J. Functional Programming **1** (1): 1–000, January 1993 © 1993 Cambridge University Press

FUNCTIONAL PEARLS
Proof-Directed Debugging

Robert Harper
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The close relationship between writing programs and proving theorems has frequently been cited as an advantage of functional programming languages. We illustrate the interplay between programming and proving in the development of a program for regular expression matching. The presentation is inspired by Lakatos's method of proofs and refutations in which the attempt to prove a plausible conjecture leads to a revision not only of the proof, but of the theorem itself. We give a plausible implementation of a regular expression matcher that contains a flaw that is uncovered in an attempt to prove its correctness. The failure of the proof suggests a revision of the specification, rather than a change to the code. We then show that a program meeting the revised specification is nevertheless sufficient to solve the original problem.

Capsule Review

The capsule review goes here.

1 Introduction

A significant challenge in an introductory programming course is to teach students to reason inductively. While it is not difficult to devise small examples to illustrate the idea, it is quite hard to convince students that these ideas are useful, even essential, in practice. What is required is a collection of compelling examples of the use of inductive reasoning methods to help solve interesting programming problems. In this note we present one such example. The problem is to implement an on-line regular expression matching algorithm in Standard ML: given a regular expression $\mathbf{r}$ and a string s determine whether or not s matches $\mathbf{r}$.[†] It is relatively easy to devise, by "seat of the pants" reasoning, an algorithm to solve the problem. The primary difficulty is with sequential composition of regular expressions, for which we use continuations. With this in mind it is easy to give a very plausible implementation of a regular expression matcher that works in nearly every case.

[†] By "on line" we mean that we do not pre-process the regular expression before matching.

However, the program contains a subtle error that we tease out by attempting to carry out a proof of its correctness. The development is inspired by Lakatos's book *Proofs and Refutations* (1976), which is concerned with the dynamics of mathematical reasoning: formulating conjectures, devising proofs, and discovering refutations. The first step is to give a precise specification of the continuation-passing regular expression matcher. This leads to the conjecture that the matcher satisfies its specification, which we proceed to investigate. Inspection of the code suggests a proof by induction on the structure of the given regular expression, with a case analysis on its outermost form. The proof proceeds along relatively familiar lines, with no serious difficulties, except in the case of iteration, where we discover that the inductive hypothesis is inapplicable. Further analysis suggests an inner induction on the length of the candidate string. Once again the proof appears to go through, but for a small gap at a critical step of the argument. Analysis of the gap in reasoning reveals a counterexample to the conjecture — the proposed implementation does *not* satisfy the specification.

A common impulse is to *change the code* to correct the error, often by an *ad hoc* method that only buries the problem, rather than eliminates it. A less obvious alternative is to *change the specification* to eliminate the counterexample — "monster barring", in Lakatos's colorful terminology. The failed proof of correctness is a valid proof of a weaker specification. But what about those "monsters"? We show that there is no loss of generality in ruling them out because every regular expression is equivalent to one that is not a "monster". By pre-processing to eliminate the "monsters", we arrive at a fully-general matching procedure.

All programs are written in Standard ML (Milner *et al.*, 1997), but there should be no difficulty transcribing the examples into other functional languages.

2 Background

We review here some basic definitions in order to establish notation.

2.1 Languages

Fix an *alphabet*, Σ, a countable set of *letters*. The set Σ^* is the set of *strings* over the alphabet Σ. The *null string* is written ϵ, and string concatenation is indicated by juxtaposition. A *language* L is any subset of Σ^* — that is, any set of strings over Σ. We will identify Σ with the ML type `char` and Σ^* with the ML type `string`.

We will need the following operations on languages (over a fixed alphabet):

Zero	0	$=$	$\emptyset$
Unit	1	$=$	$\{\epsilon\}$
Alternation	$L_1 + L_2$	$=$	$L_1 \cup L_2$
Concatenation	$L_1\,L_2$	$=$	$\{\, s_1\,s_2 \mid s_1 \in L_1,\ s_2 \in L_2 \,\}$
Iteration	$L^{(0)}$	$=$	1
	$L^{(i+1)}$	$=$	$L\,L^{(i)}$
	L^*	$=$	$\bigcup_{i\geq 0} L^{(i)}$

It is instructive to observe that L^* is the smallest language M such that $1+LM \subseteq M$ — that is, the smallest language containing the null string and closed under concatenation with L on the left. It follows that $L^* = 1 + L\,L^*$, an identity that we shall use shortly.

2.2 Regular Expressions

Regular expressions are a notation system for languages. The set of regular expressions over an alphabet Σ is given by the following inductive definition:

1. $\mathbf{0}$ and $\mathbf{1}$ are regular expressions.
2. If $a \in \Sigma$, then $\mathbf{a}$ is a regular expression.
3. If $\mathbf{r}_1$ and $\mathbf{r}_2$ are regular expressions,then so are $\mathbf{r}_1 + \mathbf{r}_2$ and $\mathbf{r}_1\,\mathbf{r}_2$.
4. If $\mathbf{r}$ is a regular expression, then so is $\mathbf{r}^*$.

The *language*, $L(\mathbf{r})$, of a regular expression $\mathbf{r}$ is defined by induction on the structure of $\mathbf{r}$ as follows:

$$\begin{array}{rcl} L(\mathbf{0}) & = & 0 \\ L(\mathbf{1}) & = & 1 \\ L(\mathbf{a}) & = & \{\,a\,\} \\ L(\mathbf{r}_1 + \mathbf{r}_2) & = & L(\mathbf{r}_1) + L(\mathbf{r}_2) \\ L(\mathbf{r}_1\,\mathbf{r}_2) & = & L(\mathbf{r}_1)\,L(\mathbf{r}_2) \\ L(\mathbf{r}^*) & = & L(\mathbf{r})^* \end{array}$$

On the left-hand side we are dealing with *syntax*, whereas on the right we are dealing with *semantics*. Thus 0 on the right-hand side stands for the empty language, 1 stands for $\{\,\epsilon\,\}$, and so on, whereas on the left-hand side $\mathbf{0}$ and $\mathbf{1}$ are just forms of expression.

We say that a string s *matches* a regular expression $\mathbf{r}$ iff $s \in L(\mathbf{r})$. Thus s never matches $\mathbf{0}$; s matches $\mathbf{1}$ only if $s = \epsilon$; s matches $\mathbf{a}$ iff $s = a$; s matches $\mathbf{r}_1 + \mathbf{r}_2$ if it matches either $\mathbf{r}_1$ or $\mathbf{r}_2$; s matches $\mathbf{r}_1\,\mathbf{r}_2$ if $s = s_1\,s_2$, where s_1 matches $\mathbf{r}_1$ and s_2 matches $\mathbf{r}_2$; s matches $\mathbf{r}^*$ iff either $s = \epsilon$, or $s = s_1\,s_2$ where s_1 matches $\mathbf{r}$ and s_2 matches $\mathbf{r}^*$. An equivalent formulation for the last case is that s matches $\mathbf{r}^*$ iff there exists $n \geq 0$ such that $s = s_1\,\ldots\,s_n$ with s_i matching $\mathbf{r}$ for each $1 \leq i \leq n$.

3 A Matching Algorithm

We are to define a function `accept` with type `regexp -> string -> bool` such that `accept r` s evaluates to `true` iff s matches $\mathbf{r}$, and evaluates to `false` otherwise.

The type `regexp` is defined as follows:

```
datatype regexp =
    Zero
  | One
  | Char of char
  | Times of regexp * regexp
  | Plus of regexp * regexp
  | Star of regexp
```

The correspondence to the definition of regular expressions should be clear. It is a simple matter to define for each regular expression $\mathbf{r}$ its *representation* $\ulcorner\mathbf{r}\urcorner$ as a value of type `regexp` in such a way that a given value v of type `regexp` is $\ulcorner\mathbf{r}\urcorner$ for exactly one regular expression $\mathbf{r}$. We shall gloss over the distinction between a regular expression $\mathbf{r}$ and its representation $\ulcorner\mathbf{r}\urcorner$ as a value of type `regexp`.

The matcher is defined using a programming technique called *continuation-passing*. We will define an auxiliary function `acc` of type

```
regexp -> char list -> (char list -> bool) -> bool
```

which takes a regular expression, a character list, and a continuation, and yields either `true` or `false`. Informally, the function `acc` matches some initial segment of the given character list against the given regular expression, and passes the corresponding final segment to the continuation, which determines the final outcome. To ensure that the matcher succeeds (yields `true`) whenever possible, we must be sure to consider *all* ways in which an initial segment of the input character list matches the given regular expression in such a way that the remaining unmatched input causes the continuation to succeed. Only if there is no way to do so may we yield `false`.

This informal specification may be made precise as follows. We call a function f of type τ`->`τ' *total* iff for every value v of type τ, there exists a value v' of type τ' such that $f(v)$ evaluates to v'. For every s of type `char list`, every r of type `regexp`, and every total function k of type `char list -> bool`

1. If there exists s_1 and s_2 such that $s = s_1\, s_2$, $s_1 \in L(\mathbf{r})$, and $k(s_2)$ evaluates to `true`, then `acc` $\mathbf{r}$ s k evaluates to `true`.
2. If for every s_1 and s_2 such that $s = s_1\, s_2$ with $s_1 \in L(\mathbf{r})$ we have that $k(s_2)$ evaluates to `false`, then `acc` $\mathbf{r}$ s k evaluates to `false`

Notice that we restrict attent to continuations k that always yield either `true` or `false` on any input. Notice as well that the specification implies that the result should be `false` in the case that there is no way to partition the input string s such that an initial segment matches $\mathbf{r}$.

Without giving an implementation of `acc`, we can define `accept` as follows:

```
fun accept r s =
    acc r (String.explode s) (fn nil => true | _ => false)
```

We "explode" the string argument into a list of characters to facilitate sequential processing of the string. The initial continuation yields `true` or `false` according to whether the remaining input has been exhausted. Assuming that `acc` satisfies the specification given above, it is easy to see that `accept` is indeed the required matching algorithm.

We now give the code for `acc`:

```
fun acc Zero cs k = false
  | acc One cs k = k cs
  | acc (Char d) nil k = false
  | acc (Char d) (c::cs) k =
    if c=d then k cs else false
  | acc (Plus (r1, r2)) cs k =
    acc r1 cs k orelse acc r2 cs k
  | acc (Times (r1, r2)) cs k =
    acc r1 cs (fn cs' => acc r2 cs' k)
  | acc (r as (Star r1)) cs k =
    k cs orelse acc r1 cs (fn cs' => acc r cs' k)
```

Does `acc` satisfy the specification given above? A natural way to approach the proof is to proceed by induction on the structure of the regular expression. For example, consider the case $\mathbf{r} =$ `Times(`$\mathbf{r}_1$`,`$\mathbf{r}_2$`)`. We have two proof obligations, according to whether or not the input may be partitioned in such a way that an initial segment matches $\mathbf{r}$ and the continuation succeeds on the corresponding final segment.

First, suppose that $s = s_1\, s_2$ with s_1 matching $\mathbf{r}$ and k`(`s_2`)` evaluates to `true`. We are to show that `acc` $\mathbf{r}$ s k evaluates to `true`. Now since s_1 matches $\mathbf{r}$, we have that $s_1 = s_{1,1}\, s_{1,2}$ with $s_{1,1}$ matching $\mathbf{r}_1$ and $s_{1,2}$ matching $\mathbf{r}_2$. Consequently, by the inductive hypothesis applied to $\mathbf{r}_2$, we have that `acc` $\mathbf{r}_2$ $(s_{1,2}\, s_2)$ k evaluates to `true`. Therefore the application `(fn cs' => acc` $\mathbf{r}_2$ `cs'` k`)` $(s_{1,2}\, s_2)$ evaluates to `true`, and hence by the inductive hypothesis applied to $\mathbf{r}_1$, the expression `acc` $\mathbf{r}_1$ s `(fn cs' => acc` $\mathbf{r}_2$ `cs'` k`)` evaluates to `true`, which is enough for the result.

Second, suppose that no matter how we choose s_1 and s_2 such that $s = s_1\, s_2$ with $s_1 \in L(\mathbf{r})$, we have that k`(`s_2`)` evaluates to `false`. We are to show that `acc` $\mathbf{r}$ s k evaluates to `false`. It suffices to show that `acc` $\mathbf{r}_1$ s `(fn cs' => acc` $\mathbf{r}_2$ `cs'` k`)` evaluates to `false`. By the inductive hypothesis (applied to $\mathbf{r}_1$) it suffices to show that for every $s_{1,1}$ and s_2' such that $s = s_{1,1}\, s_2'$ with $s_{1,1} \in L(\mathbf{r}_1)$, we have that `acc` $\mathbf{r}_2$ s_2' k evaluates to `false`. By the inductive hypothesis (applied to $\mathbf{r}_2$) it suffices to show that for every $s_{1,2}$ and s_2 such that $s_2' = s_{1,2}\, s_2$ with $s_{1,2} \in L(\mathbf{r}_2)$, we have that k`(`s_2`)` evaluate to `false`. But this follows from our assumptions, taking $s_1 = s_{1,1}\, s_{1,2}$.

The cases for $\mathbf{0}$, $\mathbf{1}$, $\mathbf{a}$, and $\mathbf{r}_1 + \mathbf{r}_2$ follow a similar pattern of reasoning.

What about iteration? Let $\mathbf{r}$ be `Star` $\mathbf{r}_1$, and suppose that $s = s_1\, s_2$ with s_1 matching $\mathbf{r}$ and k`(`s_2`)` evaluates to `true`. By our choice of $\mathbf{r}$, there are two cases to consider: either $s_1 = \epsilon$, or $s_1 = s_{1,1}s_{1,2}$ with $s_{1,1}$ matching $\mathbf{r}_1$ and $s_{1,2}$ matching $\mathbf{r}$. In the former case the result is the result of k`(`s`)`, which is k`(`s_2`)`, which is `true`, as required. In the latter case it suffices to show that `acc` $\mathbf{r}_1$ s `(fn cs' => acc` $\mathbf{r}$ `cs'` k`)` evaluates to `true`. By inductive hypothesis it suffices to show that `acc` $\mathbf{r}$ $s_{1,2}s_2$ k evaluates to `true`. It is tempting at this stage to appeal to the inductive hypothesis to complete the proof — but we cannot because the regular expression argument is the *original* regular expression $\mathbf{r}$, and not some sub-expression of it!

What to do? Let's try to fix the proof. The offending call to `acc` is on the original regular expression $\mathbf{r}$, but only after some initial segment of the string argument s has been matched by $\mathbf{r}_1$. This suggests that we proceed by an inner induction on the length of the string argument to `acc`, relying on the inner inductive hypothesis in the critical case of a recursive call to `acc` with the original regular expression $\mathbf{r}$. This seems appealing, until we realize that the initial segment $s_{1,1}$ of s matched by $\mathbf{r}_1$ might be the null string, in which case neither the regular expression nor the string argument change on the recursive call! This immediately suggests a counterexample to the conjecture: `acc` $\mathbf{0}^*$ ϵ k loops infinitely, even if k succeeds on input ϵ.

So the conjecture, as stated, is false. What to do? Following Lakatos, we observe that *the proof proves something*, it is only a question of what. Call a regular expression $\mathbf{r}$ *standard* iff whenever $\mathbf{r}_1^*$ occurs in $\mathbf{r}$, the language $L(\mathbf{r}_1)$ does not contain the null string. Observe that for a standard regular expression, if $\mathbf{r} = \mathbf{r}_1^*$ matches a string s, then either $s = \epsilon$ or $s = s_1\, s_2$, where $s_1 \neq \epsilon$ matches $\mathbf{r}_1$ and s_2 again matches $\mathbf{r}$. Thus the proof proves that *the regular expression matcher is correct for regular expressions in standard form.* Rather than change the code, we change the specification!

4 Standardization

But haven't we lost something by making the restriction to standard form? After all, $\mathbf{0}^*$ is a perfectly reasonable regular expression, yet we've ruled it out as a possible input to the matching algorithm (or, at any rate, only guaranteed the behavior of the matcher for regular expressions in standard form). Isn't this just mathematical sleight of hand?

No, because *any regular expression can be brought into standard form.* More precisely, every regular expression is equivalent to one in standard form in the sense that they both accept the same language. Moreover this equivalence is effective in that we may define an algorithm to put every regular expression into standard form. Thus we may define a fully general regular expression matcher by composing the matcher defined in the previous section with a standardization algorithm that puts regular expressions into standard form.

We rely on the equation $\mathbf{r} = \delta(\mathbf{r}) + \mathbf{r}^-$, where $\delta(\mathbf{r})$ is either $\mathbf{1}$ or $\mathbf{0}$ according to whether or not $\mathbf{r}$ accepts the null string, and where $L(\mathbf{r}^-) = L(\mathbf{r}) \setminus \{\,\epsilon\,\}$.(Berry & Sethi, 1987) The function $\delta(\mathbf{r})$ is defined as follows.

$$\begin{array}{rcl}
\delta(\mathbf{0}) & = & \mathbf{0} \\
\delta(\mathbf{1}) & = & \mathbf{1} \\
\delta(\mathbf{a}) & = & \mathbf{0} \\
\delta(\mathbf{r}_1 + \mathbf{r}_2) & = & \delta(\mathbf{r}_1) \oplus \delta(\mathbf{r}_2) \\
\delta(\mathbf{r}_1\, \mathbf{r}_2) & = & \delta(\mathbf{r}_1) \otimes \delta(\mathbf{r}_2) \\
\delta(\mathbf{r}^*) & = & \mathbf{1}
\end{array}$$

Here $\mathbf{r}_1 \oplus \mathbf{r}_2$ is defined to be $\mathbf{1}$ if either $\mathbf{r}_1$ or $\mathbf{r}_2$ is $\mathbf{1}$, and $\mathbf{0}$ otherwise. Similarly, $\mathbf{r}_1 \otimes \mathbf{r}_2$ is defined to be $\mathbf{0}$ if either $\mathbf{r}_1$ or $\mathbf{r}_2$ is $\mathbf{0}$, and is $\mathbf{1}$ otherwise.

The function $\mathbf{r}^-$ is defined as follows:

$$\begin{array}{rcl}
\mathbf{0}^- & = & \mathbf{0} \\
\mathbf{1}^- & = & \mathbf{0} \\
\mathbf{a}^- & = & \mathbf{a} \\
(\mathbf{r}_1 + \mathbf{r}_2)^- & = & \mathbf{r}_1^- + \mathbf{r}_2^- \\
(\mathbf{r}_1\,\mathbf{r}_2)^- & = & \delta(\mathbf{r}_1)\,\mathbf{r}_1^- + \mathbf{r}_1\,\delta(\mathbf{r}_2) + \mathbf{r}_1^-\,\mathbf{r}_2^- \\
(\mathbf{r}^*)^- & = & \mathbf{r}^-(\mathbf{r}^-)^*
\end{array}$$

The last two clauses deserve comment. The non-empty strings matching $\mathbf{r}_1\,\mathbf{r}_2$ are (1) the non-empty strings in $\mathbf{r}_2$, in the case that $\mathbf{r}_1$ contains the empty string, (2) the non-empty strings in $\mathbf{r}_1$, in the case that $\mathbf{r}_2$ contains the empty string, and (3) the concatenation of a non-empty string in $\mathbf{r}_1$ followed by a non-empty string in $\mathbf{r}_2$. The clause for iteration is motivated by the observation that the non-empty strings in the iteration $\mathbf{r}^*$ are simple the *non-zero* iterations of the *non-empty* strings in $\mathbf{r}$.

It is easy to check that $\delta(\mathbf{r})$ and $\mathbf{r}^-$ have the properties stated above, that $\mathbf{r}^-$ is in standard form, and that $L(\mathbf{r}) = L(\delta(\mathbf{r})+\mathbf{r}^-)$. It follows that we may relax the restriction to standard form regular expressions in the specification of the matcher by composing the matcher given in the previous section with a simple standardization algorithm based on the equations given above.

5 Conclusion

The example of regular expression matching illustrates a number of important programming concepts:

1. *Continuation-passing*: the use of higher-order functions to manage the flow of control in a program.
2. *Proof-directed debugging*: the use of a failed proof attempt to discover an error in the code.
3. *Change of specification*: once we isolated the error, we didn't change the code, but rather the specification. Debugging isn't always a matter of changing the code!
4. *Pre-processing*: to satisfy the more stringent specification we pre-processed the regular expression so that it satisfies the additional assumption required for correctness.

6 Acknowledgement

The regular expression matching problem was suggested by Frank Pfenning, to whom I am grateful for his comments and suggestions.

References

Berry, Gerard, & Sethi, Ravi. (1987). From regular expressions to deterministic automata. *Theoretical computer science*, **25**(1).

Lakatos, Imre. (1976). *Proofs and refutations.* Cambridge University Press.

Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, David. (1997). *The definition of Standard ML (revised).* MIT Press.

This research was sponsored in part by National Science Foundation (NSF) grant no. CCR-0122581.