Constructive Logic (15-317), Spring 2020
Assignment 9:

Instructor: André Platzer
TAs: Avery Cowan, Cameron Wong, Carter Williams, Klaas Pruiksma

Due: Tuesday, April 14, 2020, 11:59 pm

This homework is out of 20 points.

1 Binary

Consider the following grammar of ground terms representing binary numbers:

$$n \quad ::= \quad \epsilon \mid \texttt{b0}(n) \mid \texttt{b1}(n)$$

In class, we learned to write forward logic programs using inference rules; a forward logic programming engine will apply these inference rules until saturation is reached, and then the result of our program can be read from the saturated proof state. In the tasks that follow, you are free to introduce any auxiliary predicates that you require. You need to ensure that your rules *saturate* when new facts of the indicated form are added to the database.

In the problems that follow, you are required to implement forward logic programs by writing down systems of inference rules. You may find it useful to experiment with **DLV**, an implementation of forward logic programming which can be downloaded here: `http://www.dlvsystem.com/dlv/`. **DLV** can be used to test your ideas on specific cases and quickly determine if they are likely to work; but it is not required.

Task 1 (5 pts). Implement a forward logic program $\texttt{std}(n)$ which derives the atom `no` iff it is not the case that n is in standard form. You may assume that n is ground (i.e. not subject to unification).

Task 2 (5 pts). Next, implement a forward logic program $\texttt{succ}(m, n)$ which derives `no` when it is not the case that $m + 1 = n$. For the purpose of this exercise, you may assume that m and n are ground. You may also assume that m and n are in standard form.

2 NFAs

2.1 Background

An important construct in the study of Chomsky's hierarchy of languages is the Deterministic Finite Automata (DFA), which defines the so-called *regular languages*. A DFA over a fixed *alphabet* Σ can be defined informally[1] via the following constructs:

- Q, a set of *states*.
- q_0, an *initial* state.

[1]"informal" in this context referring to the lack of inference rules for sets, functions, bools, etc.

- $\delta : Q \times \Sigma \to Q$, a *transition function* taking a state and a character and returning a new state.
- $F : Q \to$ `bool`, a function returning *true* on *accepting* (or *final*) states.

A given DFA Q, q_0, δ, F, then, *accepts* a string $S = s_1 s_2 \ldots s_n$ exactly when $F(\delta(\delta(\ldots\delta(q_0, s_1), \ldots), s_n))$ returns *true*.

Note that, if viewed as a state machine, a DFA is "in" a unique state at any given time. This is enforced by the transition function δ being a *function* from $Q \times \Sigma$ to Q. If we relax this condition, allowing the transition function to potentially return multiple values, we get a *nondeterministic* finite automata (NFA). An NFA, then, accepts if there is *any* path leading to an accepting state (more on this later).

2.2 Saturation

The nondeterminism inherent to NFAs make it a good fit for exploring the behavior of a forward logic program, in which we *also* act "nondeterministically" if multiple rules can be applied at once.

Consider the following grammar:

$$\begin{array}{llll} \text{Characters} & C & ::= & a \mid b \mid c \\ \text{Strings} & S & ::= & \epsilon \mid CS \\ \text{States} & Q & ::= & q_0 \mid q_1 \mid q_2 \mid q_3 \mid q_4 \mid q_5 \end{array}$$

For this problem, we will encode δ and F into the rules and grammar, rather than treating them as first-class constructions themselves, as it is simpler and requires less setup.

Task 3 (5 pts)**.** Given the following forward logic program, give all predicates of the form $accept(S)$ present in the "saturated" database obtained from running the program under the following constraint:

- Predicates of the form $acceptFrom(Q, S)$ are logged only when S has length at most 5.

Your answer should be ordered according to **depth-first** chaining. Rules are applied in order of lowest rule number first where applicable.

$$\frac{acceptFrom(q_0, S)}{accept(S)}\,\text{NFA}_0 \qquad \frac{final(Q)}{acceptFrom(Q, \epsilon)}\,\text{NFA}_1$$

$$\frac{acceptFrom(q_1, S)}{acceptFrom(q_0, aS)}\,\text{NFA}_2 \qquad \frac{acceptFrom(q_2, S)}{acceptFrom(q_1, bS)}\,\text{NFA}_3 \qquad \frac{acceptFrom(q_0, S)}{acceptFrom(q_2, S)}\,\text{NFA}_4$$

$$\frac{acceptFrom(q_3, S)}{acceptFrom(q_2, S)}\,\text{NFA}_5 \qquad \frac{acceptFrom(q_4, S)}{acceptFrom(q_3, aS)}\,\text{NFA}_6 \qquad \frac{acceptFrom(q_5, S)}{acceptFrom(q_4, cS)}\,\text{NFA}_7$$

$$\frac{acceptFrom(q_3, S)}{acceptFrom(q_5, S)}\,\text{NFA}_8 \qquad \frac{}{final(q_3)}\,\text{NFA}_9 \qquad \frac{}{final(q_0)}\,\text{NFA}_{10}$$

Hints:

- This NFA recognizes the language (ab)+$(ac)^*$.
- In this case, there is no "seed" fact. What does that mean for the behavior of this program?

Task 4 (5 pts)**.** In the first two tasks, we asked you to implement a forward logic program deriving `no` when the predicate is false. In the previous problem, however, the rules more closely resemble those that might be used in a more traditional backwards-chaining program. Explain the problem with using the previous rules as a forward-logic program that determines whether the NFA accepts an input string S.