Constructive Logic (15-317), Fall 2016
Assignment 9: Forward Logic Programming

Contact: Evan Cavallo (ecavallo@cs.cmu.edu)

Due Thursday, December 1, 2016, 1:30pm

This assignment is due at the beginning of class on the above date and must be submitted electronically via autolab. Submit your homework as a **tar** archive containing:

- your written solutions in `hw9.pdf`,
- your solution to Task 1 in `infer.pl`,
- your solution to Task 4 in `unify.pl`.

After submitting via autolab, please check the submission's contents to ensure it contains what you expect. No points can be given to a submission that isn't there.

In this homework, we'll be implementing forward logic programs. We'll be using a program, `forward-chain`, which accepts rule sets written in Prolog syntax and outputs the result of running these rules to saturation. You can run this program on AFS as

```
/afs/andrew/course/15/317/bin/forward-chain solution.pl
```

Or build it for yourself from source. You will need `ml-lex` and `ml-yacc` to compile. Download the linked package, unpack and compile it by running:

```
$ tar -xzf forward-chain.tar.gz
$ cd forward-chain
$ make
```

This creates an SML image called `forward-image.(...)` (the extension of the file depends on your computer's system). You can then run this program with:

```
sml @SMLload forward-image.(...) program.pl
```

Replace the three dots and parentheses with the appropriate extension.

Note that you can also run all the commands with a flag -v that makes it print the partial databases computed at each step.

1 Type Inference

In class we saw typing judgments of the form $\Gamma \vdash e : \tau$, but only for a few specific types like `nat`. In this section, we will consider typing judgments for a more comprehensive language consisting of booleans, pairs and functions.

$$\begin{array}{rcl} e & ::= & \texttt{var}(x) \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}(e;e;e) \mid \langle e, e\rangle \mid \pi_L(e) \mid \pi_R(e) \mid \lambda x{:}\tau.e \mid ee \\ \tau & ::= & \texttt{bool} \mid \tau \times \tau \mid \tau \to \tau \end{array}$$

Typing is defined by a judgment $\Gamma \vdash e : \tau$, which asserts that expression e has type τ in a context Γ of assumptions about the types that the variables in e have. Of course, `true` is supposed to have type `bool` and the type of the pair $\langle e_1, e_2\rangle$ is supposed to be the product of the types of e_1 and e_2. Likewise, the left projection $\pi_L(e)$ of an expression e of product type $\sigma \times \tau$ has type σ. This is achieved by defining the typing judgment for the above language with the following typing rules:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash \texttt{if}(e; e_t; e_f) : \tau}$$

$$\frac{\Gamma \vdash e_L : \tau_L \quad \Gamma \vdash e_R : \tau_R}{\Gamma \vdash \langle e_L, e_R\rangle : \tau_L \times \tau_R} \qquad \frac{\Gamma \vdash e : \tau_L \times \tau_R}{\Gamma \vdash \pi_L(e) : \tau_L} \qquad \frac{\Gamma \vdash e : \tau_L \times \tau_R}{\Gamma \vdash \pi_R(e) : \tau_R}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

As hinted above, we can read these rules programmatically as taking Γ and e as inputs and producing a type τ as an output. (This is a partial function, since an expression like $\pi_L(\texttt{true})$ has no type.) The process of finding a type for a given expression is called *type inference*.

Task 1 (12 pts)**.** Write a forward-chaining program in `infer.pl` which computes the type of an expression e in a given context Γ. Assume that the input is specified by seeding the database with the following facts:

- some number of predicates $\texttt{hastype}(\texttt{var}(x), \tau)$ specifying the types of all variables in Γ, and

- a predicate `infer`(e) specifying the input expression.

When the program has run to saturation, the database should contain a predicate `hastype`(e, τ) giving the inferred type of the input expression if and only if one exists. There should be at most one type τ for which `hastype`(e, τ) holds (unique output). You can assume that distinct bound variables have distinct names and are moreover distinct from any free variables which appear (for example, the expression $\langle \lambda x$:`bool`.`true`, λx:`bool`.`false`$\rangle$ is not allowed because x is bound in two places). See the `infer.pl` file included in the handout for the concrete syntax of the operators.

Task 2 (4 pts)**.** Explain why the assumption of distinct variable names is necessary for this formulation of the algorithm.

Task 3 (4 pts)**.** Discuss the advantages and disadvantages of this implementation of type-inference as compared with a backward-chaining implementation.

2 Unification

In this section, we will implement an algorithm which tests if unification is possible using forward logic programming. We will do this for the grammar of types defined in the previous section, adding type metavariables ($\alpha, \beta, \ldots$).

$$\tau ::= \alpha \mid \texttt{bool} \mid \tau \times \tau \mid \tau \to \tau$$

Our algorithm will work by checking if unification is *impossible*. The program will take in a database of required equational constraints as input. After running to saturation, the database will include the fact `contra` ("contradictory constraints") if there is no unifier that will satisfy the equations.

One advantage of phrasing the problem this way is that we can maintain a database of equations and add new constraints dynamically, with the system alerting us if the set of equations ever becomes contradictory. You can interactively add rules in `forward-chain` using the `-i` option.

Task 4 (12 pts)**.** Implement a forward-chaining program in `unify.pl` which, when seeded with a set of facts $\{\texttt{unify}(\tau_0, \rho_0), \ldots, \texttt{unify}(\tau_n, \rho_n)\}$, generates the fact `contra` if and only if it there is no substitution which simultaneously unifies each equation $d_i \doteq e_i$. You do not have to implement the occurs check (see Task 6). (Hint: if d and e are unifiable, what else must be unifiable? Generate all relevant facts of this form and conclude `contra` if you deduce anything obviously contradictory. For example, you can deduce $\tau_1 \doteq \rho_1$ from $\tau_1 \times \tau_2 \doteq \rho_1 \times \rho_2$.)

Task 5 (4 pts)**.** Use McAllester's theorem to characterize the asymptotic complexity of your solution.

Task 6 (4 pts)**.** Recall that the occurs check, which is not present in Prolog, disallows the unification of α with τ when the metavariable α itself occurs in τ (but $\tau \neq \alpha$). For example, this prevents unifying α with $\alpha \to \texttt{bool}$. (Prolog will instead conclude that α is the "infinite type" $((\cdots \to \texttt{bool}) \to \texttt{bool}) \to \texttt{bool}$). Explain how you could add the occurs check to your solution. In other words, describe how you can modify your program so that `contra` is generated whenever the inputs imply $\alpha \doteq \tau$ with $\alpha \in \tau$ and $\alpha \neq \tau$.