Assignment 9:
Saturating Logic Programming

15-317: Constructive Logic

Out: Sunday, Nov 16, 2008
Due: Tuesday, November 25, 2008

1 Model Checking

In recitation, we discussed *CTL model checking*, an algorithm for checking properties of graphs.

The particular graphs we consider have the following properties:

- Each node is labeled with the atomic propositions true at that node
- Every node has at least one successor (which may be a self-loop)

We consider the following propositions:

1. Atomic propositions P
2. Truth $\top$
3. Conjunction $\phi_1 \wedge \phi_2$
4. Disjunction $\phi_1 \vee \phi_2$
5. Negation $\neg\phi$
6. Some next state $EX\phi$
7. All next states $AX\phi$
8. Some future state along some path $EF\phi$
9. Some future state along all paths $AF\phi$
10. All future states along some path $EG\phi$
11. All future states along all paths $AG\phi$

For a fixed graph G, these propositions are true of a state S in G (written $S \models \phi$) under the following conditions. We write $S \mapsto S'$ to mean there is an edge from S to S'.

1. $S \models P$ iff S is labeled with P

2. $S \models \top$ for all S
3. $S \models \phi_1 \wedge \phi_2$ iff $S \models \phi_1$ and $S \models \phi_2$ and
4. $S \models \phi_1 \vee \phi_2$ iff $S \models \phi_1$ or $S \models \phi_2$.
5. $S \models \neg\phi$ iff it is not the case that $S \models \phi$
6. $S \models EX\phi$ iff there exists an S' such $S \mapsto S'$ and $S' \models \phi$
7. $S \models AX\phi$ iff for all S' if $S \mapsto S'$ then $S' \models \phi$
8. $S_0 \models EF\phi$ iff there exists a path $S_0 \mapsto S_1 \mapsto S_2 \mapsto ...$ such that $S_i \models \phi$ for some $i \geq 0$.
9. $S_0 \models AF\phi$ iff for all paths $S_0 \mapsto S_1 \mapsto S_2 \mapsto ...$, $S_i \models \phi$ for some $i \geq 0$.
10. $S_0 \models EG\phi$ iff there exists a path $S_0 \mapsto S_1 \mapsto S_2 \mapsto ...$ such that $S_i \models \phi$ for all $i \geq 0$.
11. $S_0 \models AG\phi$ iff for all paths $S_0 \mapsto S_1 \mapsto S_2 \mapsto ...$, $S_i \models \phi$ for all $i \geq 0$.

In recitation, we wrote a saturating logic programming for many of these connectives.

Task 1 (10 pts). Starting from the support code, add cases for the remaining connectives: EF, AX, AF, AG. Hint: express AX, AF, AG in terms of EX, EF, EG and negation.

Note that you will need to run IRIS in well-founded mode for this problem:

```
iris program-file=<yourfile> well-founded
```

2 Edit Distance

The edit distance between an input string and an output string is a cost computed from the number of inserts, deletes, and modifications necessary to transform the input into the output. Edit distance is used in variety of applications: To suggest results for a misspelled search query, you might find a more popular query with a small edit distance from the one the user submitted. Also, many computational biology algorithms involve measuring the similarity between two proteins.

We use the following costs for edits: insert costs 100, delete costs 10, modify costs 1. For example, the possible edit distances between `'atc'` and `'agct'` include 101 (one insert, one modify), 102 (one insert, two modifies), and so on.

We will represent the input and output strings with predicates `input(n,c)` and `output(n,c)`, where `n` is a unary natural number (`zero()` or `succ(n)`) and `c` is one of `'a' 't' 'c' 'g'`.

Task 1 (10 pts). Your task is to define a predicate `dist(i,o,c)`, where `i` and `o` are natural numbers, and `c` is an integer. `dist(i,o,c)` should hold iff there is a series of edits that transform the segment of the input string from position `i` to its end into the segment of the output string from position `o` to its end, with cost `c`.

Hint: you will want to consider five cases:

1. `i` and `o` are at the end of the respective strings
2. The characters at `i` and `o` are the same

3. The characters at `i` and `o` are not the same (use `?ichar != ?ochar`)
4. The character at `i` is deleted
5. The character at `o` is inserted

The relation `ADD(c1,c2,c3)` means that `c3` is the sum of the integers `c1` and `c2`.

3 Alias Analysis

One important compiler analysis is *alias analysis*, which approximates the values of registers and memory cells. This allows you to answer questions like "do these two registers point to the same location?", which are useful both for optimization (you can move code around more if it doesn't use the same memory as other code) and for checking program invariants.

An alias analysis must *overapproximate* the values of registers and memory cells—because it is predicting the behavior of a program fragment without running it. To keep the problem simple, we will define an analysis that is *flow-insensitive* (the order of statements in the program is disregarded) and *context-insensitive* (all executions of a statement are identified). This makes the analysis less precise than it could be. There has been a great deal of research on more-precise analyses that are still effective and efficient (there are trade-offs here: *truth takes time*).

We use a very simple assembly language with four forms of statement:

- `l : x = n` Store the numeric constant `n` in register `x`.
- `l : x = < y , z >` Allocate a pair whose first component is the value of the register `y` and whose second component is the value of the register `z`, and store a pointer to this pair in the register `x`
- `l : x = y.f` Set the value of the register `x` to be the value of the `f` component of the pair pointed to by register `y`. Here `f` is either `'fst` or `'snd`.
- `l : x.f = y` Set the value of the `f` field of the pair pointed to by the register `x` to be the value of the register `y`

We want to define two relations:

- `valueofreg(r,v)` Register `r` may hold the value `v`
- `valueoffield(l,f,v)` The `f` field of the pair stored at location `l r` may hold the value `v`

Values `v` are either integers or `loc(l)` for some location `l`.

We identify memory locations with the *source line* at which the memory is allocated. So after a statement `l : x = < y , z >` , we will say that the register `x` holds the value `loc(l)`.

Task 1 (10 pts). Define a saturating logic programming that computes the above relations. See the support code for hints and test cases.

Task 2 (10 pts). Analyze the time complexity of your code in terms of the number n of statements in the program. Because your analysis should only use values that come from the program, you can regard the number of values as being $O(n)$.

- Give an upper bound on the size of the saturated database.
- Annotate each clause in your program with the number of prefix firings.
- State the overall time complexity of the program.

Please answer this task in comments in your IRIS file.

4 Handin

- You can run IRIS as follows:

  ```
  /afs/andrew/course/15/317/bin/iris program-file=<yourfile>
  ```

- To hand in your code, copy three files to your handin directory:
  Problem 1:

  ```
  /afs/andrew/course/15/317/submit/<yourid>/hw07-mc.iris
  ```

  Problem 2:

  ```
  /afs/andrew/course/15/317/submit/<yourid>/hw09-edit.iris
  ```

  Problem 3:

  ```
  /afs/andrew/course/15/317/submit/<yourid>/hw07-alias.iris
  ```