Monad-Based Programming SS 2023

Assignment 2

Deadline for solutions: 18.05.2023

Exercise 1 Programming a Proof Assistant (11 Points)

The formal system of *natural deduction* for *first-order logic* operates with the judgements of the form

$$\Sigma \mid \Gamma \vdash \phi$$

which state that a (first-order) formula ϕ whose free variables are in Σ, is derivable from the list of formulas Γ, whose variables are again in Σ. Derivability of such a judgment is interpreted as the fact that ϕ provably follows from Γ.

For example, given that *all humans are mortal*, and *Socrates is a human*, we can conclude that *Socrates is mortal*, as a derivable judgement as follows:

$$\dfrac{\dfrac{\dfrac{}{- \mid \forall x.\, \mathsf{h}(x) \Rightarrow \mathsf{m}(x), \mathsf{h}(\mathsf{S}) \vdash \forall x.\, \mathsf{h}(x) \Rightarrow \mathsf{m}(x)}\,(hyp)}{- \mid \forall x.\, \mathsf{h}(x) \Rightarrow \mathsf{m}(x), \mathsf{h}(\mathsf{S}) \vdash \mathsf{h}(\mathsf{S}) \Rightarrow \mathsf{m}(\mathsf{S})}\,(\forall E) \qquad \dfrac{}{- \mid \forall x.\, \mathsf{h}(x) \Rightarrow \mathsf{m}(x), \mathsf{h}(\mathsf{S}) \vdash \mathsf{h}(\mathsf{S})}\,(hyp)}{- \mid \forall x.\, \mathsf{h}(x) \Rightarrow \mathsf{m}(x), \mathsf{h}(\mathsf{S}) \vdash \mathsf{m}(\mathsf{S})}\,(\Rightarrow E)$$

Here, we indicated by "$-$" the fact that the list of free variables is empty. Recall that first order formulas are given by the grammar

$$\phi, \psi ::= p(t_1, \ldots, t_n) \mid \top \mid \bot \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi \mid \forall x.\, \phi \mid \exists x.\, \phi$$

where p is an n-ary symbol from a fixed *signature* of predicate symbols $\mathcal{P}$, and $t_1, \ldots, t_n$ are *first-order terms*, which are in turn specified by the grammar:

$$t_1, \ldots, t_n ::= x \mid f(t_1, \ldots, t_n)$$

where x ranges over a fixed infinite set $\mathcal{X}$ of variables and f over a fixed *signature* of function symbols $\mathcal{F}$ of a the corresponding arity (we do not include negation, which can be modelled as $\phi \Rightarrow \bot$). In the above example:

- $\mathcal{P} = \{\mathsf{m}/1, \mathsf{h}/1\}$, $\mathcal{F} = \{\mathsf{S}/0\}$, and $\mathcal{X}$ is, say, the set of all strings of ASCII-characters (by $/n$ we indicate the arity of the corresponding symbol, also, we omit parenthesis at 0-ary symbols, e.g. in S, instead of $\mathsf{S}()$).

First-order terms can be easily implemented in Haskell, e.g. with

```
data FOTerm = FOVar String | FOFun String [FOTerm]
```

(a) In a similar way, implement an inductive data type of first-order formulas. Implement functions for computing a list of free variables of a given formula, and a capture-avoiding substitution for terms and formulas, i.e. the operation that replaces given free occurrences of x in t with s, denoted $t[s/x]$ (you can adapt the definition from λ-calculus, where the binder λ behaves like the present binders $\forall$ and $\exists$).

Derivable judgements are built with the following rules:

$$\frac{\phi \ \text{ in } \ \Gamma}{\Sigma \mid \Gamma \vdash \phi}\ (\text{hyp})$$

$$\frac{\Sigma \mid \Gamma \vdash \phi \qquad \Sigma \mid \Gamma \vdash \psi}{\Sigma \mid \Gamma \vdash \phi \wedge \psi}\ (\wedge I) \qquad \frac{\Sigma \mid \Gamma \vdash \phi \wedge \psi}{\Sigma \mid \Gamma \vdash \phi}\ (\wedge E_1) \qquad \frac{\Sigma \mid \Gamma \vdash \phi \wedge \psi}{\Sigma \mid \Gamma \vdash \psi}\ (\wedge E_2)$$

$$\frac{\Sigma \mid \Gamma \vdash \phi}{\Sigma \mid \Gamma \vdash \phi \vee \psi}\ (\vee I_1) \qquad \frac{\Sigma \mid \Gamma \vdash \psi}{\Sigma \mid \Gamma \vdash \phi \vee \psi}\ (\vee I_2)$$

$$\frac{\Sigma \mid \Gamma \vdash \phi \vee \psi \qquad \Sigma \mid \Gamma, \phi \vdash \xi \qquad \Sigma \mid \Gamma, \psi \vdash \xi}{\Sigma \mid \Gamma \vdash \xi}\ (\vee E)$$

$$\frac{\Sigma \mid \Gamma, \phi \vdash \psi}{\Sigma \mid \Gamma \vdash \phi \Rightarrow \psi}\ (\Rightarrow I) \qquad \frac{\Sigma \mid \Gamma \vdash \phi \Rightarrow \psi \qquad \Sigma \mid \Gamma \vdash \phi}{\Sigma \mid \Gamma \vdash \psi}\ (\Rightarrow E)$$

$$\frac{\Sigma, x \mid \Gamma \vdash \phi}{\Sigma \mid \Gamma \vdash \forall x.\, \phi}\ (\forall I) \qquad \frac{\Sigma \mid \Gamma \vdash \forall x.\, \phi \qquad \Sigma \vdash t \ \ \mathit{term}}{\Sigma \mid \Gamma \vdash \phi[t/x]}\ (\forall E)$$

$$\frac{\Sigma \mid \Gamma \vdash \phi[t/x] \qquad \Sigma \vdash t \ \ \mathit{term}}{\Sigma \mid \Gamma \vdash \exists x.\, \phi}\ (\exists I) \qquad \frac{\Sigma \mid \Gamma \vdash \exists x.\, \phi \qquad \Sigma, x \mid \Gamma, \phi \vdash \psi}{\Sigma \mid \Gamma \vdash \psi}\ (\exists E)$$

$$\frac{\Sigma \mid \Gamma \vdash \bot}{\Sigma \mid \Gamma \vdash \phi}\ (\bot E) \qquad \frac{}{\Sigma \mid \Gamma \vdash \top}\ (\top I)$$

$$\frac{\Sigma \mid \Gamma \vdash (\phi \Rightarrow \bot) \Rightarrow \bot}{\Sigma \mid \Gamma \vdash \phi}\ (\neg\neg E)$$

Here, the auxiliary judgement $\Sigma \vdash t \ \ \mathit{term}$ states that t is a first-order term, whose variables are in Σ. Note also that Γ is treated as a proper list, which corresponds to an inverted Haskell-list, i.e. Γ, ϕ corresponds to $(\phi : \Gamma)$ in Haskell, and similarly for Σ.

The rules are designed so as to indicate how proofs of the judgements in the conclusions are obtained from the proofs of the judgements in the premises. Consider, for example, $(\vee E)$. This says: to obtain a proof of a proposition ξ (from the premises Γ), take a proof of the disjunction $\phi \vee \psi$; if this proof is actually a proof of ϕ, combine it with the premise, stating that ξ follows from ϕ; if this proof is actually a proof of ψ, combine it with the premise, stating that ξ follows from ψ.

(b) Program a proof assistant in Haskell, in the form of an interactive shell, implementing the above rules of natural deduction.

The interaction must be organised as follows: the user is asked to provide the name of a rule and possibly further input, such as a term in the $(\exists I)$ rule. The rule is either applied, if possible, or an error message is displayed, with a prompt to retry.

Hints: Essentially, you need to implement a type of proof states ProofState, which contains the information about the current state of the proof, and a function

```
proofStep :: (String, ProofState) -> (String, ProofState)
```

which takes a string from the user, the current proof state, and returns an updated proof state, and a string message for the user. It then suffices to compose proofStep with the following function, which provides interaction with IO:

```
statefulInteract  :: ((String, a) -> (String, a)) -> (String, a) -> IO ()
statefulInteract  f (s , x) = putStr s >> getLine >>= (return . \s -> f (s , x)) >>= statefulInteract f
```

(further details of interacting with IO, we defer to later). Try

```
statefulInteract  siTest  ("[0] Hi, enter a string: ", 0)
```

with

```
siTest :: (String, Int) -> (String, Int)
siTest (s , n) = ("[" ++ show n ++ "] your input is: " ++ show s ++ "\n["
                   ++ show (n + 1) ++ "] enter enother string: ", n + 1)
```

to grasp the idea.

The proof state must contain a stack of judgements to derive: initially it contains one judgement – our grand goal; if the stack is empty, the proof is complete; in the remaining cases ProofState pops a judgement from the stack, tries to match it with the conclusion of the rule, suggested by the user, and pushes the premises of the rule to the stack.

(c) Attest your implementation by proving (again, $\neg\phi$ encodes $\phi \Rightarrow \bot$):

- the above example about Socrates;
- the following *de Morgan law* for quantifiers: $- \mid - \vdash \neg(\exists x.\, P(x)) \Rightarrow \forall x.\, \neg P(x)$;
- the *excluded middle law*: $- \mid - \vdash P() \vee \neg P()$.

More concretely, you are requested to submit the log of your successful proof dialogue. The last item is tricky: make sure to use $(\neg\neg E)$ in the beginning, and try to obtain

$$- \mid \neg(P() \vee \neg P()) \vdash P() \vee \neg P()$$

as an intermediate goal.

A potential example interaction scenario is as follows:

```
> We are proving "forall x. h(x) => m(x), h(S) |- m(S)". Which rule to apply?
> just do it, I cannot wait!
> No such rule, try again.
> allE
> Enter a term.
> FOFun "S" []
> Rule mismatch, try again.
> =>E
> Enter a formula.
> FOPred "h" [FOFun "S" []]
> Success! We are proving "forall x. h(x) => m(x), h(S) |- h(S)". Which rule to apply?
> hyp
> Success! We are proving "forall x. h(x) => m(x), h(S) |- h(S) => m(S)". Which rule to apply?
> allE
> Enter a term
> FOFun "S" []
> Success! We are proving "forall x. h(x) => m(x), h(S) |- forall x. h(x) => m(x)". Which rule to apply?
> hyp
> Success! We are done.
```

Exercise 2 Small-step v.s. Big-step (9 Points)

Consider the following rules for the small-step and big-step call-by-value operational semantics of untyped λ-calculus:

Small-step semantics:

$$\frac{p \to_{\mathsf{cbv}} p'}{pq \to_{\mathsf{cbv}} p'q}\ \textbf{(l-red)} \qquad \frac{q \to_{\mathsf{cbv}} q' \qquad p \text{ is a value}}{pq \to_{\mathsf{cbv}} pq'}\ \textbf{(r-red)} \qquad \frac{q \text{ is a value}}{(\lambda x.\, p)q \to_{\mathsf{cbv}} p[q/x]}\ (\boldsymbol{\beta})$$

Big-step semantics:

$$\frac{}{\lambda x.\, p \Downarrow_{\mathsf{cbv}} \lambda x.\, p}\ \textbf{(value)} \qquad \frac{p \Downarrow_{\mathsf{cbv}} \lambda x.\, p' \qquad q \Downarrow_{\mathsf{cbv}} q' \qquad p'[q'/x] \Downarrow_{\mathsf{cbv}} v}{pq \Downarrow_{\mathsf{cbv}} v}\ \textbf{(app)}$$

Recall that a λ-term is a *value* if and only if it has the form $\lambda x.\, t$. A *normal form* of t w.r.t. the small-step semantics is such a value v that $t \to^{\star}_{\mathsf{cbv}} v$. A *normal form* of t w.r.t. the big-step semantics is such a value v that $t \Downarrow_{\mathsf{cbv}} v$.

(a) In both styles of semantics, calculate normal forms of the term

$$(\lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\ f\ (n\ f\ x))(\lambda f.\, \lambda x.\, f(fx))(\lambda f.\, \lambda x.\, f(fx)),$$

meaning: produce the corresponding complete derivations. How can you interpret the result?

Hint: It can be useful to introduce abbreviations for *combinators* (i.e. terms without free variables), e.g. p for $\lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\ f\ (n\ f\ x)$ and t for $\lambda f.\, \lambda x.\, f(fx)$.

(b) Prove that our semantics is deterministic, i.e. whenever both $p \Downarrow_{\mathsf{cbv}} q$ and $p \Downarrow_{\mathsf{cbv}} r$, q must be equal to r.

(c) Prove that for any closed λ-term p, $p \to^{\star}_{\mathsf{cbv}} q$ with q being a value iff $p \Downarrow_{\mathsf{cbv}} q$.

Hint: For one direction of the equivalence prove the following auxiliary lemma: $p \to_{\mathsf{cbv}} q \wedge q \Downarrow_{\mathsf{cbv}} c \Rightarrow p \Downarrow_{\mathsf{cbv}} c$.

For (b) and (c), use (without a proof) the following

Well-founded Tree Induction Principle: given a set of rules S and a predicate P with the following properties:

(i) $P(t)$ for any axiom

$$\frac{}{t}$$

from S.

(ii) whenever $P(t_1), \ldots, P(t_n)$ and the rule

$$\frac{t_1 \quad \ldots \quad t_n}{t}$$

belongs to S then $P(t)$.

Then $P(t)$ for any t that can be derived using S.