Advanced Language Processing - Assignment 2

Sebastian Paaske Tørholm

March 29, 2013

1 Implementation

I chose to do my implementation in Haskell, primarily due to easy access to parser combinators, ADTs and pattern matching.

1.1 Syntax tree

The syntax tree is represented as an ADT, which can be found in `Prolog.hs`. A program is represented as a sequence of clauses, each of which have a head and list of values that comprise the body. The values are comprised of predicates, variables, negation as failure, cut, the special predicate `fail` and a unification of two variables.

While negation of failure is stored in the syntax tree initially, it is later rewritten to the form described on page 138 of the book.

1.2 Parser

For parsing, I use `ReadP`. The parser is a fairly straight forward parser combinator based parser. Most of the parser is rather straight-forward, so I won't go too much into details on it.

The `list` parser performs desugaring of list syntax directly at parse-time.

I've chosen to allow arbitrary whitespace between tokens, to provide some freedom for formatting.

The `Parser` module provides three functions:

`parseFile` Parses the contents of a file into a syntax tree.

`parseString` Parses the contents of a string into a syntax tree.

`parseQuery` Parses the contents of a string into a list of goals.

Any parse errors lead to an undescribed failure result, due to the nature of `ReadP`.

No formalized testing have been done on the parser, however all test programs in the `programs/` folder have been manually verified to be parsed correctly.

2 Prolog

The implementation follows the implementation suggested in the book rather closely.

Rather than storing bindings on a modifiable “trail”, the functions pass around an environment consisting of the currently active substitutions, as well as an identifier used for generating unique identifiers for renaming.

Additionally, instead of solve directly printing the solutions when found, I return them in a list, to simplify interaction with the user.

Generated variables and predicates are marked by having a `#` as the first character in their identifier. This allows me to simplify them out of the environment, when it is to be showed to the user. The simplification step removes all temporary variables from the environment, as well as expands bindings of non-temporary variables as much as possible.

Negation as failure is converted to cuts in the function `nafToCut`. This function traverses all clauses, and when a negation as failure is found, it performs the following rewrite:

```
p(X,Y,Z) :- ..., \+ q(X,Z), ... .
```

becomes

```
p(X,Y,Z) :- ..., #naf_0(X,Y,Z), ... .

#naf_0(X,Y,Z) :- q(X,Z), !, fail.
#naf_0(X,Y,Z).
```

Note, however, that negation as failure and cut does not work as intended. I believe that this is due to the implementation of cut as described in the book being incorrect.

When a cut is encountered in a clause, backtracking is prevented beyond that point in that clause. This can be seen in the example program `simple_cut.pl`, by performing the query `a(X,Y)`.

What the implementation outlined in the book does not do, however, is prevent backtracking from continuing on other clauses with the same head. This means, in practice, that negation as failure will never fail. This can be seen in `burger_naf.pl` or `burger_cut.pl`, by evaluating the query `enjoys(vincent, X)`, which incorrectly returns `X = b` as one of the solutions.

In order to handle this, some kind of grouping would have to be done on the clauses, grouping clauses with the same identifier on the heads. Due to time constraints, I haven’t found a good solution for this. If this problem is corrected, both cuts and negation as failure should work as expected.

The `Interpreter` module provides one value, `main`, which when run executes the program. The easiest way to execute the program is to simply execute the command

```
runghc ./Interpreter.hs <programs>
```

The interpreter accepts any number of programs as input[1]. If programs are malformed, no errors are shown in the interpreter session, and the program is silently ignored.

3 Example programs

Three example programs are attached in the `programs` folder:

`simple.pl` A very simple dictionary of names.

`append.pl` The append program shown on the slides.

`unify.pl` A program testing unification of variables.

`simple_cut.pl` A simple program using cuts.

`burger_naf.pl` A simple program using negation as failure. (Doesn't work.)

`burger_cut.pl` The same program, rewritten to use cuts. (Doesn't work.)

`8queens.pl` The 8-queens program. (Doesn't work.)

[1]Yes, even 0, though that'll give a rather boring interactive session.