, change the definition of number to

```
number  =  app mknumber (repl digit)
```

Then the value returned by `expr "(42+69)"` will be `["(", 42, "+", 69, ")"]`.

Recursive descent. If lazy evaluation is used, pattern matching in this style evaluates in the same way as recursive descent parsing with backtracking. As with recursive descent, one must be careful to avoid left-recursive definitions.

A common special case of recursive descent parsing is recursive descent parsing without backtracking. This corresponds to those patterns that can be matched using exception handling rather than backtracking. (The grammars that can be matched in this way include the LL(1) grammars [Aho and Ullman 77].)

A version of pattern matching equivalent to recursive descent parsing without backtracking can be obtained as follows. First, uses of `alt` should be replaced by `alt'`, where

```
alt' p q  =  cut . alt p q
```

This modification ensures that `alt` returns at most one alternative. Second, uses of `seq` should be replaced by `seq'`, where

```
seq' f p q  =  seq f p (nofail . q)
```

This modification ensures that if the pattern `p` is matched in a sequence, then the pattern `q` must match as well, or else a run-time error will occur. (Here `.` is function composition, `(f . g) x = f (g x)`, and `cut` is as defined in section 3, and `nofail` is defined and explained below.)

Pattern matching without backtracking is important for two reasons. First, it improves the behaviour of evaluation under lazy evaluation. Second, it makes it possible to apply these techniques even if one is using a language *without* lazy evaluation. These two points will be discussed in turn.

Behaviour under lazy evaluation. To begin, consider again the use of full backtracking. In this case, the definition of repetition given above is not quite as "lazy" as one would like. For example, consider the term

```
(5)     rep (lit 'a') "a..."
```

One would expect this to evaluate to a term of the form

```
(6)     [(['a', ...], ...), ...]
```

regardless of what comes after the "a" in "a...". This property is important for programs that wish to use lazy evaluation to process a long list, producing each new bit of the output as each new bit of the input is read.

But in fact, `(5)` does not always evaluate to `(6)`. If the input string "`a...`" is

```
'a' : ⊥
```

(where ⊥ denotes a term whose evaluation never terminates) then `(5)` evaluates to ⊥. The problem is that one expects `rep p xs` to always succeed, that is to return a non-empty list of matches, since it will match the empty list if nothing else. But this expectation is false, because `rep p ⊥` returns ⊥. Of course, ⊥ is a degenerate case, but it is easy to reason about, and hence is useful for explaining the behaviour of the program under lazy evaluation. See [Wadler 85] for a fuller explanation of this point.

This problem can be solved by introducing a new function, `nofail`. Just as `cut` guarantees that its result list contains at most one match, `nofail` guarantees that its result list contains at least one match. It is defined by

```
nofail u  =  (first (head u), second (head u)) : (tail u)
```

(Here `first (x,y) = x`, `second (x,y) = y`, `head (x:xs) = x`, and `tail (x:xs) = xs`.) If `u` is the result of a succesful match, then `nofail u` is the same as `u`; otherwise an error has occured. The definition of `rep` is changed to

```
rep p  =  nofail . (alt (seq cons p (rep p)) (empty []))
```

(This is identical to the previous definition, except for the inclusion of `nofail`.) Using this definition, `rep p xs` always returns a non-empty list of matches, and term `(5)` now evaluates in the desired way. (In particular, `rep (lit 'a') ⊥` reduces to `nofail ⊥` which returns `((⊥,⊥):⊥)`, and `rep (lit 'a') ('a':⊥)` returns `((('a':⊥),⊥):⊥)`.)

Returning to recursive descent without backtracking, it is not hard to see that the use of `nofail` in `seq′` improves the behaviour in a similar way. When matching against any sequence the first part of the result can be returned before the entire sequence has been matched, and similarly for repetitions, since these are defined using sequences. More generally, `nofail` can be used in a similar way to improve behaviour under lazy evaluation whenever a pattern is not expected to fail.

Pattern matching without lazy evaluation. In the special case of recursive descent parsing without backtracking, the use of lazy evaluation is *not* necessary. This is because the arguments to the functions `alt` and `seq` are themselves functions (and thus can be fully evaluated) and the results are lists of length at most one (and thus can also be fully evaluated).

All that is necessary is to rewrite `alt'` and `seq'` as follows:

```
alt'' p q xs
    =  CASE (p xs) OF
            []          ⇒  q xs
            [(v,xs')]   ⇒  [(v,xs')]

seq'' f p q xs
    =  CASE (p xs) OF
            []          ⇒  []
            [(v1,xs1)]  ⇒  CASE (q xs1) OF
                              [(v2,xs2)]   ⇒   [(f v1 v2, xs2)]
```

It is not hard to prove that if all patterns return a list of length zero or length one, then `alt''` and `seq''` under non-lazy evaluation are equivalent to `alt'` and `seq'` under lazy evaluation.

Generalizations of pattern matching. The functions `alt`, `seq`, and `rep` have wider application than simple pattern matching against strings. In general, a pattern can be regarded as a function that takes a "problem" or "goal" `x`, and returns a list of pairs of the form `(v,x')`, where `v` is a partial solution, and `x'` is a remaining (and one hopes, smaller) sub-goal that must be solved.

String matching patterns are an instance of this, where the problem is a string to be matched, the partial solution is the value (e.g., parse tree) of the portion matched, and the remaining sub-problem is the remainder of the string to be matched.

For another instance, consider the design of a theorem prover. Here the problem is a list of formulas to be proved, the partial solution is a step in the proof, and the remaining sub-problem is a transformed list of formulas to be proved. The theorem proving system LCF structures its proofs in a way similar to this, and the "tacticals" from which LCF constructs proofs, `ORELSE`, `THEN`, and `REPEAT`, are very similar to `alt`, `seq`, and `rep`. Similar tacticals are also useful for applying rewrite rules to terms and formulas [Paulson 83].

Some people have said that exception handling in ML is necessary, because it is used to construct tacticals in LCF. I believe that one could construct tacticals just as easily by using the above methods, and so exception handling is not necessary to support the use of tacticals. Further, since tacticals in LCF do not use backtracking, the techniques above can be applied even if lazy evaluation is not present.

5. Conclusions

This paper has shown a simple method whereby programs that use exception handling, backtracking, and pattern matching can be written in a lazy functional language without the use of any special features. In the important special case of pattern matching without backtracking, the same method can be used in a functional language without lazy evaluation.

This method may be of use to practising functional programmers. Also, this method shows that features that were previously thought to be necessary can perhaps be omitted. In particular, it is suggested that exception handling is not necessary in ML to support the use of "tacticals" in LCF.

Finally, this method further demonstrates the power of lazy evaluation. Whereas most languages must introduce new features to support new control structures, a lazy functional language can often support the same control structures just by defining new data structures and new functions. This suggests a comparison between lazy functional languages and Lisp. Whereas the power of Lisp is that it is relatively easy to define new language features, the power of lazy functional languages is that one does not need to.

Acknowledgements.

Jim Morris first introduced me to the idea of pattern matching in functional languages. I'd also like to thank Richard Bird, Mike Gordon, Tony Hoare, Larry Paulson, and Bernard Sufrin for their comments.

Part of this research was performed while on a research fellowship supported by ICL.

References

[Aho and Ullman 77] Aho, A. V., and Ullman, J. D. Principles of Compiler Design. Addison-Wesley, 1977.

[Black 82] Black, A. P. Exception handling: the case against. D. Phil. dissertation, Oxford University, 1982. Also available as University of Washington technical report TR 82-01-02, 1982.

[Bobrow and Raphael 74] Bobrow, D. G., and Raphael, B. New programming languages for artificial intelligence research. Computing Surveys, 6(3): 155-174, September 1974.

[Friedman and Wise 76] Friedman, D. P., and Wise, D. S. Cons should not evaluate its arguments. In Michaelson and Milner (editors), Automata, Languages, and Programming, 257-284, Edinburgh University Press, 1976.

[Gordon et al 79] Gordon, M. J., Milner, R., and Wadsworth, C. P. Edinburgh LCF. Lecture Notes in Computer Science 78. Springer-Verlag, 1979.

[Griswold 82] Griswold, R. E. The evaluation of expressions in Icon. ACM Transactions on Programming Languages and Systems, 4(4): 563-584, October 1982.

[Griswold 84] Griswold, R. E. The evaluation of expressions in Icon. Symposium on Lisp and Functional Programming, ACM, Austin, TX, August 1984.

[Henderson and Morris 76] Henderson, P., and Morris, J. H. A lazy evaluator. In 3'rd Symposium on Principals of Programming Languages, 95-103, ACM, Atlanta, GA, 1976.

[Henderson 80] Henderson, P. Functional Programming: Application and Implementation. Prentice-Hall, 1980.

[Kowalski 79] Kowalski, R. Algorithm = Logic + Control. Communications of the ACM, 22(7), July, 1979.

[Morris et al 80] Morris, J. H., Schmidt, E., and Wadler, P. L. Experience with an applicative string processing language. 7'th Symposium on Principles of Programming Languages, ACM, Las Vegas, Nevada, 1980.

[Paulson 83] Rewriting in Cambridge LCF. Science of Computer Programming, 3: 119-149, Sept., 1983.

[Turner 81] Turner, D. A. Recursion equations as a programming language. In Darlington, et al (editors), Functional Programming and Its Applications, Cambridge University Press, 1981.

[Wadler 85] Wadler, P. L. A splitting headache: Strict vs. lazy semantics for pattern matching in lazy functional languages. To appear.