Final Exam

15-317: Constructive Logic

December 9, 2008

Name: Andrew ID:

Instructions

- This exam is open-book.
- You have 3 hours to complete the exam.
- There are 8 problems.

	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Prob 7	Prob 8	Total
Score									
Max	60	40	45	45	40	40	20	10	300
Grader									

1 Quantifiers

Recall the proof terms for quantifiers:

$$\frac{\begin{array}{c}\overline{x{:}\tau}\\ \vdots\\ M : A\end{array}}{\lambda x.M : \forall x{:}\tau.A}\ \forall I^x \qquad \frac{M : \forall x{:}\tau.A \quad t : \tau}{Mt : A[t/x]}\ \forall E \qquad \frac{t : \tau \quad M : A[t/x]}{(t, M) : \exists x{:}\tau.A}\ \exists I \qquad \frac{M : \exists x{:}\tau.A \qquad \begin{array}{c}\overline{x{:}\tau}\ ,\ \overline{u{:}A}\\ \vdots\\ N : C\end{array}}{\mathsf{let}\ (x, u) = M\ \mathsf{in}\ N : C}\ \exists E^{x,u}$$

Task 1 (15 pts). Give a **natural deduction proof tree** for the following entailment. Note that x does not occur in B. Be sure to label each inference rule.

$$\frac{}{\forall x{:}\tau.(A(x) \supset B)}\ u$$

$$\overline{(\exists y{:}\tau.A(y)) \supset B}$$

Task 2 (15 pts). Give a **proof term** for the following entailment. Note that x does not occur in B.

$$f : (\exists x{:}\tau.A(x)) \supset B \vdash ??? : \forall y{:}\tau.(A(y) \supset B)$$

Task 3 (15 pts). Give a **natural deduction proof tree** for the following entailment. Note that x does not occur in B. Be sure to label each inference rule.

$$\frac{}{\exists x{:}\tau.(A(x) \vee B)}\ u$$

$$\frac{}{(\exists y{:}\tau.A(y)) \vee B}$$

Task 4 (15 pts). Give a **proof term** for the following entailment. Note that x does not occur in B.

$$s : (\exists x{:}\tau.A(x)) \wedge B \vdash ??? : \exists y{:}\tau.(A(y) \wedge B)$$

2 Induction

Sometimes we have written recursive functions using a pattern-matching *recursion schema*. For `nat`, the recursion schema is the following:

```
f z = M0
f (s n) = M1(n, f(n))
```

Using the schema, we can write

```
double : nat -> nat
double z = z
double (s n) = s (s (double n))
```

for the proof term

```
fn x:nat => natrec(x, z, n.r. s(s(r)))
```

where the variable `r` stands for the recursive call.

Consider the following intro and elim rules for lists of natural numbers, annotated with proof-terms

$$\frac{}{[] : \mathsf{list}} \qquad \frac{n : \mathsf{nat} \quad l : \mathsf{list}}{(n :: l) : \mathsf{list}} \qquad \frac{l : \mathsf{list} \quad M_1 : J([]) \qquad \overset{\overline{x : \mathsf{nat}} \;,\; \overline{xs : \mathsf{list}} \;,\; \overline{J(xs)}^{\,u}}{\vdots} \qquad M_2 : J(x :: xs)}{\mathsf{listrec}(l, M_1, x.xs.u.M_2) : J(l)}$$

Task 1 (15 pts). Give a primitive recursion schema for list induction, analogous to the above schema for `nat`:

Consider the following proof-term s which has type $\mathsf{list} \to \mathsf{nat} \to \mathsf{list}$:

$$\lambda x{:}\mathsf{list}.\mathsf{listrec}(x, \lambda a{:}\mathsf{nat}.[], x{:}\mathsf{nat}.xs{:}\mathsf{list}.r{:}\mathsf{nat} \to \mathsf{list}.\lambda a{:}\mathsf{nat}.(x + a) :: (r(x + a)))$$

Task 2 (10 pts). Translate this proof-term to a pattern-matching function definition using your recursion schema.

listrec has the following local reductions:

$$\begin{array}{rcl} \mathsf{listrec}([], M_1, x.xs.u.M_2) & \Rightarrow_R & M_1 \\ \mathsf{listrec}(n :: l, M_1, x.xs.u.M_2) & \Rightarrow_R & [n/x][l/xs][\mathsf{listrec}(l, M_1, x.xs.u.M_2)/u]M_2 \end{array}$$

Task 3 (15 pts). Compute the list that

$$s\ (1 :: (2 :: (3 :: [])))\ 0$$

reduces to (you need to show only the final result, not each intermediate step) and explain what this function does.

3 Prolog

In this problem, we will consider some Prolog code for matching a string S against a regular expression R. We consider the following regular expressions:

- The empty string `[]` matches the empty regular expression `epsilon`.
- The singleton string `[c]` matches the singleton regexp `single(c)`.
- The string `S` matches the regexp `concat(R1,R2)` (usually written $R_1 R_2$) if S splits as `S1` followed by `S2`, where `S1` matches `R1` and `S2` matches `S2`.
- The string `S` matches the regexp `star(R)` (usually written R^*) if $S = []$ or it splits as `S1` followed by `S2`, where `S1` matches `R` and `S2` matches `star(R)`.

Disjunctive regular expressions $(R_1 \mid R_2)$ can be explained similarly, but we elide them for brevity.

Consider the following Prolog code for regular expression matching:

```
1  append([],Ys,Ys).
2  append([X|Xs],Ys,[X|Zs]) :-
3          append(Xs,Ys,Zs).
4
5
6  match([],epsilon).
7
8  match([C],single(C)).
9
10 match(S,concat(R1,R2)) :-
11         append(S1,S2,S),
12         match(S1,R1),
13         match(S2,R2).
14
15 match([],star(R)).
16 match(S,star(R)) :-
17         append(S1,S2,S),
18         match(S1,R),
19         match(S2,star(R)).
```

Task 1 (10 pts). We will consider `match(S,R)` with mode `+ +`: both the string and the regular expression are inputs.

What mode does `append` need to have for `match` to have this mode?

Task 2 (20 pts). There are some ground S and R such that `match(S,R)` fails to terminate.

For each of the following termination metrics, identify a rule that violates that termination order and explain which subgoal violates it:

1. The regular expression R gets smaller.

   The subgoal on line _____ violates this termination order because

2. The string S gets smaller.

   The subgoal on line _____ violates this termination order because

3. The string S gets smaller, or it stays the same and the regexp R gets smaller.
   The subgoal on line _____ violates this termination order because

4. The regular expression R gets smaller, or it stays the same and the string S gets smaller.
   The subgoal on line _____ violates this termination order because

Task 3 (5 pts). Give a specific string `S` and regexp `R` for which `match(S,R)` fails to terminate. (Hint: your non-terminating input may be a query on which `match` should fail, but instead loops.)

Task 4 (10 pts). It is possible to make the above code terminate on well-moded calls by adding one extra subgoal to one rule. Viewed as inference rules, this revised code should define the same relation as the original code; but the revised code will terminate under Prolog's depth-first proof strategy.

Show the revised rule, and explain why it satisfies one of the above termination orders. Hint: you may use term equality $M = N$ or disequality $M \neq N$.

4 IRIS

In this problem, we will consider a different implementation of regular expression matching, using the saturating logic programming language IRIS.

We represent the characters in the string using relation `at` identifying the character at each position (recall the edit distance problem in Homework 9, and the CKY parsing example presented in lecture). For example, the string `atcg` is represented by

```
at(0,a).
at(1,t).
at(2,c).
at(3,g).
```

We will represent regular expression matching with a relation

`match(?s,?e,?r)`

meaning that the portion of the input string from `?s` (inclusive!) to `?e` (**exclusive**!) matches the regular expression `?r`. In the above example, we will have

```
match(0,1,single(a))
match(0,0,epsilon)
```

because the `?s` character is included in the match but `?e` character is not.

Task 1 (25 pts). Give IRIS rules for regular expression matching. Do not yet worry about saturation (see the next question).

```
// match(?s,?e,?r)

// EXAMPLE: rules for epsilon:
// There is an epsilon at the beginning of the string and after each character.
match(0,0,epsilon).
match(?s1,?s1,epsilon) :- at(?s,?c), ?s + 1 = ?s1.

// TODO rule for single(?c)

// TODO rule for concat(?r1,?r2)

// TODO three rules for star(?r)
```

Unless you already thought of this, your IRIS code will not saturate, because it will attempt to saturate the database with **all** regular expressions matching a string (and there may be infinitely many).

We need to restrict attention to those regular expressions that are subexpressions of the input (recall the model checking problem in Homework 9).

Task 2 (10 pts). Assume the database is seeded with the fact `subexp(r0)` for the input regular expression `r0` that we are interested in. Give rules for `subexp` so that `subexp(R)` holds iff `R` is a subexpression of the initial expression `r0`.

Task 3 (10 pts). Add `subexp` subgoals to your rules on the previous page so that the matching algorithm only considers subformulas of the input. The revised code should satisfy the following invariant: if `match(?s,?e,?r)` then `subexp(?r)`. Avoid redundant `subexp` subgoals—check it only where it is necessary.

5 Linear Logic

For each of the following linear logic entailments, circle **Derivable** or **Not Derivable** and:

- If the entailment is **Derivable**: give a sequent calculus derivation
- If the entailment is **Not Derivable**: explain why no derivation exists (i.e., attempt a derivation and explain why you must get stuck)

Clearly label each inference rule.

Task 1 (14 pts).

Derivable / **Not Derivable**

$$\overline{(A \otimes B) \multimap C \Vdash A \multimap (B \multimap C)}$$

Task 3 (13 pts).

Derivable / **Not Derivable**

$$\overline{(A \multimap B)\&(A \multimap C) \Vdash A \multimap (B\&C)}$$

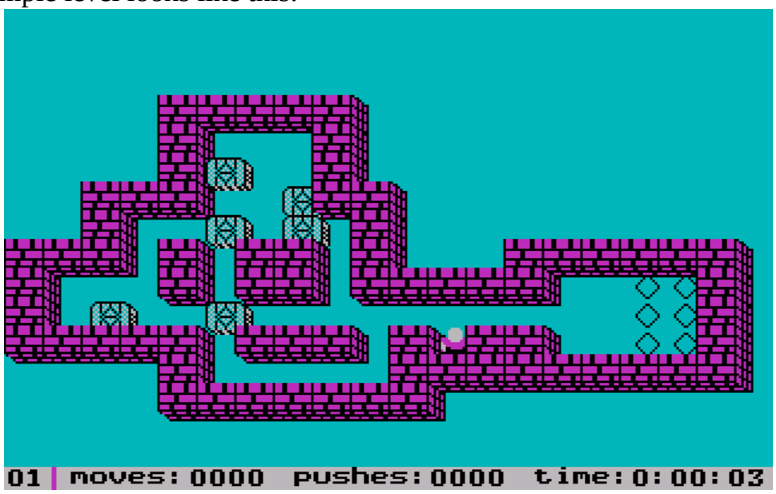
Task 3 (13 pts).

Derivable / **Not Derivable**

$$\overline{(A \multimap B) \otimes (A \multimap C) \Vdash A \multimap (B \otimes C)}$$

6 Linear logic programming

In the game Sokoban, a player must move a collection of boxes to certain specified goal positions. An example level looks like this:



The player can move in four directions (north, south, east, west) on a grid. Aside from the player's current location, each space on the grid is either unoccupied, or it contains a box, or it contains a wall. The player can freely move to any unoccupied space. The player can move boxes only by pushing them to an open square; he cannot pull them. (For example: A block with walls to the north and to the east cannot be moved.)

We will write a linear logic program to solve Sokoban puzzles.

We'll represent the board as a collection of cells with the following predicates:

- `!east(C1,C2)` The cell to the `east` of cell `C1` is the cell `C2`. Similarly for `!south(C1,C2)`. The board layout does not change over time.
- `player(C)` The player is at cell `C`.
- `box(C)` There is a box at cell `C`.
- `clear(C)` Cell `C` is clear.

Walls are not specified explicitly—they are just the cells that are neither player nor box nor clear.

In this problem, you will give rules for two actions; the remaining rules are analogous. You can write your answer using either the rule notation we used in lecture, with inference rules that consume their premises (except the `!` premises):

```
premise1,
!premise2,
...
-------------
conclusion1,
conclusion2,
...
```

or in linear logic notation, as you did in Homework 10.

Task 1 (15 pts). Give a rule for the following action:

Move East: If the player is at a cell, and the cell to the east is clear, then the player can move to that cell.

Task 2 (15 pts). Give a rule for the following action:

Push East: If the player is at a cell, and the cell to the east has a box, and the cell to the east of that is clear, then the player and the block can each move one cell to the east.

Task 3 (10 pts). Give a linear logic proposition whose proofs are solutions to the following Sokoban problem. You may assume that the layout of the board (`east`, `south`) is already in the database.

- Initial state: The player is at cell `C1`, a box is at cell `C2`, and the cells `C3`, `C4`, `C5` are clear.
- Final state: The box is at cell `C4`, and the player can be anywhere.

7 Lax Logic

Recall the proof terms for lax logic, which correspond to effectful computation:

$$\frac{E \div A\,\mathsf{lax}}{\{E\} : \{A\}\,\mathsf{true}}\ \{\}I \qquad \frac{M : A\,\mathsf{true}}{M \div A\,\mathsf{lax}} \qquad \frac{M : \{A\}\,\mathsf{true} \quad \begin{array}{c}\overline{x : A\,\mathsf{true}}\\ \vdots \\ E \div C\,\mathsf{lax}\end{array}}{(\mathsf{let}\,\{x\} = M\,\mathsf{in}\,E) \div C\,\mathsf{lax}}\ \{\}E$$

We'll add a term

$$\overline{\mathsf{flip} \div (\top \vee \top)\,\mathsf{lax}}$$

which non-deterministically returns true (represented as $\mathsf{inl}\langle\rangle$) with some probability and otherwise returns false (represented as $\mathsf{inr}\langle\rangle$).

Task 1 (20 pts). Write a proof-term

$$\mathsf{counttrues} : (\mathsf{nat} \supset \{\mathsf{nat}\})\,\mathsf{true}$$

where ($\mathsf{counttrues}\,n$) flips the coin n times and returns the number of heads.

```
counttrues zero =


counttrues (succ n) =
```

8 Classical Logic

In English usage, it is common to use a double-negative to express *weak affirmation*. For example, the statement

> "It's **not unlikely** that I will be at your party tonight."

is weaker than the statement

> "It's **likely** that I will be at your party tonight."

I.e., the first expresses a lower probability that I will come to the party than the second.

Task 1 (10 pts). Does this usage make more sense in constructive logic or in classical logic? Explain why.