Constructive Logic (15-317), Fall 2009
Assignment 10: Bottoms Up!

William Lovas (wlovas@cs)

Out: Friday, November 20, 2009
Due: Thursday, December 3, 2009 (before class)

Celebrate the final assignment of the course with bottom-up logic programming! In this assignment, you'll experiment with various bottom-up logic programs, including both saturating programs and stateful fact-consuming programs. The problems ask you to write programs using a bottom-up linear logic programming language called Lollibot; see Appendix A for a brief primer.

Submit your Lollibot code by copying it to the directory

```
/afs/andrew/course/15/317/submit/<userid>/hw10
```

where <userid> is replaced with your Andrew ID. Some problems ask you to analyze the complexity of your programs; you may do so in comments.

You have two weeks to complete this assignment, thanks to Thanksgiving. Enjoy!

Note: If you have any difficulties with the Lollibot implementation, please send an email to Rob Simmons (rjsimmon@cs.cmu.edu) and myself. We'll address any problems as promptly as possible!

1 Saturating Logic Programming (25 points)

1.1 Program Analysis: Reaching Definitions

In lecture, we saw how liveness analysis could be expressed succinctly as a saturating bottom-up (forward search) logic program. Many other standard dataflow analyses are similarly natural in the bottom-up style.

Many optimizations and program analyses require that we know the *reaching definitions* for a program point. A definition of the variable x at line l of a program *reaches* a line k if there is some program path between l and k along which x is not redefined.

Recall the simple command language from lecture, recapitulated in Figure 1. An instruction is either an assignment, a conditional branch, an unconditional branch,

Concrete syntax	Lollibot notation
$l : x = op(y, z)$	`!inst` l `(assign` x op y z`)`
l : if x goto k	`!inst` l `(if` x k`)`
l : goto k	`!inst` l `(goto` k`)`
l : halt	`!inst` l `halt`

Figure 1: A simple command language.

or the halt instruction. The commands of a program are represented in Lollibot as persistent facts in the initial database.

Task 1 (10 pts). Write a saturating predicate `reaches L X K` which holds after saturation if the definition of `X` at line `L` reaches line `K`. Put your code in a file called `reaches.lob`.

Task 2 (5 pts). Analyze the complexity of your program: give an asymptotic upper bound on the size of the saturated database, annotate each rule with an asymptotic bound on its number of prefix firings, and use these to compute an asymptotic bound on the running time of the program as a whole.

1.2 Bottom-Up from Top-Down: Magic Sets

Recall the *magic templates transformation* (or *magic sets transformation*) from lecture, a general procedure for converting a terminating top-down (backward search) logic program into a saturating bottom-up (forward search) logic program. To apply the transformation, we must know the intended mode of the predicate we're converting. First, we create an auxiliary predicate that generates all of the inputs we care about—our initial database will be seeded with this predicate applied to the inputs of our query. Then, we guard all of the clauses of the original top-down backward search predicate with calls to the auxiliary predicate in order to ensure that we only generate facts that are pertinent to our query.

For example, consider the `plus(+M, +N, -P)` predicate on natural numbers.

```
plus(z, N, N).             %% bw clause 1

plus(s(M), N, s(P)) :-     %% bw clause 2
    plus(M, N, P).
```

Our first step is to create the auxiliary predicate that generates all of the necessary inputs. We call this predicate `go-plus(M, N)`, and it only operates on the inputs of the `plus` predicate. Its clauses mimic the input parts of the recursive clauses of `plus`, but swapping the premise and the conclusion.

```
go-plus(M, N) :-          %% go clause
    go-plus(s(M), N).
```

Intuitively, this clause says that if we want to compute the addition of `s(M)` and `N`, we may need to compute the addition of `M` and `N`, so the conclusion records that fact in the database.

Next, we guard the clauses of the original `plus` predicate with `go-plus` facts in order to restrict the set of `plus` facts forward search will generate to only the ones that are pertinent to our query.

```
plus(z, N, N) :-          %% fwd clause 1
    go-plus(z, N).

plus(s(M), N, s(P)) :-    %% fwd clause 2
    go-plus(s(M), N),
    plus(M, N, P).
```

Now, to use our new bottom-up version of `plus` to add two numbers like `2` and `3`, we seed the initial database with `go-plus(2, 3)` and run the rules to saturation. Then we look in the database for a fact of the form `plus(2, 3, X)`; if we find one, the term `X` is our answer.

Rewriting everything in the forward notation of Lollibot—and taking care to make all of the predicates persistent along the way—we have:

```
!go-plus (s M) N          %% go clause
  -o !go-plus M N.

!go-plus z N              %% fwd clause 1
  -o !plus z N N.

!go-plus (s M) N,         %% fwd clause 2
!plus M N P
  -o !plus (s M) N (s P).

% test by seeding initial database:
%exec * !go-plus (s (s z)) (s (s (s z))).
```

If we put this code into a file `plus.lob`, we can run `lollibot` on it to get the following output:

```
=== plus.lob ===

!go-plus (s M) N -o !go-plus M N.
!go-plus z N -o !plus z N N.
!go-plus (s M) N, !plus M N P -o !plus (s M) N (s P).
```

```
%exec * !go-plus (s (s z)) (s (s (s z))).
-- After 0 steps:
-- !plus (s(s(z))) (s(s(s(z)))) (s(s(s(s(s(z)))))),
   !plus (s(z)) (s(s(s(z)))) (s(s(s(s(z))))),
   !plus z (s(s(s(z)))) (s(s(s(z)))),
   !go-plus z (s(s(s(z)))),
   !go-plus (s(z)) (s(s(s(z)))),
   !go-plus (s(s(z))) (s(s(s(z))))
```

The first fact listed in the saturated database tells us our answer: `5`.

In lecture, we saw how to apply the magic templates transformation to the predicate `append(+Xs, +Ys, -Zs)`, the top-down append predicate taking two input lists and returning their concatenation:

```
append(nil, Zs, Zs).

append(cons(X, Xs), Ys, cons(X, Zs)) :-
    append(Xs, Ys, Zs).
```

But remember from our Prolog days that we can interpret this same code with mode `(-, -, +)` and use it to enumerate all the ways of splitting a given list into two pieces.

Task 3 (5 pts). Convert the top-down version of `append(-Xs, -Ys, +Zs)` to a bottom-up logic program using the magic templates transformation. Put your code in a file called `append.lob`.

Task 4 (5 pts). Analyze the complexity of the transformed program: give an asymptotic upper bound on the size of the saturated database, annotate each rule with an asymptotic bound on its number of prefix firings, and use these to compute an asymptotic bound on the running time of the program as a whole.

2 Imperative Logic Programming (15 points)

2.1 Making Change

Linear logic programming can be used to naturally model many imperative state-changing systems like the "blocks world" example we saw in lecture.

Consider a state-transition system whose states are collections of facts of the form `penny`, `nickel`, `dime`, and `quarter` representing collections of coins. Two states are equivalent if their coins have the same monetary value.

Task 5 (5 pts). Write clauses representing transitions that allow us to replace any set of coins in our database with any other set of coins of equal value. Like the linear logic model of blocks world, your rules need not terminate in any sense—they should just make it possible to transition from any state to any equivalent one. Put your code in a file called `change.lob`.

Concrete syntax	Lollibot notation
z	`z`
s(e)	`s` e
$e_1 + e_2$	`plus` e_1 e_2
$e_1 \times e_2$	`times` e_1 e_2
s(n)	`s` n

Figure 2: A simple arithmetic language.

2.2 Evaluating Arithmetic Expressions

State-transition systems can be used to model algorithms in a way that is imminently parallelizable.

Consider the simple arithmetic language with unary natural numbers:

$$e ::= \mathsf{z} \mid \mathsf{s}(e) \mid e_1 + e_2 \mid e_1 \times e_2$$
$$n ::= \mathsf{z} \mid \mathsf{s}(n)$$

This language can be represented by Lollibot terms as shown in Figure 2.

We can represent parallel arithmetic evaluation by a linear state-transition system as follows:

- A state is a collection of facts `eval(`e_1`), ..., eval(`e_m`), result(`n`)`, where $e_1, \ldots, e_m$ are expressions to be evaluated and summed up, and n is the result accumulated so far,
- An initial state is one in which there is a single `eval(`e`)` fact and a single fact `result(z)` representing a starting point of zero, and
- A final state is one in which there are no more facts of the form `eval(`e_i`)` and there is a single fact `result(`n`)` representing the final accumulated result of evaluation.

The transitions of the system replace individual `eval(`e`)` facts with other facts in a way that moves ever towards a final state.

Task 6 (10 pts). Implement the arithmetic evaluation state-transition system in Lollibot. Put your code in a file called `eval.lob`.

Extra Credit Task 1 (3 extra credit pts). What changes are required to the specification of the state transition system if we want to support negation expressions, `neg(`e`)`? Extend your bottom-up evaluator to handle negations.

A Lollibot Primer

Lollibot is a bottom-up linear logic programming language that implements saturation for persistent facts and quiescence for ephemeral facts, the two termination strategies from lecture. Its syntax is similar to Twelf's, with a few important caveats:

- Rules are not named, and terms and predicates need not be declared
- Rules are written only in the left-to-right direction
- Instead of `->`, Lollibot uses the linear logic lollipop, rendered in ASCII as `-o`
- Rules may have multiple premises and multiple conclusions, using comma `,` to join facts and `()` to represent none at all
- Predicates are treated as ephemeral unless marked with a bang `!`

To run a set of rules, you use one of the following declarations:

```
%exec <steps> <initial facts>.
%trace <steps> <initial facts>.
```

where `<steps>` is the number of steps to run for and `<initial facts>` is the initial database of facts, separated by commas. To run until saturation or quiescence, use a `<steps>` value of `*`. The `%exec` declaration displays only the final database, while the `%trace` declaration additionally displays each step of execution.[1]

As a simple example illustrating the new syntax, recall the linear logic program from lecture that decomposes a list and then reassembles it into a permutation of a sublist, possibly dropping some elements.

```
% list Xs --> perm Ys, where Ys is a permutation of a sublist of Xs.

list (cons X Xs)  -o  elem X, list Xs.
list nil  -o  perm nil.
elem X  -o  ().
elem X, perm Xs  -o  perm (cons X Xs).

% test case: permute [1, 2, 3], showing all steps
%trace * list (cons 1 (cons 2 (cons 3 nil))).
```

Lollibot also has built-in primitive (non-saturating) predicates for testing equality and disequality of terms. To test whether `X` and `Y` are equal, use the predicate `X == Y`. To test whether `X` and `Y` are unequal, use the predicate `X <> Y`. For instance, the following code generates all unequal pairs of items:

```
!item A, !item B, A <> B  -o  !pair A B.
%exec * !item(1), !item(2), !item(3).
```

Many more Lollibot examples are available from the course's Software page.

[1]... but saturation is regarded as a single step.