FORMAL DERIVATION OF A PATTERN MATCHING ALGORITHM

R.S. BIRD, J. GIBBONS and G. JONES
Programming Research Group, Oxford University, 11 Keble Road, Oxford, OX1 3QD,
United Kingdom

Communicated by M. Sintzoff
Received February 1988
Revised October 1988

1. Introduction

This paper is devoted to the synthesis of a functional version of the Knuth-Morris-Pratt algorithm for pattern matching. This algorithm was first discussed by Knuth [6]; since then formal developments have been given by Dijkstra [4] and Dromey [5], among many others. The novel aspects of the present treatment are: (i) the result is expressed as a (very short) functional program; and (ii) the derivation makes use of the calculus of lists described by Bird [1]. In order to make this paper as self-contained as possible, we shall indicate the concepts and notations of [1] used in our derivation as we go along. However, we do assume some familiarity with the basic ideas of functional programming. (For an introduction to functional programming, see Bird and Wadler [3].)

The calculus of lists is based on the definition of lists as an algebraic structure (actually, a monoid) under the primitive operation of concatenation, denoted by the sign ++. Thus

\[ [a_1, a_2, \ldots, a_n]++[b_1, b_2, \ldots, b_m] = [a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m]. \]

Concatenation is an associative operation and has the empty list, denoted by [ ], as its identity element.

The above information is sufficient for us to be able to specify the pattern matching problem. Let \( w \) (called the pattern) and \( t \) (called the text) be lists of characters. Required is an efficient algorithm for computing a boolean function \( \text{match} \) such that \( \text{match} \ w \ t \) holds just in the case that there exists lists \( u \) and \( v \) so that \( t = u ++ w ++ v \). This condition is just the assertion that \( w \) is a segment of \( t \). Let the function \( \text{segs} \) return a list of all the segments of a given list. An equivalent statement of the problem is now given by:

\[ \text{match} \ w \ t = w \in \text{segs} \ t \]
The formal definition of \( \text{segs} \) given in [1] is based on two subsidiary functions, \( \text{inits} \) and \( \text{tails} \), which return the list of initial and final segments of a list, respectively. Thus,

\[
\text{inits}[a_1, a_2, \ldots, a_n] = [[], [a_1], \ldots, [a_1, a_2, \ldots, a_n]],
\]

\[
\text{tails}[a_1, a_2, \ldots, a_n] = [[a_1, a_2, \ldots, a_n], [a_2, \ldots, a_n], \ldots, [a_n], []].
\]

Informally speaking, the segments of a list can be computed by taking the tail segments of all the initial segments and concatenating the results. (An alternative method is to take the initial segments of all the tail segments.) To formalize this procedure we need to introduce two important operators in the theory of lists, the map operator \( \ast \) and the reduction operator \( / \).

The operator \( \ast \) takes a function on its left and a list on its right; we have

\[
f \ast [a_1, a_2, \ldots, a_n] = [fa_1, fa_2, \ldots, fa_n].
\]

Thus, \( (f \ast ) \) takes a list and applies \( f \) to every element. A simple yet important fact about \( \ast \) is the **map distributivity** law:

\[
(f \cdot g) \ast = (f \ast ) \cdot (g \ast).
\]

Here, the sign \( (\cdot) \) denotes functional composition.

The reduction operator \( / \) takes an operator on its left and a list on its right; we have

\[
\oplus /[a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n.
\]

The operator \( \oplus \) used in a reduction must be associative (there are no brackets on the right of the above equation). If, in addition, \( \oplus \) has an identity element \( e \), then we define

\[
\oplus /[1] = e.
\]

Using the above functions we can now define \( \text{segs} \):

\[
\text{segs} = \oplus / \cdot \text{tails} \ast \cdot \text{inits}.
\]

We can also formally define the list membership operation \( (a \in) \):

\[
(a \in) = \lor / \cdot (a =) \ast.
\]

Here, \( \lor \) is the operation of logical disjunction. In words, \( a \in x \) holds if the list of boolean values, obtained by comparing \( a \) with every element of \( x \), contains an occurrence of the value \( \text{True} \).

Functions of the form \( h = \oplus / \cdot f \ast \) define homomorphisms on lists, and are studied extensively in [1]. For present purposes we require only the following fact: if \( h = \oplus / \cdot f \ast \), then

\[
h \cdot \oplus / = \oplus / \cdot h \ast.
\]
In particular we can now calculate that:

\[
\begin{align*}
\text{match } w \\
= \text{definition of } \text{match} \\
(w \in ) \cdot \text{segs} \\
= \text{definition of } \text{segs} \\
(w \in ) \cdot +/ \cdot \text{tails } \ast \cdot \text{inits} \\
= \text{definition of } \in \text{ and fact above} \\
\lor / \cdot (w \in ) \ast \cdot \text{tails } \ast \cdot \text{inits} \\
= \text{map distributivity} \\
\lor / \cdot \text{endswith } w \ast \cdot \text{inits}
\end{align*}
\]

where we introduce

\[
\text{endswith } w \times = (w \in \text{tails } x).
\]

This calculation gives a new expression for \text{match}. For example:

\[
\begin{align*}
\text{match } "\text{aba}" "\text{ababa}" \\
= \lor / \text{endswith } "\text{aba}" \ast \text{inits } "\text{ababa}" \\
= \lor / \text{endswith } "\text{aba}" ["", "a", "ab", "aba", "abab", "ababa"] \\
= \lor / [\text{False, False, False, True, False, True}] \\
= \text{True}.
\end{align*}
\]

The derived expression for \text{match} is executable and requires \(O(m^2n)\) steps, where \(m\) and \(n\) are the lengths of \(w\) and \(t\), respectively. This time can be reduced to \(O(mn)\) steps by optimizing \text{endswith } w\ to run in \(O(m)\) steps. The KMP algorithm, on the other hand, requires only \(O(m + n)\) steps.

2. **Strategy**

The strategy for calculating an efficient algorithm for \text{match} is based on the following algebraic identity (in fact, [1, Lemma 5]):

\[
(\oplus \rightarrow e) \ast \text{inits } x = (\oplus \neq e) x \tag{1}
\]

for all \(\oplus\), \(e\) and \(x\). The left-hand side of (1) denotes the application of a left-reduction \((\oplus \rightarrow e)\) (called foldl in [3]) to every initial segment of \(x\). The informal definition of \((\oplus \rightarrow e)\) is:

\[
(\oplus \rightarrow e)[x_1, x_2, \ldots, x_n] = (((e \oplus x_1) \oplus x_2) \oplus \cdots) \oplus x_n.
\]
Thus, left-reductions are like general reductions, except that an explicit value \( e \) is provided as the value of the reduction on the empty list, and the order of association is from left to right. Given a fixed order of association, it is no longer necessary to assume that \( \oplus \) is associative. In fact the type of \( \oplus \) can take the general form

\[
\oplus : (\beta \times \alpha) \rightarrow \beta
\]

so \( \oplus \) need not even satisfy the type constraints on an associative operator (namely, that it should have type \( \alpha \times \alpha \rightarrow \alpha \) for some \( \alpha \)).

Formally, \( (\oplus \rightarrow e) \) is defined recursively by the equations

\[
(\oplus \rightarrow e)[ ] = e,
\]

\[
(\oplus \rightarrow e)(\langle x \rangle + [a]) = (\oplus \rightarrow e)x \oplus a.
\]

Note that direct evaluation of the left-hand side of (1) requires \( O(n^2) \) calculations involving \( \oplus \), where \( n \) is the length of \( x \).

The right-hand side of (1) denotes a left-accumulation (called \( \text{scan} \) in [3]):

\[
[e, e \oplus x_1, (e \oplus x_1) \oplus x_2, \ldots, ((e \oplus x_1) \oplus x_2) \oplus \cdots \oplus x_n]
\]

where \( x = [x_1, x_2, \ldots, x_n] \). Formally, we have:

\[
(\oplus \# e)[ ] = [e],
\]

\[
(\oplus \# e)([a] ++ x) = [e] ++ (\oplus \# (e \oplus a))x.
\]

Thus, \( (\oplus \# e) \) \( x \) can be computed in \( O(n) \) calculations involving \( \oplus \). Equation (1) can therefore be viewed, from left to right, as an efficiency improving transformation.

The use of equation (1) in our problem is indicated by the form of the right-hand side of the derived expression for \( \text{match} \). It suggests the following strategy for our problem: if we can express \( \text{endswith} \ w \) as a left-reduction, involving an operator \( \oplus \) whose values can be computed quickly, then \( \text{match} \) can be computed as a left-accumulation. More precisely,

\[
\text{match} \ w
\]

\[
= \text{previous calculation}
\]

\[
\lor \cdot \text{endswith} \ w \ast \cdot \text{inits}
\]

\[
= \text{consequence of successful strategy}
\]

\[
\lor \cdot (\oplus \rightarrow e) \ast \cdot \text{inits}
\]

\[
= \text{equation (1)}
\]

\[
\lor \cdot (\oplus \# e).
\]

Unfortunately, the suggested strategy will not work. The function \( \text{endswith} \ w \) returns a single boolean value, and this is insufficient information for it to be expressible as a left-reduction. The next best thing is to express \( \text{endswith} \ w \) as the composition of a left-reduction with another function. More precisely, we shall show

\[
\text{endswith} \ w = h \cdot (\oplus \rightarrow e)
\]
for a suitable function \( h \) and operator \( \oplus \). This leads to the conclusion that

\[
\text{match } w = \nu / \cdot h \cdot (\oplus \neq e).
\]

In fact, every function over lists can be expressed as the composition of some function with a left-reduction (just take the left-reduction that computes the identity function on lists). The trick, however, is to choose \( h \) and \( \oplus \) in (2) so that the resulting form for \( \text{match} \) can be computed in linear time.

The minimal generalization which enables us to make progress is to replace \( \text{endswith } w \ x \) by a value that gives the maximum amount of overlap between \( w \) and \( x \). The function \( \text{overlap } w \) returns the longest initial segment of \( w \) that ends \( x \). If this longest segment is \( w \) itself, then \( x \) ends with \( w \); otherwise not.

The function \( \text{overlap} \) is defined formally by the equations:

\[
\text{overlap } w \ x = \uparrow_w / \text{starts } w \leftarrow \text{tails } x,
\]

\[
\text{starts } w \ z = (z \in \text{inits } w).
\]

This says that \( \text{overlap } w \ x \) is the longest final segment of \( x \) that is also an initial segment of \( w \). The expression \( p \leftarrow x \) denotes the sublist of those elements of \( x \) that satisfy the predicate \( p \) and \( \uparrow_w \) is an operator that returns the longer of two lists.

The relationship between \( \text{endswith} \) and \( \text{overlap} \) is that

\[
\text{endswith } w \ x = (w = \text{overlap } w \ x).
\]

Assuming that \( \text{overlap } w \) can be expressed as a left-reduction \((\oplus \Rightarrow e)\), we therefore have

\[
\text{match } w = \nu / \cdot (w =) \cdot (\oplus \neq e).
\]

This expression for \( \text{match} \) is not quite good enough for the final algorithm, since it contains the presumably expensive test \((w =)\). We shall show how to avoid it after first expressing \( \text{overlap } w \) as a left-reduction.

### 3. Derivation

The construction of \( \text{overlap } w \) as a left-reduction is based on the definition

\[
\text{overlap } w \ x = \uparrow_w / \text{starts } w \leftarrow \text{tails } x,
\]

\[
\text{starts } w \ z = (z \in \text{inits } w)
\]

of \( \text{overlap} \) given above. The aim is to try and find \( e \) and \( \oplus \) such that \( \text{overlap } w = (\oplus \Rightarrow e) \), that is, we require

\[
\text{overlap } w \ [ ] = e,
\]

\[
\text{overlap } w \ (x++[a]) = \text{overlap } w \ x \oplus a.
\]

We will need the following two properties of \( \text{starts } w \):

\[
\text{starts } w \ [ ] = \text{True}, \quad (3)
\]

\[
\text{starts } w \ (x++[a]) = \text{starts } w \ x \land x \neq w \land a = \text{hd} (w \rightarrow x). \quad (4)
\]
In words, the first equation says that \([\ ]\) is an initial segment of \(w\), and the second equation says that \(x++[a]\) is an initial segment of \(w\) just in the case that \(x\) is a (proper) initial segment of \(w\) and \(a\) is the first element of the sequence that remains when \(x\) is removed from \(w\). The operator \(\rightarrow\) is defined by the rule

\[(x++y)\rightarrow x = y\]

and \(hd\) (short for \(head\)) is defined for nonempty lists by

\[hd([a]++x) = a\]

(The companion function \(tl\) (short for \(tail\)), defined by

\[tl([a]++x) = x\]

is needed below.)

Equation (4) can also be described using terminology from [1]: a predicate \(p\) is said to be \textit{prefix-closed} if

\[p(x++y) \Rightarrow p x\]

for all \(x\) and \(y\). In words, \(p\) is prefix-closed if it holds for every initial segment of \(x\) whenever it holds for \(x\). It follows that a prefix-closed predicate must hold for the empty list if it holds at all. For simplicity, we limit the definition of prefix-closure to exclude the everywhere false predicate. Thus, we suppose

\[p[\ ] = True\] (5)

Any function \(\delta\) which satisfies the equation

\[p(x++[a]) = p x \land \delta a x\] (6)

for all \(a\) and \(x\) is called a \textit{derivative} function of \(p\). Now, equation (4) says that \(\text{starts}\ w\) is a prefix-closed predicate with derivative function

\[\delta w a x = (x \neq w) \land (a = hd(w \rightarrow x)).\] (7)

For the rest of this section we consider properties of an arbitrary prefix-closed predicate. In the final section we return again to the particular case of the predicate \(\text{starts}\ w\), and thence to the solution of the pattern matching problem.

The following lemma was given in [1].

\textbf{Lemma 3.1.} \textit{Suppose \(p\) is a prefix-closed predicate, and let}

\[f x = \uparrow_{\neq} / p \leftarrow \text{tails} x\]

\textit{Then} \(f = (\oplus \rightarrow [\ ]\), where \(\oplus\) is given by

\[x \oplus a = \uparrow_{\neq} / p \leftarrow \text{tails}(x++[a])\]
Equivalently, $\oplus$ can be defined recursively by

$$x \oplus a = \begin{cases} 
  x++[a], & \text{if } p(x++[a]), \\
  [], & \text{if } \neg p(x++[a]) \land x = [], \\
  f(tlx) \oplus a, & \text{if } \neg p(x++[a]) \land x \neq [].
\end{cases}$$

**Proof.** We give an informal proof. The crucial observation is that since $p$ is prefix closed, the value of $f(x++[a])$ must be some final segment of $fx++[a]$. It cannot be longer than $fx++[a]$ because, if it were, then removing the last element gives a sequence that satisfies $p$ and is longer than $fx$. But this contradicts the definition of $fx$. From this fact we obtain

$$f(x++[a]) = \uparrow_x / \text{tails}(fx++[a]).$$

Now, by the definition of $\oplus$, the right side of this equation is just $fx \oplus a$. Since we also have $f[ ]$, we therefore get that

$$f = (\oplus \rightarrow [ ])$$

as required. The recursive definition of $\oplus$ is easy to calculate: it expresses the straightforward procedure of finding the longest tail segment of $(x++[a])$ satisfying $p$ by inspecting the final segments of $(x++[a])$ in decreasing order of length. \qed

An immediate consequence of the lemma is that

$$x \oplus a = f(x++[a]) = fx \oplus a$$

for all $x$. We can use this fact in the last clause of the recursive definition of $\oplus$ to obtain

$$x \oplus a = \begin{cases} 
  x++[a], & \text{if } p(x++[a]), \\
  [], & \text{if } \neg p(x++[a]) \land x = [], \\
  f(tlx) \oplus a, & \text{if } \neg p(x++[a]) \land x \neq [].
\end{cases}$$

The advantage of the new equations for $\oplus$ is that values of $x \oplus a$ are required only for lists $x$ satisfying $p$. Since the left-hand argument of $\oplus$ decreases in length at each recursive call, the new equations serve as an alternative recursive definition of $\oplus$.

Now, suppose that $p$ has derivative function $\delta$, so that

$$p(x++[a]) = px \land \delta a x.$$  

Since we know that $px$ holds for each computed value of $x \oplus a$, we can rewrite the definition of $\oplus$ once again, this time in the form

$$x \oplus a = \begin{cases} 
  x++[a], & \text{if } \delta a x, \\
  [], & \text{if } \neg \delta a x \land x = [], \\
  f(tlx) \oplus a, & \text{if } \neg \delta a x \land x \neq [].
\end{cases}$$

The advantage of this step is that possibly expensive computations of $p$ are eliminated. More precisely, they are traded for additional computations of $f$. This will
only lead to a saving if expressions of the form \( f(tlx) \) can be computed efficiently and, since their values may be required more than once, never recomputed. To illustrate this situation, suppose that \( p \) holds just for a finite set \( S \) of lists. Then, for the purpose of computing \( \oplus \), \( f \) is required only for a finite number of arguments (those of the form \( tlx \) for \( x \) in \( S \)) and these can be precomputed and stored in a table. This process is known as tabulation (see [2] for a general discussion of the idea). If we can assume the existence of a constant time look-up function, then the efficient computation of \( \oplus \) can be expressed in the form:

\[
x \oplus a = \begin{cases} 
  x++[a], & \text{if } \delta a x, \\
  [ ], & \text{if } \neg \delta a x \land x = [ ], \\
  \text{lookup } f(tlx) \oplus a, & \text{if } \neg \delta a x \land x \neq [ ]. 
\end{cases}
\]

Although each individual computation of \( x \oplus a \) may require more than a fixed number of steps, the evaluation of \( (\oplus \oplus [ ]) \) on a list of length \( n \) will take no more than \( O(n) \) steps, assuming \( \delta a \) requires constant time.

The second step of our development is to obtain this efficiency by choosing a suitable representation of the table and showing how it can be constructed. We shall not assume the table is finite; this means that we do not assume that the table is built first and only afterwards used. Rather, the pieces of the table are built as and when they are needed. Of course, only a finite portion of the table is required in a finite computation. This “lazy” approach (familiar to functional programmers) is crucial to the success of the algorithm.

We shall implement the table as an element of the following type:

\[
table \alpha ::= \text{Null} \mid \text{Node}(\alpha], \text{table } \alpha, \alpha \rightarrow \text{table } \alpha).\]

The syntax for type declarations used here is that of [3] and is common in functional programming. The declaration says that \( \text{Null} \) is an element of \( table \alpha \), and so is \( \text{Node}(x, t, \phi) \) for all sequences of \( \alpha \)-values \( x \), elements \( t \) of type \( table \alpha \), and functions \( \phi : \alpha \rightarrow table \alpha \).

Informally, the table structure can be thought of as a linked list of nodes. The contents of the node \( \text{Node}(x, t, \phi) \) is a string \( x \), and \( t \) is a pointer to the node whose contents are \( f(tlx) \) (if \( x \neq [ ] \)), and the special node \( \text{Null} \) otherwise. Thus, the look-up function mentioned above is implemented by extracting the second component of nodes. The third component \( \phi \) is a function that returns, for argument \( a \), a pointer to the node with contents \( x++[a] \). We should emphasize that the words “linked list”, “contents” and “pointer” have no formal meaning in our notation, but are used for expository purposes only.

Informally, the relationship between tables and sequences of \( \alpha \)-values is described by a representation function \( \Pi : [\alpha] \rightarrow table \alpha \), which satisfies the equations

\[
\Pi([ ]) = \text{Node}(\[ ], \text{Null, } \lambda a.\Pi([a])),
\]

\[
\Pi(x) = \text{Node}(x, \Pi(f(tlx)), \lambda a.\Pi(x++[a])), \quad \text{if } x \neq [ ].
\]
the function $\Pi$ is injective with $\Pi^{-1}$ defined by

$$\Pi^{-1}(\text{Node}(x, t, u)) = x.$$ 

Under the representation given by $\Pi$, the operation $\oplus$ is translated into an operation $\otimes$ satisfying

$$\Pi(x) \otimes a = \Pi(x \oplus a). \quad (8)$$

It follows (by an easy induction) from (8) that

$$\Pi \cdot (\oplus \to [ ]) = (\oplus \to \Pi([ ]))$$

and so, if $f = (\oplus \to [ ])$, that

$$f = \Pi^{-1} \cdot (\oplus \to \Pi([ ])).$$

It is worth restating this idea of a change in representation somewhat more generally since it arises frequently in the calculation of efficient programs. Suppose we want to compute $(\oplus \to e)$, where $e$ is an element of some type $\beta$, and $\oplus : \beta \times a \to \beta$. Suppose also that direct evaluation of this expression is unacceptably inefficient. Let $\Pi : \beta \to \text{struct } \beta$ be an injective function specifying some correspondence between elements of $\beta$ and elements of some, probably more elaborate, structure based on $\beta$. The essential idea is that if $\otimes$ satisfies equation (8), then

$$(\oplus \to e) = \Pi^{-1} \cdot (\otimes \to \Pi(e))$$

and evaluation of the right-hand side may prove more efficient.

Two tasks remain: first, to convert the nonconstructive definition of $\otimes$ given by equation (8) into a constructive one; and second, to massage the definition of $\Pi$ so that the starting value $\Pi([ ])$ for the computation of $f$ can be computed efficiently.

Using the definition of $\Pi$, the definition of $\oplus$ given in Lemma 3.1 and equation (8), we can calculate the following constructive definition of $\otimes$:

$$\text{Node}(x, t, \phi) \otimes a = \begin{cases} \phi a, & \text{if } \delta a x, \\ t_0, & \text{if } \neg \delta a x \wedge x = [ ], \\ t \otimes a, & \text{if } \neg \delta a x \wedge x \neq [ ], \end{cases}$$

where $t_0 = \Pi([ ])$. If we set

$$\text{Null} \otimes a = t_0,$$

then the second two clauses of the definition of $\otimes$ can be compressed into one:

$$\text{Node}(x, t, \phi) \otimes a = \begin{cases} \phi a, & \text{if } \delta a x, \\ t \otimes a, & \text{otherwise.} \end{cases}$$

It remains to show how to compute $t_0$ more efficiently than through direct use of the representation function $\Pi$. Given a suitable functional programming language, the equations specifying $\Pi$ as they stand can be used to compute $t_0 = \Pi([ ])$, even though the computation will never terminate. The important point is that, when information about some node in the table is required, the further construction of
the table will always make sufficient progress to enable the information to be supplied. The disadvantage of using the equations for \( \Pi \) is only that evaluation may involve many recomputations of the second components of nodes.

Instead, we shall use a modified representation function \( \Psi \) that satisfies:

\[
\Psi([ ], \text{Null}) = \Pi([ ]),
\]

\[
\Psi(x, \Pi(f(tx))) = \Pi(x), \quad \text{if } x \neq [ ].
\]

In brief, the second argument to \( \Psi \) is just the value required for the second component of the node under construction.

From the first equation for \( \Psi \) we get \( t_0 = \Psi([ ], \text{Null}) \) and from the second we get

\[
\Psi(x, \Pi(f(tx))) = \text{definition of } \Psi
\]

\[
\Pi(x) = \text{definition of } \Pi
\]

\[
\text{Node}(x, \Pi(f(tx)), \lambda a.\Pi(x+[a])) = \text{definition of } \Psi
\]

\[
\text{Node}(x, \Pi(f(tx)), \lambda a.\Psi(x+[a], \Pi(f(tx+[a])))).
\]

To simplify the last expression, observe that if \( x \neq [ ] \), then

\[
\Pi(f(tx+[a])) = x \text{ not empty}
\]

\[
\Pi(f(tx+[a])) = \text{since } f = (\oplus \rightarrow [ ])
\]

\[
\Pi(f(tx) \oplus a) = \text{equation (8)}
\]

\[
\Pi(f(tx)) \otimes a.
\]

Hence, setting \( t = \Pi(f(tx)) \), we get

\[
\Psi(x, t) = \text{Node}(x, t, \lambda a.\Psi(x+[a], t \otimes a)).
\]

This completes the major portion of the development. Let us summarize what we have done as a second lemma.

**Lemma 3.2.** Under the assumptions of Lemma 3.1, we have

\[
f = \Pi^{-1} \cdot (\otimes \Rightarrow t_0),
\]
where
\[
\Pi^{-1}(\text{Node}(x, t, \phi)) = x,
\]
\[
\text{Null} \otimes a = t_0,
\]
\[
\text{Node}(x, t, \phi) \otimes a = \begin{cases} 
\phi a, & \text{if } \delta(a, x), \\
t \otimes a, & \text{otherwise,}
\end{cases}
\]
\[
t_0 = \Psi([], \text{Null})
\]
\[
\Psi(x, t) = \text{Node}(x, t, \lambda a. \Psi(x+a[], t \otimes a)).
\]

4. The KMP algorithm

Now let us return to the particular case \( p = \text{starts } w \). Here, the derivative function is
\[
\delta(w, a, x) = (x \neq w) \land (a = \text{hd}(w \rightarrow x))
\]
and this definition can be substituted for \( \delta \) in the conclusion of Lemma 3.2. There are two further optimizations we can make. First, in order to avoid expensive computations of \( \text{hd}(w \rightarrow x) \), we can store \( z = w \rightarrow x \) rather than \( x \) as the first component of nodes; this leads to the changes
\[
\Pi^{-1}(\text{Node}(z, t, \phi)) = w \leftrightarrow z,
\]
\[
\text{Null} \otimes a = t_0
\]
\[
\text{Node}(z, t, \phi) \otimes a = \begin{cases} 
\phi a, & \text{if } z \neq [] \land a = \text{hd} z, \\
t \otimes a, & \text{otherwise,}
\end{cases}
\]
\[
t_0 = \Psi(w, \text{Null})
\]
\[
\Psi(z, t) = \text{Node}(z, t, \lambda a. \Psi(z+a[], t \otimes a)).
\]
In the first line, the operator \( \leftrightarrow \) is defined by the equation
\[
(x \leftrightarrow y) \leftrightarrow y = x.
\]
We now have
\[
\text{overlap } w = \Pi^{-1} \cdot (\otimes \rightarrow t_0).
\]
Recalling the definition of \text{endswith}, we have
\[
\text{endswith } w x = (w = \text{overlap } w x)
\]
and so we obtain
\[
\text{endswith } w = \text{test} \cdot (\otimes \rightarrow t_0),
\]
where
\[
\text{test}(\text{Node}(z, t, \phi)) = (w = \Pi^{-1}(\text{Node}(z, t, \phi)))
= (w = w \leftarrow z)
= (z = []).\]

For the final optimization, observe that in the particular case \( p = \text{starts} \) \( w \) the third component of nodes, namely the function \( \phi \), is applied only to one value, namely the head of the first component. We can therefore replace \( \phi \) by the result of this application.

With this second change the algorithm for computing \( \text{match} \) can be expressed finally as:
\[
\text{match} w = v / \cdot \text{test} \cdot (\otimes \# t_0),
\]
where
\[
\text{test}(\text{Node}(z, t, u)) = (z = []),
\]
\[
\text{Null} \otimes a = t_0,
\]
\[
\text{Node}(z, t, u) \otimes a = \begin{cases} u, & \text{if } z \neq [] \land a = \text{hd } z, \\ t \otimes a, & \text{otherwise,} \end{cases}
\]
\[
t_0 = \Psi(w, \text{Null}),
\]
\[
\Psi(z, t) = \text{Node}(z, t, \Psi(tl z, t \otimes (\text{hd } z))).
\]

These six equations give the KMP algorithm expressed as a functional program.

Acknowledgement

The authors would like to thank Michael Goldsmith, Andrew Kay, and Chris Wright (all from the Programming Research Group, Oxford), for many enjoyable discussions about the derivation of the KMP algorithm. An earlier version was discussed at the Eindhoven Tuesday Afternoon Club, and the authors are extremely grateful to Netty van Gasteren for forwarding the Club’s comments and suggestions for improving the presentation of the paper.

References