Imperial College London - Department of Computing

MSc Advanced Computing, MSc Computing (Spec.) Comp. 4th year, ISE 4th year & JMC 4th year

480 AUTOMATED REASONING CBC Otter Practice (Assessed)

Issued: 30 October 2012 **Due:** 12 November 2011

Before running Otter

You need to create your text input files by using the text editor of your choice. Create a file named **example_1.ot** with the following contents:

```
set(binary_res).
     list(sos).
          p( a ).
          - p( x ) | p( f( x )).
          - p( f( f( a ))).
     end_of_list.
```

Note that the suffix **.ot** (and **.oout** used further on) are not required by Otter but it is suggested that you adopt this or a similar mechanism in order to keep your files organised.

Running Otter

The version of Otter install the example just created type the command (at shell level):

```
otter < example_1.ot > example_1.oout
```

where > and < are the standard input and output channel redirection of Linux (> will actually overwrite the contents of any existing file **example_1.oout**).

If the search initiated by Otter is successful, it will terminate returning the control back to the shell. The result of the search can then be inspected by `more`-ing, `less`-ing, printing or editing the file **example_1.oout**. If you don't use the output redirection, e.g. **otter < example_1.ot**, the output will be displayed onto the screen, but it is bound to be too long and fast, hence the use of > is strongly recommended.

Interaction during the search

Otter offers a facility to interrupt the search so that the user can modify the options and then resume the search. The normal way to trigger the interrupt is to press `<control>-C`: it then enters a loop of reading and executing commands. One of the main uses for this type of interrupt is to examine searches that seem to be very time consuming. A second way to interact is to use the `set( interactive_given ).` command directly in the input file. (Note all commands must end with a full stop.)

Some of the accepted interactive commands for version 3.3 are:

`help.` Give simple help.

`set(` *flag-name* `).` Set a flag.

`clear(` *flag-name* `).` Clear a flag.

`assign(` *param-name*`,` *value* `).` Assign a value to a parameter.

`stats.` Send statistics to standard output and terminal.

`usable.` Print list usable on terminal.

`sos.` Print list sos on terminal.

`passive.` Print list passive on terminal.

`continue.` Continue the search.

`kill.` Send statistics to standard input, and exit.

`fork.` Fork and run the child process.

`options.` Allows to see the actual state of the flags and the values of the parameters.

Create a file named **example_2.ot** with the following contents:

```
set( binary_res ).
clear( factor ).
clear( back_sub ).
clear( for_sub ).
list( sos ).
      p( u, u, c ) | p( v, b, d ).
end_of_list.
list( usable ).
      - p( a, x, z ) | - p( b, y, z ).
end_of_list.
```

Use the command **otter** < **example_2.ot** > **example_2.oout** to start the search, but press `<control>-C` after a few seconds. Use the commands `sos` and `usable` to see what is happening; eventually, type `set( factor ).` which will set the factor flag, essential to solve this problem, and then type `continue`. You now can obtain a proof.

Documentation and Examples

The documentation and the set of examples distributed with the software package can be accessed at `/usr/share/doc/otter`.

Provided Files

A small set of examples chosen from the tutorial exercises of the course **"Automated Reasoning"** can be obtained in zip format from the exercises timetable on CATE.

Assessed Coursework Task

1. For each of the provided examples taken from the tutorial exercises **dataOt2.ot** – **dataOt5.ot** (4 in all) you are asked to do the following:

   (a) Experiment with various different settings of flags (including the choice of how to distribute clauses between SOS and USABLE lists) to find a refutation;

   (b) for at least one (successful) setting produce a listing of the execution up to the first refutation i.e. the **.oout** file, and

   (c) justify your settings and annotate any interesting features you observed in the proof provided or in other proofs you found (do this by editing the **.oout** file(s) produced).

   Please, if the `.oout` file is very long consider cutting some of it, or refining your choice of flags and/or parameters in order that less is produced.

2. For the first provided example (called **dataOt1.ot**) you should find **two** refutations using the `interactive_given` command as follows.

   (a) Using binary resolution (i.e. `set(binary_res.)` find a clause selection order that simulates a saturation search. You will need to be careful that Otter doesn't take short cuts that violate the saturation search strategy. For instance, `unit_deletion` is one such, so this flag is probably better cleared for this part.

   For the chosen selection order, investigate the effects of setting or clearing the flags `for_sub` and `back_sub`.

   (b) Using hyper-resolution and any subsumption or factoring flags that you wish, investigate the effect of using different predicate orderings on the positive literals in electrons. (See below for information on predicate ordering in hyper-resolution in Otter.)

   In both cases the listing(s) should be produced in a **.oout** file (call them **dataOt1a.oout** and **dataOt1b.oout**) and annotated to explain the output, including for (1) why you selected clauses in the order that you did, and for (2) explaining the effect the different orderings made on the generated proofs.

Submit the 6 **.oout** files via the CATE system.

Notes about hyper-resolution and ordering in Otter

The Otter hyper-resolution rule chooses the **maximal** literal in an electron clause. (*Note that this is contrary to what we did in lectures, which was to choose the minimal literal.*) This restriction can be removed by using

```
clear(order_hyper)
```

which tells Otter to use any literal from an electron if possible.

The default order in Otter for predicates puts ternary predicates $\prec$ binary predicates $\prec$ unary predicates, and within the predicates of the same arity they are ordered alphabetically. So the predicates `on`, `blue` and `green` are ordered in that order, meaning that `green` is maximal and is selected before `blue`, which is selected before `on`.
You can set a different predicate order for hyper-resolution using the `lex` command. For example,

```
lex([green(_),on(_,_),blue(_)])
```

tells Otter to select `blue` before `on` before `green` when using hyper-resolution. (The maximal predicate here is `blue`.) The underscores are used to denote argument positions. The use of `clear(order_hyper).` clears the use of predicate ordering in hyper-resolution. It doesn't matter where the command is placed relative to any `lex` commands you may have used.
Note that predicate ordering is only available in Otter for hyper-resolution (or negative hyper-resolution).

Notes about the `passive` list

A third list, `list(passive)` can be useful for controlling the proof process. Otter uses this list in two ways:

1. If a unit clause is in the `passive` list and a complementary unit clause is derived then Otter immediately derives the empty clause.

2. Any derived clause that is subsumed by a clause in the list will not be kept.

   For example, the clause $p(f(x))$ in the list would subsume clauses containing literals such as $p(f(a))$ and $p(f(f(y)))$ (this for all y). If the user thinks such clauses are unlikely to contribute to a proof, this is a good way of preventing them from leading to non-useful clauses.

Problems and Bugs

Please e-mail Krysia Broda (kb@imperial.ac.uk) about any problems you encounter.