Harvey Mudd College
Computer Science 80
Logic for Computer Science
Fall Semester 2001

Optional Project 3: Fun with `otter`

Due 5:00pm, Saturday, December 22, 2001

The purpose of this project is to give you experience in working with an industrial-strength theorem proving system. The example we have chosen is `otter`.

Running `otter`:
The general command for running `otter` on `turing` is

```
otter <infile >outfile
```

You can omit the `outfile` part if you want to see the output flash by on the screen.

There is documentation for `otter` in PDF form on the course homepage, as well as in the directory `/usr/local/src/otter-3.0.6` on `turing` and on the `otter` homepage at `http://www-unix.mcs.anl.gov/AR/otter`. The latter web address also has `otter` source code for Unix computers and binaries for PC's and Macintosh. It is recommended that you read at least through Section 5 of the manual to get the full details of basic input syntax, etc.

(When `otter` is launched on the Macintosh it presents a dialogue box for specifying input and output files as well as providing a field for entering other command-line arguments.)

`otter` will stop automatically when it is finished. If you think it is in an infinite loop or you are tired of waiting, type control-C and then `kill.` (Include the period.) Note that `otter` can consume considerable system resources. For this project, no run should take more than a couple of minutes. Use the `nice` command if you run a process for a longer time. (See the man page for `nice`.)

Input to `otter`:
`otter` takes a file of formulas as input and attempts to build a refutation using resolution, and several variants of that algorithm, as well as a number of heuristics. Thus, if you want to know whether a consequence $\Gamma \models \Phi$ holds, the file should contain the formulas in Γ and the formula $\neg\Phi$.

`otter` can accept fomulas in two forms: directly as first-order formulas, in which case all the operators are available to you, and all variables must be bound by explicit quantification; or in clausal form, in which case you provide a list of clauses (disjunctions of literals), and all free variables are presumed to be universally quantified at the outside of the clause.

If you are using the formula representation, lists of formulas are bounded by the delimiters "`formula_list(`*type*`).`" and "`end_of_list.`", where *type* is most commonly either "`usable`", or "`sos`", which will be explained briefly below. The `otter` syntax for logical formulas is fairly natural. The usual connectives are written with the keyboard symbols `-` for negation, `&` for conjunction, `|` for disjunction, and `->` for implication. Parentheses should be used for grouping. (The `otter` manual gives default operator precedence information.) Quantifiers are written as `all` and `exists` followed by the quantified variable and a subformula in parentheses. If you have several quantifiers in a row of the same type, you can just give a series of variable names after the quantifier of that type. Thus the formula:

$$\forall x(\exists y(\exists z((p(x) \wedge q(y)) \Rightarrow r(z))))$$

would be written:

```
all x (exists y z ((p(x) & q(y)) -> r(z))).
```

Function symbols are permitted, as is equality. Negation of equality can be written `!=`. A period marks the end of a formula.

If you are using the clausal notation, then the delimiter "`formula_list(`*type*`).`" is replaced by "`list(`*type*`).`" In addition, the only operators allowed are `-`, applied to atoms, and `|`, between literals. Any name beginning with the letters `u`, `v`, `w`, `x`, `y`, or `z` is assumed to be a variable, and is treated as universally bound. Any other name in term position is assumed to be a constant. (Alternately, there is a flag that can be set to make the system abide by the Prolog convention of beginning variables with upper-case characters.) Skolem constants should be identified at the begining of the file using the declaration `skolem`. Thus, the formula above, which is equivalent to:

$$\forall x(\exists y(\exists z((\neg p(x) \vee \neg q(y) \vee r(z)))))$$

would be rendered in clausal form as:

```
-p(x) | -q(sk1) | r(sk2).
```

with the addition of the declaration `skolem([sk1,sk2]).` somewhere at the beginning of the file.

It is important to note that if you say you are using clausal form (through a `list(`*type*`)` declaration) but use other operators in the formula, otter **will not** report any sort of syntax error, but will not function correctly.

Three sample input files are attached. The first two show the same file, the first using formula notation, and the second using clausal notation. Both expect `otter` to run in the most automated mode, with the declaration

```
set(auto).
```

at the top and then a list of formulas for refutation. Notice that every directive and every clause ends with a period. In the `auto` mode, `otter` tries to determine on its own which proof strategies would be most effective.

For this example, `otter` is not very effective. It works better if we use the third file. There, we specify the search strategy, with

```
set(binary_res).
```

calling for ordinary resolution. We also divide the formulas into two lists, `usable` and `sos` (for *set of support*). The formulas in the `sos` list are never candidates for mutual refutation. They correspond to "facts" in a database. The `sos` list is useful in bounding `otter`'s searches. The third sample produces a refutation which demonstrates that Bart and Lisa really are siblings.

The fourth example file demonstrates the trade-off decisions you must often make in designing an axiomatization. In the first three versions of the file, motherhood and fatherhood are represented by functions mapping from the child to the parent. In the fourth example they are represented as relations, the same as parenthood. In some cases this sort of representation has advantages. In this case, this example can be run in `auto` mode successfully, unlike the first and second examples. (I am not sure if the other style of representation ever has advantages over this style.)

In addition to `auto` and `binary_res`, you can set `hyper_res`, `knuth_bendix`, and others. They are listed, and briefly described, in the `otter` documentation. The `auto` mode must be used alone, but the other strategies may be used in any combination. With `auto`, you *cannot* specify an `sos` list. With the other strategies, you *must* have an `sos` list (but it can be the entire list if you are unsure where to start).

Output from `otter`:
There are three possible outcomes when you run an `otter` program:

- The program will run for a very long time, perhaps forever.
- The program will halt with no refutation. This can either be because the clauses are satisfiable, or it could be because `otter` just could not find a refutation, since its heuristics are incomplete. In these situations you will have to specify a different strategy. Note that there is no way to tell from `otter`'s output which of these two situations caused the failure to find a refutation.
- The program will produce a refutation. The most important part of the output file, for our purposes, is the transcript of the refutation.

  Among the many lines of output for the third sample file is the following proof, which should look familiar to students of resolution:

```
---------------- PROOF ----------------

1 [] -Sibling(bart,lisa).
2 [] Sibling(x,y)| -Parent(z,x)| -Parent(z,y)|x=y.
3 [] Parent(x,y)|x!=father(y).
6 [] bart!=lisa.
18 [] homer=father(bart).
19 [] homer=father(lisa).
22 [binary,6.1,2.4,unit_del,1] -Parent(x,bart)| -Parent(x,lisa).
40 [binary,18.1,3.2] Parent(homer,bart).
43 [binary,19.1,3.2] Parent(homer,lisa).
49 [binary,22.1,40.1] -Parent(homer,lisa).
50 [binary,49.1,43.1] $F.

------------ end of proof -------------
```

Any time the sytem halts, either with or without a refutation, it produces extensive statistics about the proof search. A variety of switches exists to control the details presented.

Answer Substitutions:
The fifth sample file changes the query to attempt to prove that Bart has a sibling, rather than that a particular person, Lisa, is that sibling.

When the query is an existential, we are generally interested in obtaining the answer substitution (the *witness* or *exemplar* of the proof) where possible. Since there is no particular indication in the file which formula or clause is the "query", `otter` has no idea which existential we want the answer substitution for. To solve this the sytem allows you to insert *answer literals* wherever you'd like. An answer literal is any literal in which the predicate name begins with `$ans`. These literals are simply carried along for the ride during the dearch for a refutation. The system does not consider them when looking for clashing literals, and

it considers any clause that contains only answer literals to be the same as the empty clause (box).

By including in the query clause an answer literal containing the variables for which we want the instantiations, as is done in the fifth example file, we can extract the witness from the refutation. The fifth file generates the following refuation, with the name of a sibling in the last line:

```
---------------- PROOF ----------------

1 [] -Sibling(bart,x)| -$ans(x).
2 [] -Parent(z,x)| -Parent(z,y)|x=y|Sibling(x,y).
3 [] -father(x,y)|Parent(x,y).
6 [] bart!=lisa.
18 [] father(homer,bart).
19 [] father(homer,lisa).
22 [binary,6.1,2.3,unit_del,1] -Parent(x,bart)| -Parent(x,lisa)| -$ans(lisa).
34 [binary,18.1,3.1] Parent(homer,bart).
35 [binary,19.1,3.1] Parent(homer,lisa).
59 [binary,22.1,34.1] -Parent(homer,lisa)| -$ans(lisa).
60 [binary,59.1,35.1] -$ans(lisa).

------------ end of proof -------------
```

Notice that this last example file also includes a setting that specifies that we would like to see up to four different proofs (i.e. refutations) if they can be found. This may enable us to find more than one answer substitution.

Project Problems:

In the next three pages are a series of problems requiring you to prove facts about a variety of systems in `otter`.

For each of the first 7 problems you should concatenate `otter`'s output to you input file and use `cs80submit` to submit the result as the solution to that particular problem of project 3. For part 4 you should submit the human-readable proofs on paper.

PART 1. This first problem is a warm-up in propositional logic, but nevertheless proves the correctness of an important programming technique. In particular, you may or not be aware that in languages that support bitwise operations, you can swap the values in two variables without using a "temp" variable as an intermediary. The technique involves three identical applications of the exclusive-or operation. For example, to swap two integer variables `a` and `b` in Java, in which the exclusive-or operations is written "`^`", you would write:

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

Historically, this was the most efficient way to swap two variables. That is not necessarily the case any more, and there are cases in which the technique is dangerous. (If, for example, the two variables are pointers, there is a problem if garbage collection occurs between the first and third lines.) Nevertheless the technique is still of interest.

If you are not familiar with this technique, you should first convince yourself that it works by trying it on two one-bit variables.

Problem 1. Have `otter` prove that this method is correct for one-bit values (which are the same as propositional variables). In order to accomplish this you should write down a series of equivalences that relate succesive new values of each of the program variables, `a` and `b`, to old values. Then prove that the final value of `a` is logically equivalent to the initial value of `b`, and that the final value of `b` is equivalent to the original value of `a`.

You will need one propositional variable for each value that a given program variable takes on in the course of the computation. Thus you will need three propositional variables for the values of `a` (initial, after the first line, and final) and two for `b` (initial and final).

Since `otter` does not have an exclusive-or operator, you should use one of the following alternatives (Note that the standard representation on exclusive-or is $\oplus$.):

$$(A \oplus B) \equiv ((A \vee B) \wedge \neg(A \wedge B)) \equiv (\neg(A \equiv B))$$

PART 2. Let `f` be a binary, associative function with a right identity and right inverses. (Think about addition of numbers or multiplication of matrices.) These properties are characterized by the following axioms, in which g is the right inverse function:

```
f(x,e) = x                  % e is right identity
f(x,g(x)) = e               % g is the right inverse function
f(x,f(y,z)) = f(f(x,y),z)   % f is associative
```

Use `otter` to prove the following facts from the three axioms:

Problem 2. Right identity is also Left identity: `f(e,x) = x`

Problem 3. Right inverse is also Left inverse: `f(g(x),x) = x`

Problem 4. Inverse formula: `g(f(x,y)) = f(g(y),g(x))`

It does not follow from the three axioms that the operation `f` is commutative, but we can *prove* commutativity if we assume the "naive" inverse formula, `g(f(x,y)) = f(g(y),g(x))`.

Problem 5. Given the three axioms plus the naive inverse formula, use `otter` to prove Commutativity: `f(x,y) = f(y,x)`

PART 3. We are going to use the deductive powers of `otter` to investigate proofs in a system of propositional logic. The so-called Hilbert systems use only two connectives $\Rightarrow$ and $\neg$. The remaining connectives are considered to be abbreviations for expressions in terms of these two.[1]

There is one rule of inference, *modus ponens:* From p and $p \Rightarrow q$, one can conclude q. This rule is used to construct proof trees in which the leaves are one of a set of *axiom schema* of the system. There are various axiom systems, one of which has three schemas and is attributed to Church:

Schema 1. $A \Rightarrow (B \Rightarrow A)$

Schema 2. $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (B \Rightarrow C))$

Schema 3. $(\neg B \Rightarrow \neg A) \Rightarrow (A \Rightarrow B)$

Using `i( , )` for implication and `n( )` for negation, we can express such formulas as *terms* in `otter`. We can also introduce a predicate `P( )` to denote provability. Instances of the three axiom schemas are obviously provable; that fact is expressed by the following assertions (in clausal form, where `x`, `y`, and `z` are considered as universally quantified variables):

```
P(i(x,i(y,x))).
P(i(i(x,i(y,z)),i(i(x,y),i(x,z)))).
P(i(i(n(y),n(x)),i(x,y))).
```

By adding an assertion representating the *modus ponens* inference schema, use `otter` to determine that there are Hilbert-style proofs of the following formulas.

Problem 6. $p \Rightarrow p$

Problem 7. $(p \Rightarrow q) \Rightarrow ((q \Rightarrow r) \Rightarrow (p \Rightarrow r))$ Hint: There is a proof that does not use axiom schema 3.

PART 4: Extract human-readable Hilbert proofs from the `otter` refutations in problems 6 and 7 above. (A sample Hilbert proof for number 5 appears at the bottom of page 20 in the notes at the end of your course packet. Does `otter` come up with the same proof?)

[1] It follows that Hilbert systems are inherently classical, because the definition of $p \vee q$ is $\neg p \Rightarrow q$. With that characterization of disjunction, the law of the excluded middle is just $\neg p \Rightarrow \neg p$, which is proved in problem number 6.

```
%
%  The first sample input file to otter.
%
%  Josh Hodas
%  April 21, 2001
%  CS 80
%

set(auto).

formula_list(usable).

  -Sibling(bart, lisa).   % refutation goal

  all x y z ((Parent(z,x) & Parent(z,y) & x != y) -> Sibling(x,y)).
  all x y (x = father(y) -> Parent(x,y)).
  all x y (x = mother(y) -> Parent(x,y)).

  bart  != homer.
  bart  != lisa.
  bart  != maggie.
  bart  != marge.
  homer != lisa.
  homer != maggie.
  homer != marge.
  lisa  != maggie.
  lisa  != marge.
  marge != maggie.

  marge = mother(bart).
  marge = mother(lisa).
  marge = mother(maggie).
  homer = father(bart).
  homer = father(lisa).
  homer = father(maggie).

end_of_list.
```

```
%
%  The second sample input file to otter.
%  The same as the last file, but using clausal form.
%  (This actually only effects the three core formulas.)
%
%  Rett Bull
%  April 20, 2001
%  CS 80
%

set(auto).

list(usable).

  -Sibling(bart, lisa).   % refutation goal

  Sibling(x,y) | -Parent(z,x) | -Parent(z,y) | x = y.
  Parent(x,y) | x != father(y).
  Parent(x,y) | x != mother(y).

  bart  != homer.
  bart  != lisa.
  bart  != maggie.
  bart  != marge.
  homer != lisa.
  homer != maggie.
  homer != marge.
  lisa  != maggie.
  lisa  != marge.
  marge != maggie.

  marge = mother(bart).
  marge = mother(lisa).
  marge = mother(maggie).
  homer = father(bart).
  homer = father(lisa).
  homer = father(maggie).

end_of_list.
```

```
%
%  The third sample input file to otter.
%  This one specifies a specific strategy, and, more
%  importantly, in this case, an initial set-of-support.
%
%  Rett Bull
%  April 20, 2001
%  CS 80
%
set(binary_res).

list(usable).

  -Sibling(bart, lisa).   % refutation goal

  Sibling(x,y) | -Parent(z,x) | -Parent(z,y) | x = y.
  Parent(x,y) | x != father(y).
  Parent(x,y) | x != mother(y).

end_of_list.

list(sos).

  bart  != homer.
  bart  != lisa.
  bart  != maggie.
  bart  != marge.
  homer != lisa.
  homer != maggie.
  homer != marge.
  lisa  != maggie.
  lisa  != marge.
  marge != maggie.

  marge = mother(bart).
  marge = mother(lisa).
  marge = mother(maggie).
  homer = father(bart).
  homer = father(lisa).
  homer = father(maggie).

end_of_list.
```

```
%
%  The fourth sample input file to otter.
%  This one, based on the last, uses two-place
%  predicates rather than function symbols.
%
%  Josh Hodas
%  April 22, 2001
%  CS 80
%
set(auto).

formula_list(usable).

  -Sibling(bart, lisa).   % refutation goal

  all x y z ((Parent(z,x) & Parent(z,y) & x != y) -> Sibling(x,y)).
  all x y (father(x,y) -> Parent(x,y)).
  all x y (mother(x,y) -> Parent(x,y)).

  bart  != homer.
  bart  != lisa.
  bart  != maggie.
  bart  != marge.
  homer != lisa.
  homer != maggie.
  homer != marge.
  lisa  != maggie.
  lisa  != marge.
  marge != maggie.

  mother(marge,bart).
  mother(marge,lisa).
  mother(marge,maggie).
  father(homer,bart).
  father(homer,lisa).
  father(homer,maggie).

end_of_list.
```

```
%
%  The fifth sample input file to otter.
%  This one has an existential query with answer literal.
%
%  Josh Hodas
%  April 22, 2001
%  CS 80
%
set(binary_res).

assign(max_proofs,4).

formula_list(usable).

  -(exists x (Sibling(bart, x) & $ans(x))).   % refutation goal with answer literal

  all x y z ((Parent(z,x) & Parent(z,y) & x != y) -> Sibling(x,y)).
  all x y (father(x,y) -> Parent(x,y)).
  all x y (mother(x,y) -> Parent(x,y)).

end_of_list.

formula_list(sos).


  bart  != homer.
  bart  != lisa.
  bart  != maggie.
  bart  != marge.
  homer != lisa.
  homer != maggie.
  homer != marge.
  lisa  != maggie.
  lisa  != marge.
  marge != maggie.

  mother(marge,bart).
  mother(marge,lisa).
  mother(marge,maggie).
  father(homer,bart).
  father(homer,lisa).
  father(homer,maggie).

end_of_list.
```