ITP Exercise 6

due Friday 2nd June

1 Exercise 5

Please finish question 1.3 and 1.4 from exercise 5. You might find the simplifier helpful.

2 Final Project

We have 3 more weeks to go on the lecture. For the last 3 weeks, I would like to set a small project to solve for everyone. There will be a default project (see below). However, if you want you can also propose your own project. This has to be approved by me, before you can start working on it. It should satisfy the following requirements:

- You should build a formal model of some description. This description can e. g. be a natural language text or some computer program.
- You should test your model against the description / implementation.
- You should prove some interesting properties.
- It should be do-able in a reasonable amount of time (ideally 3 weeks). You have to either convince me that it is doable in 3 weeks or worth both your and my additional time.

Please read the default project proposal below. Either decide to do this project or think of an alternative final project. In any case, discuss your choice with me. **Whoever has not discussed a final project with me by June 2nd will be forced to do the default project.**

3 Default Project - Regular Expressions in HOL

There is a fun paper on regular expressions: *A Play on Regular Expressions* by Sebastian Fischer, Frank Huch and Thomas Wilke published as a functional pearl at ICFP 2010 (`http://www-ps.informatik.uni-kiel.de/~sebf/pub/regexp-play.html`). In this paper an implementation of marked regular expressions in Haskell is described. The task is to formalise the simple parts of this work in HOL, verify the correctness of the implementation and export trustworthy code into an SML library.

You should develop this project such that (in theory) it could be added to the examples directory of HOL. Therefore, I want you to create a git-repository for your project and give me access to it. You should create one or more HOL-theories that can be compiled by Holmake. There will be multiple SML files as well. These should compile decently and have a signature. Please provide a selftest for your development. Write decent documentation. There should be a (very short) `README` as well as sufficient comments in the code.

3.1 Basic Regular Expression Semantics

Read Act 1, Scene 1. Implement the `Reg` datatype in HOL. Like later in the paper, replace the type `Char` with a free type variable `'a`. The intention is to define regular expressions on lists of type `'a`. Define a function `language_of : 'a Reg -> ('a list) set` that returns the language accepted by a regular expression. The definition of `language_of` should be as clean and simple as possible. It does not need to be executable.

3.2 Executable Semantics

Now define the function `accept` in HOL. While doing so replace the type `String` with `'a list` to match the changes to `Reg`. You will need to implement the auxiliary functions `parts` and `split`. Test your definitions and apply formal sanity checks.

3.3 Code Extraction and Conformance Testing

Familiarise yourself with `EmitML`. Use it to extract your datatype `Reg` and the function `accept` to ML. Test `accept` against the regular expression implementation in `regexpMatch.sml` that comes with HOL.

`EmitML` has not been discussed in the lecture and is not well documented. Part of this challenge is to find information for yourself about HOL libraries and learn from examples and source code.

3.4 Correctness Proof

Prove that `accept` and `language_of` agree with each other, i. e. prove the statement `!r w. accept r w <=> w IN (language_of r)`.

3.5 Marked Regular Expressions

Continue reading the paper. Act 1, Scene 2 is interesting, but we are here not interested in weights. Instead focus in Act 2, Scene 1. Implement a datatype for marked regular expressions called `MReg`. Use first the simple version with caching the values of `empty` and `final`. Provide a function `MARK_REG : 'a Reg -> 'a MReg` that turns a regular expression into a marked expression without any marks set. Implement a function `acceptM : 'a Reg -> 'a list -> bool` following the idea of the `accept` function in the paper.

Test your definitions and perform formal sanity checks.

3.6 Correctness Proof Marked Regular Expressions

Show that `acceptM` is correct, i. e. show `!r w. acceptM r w <=> w IN (language_of r)`.

3.7 Cached Marked Regular Expressions

Now let's also implement the caching of `empty` and `final`. Call the resulting mutually recursive datatypes `CMReg` and `CMRe`. Write a function `CACHE_REG : 'a MReg -> 'a CMReg` that turns a marked regular expression into a cached marked one with valid caches. Implement a function `acceptCM : 'a Reg -> 'a list -> bool` that is similar to `acceptM`, but more efficient due to using the caches.

Test your definitions and perform formal sanity checks. As part of formal sanity, define a well-formedness predicate for cached marked regular expressions stating that the cached values for `empty` and `final` are correct. Moreover, define the inverse function `UNCACHE_REG : 'a CMReg -> 'a MReg` of `CACHE_REG` and show that these functions are really inverses.

3.8 Correctness Proof Caches

Show that `acceptCM` is correct, i. e. show `!r w. acceptCM r w <=> w IN (language_of r)`.

3.9 SML Library

Use `EmitML` to extract your code to SML. Provide an interface for regular expressions on strings. The interface should contain a type for regular expression on strings similar to `char Reg`. It should provide a function `match` that checks whether such a regular expression matches a given string. Build 4 instances of this interface, one with the regular expression library `regexpMatch.sml` and ones for `accept`, `acceptM` and `acceptCM`. Write some simple tests and run them against all these instantiations (e. g. via a functor). Perform some simple performance measurements.