Regular Expression Patterns

Niklas Broberg d00nibr@dtek.chalmers.se
Andreas Farre d00farre@dtek.chalmers.se
Josef Svenningsson josefs@cs.chalmers.se
Chalmers University of Technology

Abstract

We extend Haskell with regular expression patterns. Regular expression patterns provide means for matching and extracting data which goes well beyond ordinary pattern matching as found in Haskell. It has proven useful for string manipulation and for processing structured data such as XML. Regular expression patterns can be used with arbitrary lists, and work seamlessly together with ordinary pattern matching in Haskell.

Our extension is lightweight, it is little more than syntactic sugar. We present a semantics and a type system, and show how to implement it as a preprocessor to Haskell.

Categories and Subject Descriptors

D.3.2 [Language Classifications]: Applicative (functional) languages; D.3.3 [Language Constructs and Features]: Patterns

General Terms

Languages

Keywords

Regular expressions, pattern matching, Haskell

1 Introduction

Pattern matching as found in many functional languages is a nice feature. It allows for clear and succinct definitions of functions by cases and works very naturally together with algebraic data types. But sometimes ordinary pattern matching is not enough. A distinct feature of this form of pattern matching is that it only examines the outermost constructors of a data type. While this allows for efficient implementations it is also a rather limited construct for analysing and retrieving data.

A well-known example of a construct that provides deeper and more complex retrievals are regular expressions for strings. While this is not a very common feature among programming languages it is one of the key constructs that have made Perl so popular. Regular expressions are ideal for various forms of string manipulation, text extraction etc, however, they remain a very domain specific and ad-hoc construct targeted only for one particular data structure, namely strings.

On another axis we find the recent trend in XML centric languages. The first attempts at such languages used the ordinary pattern matching facility of functional languages to analyze XML fragments [MS99]. This was found to be too restrictive, so in order to be able to express more sophisticated patterns and transformations on XML fragments the notion of regular expression patterns were invented. Examples of languages including this feature are XDuce [HP03] and CDuce [BCF03]. While this is a great boost for the XML programmer, in the case of XDuce it only works for XML data and not for any other data. Furthermore those pattern matching constructs are closely tied to rather sophisticated type systems which makes them somewhat heavyweight.

In this paper we extend Haskell with regular expression patterns. Our extension has the following advantages:

- Our proposal is lightweight. It is hardly more than syntactic sugar. Most notably it does not require any complex additions to the type system.
- It works for arbitrary lists. It is a general construct and not tied to a specific data type for elements. But it should be noted that it works in particular for strings since strings are just lists of characters in Haskell.
- It fits seamlessly with the ordinary pattern matching facility found in Haskell.

In this paper we give a detailed semantics and type system of regular expression patterns. The extension has been implemented as a preprocessor to Haskell, and we sketch the implementation.

While we have chosen to focus on Haskell in this paper there are very little Haskell specific details. We are quite confident that our proposal could be adapted to any similar functional language.

In recent years a number of papers have been devoted to developing efficient pattern matching and efficient regular matching [Fri04, HM03, Lev03]. This is not the concern of this paper. Although efficiency is an important consideration we focus only on language design.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP’04, September 19–21, 2004, Snowbird, Utah, USA.
Copyright 2004 ACM 1-58113-035-5/04/0009 $.50
Another issue that we do not address is the question of overlapping and exhaustive patterns. We are confident that the existing techniques developed for XML-centric languages will do the job nicely [HVP00]. Note also that in general, it is undecidable to check whether patterns are overlapping or non-exhaustive in Haskell because of guards, so in our setting it is something of a non-issue.

2 Regular expression patterns by example

2.1 Ordinary pattern matching

Assume that we have the following data type representing an entry in an address book.

\[
\text{data Contact = Person Name \mid ContactMode}
\]

\[
\text{data ContactMode = Tel TelNr}
\]

We can assume that the types Name and TelNr are type synonyms for String. The reason for not including TelNr in Contact is because we will later want to add other means of contact, e.g. email addresses, to our address book.

Now consider two different functions that extract information from a contact; firstTel will return the first TelNr in the list of contact modes associated with a contact. lastTel will analogously return the last associated TelNr. The first is easy to write using simple pattern matching on a contact:

\[
\text{firstTel :: Contact \rightarrow Tel Nr}
\]

\[
\text{firstTel (Person \_ \_Tel nr : \_)} = \_nrs
\]

\[
\text{firstTel (Person \_ \_| \_)} = \text{error "No Tel"}
\]

The second function, although its functionality is very similar to firstTel, cannot be written in the same simple way. We must instead resort to recursion and an auxiliary function to step through the list until we reach the end.

\[
\text{lastTel :: Contact \rightarrow Tel Nr}
\]

\[
\text{lastTel (Person \_ nrs) = aux nrs where aux \_| = \text{error "No Tel"}
\]

\[
\text{aux \_ Tel nr} = \_nrs
\]

\[
\text{aux \_ nrs} = \text{aux nrs}
\]

Although the two functions have very similar functionality, only one of them can be written using direct pattern matching. Why is this so? The answer lies, of course, in the list data type. A (non-empty) list has a head and a tail, so extracting the first element is easy. To get to the last element, however, we must recursively look at the tail for its last element. In other words, we must first match on the structure of the list, before being able to look at the elements.

Haskell has a construct for matching directly on elements of a list, but only for fixed-size lists. If we know that a contact never has more than three phone numbers, we could write lastTel as we will ignore the erroneous case from now on.

\[
\text{lastTel (Person \_ \_Tel nr)} = \_nrs
\]

\[
\text{lastTel (Person \_ \_Tel nr)} = \_nrs
\]

\[
\text{lastTel (Person \_ \_Tel nr)} = \_nrs
\]

Clearly this is not a very good solution. Even for this very small task we must write far more than we are comfortable with, and for larger lists or more complex data types this approach quickly becomes infeasible. What we need is a way of saying "match a list containing a Tel, preceded by any number of other elements". This is where regular expression patterns enter the picture.

2.2 Regular expression patterns

Mathematically a regular expression defines a regular language, where language in this context means a (possibly infinite) set of words, and each word is a sequence of elements taken from some alphabet. We can use a regular expression as a validator and try to match an arbitrary word against it to find out if the word belongs to that regular language or not. The basic regular expression operators are repetition, concatenation and choice. Concatenation is straightforward, \(ab\) means \(a\) followed by \(b\). Choice \((a \lor b)\) means either \(a\) or \(b\). Repetition \(a^*\) means zero or more occurrences of \(a\). Repetition can be defined using choice and recursion as \(a^* = \varepsilon \lor a^*a\) where \(\varepsilon\) denotes the empty sequence. As an example, consider the regular expression \(e = a^*b\). The language defined by \(e\), denoted \(L(e)\), is the set of all words consisting of only \(a\)’s or only \(b\)’s, including the empty word. We have that \(aa \in L(e), bbb \in L(e), \text{ but } ab \notin L(e)\). In other words, \(aa\) and \(bbb\) both match the regular expression \(e\), but \(ab\) doesn’t.

This notion of treating a regular expression as a validator is very similar to the concept of pattern matching in Haskell. We take a Haskell value (a word) and a pattern (a regular expression) and try to match them, getting a yes or no as the result. Combining these two concepts is straightforward, yielding what we call regular expression patterns. As noted, a regular expression can be matched against a sequence of elements from some alphabet. Lifting this idea into Haskell, a regular expression pattern can be matched against a list of elements of some datatype. When we speak of a sequence, we mean a sequence of elements in the abstract sense. In contrast, when we speak of a list, we mean the list data type that is used to encode sequences in Haskell.

Returning to our lastTel function, we can now easily write it with a single pattern match by using a repetition regular expression pattern:

\[
\text{lastTel (Person \_ \_*, Tel nr)} = \_nrs
\]

We write concatenation using commas as with ordinary Haskell lists, and we denote repetition with \(^*\). As we can see from the example, regular expression patterns are actually more flexible than bare regular expressions. A regular expression is built from elements of some alphabet, the same alphabet that the words it may match are built from. A regular expression pattern on the other hand is built from patterns over elements of some datatype, allowing us to use constructs like wildcards and pattern variables. We use the term regular expression pattern both for the sub-patterns (repetition, choice etc.) and for a top-level list pattern that contains the former. It should be clear from the context which we are referring to.

2.3 Repetition and Ambiguities

Let us see what else we can do with regular expression patterns. First, as promised, we extend our data type with email addresses.

\[
\text{data Contact = Person Name \mid ContactMode}
\]

\[
\text{data ContactMode = Tel TelNr | Email EAddr}
\]

If we only have ordinary pattern matching we cannot even write firstTel without resorting to recursion and auxiliary functions.

\[
\text{firstTel (Person \_ cmodes) = aux cmodes where aux (Tel nr : \_) = nr}
\]

\[
\text{aux (Tel nr : cmodes)} = \text{aux cmodes}
\]

Using a regular expression pattern, we can write it in one go:
firstTel (Person _ (Email _), Tel nr, _*)! = nr

The straight-forward intuition of the pattern above is that the first
Tel in the list is preceded by zero or more Emails (but no Tels), and
any number of other elements may follow it. We can easily write
lastTel in a similar way as

lastTel (Person _ !*, Tel nr, (Email _*)!) = nr

But seeing these two definitions leads to an interesting question:
What happens if we write the function

someTel :: Contact -> TelNr
someTel (Person _ _*, Tel nr, _*)! = nr

i.e. where the Tel in question may both be preceded and suc-
ceded by other Tels? Clearly this pattern is ambiguous, since if we
match it to e.g. Person "Niklas" [Tel 12345, Tel 23456, 
Tel 34567] we can derive a match for either of the three TelNr
or be bound to nr, by letting the first _* match either 0, 1 or 2 Tels.
To disambiguate such issues, we adopt the policy that a repetition
pattern will always match as few elements as possible while still letting
the whole pattern match the given list. In standard terminology, our
repetition regular patterns are non-greedy. This policy means that
someTel above will be exactly the same as our firstTel function,
since the first _* will now try to match as few elements as possible.

In some cases though, such as lastTel, we want the greedy behav-
ior. To this end we let the programmer specify if a repetition pattern
should be greedy by adding an exclamation mark (!) to it, e.g. in
the following definition of lastTel:

lastTel (Person _!*!, Tel nr, _*)! = nr

2.4 Choice patterns

Now that we’ve seen the power of repetition patterns, we turn
our attention to choice patterns. Assume that we want a function
allTels that returns a list of all telephone numbers associated with
a contact. Without regular expression patterns we must once more
resort to recursion and auxiliary functions.

allTels :: Contact -> [TelNr]
allTels (Person _ cmodes) = aux cmodes
where aux [] || = []
   aux (Tel nr : cmodes)
     = nr : aux cmodes
   aux _ : cmodes = aux cmodes
Using a combination of repetition and choice, we can write it as

allTels (Person _ | (Tel nr | _*)!) = nr

The intuition here is that each element in the list of contact modes
is either a Tel or something else (_). Every time that we encounter
a Tel, we should include the associated TelNr in the result. As
the example shows we can achieve this accumulation of TelNr
with a single pattern variable. Since the intuition of a repetition
pattern is that its subpattern, i.e. the pattern it encloses, should be
matched zero or more times, the same must be true for any pattern
variables inside such a pattern. For each repetition, such a variable
will match a new value. Clearly the only sensible thing to do is to
let that variable bind to a list of all those matched values.

This treatment of variables breaks one aspect of Haskell’s linearity
property – that the occurrence of a variable in a pattern will bind
that variable to exactly one value of the type that it matches. We
will therefore call such a variable non-linear. A non-linear variable
will be bound to a list of values that it matches, in the order that they
were matched (i.e. the order in which they appeared in the matched
list). When we speak of a non-linear binding, we mean a binding
of a non-linear variable to a list of values. We will also use the
terms non-linear context to mean a context in which linear variables
cannot appear, and non-linear patterns, by which we mean patterns
whose subpatterns will always be matched in a non-linear context.

By the example above we see that a repetition pattern is a non-
linear pattern, and consequently that the variable nr appears in
a non-linear context. Similarly a choice pattern is also non-linear.
If we remove the repetition from the regular expression pattern in
allTels we get the pattern |Tel nr| for matching a list of ex-
actly one element. If that element is a Tel we will have a value to
bind to nr, but if it is an Email we have none! Thus we still cannot
guarantee that a variable gets one value; in this case nr will be
bound to a list with zero or one element.

The function allTels shows how regular expression patterns can be
used for filtering a list based on pattern matching. We can go
one step further and do partitioning, e.g.

allTelsAndEmails :: Contact -> ([TelNr], [Email])
allTelsAndEmails (Person _ ![Email addr]*) = (nr, email)

A choice pattern can also be ambiguous if any of its subpatterns
overlap, as in

sillyAllTels :: Contact -> ([TelNr], [TelNr])
sillyAllTels (Person _ | (Tel nr | Tel nr | _*)!) = (nr, nr)

To disambiguate this we adopt a first-match policy, much like that
of Haskell pattern matching. Thus we first check if the first sub-
pattern matches, and consider the kth subpattern only if no pattern
i < k matches. Note that we allow choice patterns to contain more
than two subpatterns. Choice patterns are right associative so for
example | (Tel nr | Tel nr | _*)! is parenthesised like |
(Tel nr | (Tel nr | (_*)! |). Another interesting thing about
choice patterns is that we allow a variable to appear in both subpat-
terns assuming that it binds to values of the same type. For instance,
if our datatype for modes of contact was defined as
data ContactMode = HomeTelNr | WorkTelNr
we could define allTels as

allTels (Person _ | (Home nr | Work nr)!*) = nr

Variables in choice patterns are still non-linear even if they appear
in all subpatterns, so the function

singleTel (Person _ | (Home nr | Work nr)) = nr

will have the type Contact -> [TelNr].

2.5 Subsequences and option patterns

Regular expressions allow grouping of elements and subexpress-
sions using parentheses. For example, the regular expression e =
(ba)* will match the words ba, baba etc. To add this feature to our
regular expression patterns we need to introduce some new syntax,
since using ordinary parentheses in Haskell will denote tuples, as in
The variable all will be bound to the (non-empty) list of ContactModes associated with a contact. This is a very useful feature to have for regular expression patterns as well, for instance we may want to write a function that picks the first two elements from a list as

twoFirst :: [a] -> [a]
twoFirst [a|/\ b]*! = a

However, adding this feature raises some interesting questions. Firstly, what will the type of a variable bound to a regular expression pattern be? For a subsequence it seems fairly obvious that it will have a list type, but what about repetitions, choices and optional patterns? To this issue there is no obvious right answer, one way might be to let a variable be bound to all elements matched by the subpattern in analogy with implicitly bound variables. We have chosen a slightly different approach in which we assign different types to patterns to mirror the intuition behind them.

Subsequences and repetition patterns will both have list types since they represent sequences. There’s a difference between them though: a subsequence is just what the name implies, a subsection of the original sequence. Thus a variable bound to it will always have the same type as the input list, i.e. a list of elements. A repetition pattern on the other hand is a repetition of some subpattern, and so it will have the type of a list of that subpattern. For choice patterns we make use of Haskell’s built-in Either type defined as

data Either a b = Left a | Right b

By using this type we can allow the left and right subpatterns of a choice pattern to have different types, for instance

singleCM :: [ContactMode] -> Either ContactMode ContactMode
singleCM [a@Tel _ | Email _ | a] = a

maybeSingleTel :: [ContactMode] -> Either ContactMode [ContactMode]
maybeSingleTel [a@Tel _ | _] = a

Similarly for optional patterns we use another built-in Haskell type:

data Maybe a = Nothing | Just a

so if we write a function

singleOrNoTel :: [Email _]*, a@Tel _?, (Email _)*! = a

it will have the type [ContactMode] -> Maybe ContactMode.

One way to think about this is to see the regular expression pattern operators as special data constructors. In an analogy with ordinary Haskell, we don’t expect a to have the same type in the two uses a@Tel _ and a@Tel _? as the type uses a@Tel _ and a@Tel _?. Nor do we expect the a in a@Tel _? to have the same type as the a in a@Tel _?

The second issue concerns linear vs. non-linear binding. We have already seen that implicit bindings, i.e. bindings that arise from the use of ordinary pattern variables, are context dependent; in linear context they get the ordinary types, whereas in non-linear context they get list types. This context dependence unfortunately makes it easy for the programmer to make mistakes, since it isn’t clear just by looking at a variable in the pattern what type it will have. We cannot do anything about implicit bindings, but we can avoid the same problem for explicit binding. Therefore we let the ordinary @ operator signify linear explicit binding, the only kind available
in ordinary Haskell. For non-linear explicit binding we introduce a new operator \( \forall \) (read "as cons" or "accumulating as"). The former may not appear in non-linear context, whereas the latter may appear anywhere inside a regular expression pattern. Their differences are shown by the following examples:

\[
\begin{align*}
|a@(Tel _) \, _\forall | & \Rightarrow a :: ContactMode \\
|a@(Tel \_\_ \_\_ | \Rightarrow a :: ContactMode \\
|\{Tel a\_\_\_\_ \, _\forall | & \Rightarrow a :: TelNo \\
|\{Tel a\_\_\_\_ \, _\forall | & \Rightarrow Not allowed! \\
|\{Tel a\_\_\_\_ \, _\forall | & \Rightarrow a :: TelNr
\end{align*}
\]

We can define the semantics of implicit bindings in terms of explicit bindings. In linear context we have that a pattern variable \( a \) is equivalent to the pattern \( a\_\forall \). This can be seen in the example \( \{Tel a\_\_\_\_ \, _\forall | which is clearly equivalent to \( \{Tel a\_\_\_\_ \, _\forall |. In non-linear context, \( a \) is equivalent to \( a\_\forall \) as in the examples \( \{Tel a\_\_ \, _\forall | and \( \{Tel a\_\_\_\_ \, _\forall |. 

### 2.8 Further examples

Now that we’ve seen all the basic building blocks that our regular expression patterns consist of, let us put them to some real use.

Traditionally regular expressions have been used in programming languages for text matching purposes, and certainly our regular expression patterns are well suited for this task. As an example, assume we have a specification of a simple options file. An option has a name and a value, written on a single row, where name and value are separated by a colon and a whitespace. Different options are written on different lines. Here are the contents of a sample options file:

```
author: Niklas Broberg
author: Andreas Farre
author: Josef Svenningsson
title: Regular Expression Patterns
submitted: ICFP 2004
```

A simple parser for such option files can be written using a regular expression pattern as

```
parseOptionFile :: String -> [(String, String)]
parseOptionFile
| (// names@\_\_\_, ':', ' ', vals@\_\_\_, '\n' '/\*]' | 
  \- zip names vals
```

where `zip` is a function that takes two lists and groups the elements pair-wise.

XML processing is another area that greatly benefits from regular expressions, since "proper pattern matching on XML fragments requires... matching of regular expressions" [MvV01]. Indeed several recent XML-centric languages (Xduce, CDuce) include regular expressions as part of their pattern matching facilities.

As an example we encode XML in Haskell using a simple datatype

```haskell
data XML = Tag String | XML | PCDATA String
```

An XML fragment is either a Tag, e.g. `<P> ... </P>`, which has a name (a String) and a list of XML children, or it is PCDATA (XML lingo for a string inside tags). This model is of course extremely simplified, we’ve left out anything that will not directly add anything to our example, most notably XML attributes. Now assume that we have a simple XML email format, where a sample email message in this format might look like:

```
<MSG>
  <FROM>d00nibro@dtect.chalmers.se</FROM>
  <RCPTS>
    <T0>d00farre@dtect.chalmers.se</T0>
    <T0>josefs@cs.chalmers.se</T0>
  </RCPTS>
  <SUBJECT>Regular Expression Patterns</SUBJECT>
  <BODY>
    <P>Regular expression patterns are useful</P>
  </BODY>
</MSG>
```

which would be encoded in our XML datatype as

```
Tag "MSG" |
Tag "FROM" [PCDATA "d00nibro@dtect.chalmers.se"],
Tag "RCPTS" |
  Tag "T0" [PCDATA "d00farre@dtect.chalmers.se"],
  Tag "T0" [PCDATA "josefs@cs.chalmers.se"] |
  Tag "SUBJECT" [PCDATA "Regular Expression Patterns"],
  Tag "BODY" |
  Tag "P" [PCDATA "Regular expression patterns are useful"] |
```

We can write a function to convert messages from this XML format into the standard RFC822 format using regular expression patterns:

```
xmToRfc822 :: XML -> String
xMToRfc822
| Tag "MSG" |
  Tag "FROM" [PCDATA from],
  Tag "RCPTS" |
    (Tag "T0" [PCDATA tos]),
  Tag "SUBJECT" [PCDATA subject],
  Tag "BODY" |
    (Tag "P" [PCDATA paras])
| |
| = concat
  ["From: ", from, crlf,
   "To: ", concat (intersperse " ", tos),
   crlf,
   "Subject: ", subject, crlf, crlf,
   concat (intersperse crlfs paras),
   crlf]
  where crlf = "\r\n"
```

### 3 Syntax

The previous section has gone over all of regular expression patterns by example. This section starts the formal treatment by giving a grammar for the syntax, which can be seen in figure 1. We refer to the nonterminal for Haskell’s ordinary patterns as `pattern` and extend it with a new production for regular expression patterns.

The concrete syntax is quite close to that of e.g. Perl [Perl] or CDuce [BCF03] with the notable exception that we have non-greedy patterns as default. An extra exclamation mark indicates greediness.
Ordinary Haskell patterns are regular expressions patterns. The operators are repetition (*), non-empty repetition (+) and option (?). Furthermore there are choice patterns indicated by a vertical bar and subsequences are enclosed in subsequence brackets. Regular expression patterns can be enclosed in parenthesis. The last two productions are for linear and non-linear variable bindings. Precedence of the operators is as follows: *+, ?, *, +, and ? binds strongest. They are followed by choice patterns which are also right associative. Lastly we have @ and #: which bind weakest. All constructs in regular expression patterns bind stronger than constructor application.

4 Semantics

In this section we turn to the formal semantics for regular expression patterns. Our semantics divides mutually into two parts: one for linear and one for non-linear patterns. The reason for this division is that variable bindings are treated differently.

4.1 Structure of semantics

We give the semantics as an all-match semantics. This leads to possibly ambiguous matches, the same list can be matched in many different ways. Since this may affect how variables are bound to their values we need to disambiguate our rules. We follow the approach taken by Hosoya and Pierce [HP03] and introduce an ordering on the rules indicating which rule will have precedence when several rules can match. The order is given by numbers in the name of the rules, where lower numbers have higher precedence. Intuitively this means that when building the derivation tree for a match, one must always try to use the rule with the highest precedence first, and choose the other rule only if choosing the first rule cannot lead to a match.

Before we begin with the semantics we will define some concepts which will be used in our explanation of the semantics. We will use sets of variable bindings to map variables to values. A variable binding is denoted \( x \rightarrow v \). In repetition patterns we will need to merge sets of variable bindings with overlapping domains. We use \( \bowtie \) to this end and define it as follows:

\[
\{ x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n \} \bowtie \{ x_1 \rightarrow \psi_1, \ldots, x_n \rightarrow \psi_n \} = \{ x_1 \rightarrow v_1 \bowtie \psi_1, \ldots, x_n \rightarrow v_n \bowtie \psi_n \}
\]

When giving a semantics for subsequence patterns we will use a type indexed function flaten to merge lists of values. It is defined as follows:

\[
\text{flatten}_{\tau}([\ ] = [\ ]
\text{flatten}_{\tau}([\ ] = [\ ]
\text{flatten}_{\tau}(v, v_s) = \text{flatten}_{\tau}(v) + \text{flatten}_{\tau}(v_s)
\text{flatten}_{\tau}(\text{Nothing}) = [\ ]
\text{flatten}_{\tau}(\text{Just} v) = \text{flatten}_{\tau}(v)
\text{flatten}_{\tau}(\text{Left} v) = \text{flatten}_{\tau}(v)
\text{flatten}_{\tau}(\text{Right} v) = \text{flatten}_{\tau}(v)
\]

We will refer to the set of bound variables in a pattern \( p \) as

\[
\text{Var}(\text{reg pat})
\]

The semantics for linear regular expression patterns can be found in figure 2. Due to space reasons we only give a few of the rules as we explain below.

The judgement for matching linear patterns is denoted \( \varepsilon \vdash p \rightarrow v; \beta; \ell \). It should read as "\( p \) is matched by a pattern \( p \) yielding a value \( v \), a set of variable bindings \( \beta \), and a remainder list \( \ell \). \( \varepsilon \) and \( \ell \) range over Haskell lists, where \( \varepsilon \) is the list we wish to match and \( \ell \) is a (possibly empty) suffix of \( \varepsilon \) that wasn't matched.

First of all we have a rule HM-REGPAT that extends Haskell’s pattern matching semantics, denoted \( \varepsilon \vdash p \), with regular expression patterns. It does so by performing a linear match.

\[
\begin{align*}
\varepsilon \vdash ((\mid p_1 \ldots p_n) \rightarrow l; \beta; \ell) \\
\varepsilon \vdash [p_1 \ldots p_n] \rightarrow \beta \\
\end{align*}
\]

Here we require that the remainder list is empty i.e. that the whole input list is successfully matched. This requirement together with the ordering on the rules determines which derivation must be chosen.

The base rule, \( \text{LM-BASE} \), is that where the pattern to match is a normal Haskell pattern. In this case we piggy-back on Haskell’s normal mechanism for binding variables from patterns.

\[
\begin{align*}
\varepsilon \vdash p \rightarrow \beta \\
\varepsilon \vdash [p_1 \ldots p_n] \rightarrow \beta \\
\end{align*}
\]

A part from ordinary Haskell patterns there are two ways that we can bind variables to values at top level, given by the rules LM-AS and LM-ACCAs. The \( \theta \) operator simply binds the variable to a value, whereas the \( \theta \) operator binds the variable to a list containing the value. The behavior of \( \theta \) clearly makes more sense in a non-linear context, where the number of bound values may vary, but since it is harmless to do so we have chosen to allow it to appear in linear contexts as well.

For subexpressions we simply match each pattern in the sequence in order, as stated by the rule LM-SEQ. The values produced after matching are concatenated and the resulting disjoint sets of variable bindings are merged. The value yielded by matching a subexpression should always be a list of elements, so before we can concatenate the values of the sub-matches we need to flatten these values to simple lists. Here we need to use the typing relation on patterns defined in section 5. The typing relation is defined relative to some base type \( T \) that during the actual matching will be instantiated to the type of the elements in the matching list.

Matching a non-linear pattern in a linear context is identical to matching it in a non-linear context. This is exemplified by the rule
LM-STAR. The rules for the rest of the operators are similar and are left out due to space restrictions.

4.3 Semantics for non-linear patterns

The relation for matching in a non-linear context, denoted \( l \in p \rightarrow v_1; \beta_1; l' \) (the only difference in syntax is that we drop the subscript on \( e \)), is similar to the relation for linear contexts. It differs in two crucial aspects, namely variable bindings and that we handle non-linear patterns. The rules can be found in figure 3.

The base rule M-BASE is once again that where the pattern to match is an ordinary Haskell pattern. Since the matching now takes place in a non-linear context, the values of variables being bound while matching this pattern are put into lists instead of just being bound outright. Binding variables explicitly in a non-linear context can only be done using the \( \theta : \) (accumulating as) operator that binds its variable argument to a list of the value matching its pattern argument, as shown in the rule M-ACCUMAS.

The rule for matching a subsequence, M-SEQ, is identical to LM-SEQ except that subpatterns in the sequence are also matched in a non-linear context.

The rules for a repetition pattern, M-STAR1 and M-STAR2, give a non-greedy semantics to the operator by giving the rule for not matching higher precedence than the rule for actually matching the subpattern. The first rule simply doesn’t try to match anything, whereas the second rule matches the given subpattern \( p' \) once and then recurses to obtain more matches. The value obtained from matching \( p \) is then prepended to the result values of the recursive second premise. Similarly the values of bound variables are prepended to the bindings from the recursive call. To get a greedy semantics in the rules M-GSTAR1 and M-GSTAR2 we simply swap the order of the rules to give precedence to preforming a match.

The non-empty repetition pattern operator \( p^* \) is defined as \( p^* \equiv pp' \), similarly its greedy counterpart \( p^! \equiv pp' \), and the rules M-PLUS and M-GPLUS can easily be derived from these facts.

The rules M-OPT1 and M-OPT2 for optional patterns are very similar to the rules for repeating patterns, only that no recursion to obtain more matches is done. The values returned by an optional pattern are of the Haskell Maybe type for optional values.

For choice regular expression patterns we return values of the Haskell Either type to indicate which choice was taken. In the rules M-CHOICE1 and M-CHOICET2 we give precedence for matching the left pattern. Furthermore all variables occurring only in the branch not taken are assigned empty lists.

5 Well-formed regular expression patterns

We now turn our attention to the static semantics of regular expression patterns. We will refer to the static semantics as well-formedness of regular expression patterns.

There are two reasons why we need a static semantics. The first reason concerns where and how a variable is bound in a pattern. In ordinary patterns a variable may appear only once, with the notable exception for \( \alpha \)-patterns found in Ocaml and SML/NJ. In these languages all alternatives must bind exactly the same set of variables. We have similar yet more liberal restrictions on variable bindings. Bound variables must not necessarily be bound in all alternatives in a choice pattern.

The second reason is that we need to ensure that the types of the bound variables are correct. The same variable should in particular have the same type for all its occurrences in a choice pattern.

To express the well-formedness of a regular expression pattern we use the judgment \( \Delta \vdash \gamma \) which says that a (linear) regular expression pattern \( p \) is well-formed in the typing context \( \Delta \). The typing context \( \Delta \) gives types to the variables bound in the pattern. When checking the validity of patterns in a non-linear context we use the judgment \( \Delta \vdash \gamma \) which is similar to the judgment for linear patterns. We will also refer to the well-formedness of patterns in Haskell, using the judgment \( \Delta \vdash \gamma : p \). We refer to Faxén’s paper for a static semantics of Haskell patterns [Fax02]. We require that \( \Delta \vdash \gamma : p \) can only be derived if \( p \) binds exactly the variables in the typing context \( \Delta \). Finally we will need a notion of types for regular expression patterns. We use the judgment \( p : \tau \) to say that the pattern \( p \) has the type \( \tau \).

Checking the well-formedness of a regular expression pattern as an ordinary pattern in the host language is done using the following rule. It is noteworthy that we split the typing context. All the typing contexts \( \Delta \) must bind different names. We use this to enforce that a variable may only be bound once.

\[
\frac{\Delta ; p_1 \ldots \Delta_n \vdash p_n}{\Delta ; \ldots \Delta_j \vdash_{h \mid \vdash \Delta_j \neq \Delta_i} \forall i \neq j}
\]

The rules for establishing well-formedness of linear patterns can be found in figure 4. In this section we only present the rules for non-greedy operators as the rules for greedy counterparts are exactly the same. The only interesting thing to note about the rules for \( + \) and \( ? \) is the fact that when checking their subpatterns we are in a non-linear context and therefore use the corresponding judgment for the premises. The rule for sequences is reminiscent of that for regular expression patterns in the context of ordinary patterns explained above.
\[ e \in_{h} \pi \rightarrow \beta \]
\[ e, T \in \pi \rightarrow e; \sigma ; t \sigma = \{ x \mapsto [v] \mid x \in \beta \} \]

**M-Base**
\[ l_1 \in p \rightarrow v_1; \beta_1; l_2 \]
\[ l_1 \in (p_1 \cdots p_n) \rightarrow v_1 \beta_1 \cdots \beta_n l_1 \]

**M-Seq**
\[ l_1 \in p_1 \rightarrow v_1; \beta_1; l_2 \cdots l_n \in p_n \rightarrow v_n; \beta_n; l_f \]
\[ l_1 \in (p_1 \cdots p_n) \rightarrow v_1 \beta_1 \cdots \beta_n l_1 \]

**M-Seq**
\[ l \in p' \rightarrow [\beta; l] \rightarrow \{ x \mapsto [v] \mid x \in \text{vars}(p) \} \]

**M-Plus**
\[ l_1 \in p \rightarrow v_1; \beta_1; l_2 \]
\[ l_3 \in p^! \rightarrow v_1; \beta_2; l_3 \]
\[ l_1 \in p^! \rightarrow v_1; \beta_2; l_3 \]

**M-Plus**
\[ l_1 \in p^! \rightarrow \text{Nothing}; \beta; l \rightarrow \{ x \mapsto [v] \mid x \in \text{vars}(p) \} \]

**M-Opt**
\[ l_1 \in p \rightarrow v_1; \beta_1; l_2 \]
\[ l_1 \in p^! \rightarrow \text{Nothing}; \beta_1; l_2 \rightarrow \{ x \mapsto [v] \mid x \in \text{vars}(p) \} \]

**M-Choice**
\[ l_1 \in (p_1 \cap p_2) \rightarrow \text{Nothing}; \beta_1; l_2 \rightarrow \{ x \mapsto [v] \mid x \in \text{vars}(p_1) \} \]

**M-Choice**
\[ l_1 \in (p_1 \cap p_2) \rightarrow \text{Nothing}; \beta_1; l_2 \rightarrow \{ x \mapsto [v] \mid x \in \text{vars}(p_2) \} \]

**Figure 3. Semantics for non-linear regular expression patterns**

\[ \Delta \vdash p \quad \Delta \vdash q \quad \Delta \vdash \Delta_{1} \cup \Delta_{2} \quad \Delta \vdash p \quad \Delta \vdash q \quad \Delta = \Delta_{1} \cup \Delta_{2} \]
\[ \Delta \vdash p \quad \Delta \vdash q \quad \Delta = \Delta_{1} \cup \Delta_{2} \]

\[ \Delta \vdash p \quad \Delta \vdash q \quad \Delta = \Delta_{1} \cup \Delta_{2} \]

\[ \Delta \vdash p \quad \Delta \vdash q \quad \Delta = \Delta_{1} \cup \Delta_{2} \]

\[ \Delta \vdash p \quad \Delta \vdash q \quad \Delta = \Delta_{1} \cup \Delta_{2} \]

\[ \Delta \vdash p \quad \Delta \vdash q \quad \Delta = \Delta_{1} \cup \Delta_{2} \]

\[ \Delta \vdash p \quad \Delta \vdash q \quad \Delta = \Delta_{1} \cup \Delta_{2} \]

**Figure 4. Wellformed linear regular expression patterns**

**Figure 5. Wellformed regular expression patterns**

**Figure 6. Typing rules for regular expression patterns**

**The variable binding rules are interesting to contrast against each others. As-patterns are well-formed if the variable is bound to a pattern with the same type as the variable. Accumulating as-patterns on the other hand may match several times so the type of the variable must be a list.**

In figure 5 we present the rules for establishing the well-formedness of non-linear patterns. Most of the rules carry over straightforwardly from those for linear patterns. It should be noted though that the rule for ordinary patterns rebuilds the typing context so that all variables have list types.

Figure 6 gives the typing rules for regular expression patterns. The intuition behind these rules is that a pattern has a type which reflects the ways it can match. For example a pattern which can match many times has a list type, hence variables bound to * and + patterns get list types. Choice patterns can match one of two things which is captured by the Either type of Haskell. A sequence pattern matches yields a sequence and hence it also has a list type. Variable binding patterns don’t affect the typing. The last typing rule for ordinary patterns in the underlying language is more surprising, since it refers to a specific type ??. This means that the typing rules should be interpreted in a context where we are matching a list of type [], i.e. ?? is the type of the elements of the list.
6 Implementation

We currently have an implementation of our regular expression pattern system that works as a preprocessor for GHC. It takes a source code file possibly containing regular expression patterns and translates it into semantically equivalent vanilla Haskell code. It also comes with a matching engine, which we implement as a simple parser monad. The preprocessor does not check any types, instead we rely on GHC’s type checker to catch type errors.

6.1 Matching engine

The datatype for a matching parser, which we from now on will refer to as a matcher, looks like

```haskell
data Matcher e a = Matcher (|e| -> |(a,|e|)|)
```

It is essentially a function that takes an input list, conducts a match, and returns a list of results. Each result will consist of a value, a set of values for bound variables, and a remainder list. All of this is read directly from our semantic rules.

Since different variables will be bound to values of different types, we need to model the set of bindings as a tuple, with each entry corresponding to the value(s) for one specific variable. As customary, we let the remainder list be the state of the matcher monad, so that it is implicitly threaded through a series of matches. The individual matcher functions then need to return a value for future bindings, and a tuple with values for variables.

To account for our all-match semantics the parser generates a list of results at each step. At places where we need to branch we can use the `++` operator which lets us proceed with two different matchers. We define `++` as

```haskell
(++) :: Matcher e a -> Matcher e a -> Matcher e a
(Matcher f) ++ (Matcher g) = Matcher (
let es -> let aes1 = f es
    aes2 = g es
    in aes1 ++ aes2)
```

As we can see from the definition `++` is left-biased, i.e. any results from its left operand will end up before any results from its right operand in the list of results. This allows us to define a function that conducts the full matching by, as defined by our first-match policy, selecting the first result in this list of results for which the matcher has reached the end of the input list (i.e. the remainder list is empty). This function, called `runMatch`, corresponds to the rule `HM-REGPAT` from figure 2, and is defined as

```haskell
runMatch :: Match e a -> |e| -> Maybe a
runMatch (Matcher f) es =
  let allps = f es
      allMatches = filter (null . snd) allps
    in case allMatches of
        [] -> Nothing
        [(|l|, vars):_] -> Just vars
```

6.2 Translation

The basic idea behind translating a regular expression pattern into vanilla Haskell is to generate a matcher for each subpattern, all the way down to ordinary Haskell patterns, and then combine these to form a top-level matcher corresponding to the whole of the pattern. 6.2.1 Base patterns

The base case is when the pattern in question is an ordinary Haskell pattern. First we must generate a function that actually takes an element from the input list and tries to match it to the given pattern. For example, if the pattern in question is `Tel nr`, the corresponding function would look like

```haskell
match0 :: CMode -> Maybe TelNr
match0 e = case e of
  Tel nr -> Just (nr)
  _ -> Nothing
```

No type signatures are actually generated, we just supply them here to simplify understanding. To avoid overly long signatures we abbreviate `ContactMode` with `CMode` in our examples.

What the function returns if the match succeeds is a tuple containing the values of bound variables. The function above works in linear context since we return the bound variable as is. If we instead wanted a function to work in non-linear context, we would wrap the values in lists, like

```haskell
match0 :: CMode -> Maybe [TelNr]
match0 e = case e of
  Tel nr -> Just (nr: [])
  _ -> Nothing
```

We also need to lift a generated matching function into the matcher monad. This lifting works identically regardless of what the pattern is, so we have a function in the matcher engine that does this, defined as

```haskell
baseMatch :: (e -> Maybe a) -> Matcher e (e, a)
baseMatch matcher = do
  e <- getInput
  case matcher e of
    Nothing -> mfail
    Just b -> do discard
      return (e, b)
```

The functions used by `baseMatch` are inherent to our matcher monad. `getInput` retrieves the head of the input list, `discard` drops the head of the input list, and `mfail` is a matcher that always returns an empty list of results. We now need to generate a matcher by applying `baseMatch` to our generated function, i.e.

```haskell
match1 :: Matcher CMode (CMode, TelNr)
match1 = baseMatch match0
```

The type states that `match1` is a matcher for a list of `CModes`. The value matched is a `CMode`, and the only variable bound is of type `TelNr`. The numbers 0 and 1 in the names of these functions signify that each name is fresh, i.e. these numbers could be any positive integers, but no two functions share the same integer.

For Haskell patterns that are guaranteed to always match, i.e. pattern variables and wildcards (\_), we can simplify these steps. For a wildcard, what we need to generate is the matcher

```haskell
match0 :: Matcher e (_)
match0 = baseMatch (_ -> Just ())
```

meaning we will always match, and no variables are bound. The only difference for a pattern variable is that the variable in question is also bound, e.g. for the pattern `e` we get
match0 :: Matcher e (e, e)
match0 = baseMatch \(\lambda a \rightarrow\) Just \(\langle a \rangle\)

Once again the shown function works in linear context, in non-
linear context we would wrap the returned \(\lambda\) in a list.

6.2.2 Repetition

All regular expression patterns have one or more subpatterns, and
the first step when translating a regular expression pattern will be to
translate these subpatterns. For a repetition pattern, \(ps\), we would
first translate the subpattern \(p\) into some matcher function \(\text{match}X\).
According to the rules M-STAR1 and M-STAR2, a matcher for a
repetition pattern should if possible continue without trying to
match anything, otherwise it should match one element and then re-
cursively match the repetition pattern again. This behavior is com-
mon to all repetition patterns so we define it as a function in the
matching engine:

\[
\text{manyMatch} :: \text{Matcher } e \ a \rightarrow \text{Matcher } e \ \langle[a]\rangle \\
\text{manyMatch} \ 	ext{matcher} = (\text{return } []) +++++
(\text{do } a \leftarrow \text{matcher}
\quad \text{as } \leftarrow \text{manyMatch} \ 	ext{matcher}
\quad \text{return } (a:a))
\]

The problem with this definition is that \text{manyMatch} returns a list in
which each element is the result of one step of the recursion. We
need to unpack this list so that we instead return a tuple, in which
each entry is a list of results for a specific variable binding. We
cannot do this generically since the number of bound variables, and
thus the size of the tuple, will vary. Therefore we must supply an
appropriate unzipping function that works for the correct number of
variables. The exact function to use can be determined by the
preprocessor, that has the necessary meta-information on what vari-
ables are bound. Note that all variables inside the repetition will be
non-linear, so the result of matching a variable in each step of the
recursion will be a list of values. If we only unzip to get a list of
such results for each variable, what we would really get is a list of
lists of values. Thus to get a list of values we should also let the
unzipping function concatenate the results for each variable in the
resulting tuple.

Inside \text{manyMatch} the unpacking will be done in two steps. The first
is to simply unzip the list into two lists, one containing all values \(v_i\)
from the rules, the other containing all values of bound variables.
In the second step we need to apply the supplied unzipping-and-
concatenating function to the latter list to get the variable values
proper. This new improved \text{manyMatch} will thus look like

\[
\text{manyMatch} :: \text{Matcher } e \ (a, b) \rightarrow \text{Matcher } e \ \langle [a], c \rangle \\
\text{manyMatch} \ 	ext{matcher} \ 	ext{unzipper} = \text{do}
\quad \text{res } \leftarrow \text{mMatch} \ 	ext{matcher}
\quad \text{let } (\text{vals}, \text{vars}) = \text{unzip res}
\quad \text{vs } = \text{unzipper} \ 	ext{vars}
\quad \text{return } (\text{vals}, \text{vs})
\]

where \text{mMatch} is our old definition of \text{manyMatch}.

As an example, we show the translation of the pattern \((\text{Tel } n)r)*\.
The first step is to translate the subpattern \(\text{Tel } a\), which we have
already seen how to do. The new function that we generate will
then look like

\[
\text{match2} :: \text{Matcher } \text{CMode} \ [(\text{CMode}|,|\text{TelNr})]
\text{match2} = \text{manyMatch} \ 	ext{match1 \ unzipper}
\]

assuming the matcher for the subpattern is called \text{match1}. The
function \text{unzipper} here is simply the concat function, since there is
only one variable bound. To account for the greedy version of a
repetition pattern, \(\ast\), we simply flip the arguments to +++ in
\text{manyMatch}, which will give a higher priority to the case when we
actually match an element.

Non-empty repetition patterns, +, are very similar to ordinary repet-
ition patterns, the only difference is of course that we make an
initial match before starting the recursion, as shown in

\[
\text{neManyMatch} :: \text{Matcher } e \ (a, b) \rightarrow \text{Matcher } e \ \langle [b], c \rangle
\rightarrow \text{Matcher } e \ \langle [a], c \rangle
\text{neManyMatch} \ 	ext{matcher} \ 	ext{unzipper} = \text{do}
\quad \text{res1 } \leftarrow \text{matcher}
\quad \text{res } \leftarrow \text{mMatch} \ 	ext{matcher}
\quad \text{let } (\text{vals}, \text{vars}) = \text{unzip res1:res}
\quad \text{vs } = \text{unzipper} \ 	ext{vars}
\quad \text{return } (\text{vals}, \text{vs})
\]

6.2.3 Choice and Optional patterns

Choice patterns are slightly trickier to handle because of the way
variables are bound. As we saw in the rules M-CHOICE1 and M-
CHOICE2, any variables appearing in the other branch than the one
being matched should be bound to empty lists. This is very difficult
to handle generically since we need access to the meta-information
of variable names. Thus we instead generate the full code for the
choice pattern during translation. As an example we translate the
pattern \((\text{Tel } nr) | \text{ Email } eaddr\)\. We start by translating the
subpatterns, resulting in two functions that we assume are named
\text{match1} and \text{match2}. The code generated for the choice pattern will
be

\[
\text{match3} :: \text{Matcher } \text{CMode} \\
\quad \text{Either } \text{CMode} \text{CMode}, (\langle\text{TelNr}\rangle,|\text{EAddr}\rangle))
\text{match3} = (\text{do } (\text{val}, (a)) \leftarrow \text{match1}
\quad \text{return } (\text{left } \text{val}, (a, [])))
\quad +++
\quad (\text{do } (\text{val}, (b)) \leftarrow \text{match2}
\quad \text{return } (\text{right } \text{val}, ([], b)))
\]

where we have tagged the result value of the pattern match with the
respective constructors from the \text{Either} type.

The story is very similar for optional patterns, but this time all vari-
ables should be bound to empty lists if no match is done. For the
pattern \((\text{Tel } n)r)?\ we get

\[
\text{match4} :: \text{Matcher } \text{CMode (Maybe } \text{CMode), [\text{TelNr})}
\text{match4} = (\text{return } (\text{Nothing }, [])) +++
\quad (\text{do } (\text{val}, (a)) \leftarrow \text{match1}
\quad \text{return } (\text{Just } \text{val}, a))
\]

For a greedy optional pattern we would simply switch the argu-
ments to ++, just as for repetition patterns.

6.2.4 Subsequences

The trickiest pattern to implement is subsequence, due to the need
for flattening. As we saw in section 5, flattening is done based on
the type of a subpattern (with respect to some base type for ele-
ments in the input list), which means that the preprocessor must
keep track of these types in order to insert the proper flattening
functions. For a pattern \(/ (/ \text{Tel } n), (\text{Email } eaddr)* /\) we get
the following translation, assuming the two subpatterns are translated into matcher functions match1 and match2 respectively:

```haskell
match5 :: Matcher CMode ((CMode, (TelNr, | Email eaddr)))
match5 = do (val, (a, b)) <- match1
           (val2, (b, a)) <- match2
           let v1f = maybe [] (\(v, n\) -> [v]) v1
               v2f = concatMap (\(v, n\) -> [v]) v2
           return (v1f ++ v2f, (a, b))
```

The value \(v1\) is the result of \(match1\), i.e. the matcher for \((Tel\ nr)\), so it will have type \(Maybe\ CMode\). To flatten it we use the built-in Haskell function \(maybe\) that takes two arguments, one that is a default value to return if it encounters a \(Nothing\ in\ this\ case\ \([]\)), the other a function to apply to a value held by a \(Just\ (in\ this\ case\ the\ flattening\ function\ for\ a\ value\ of\ the\ base\ type)\). Similarly \(v2\) comes from \(match2\), so its type will be \(CMode\). We flatten it using the built-in function \(concatMap\) that takes a function, applies it to all elements of a list, and then concatenates the results.

### 6.2.5 Variable bindings

Finally we turn to the explicit binding operators. Binding a variable to a value in our matcher means to add that value to the result tuple. Since an explicitly bound variable syntactically appears to the left of any variables in its subpattern, we add the value in the leftmost position in the tuple, i.e. before those bound in the subpattern. Thus we know that the values in the result of the top-level matcher should be bound to variables from left to right in the order they appear in the pattern. As an example consider the pattern \(a\@\(Tel\ nr\ | Email\ eaddr\)). We first translate the subpattern \((Tel\ nr\ | Email\ eaddr)\) into a matcher \(match1\). The matcher generated for the variable binding will then be

```haskell
match2 :: Matcher CMode (Either CMode CMode,
                   (Either CMode CMode, [TelNr, | Email eaddr]))
match2 = do (val, (nr, eaddr)) <- match1
           return (val, (val, nr, eaddr))
```

If we had instead used non-linear binding, i.e. \(a\@\(Tel\ nr\ | Email\ eaddr)\), we would get a list for the returned value, i.e.

```haskell
match2 :: Matcher CMode
                   (Either CMode CMode,
                   ([Either CMode CMode], [TelNr, | Email eaddr]))
match2 = do (val, (nr, eaddr)) <- match1
           return (val, ([val], nr, eaddr))
```

### 6.3 Matching

Now we know how to translate a regular expression pattern into a top-level matcher function, what is left is to insert and invoke the generated matcher at the right place to preserve the pattern matching semantics. To this end we use Haskell pattern guards [EPJ00] that allow us to evaluate a function and pattern match on the result as part of the original pattern match. The function that we so wish to evaluate is \(runMatch\) applied to our generated top-level matcher and the input list that we wish to match. For our matcher functions to be in scope we add them to the \(where\) clause of the declaration that the regular expression pattern appears in. To show a complete example of the translation of a function declaration we revisit our function \(allTels\) defined as

```haskell
allTels (Person _ | (Tel nr | _)*!) = nr
```

Since it contains several different features of regular expression patterns, the translated version of this function will look like

```haskell
allTels (Person _ arg0)
  | Just (nr) <- runMatch match5 arg0 = nr
  where match0 e = case e of
    Tel nr -> Just (\(n\r\)
      _ -> Nothing
    )
match1 = baseMatch match0
match2 = baseMatch (\(v\) -> Just ()
match3 = (do (val, (nr)) <- match1
           return (Left val, (nr)))
        ++
        (do (val, ()) <- match2
           return (Right val, (\())
match4 = manyMatch match3 unzip1
match5 = (do (val, (nr)) <- match4
           let v1f = concatMap (\(v\) -> [v])
               v2f = concatMap (\(v\) -> [v])
           return (v1f ++ v2f, (a, b))
```

The functions \(match0\) and \(match1\) together correspond to the pattern \((Tel\ nr)\). Note the list around the returned variable \(nr\) signaling that the pattern is matched in a non-linear context. \(match2\) corresponds to the pattern \(_\). Combining these two into a choice patterns yields \((Tel\ nr | _)\), which is translated to \(match3\). On top of that we add a repetition, which gives us \(match4\) when translated. Finally since the top-level pattern should be matched as a subsequence, as seen in the rule \(HM-RegPar\), we translate it into \(match5\). The actual matching is done in the pattern guard that applies \(runMatch\) to the matcher and the input list. The latter is held by an automatically generated fresh variable, in this case \(arg0\). It is also interesting to note that the actual binding of variables to values does not happen until \(runMatch\) is evaluated. Any mention of variable names in the matcher functions, e.g. \(nr\ in\ match0\), are only there as mnemonic aids to a human reader. We could change all such names to freshly generated variable names without changing any semantics.

In Haskell, patterns can appear in numerous places such as function declarations, case expressions, let expressions, statements etc. Translating regular expression patterns into vanilla Haskell is slightly different depending on just where the pattern appears. The generated matchers will be identical in all cases, but the placement of them and of the evaluation may differ. We will not go through these differences in detail, but our implementation handles all cases correctly. Inefutable (lazy) patterns also require special care, and we have yet to implement support for them in full.

### 7 Related Work

Pattern matching is a well-known and much studied feature of functional languages [Aug85, Wad87, Mar92, Mar94]. It provides the starting point for the work presented in this paper.

Regular expressions have been used in programming for a long time, mostly for text matching purposes. Perl’s support for regular expressions is probably one of the most well-known [Perl], but most mainstream languages, including Haskell, have some library support for regular expression text matching. Regular expressions in such libraries are themselves encoded as strings. Matching them means taking two strings, where one encodes a regular expression, and match them to each other. This is in some sense very low-
level when compared to our regular expression patterns since there are no guarantees that regular expressions encoded as strings are well-formed, and there is no direct way to bind variables to values during a match. Yet another drawback is of course that such regular expressions work on strings only, whereas our regular expression patterns work over lists of any datatype.

The recent trend in XML-centric languages has led to several new languages with support for regular expression pattern matching such as XMLambda [MS99], XDupe [HP03] and CDupe [BCF03]. Most similar to ours is probably CDupe, a general purpose XML-centric programming language. The main focus in this language is its regular expression types which are used to validate XML documents. Borrowing from XDupe they also have regular expression patterns which are tightly coupled with the type system. This allows for very precise type information to be propagated in the right hand side of a pattern. The main difference with our work is the close connection with the type system. Our extension is little more than just syntactic sugar which makes it very easy to implement.

Another recently developed language that features regular expression patterns is Scala [Scala]. Scala is a multi paradigm language supporting both object orientated and functional programming. Its regular expression facility is rather similar to ours but differs at the following points: Firstly, there is only one variable binding construct which has a context dependent behaviour. Secondly, Scala has non-greedy operators just as we do but have no greedy counterparts. This can make some patterns awkward to express. Scala’s regular expression patterns work for arbitrary sequences.

There has been some work in extending Haskell with the full power of XDupe, called XHaskell [LS04]. This work focuses on fitting the type system of XDupe into Haskell and encoding it using Haskell’s class system. They also have regular expression patterns but these are intimately coupled with regular expression types and do not work together with ordinary pattern matching.

8 Future Work

There are several areas where our regular expression patterns extension can be improved. It is not obvious that our implementation using a monadic parser is the most efficient approach, on the contrary. There has been lots of work on efficient matching of regular expressions and it is likely that some of these techniques could be used with our system to make it more efficient.

We will need to devise and implement a type checking algorithm for our regular expression patterns on top of Haskell’s type checking mechanism. Being able to type check our regular expression patterns before translating them into vanilla Haskell, as opposed to our current implementation that first translates and then lets a Haskell type checker do the work, would, if nothing else, lead to much improved error messages.

9 Acknowledgement

We would like to thank our shepherd Erik Meijer for his many suggestions which improved the paper enormously. Thanks also to Karol Ostrowsky and David Sands who gave valuable feedback on draft versions of this paper. The participants of the Multi Meeting provided insightful comments when we presented the material in this paper. Lastly thanks to the anonymous referees for their comments.

This work was partially funded by the Swedish Foundation for Strategic Research.

10 References


