Notes on Regular Expression Simplification

Robert Harper, Spring 1997
edited by Frank Pfenning, Fall 1997

Draft of September 26, 1997

1 Introduction

Symbolic computation systems such as *Mathematica* and *Maple* provide a general means of simplifying expressions using a variety of rules. A typical example is the simplification of a polynomial by reducing it to "standard form" $a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$. Our goal is to explore the implementation of such simplifiers using the method of *tactics*. The general idea is to define a set of *primitive rules* for simplifying expressions, and to combine these using a variety of functions to build complex simplifiers from the basic rules.

2 Algebraic Laws for Regular Expressions

We will build a simplifier for regular expressions based on the following simple algebraic laws:

$$\begin{array}{rcl}
r + \mathbf{0} & = & r \\
\mathbf{0} + r & = & r \\
(r_1 + r_2) + r_3 & = & r_1 + (r_2 + r_3) \\
\\
\mathbf{1}r & = & r \\
r\mathbf{1} & = & r \\
r\mathbf{0} & = & \mathbf{0} \\
\mathbf{0}r & = & \mathbf{0} \\
(r_1r_2)r_3 & = & r_1(r_2r_3)
\end{array}$$

These are called *associative* and *cancellation* laws for alternation and concatenation.

These laws are valid in the sense that if $r_1 = r_2$ according to the above laws, then $L(r_1) = L(r_2)$. In other words, the left- and right-hand sides determine the same language.

3 Simplification

We will use these laws to put regular expressions into standard form, by which we mean that

1. All uses of alternation and concatentation are right-associated.
2. All uses of $\mathbf{0}$ in an alternation expression and all uses of $\mathbf{1}$ in a concatenation expression are eliminated.

Thus $(\mathbf{a}+\mathbf{b})+\mathbf{0}+\mathbf{c}^*$ would be rewritten to the standardized form $\mathbf{a}+(\mathbf{b}+\mathbf{c}^*)$.

How is simplification achieved? The general idea is to *orient* the equations from left to right, regarding them as *rewriting rules* in which the left-hand side is re-written to the right-hand side. Orienting the requations given above, we obtain the following rewriting rules:

$$\begin{array}{rcl}
r+\mathbf{0} & \to & r \\
\mathbf{0}+r & \to & r \\
(r_1+r_2)+r_3 & \to & r_1+(r_2+r_3) \\
\\
\mathbf{1}r & \to & r \\
r\mathbf{1} & \to & r \\
\mathbf{0}r & \to & \mathbf{0} \\
r\mathbf{0} & \to & \mathbf{0} \\
(r_1r_2)r_3 & \to & r_1(r_2r_3)
\end{array}$$

When thought of as a rewriting rule in this manner, the left-hand side is called a *redex* and the right-hand side its *contractum*.

To simplify an expression, we apply these rules according to a specific *rewriting strategy* until no further simplifications are possible. The resulting expression will then be in standard form. The rewriting strategy that we shall use is called the *leftmost-outermost strategy* because at each step we rewrite the expression by applying rules from the *outside in* (i.e., starting with the whole expression and working progressively through sub-terms) and *left-to-right* (in the case of alternation and concatenation). This strategy is *complete* in the sense that we will not miss an opportunity to perform a simplification if there is one.[1]

It is important to realize that performing one simplification can enable another. For example, if we have the expression $r = r' + \mathbf{c}$, and $r' = (\mathbf{b}+\mathbf{c})\mathbf{1}$, then simplification of r' by cancellation exposes the alternation, yielding the expression $(\mathbf{a}+\mathbf{b})+\mathbf{a}$, which can then be further simplified by right-associating.

Our overall strategy will be to repeatedly right-associate expressions, then cancel $\mathbf{0}$'s and $\mathbf{1}$'s, until no further reductions are possible.

4 Implementation

The first step is to represent the primitive reduction steps as ML functions that, when applied to a regular expression, either rewrite the regular expression in accordance with that rule, or else raise the exception `Fail` to indicate failure to apply. Here are some example primitive rules:[2]

```
fun rassoc_plus (Plus (Plus (r, s), t)) = Plus (r, Plus (s, t))
  | rassoc_plus _ = raise Fail
fun rcancel_plus (Plus (r, Zero)) = r
  | rcancel_plus _ = raise Fail
fun lcancel_plus (Plus (Zero, r)) = r
  | lcancel_plus _ = raise Fail
```

These functions (and the others for the concatenation operation) directly express the primitive rewriting steps given above.

[1]We shall not prove this important fact here.

[2]The complete code may be found in the file `code/lecture10.sml` in the course directory.

The second step is to define the leftmost-outermost rewriting strategy. The idea is that, given a rule, we apply that rule to the leftmost, outermost expression on which it does not fail. (If it does not apply anywhere, then fail.) Here is the code:

```
fun LMOM rule r =
      rule r handle Fail => LMOM' rule r
and LMOM' rule (Plus (r, s)) =
      (Plus (LMOM rule r, s) handle Fail => Plus (r, LMOM rule s))
  | LMOM' rule (Times (r, s)) =
      (Times (LMOM rule r, s) handle Fail => Times (r, LMOM rule s))
  | LMOM' rule (Star r) = Star (LMOM rule r)
  | LMOM' rule r = raise Fail
```

The function `LMOM` is defined mutually-recursively with the auxiliary function `LMOM'`. The idea is that `LMOM` attempts to apply the rule to the given expression. If it succeeds, then the result of applying the rule is the result; if it fails, then we proceed into sub-terms, considering multiple sub-terms from left-to-right.

Now we define the simplifier in stages using *rewriting tactics*. Here are the definitions:

```
val rassoc = REPEAT (LMOM (rassoc_times ORELSE rassoc_plus))
val cancel =
    LMOM (rcancel_times ORELSE rcancel_plus ORELSE
          lcancel_times ORELSE lcancel_plus)
val simplify = REPEAT (rassoc THEN cancel)
```

The functions `rassoc`, `cancel`, and `simplify` are called *rewriting tactics*.

The function `rassoc` right-associates alternation and concatenation operators by repeatedly applying the primitive associative rewritings to the leftmost, outermost position in the given regular expression. Upon completion of `rassoc`, all uses of alternation and concatenation are right-associated.

The function `cancel` cancels **0**'s and **1**'s using the cancellation laws at the leftmost, outermost position. At most one cancellation is performed by a call to `cancel`.

The function `simplify` repeatedly right-associates, then cancels, until no further simplifications can be performed.

These rewriting tactics are implemented using the generic *tacticals* `ID`, `THEN`, `ORELSE`, `FAIL`, `TRY`, and `REPEAT`. These are provided in a module with the signature `REWRITE`:

```
signature REWRITE =
sig
  exception Fail
  type 'a rewriter = 'a -> 'a (* may raise Fail *)
  (* infixr THEN ORELSE *)
  val THEN : 'a rewriter * 'a rewriter -> 'a rewriter
  val ID : 'a rewriter                    (* unfailing *)
  val ORELSE : 'a rewriter * 'a rewriter -> 'a rewriter
  val FAIL : 'a rewriter
  val TRY : 'a rewriter -> 'a rewriter    (* unfailing *)
  val REPEAT : 'a rewriter -> 'a rewriter (* unfailing *)
end;
```

Notice that the definition of the type `'a rewriter` to be `'a -> 'a` appears in the signature. Thus a rewriter is just a function from some type to itself. In our case we choose `'a` to be the type `RegExp.regexp`, but these tacticals are not limited to regular expressions.

In practical use of rewriters, it is extremely important to keep in mind which tactics may fail, and which will always succeed (possibly keeping the expression unchanged). For example, f `THEN` g may clearly fail if f fails or g fails on the result of f. On the other hand, `TRY` f will never fail: if f does not apply, it returns its argument unchanged.

How are the rewriting tacticals implemented? Using exceptions. Here's the code:

```
structure Rewriter :> REWRITE =
struct
  exception Fail
  type 'a rewriter = 'a -> 'a (* may raise Fail *)
  infixr THEN ORELSE
  fun (f1 THEN f2) x = f2(f1(x))
  fun ID x = x
  fun (f1 ORELSE f2) x = f1 x handle Fail => f2 x
  fun FAIL x = raise Fail
  fun TRY f x = (f ORELSE ID) x
  fun REPEAT f x = TRY (f THEN (REPEAT f)) x
end
```

We use infix syntax for `THEN` and `ORELSE`. The rewriter f_1 `THEN` f_2 first applies f_1, then applies f_2 to the result; this is just function composition. The rewriter `ID` just returns the argument without modification; it is the "null" or "identity" rewriter. The rewriter f_1 `ORELSE` f_2 rewrites using f_1; if this fails (i.e., raises the exception `Fail`), then it rewrites using f_2 instead. The rewriter `FAIL` always fails immediately. The rewriter `TRY` f tries to rewrite using f; if it succeeds, the result of applying f is returned, otherwise the original expression is returned untouched. Finally `REPEAT` f tries f repeatedly until it fails, yielding the result of all successful rewritings.