ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

D. Basin and P. Müller

Formal Methods and Functional Programming

Exercise Sheet 2: Natural Deduction and Recursion

Submission deadline: March 4th, 2012

Assignment 1:

In this exercise, we work with intuitionistic predicate logic. The corresponding natural deduction rules are listed below.

$$\frac{}{\Gamma, A \vdash A}\ \textit{axiom} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to I \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \to E$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot E \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg I \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash B} \neg E$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge I \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge EL \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge ER$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee IL \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee IR \qquad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C} \vee E$$

$$\frac{\Gamma \vdash P(x)}{\Gamma \vdash \forall x.\, P(x)} \forall I^{*} \qquad \frac{\Gamma \vdash \forall x.\, P(x)}{\Gamma \vdash P(t)} \forall E$$

$$\frac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x.\, A(x)} \exists I \qquad \frac{\Gamma \vdash \exists x.\, A(x) \qquad \Gamma, A(x) \vdash B}{\Gamma \vdash B} \exists E^{**}$$

Side conditions: (*) x not free in Γ and (**) x not free in Γ or B.

Recall that $\to$ is right-associative, while $\wedge$ and $\vee$ are left-associative. Moreover, $\neg$ binds stronger than $\wedge$, which binds stronger than $\vee$, which in turn binds stronger than $\to$. Also note that the scope of the quantifiers extends as far to the right as possible.

Prove that the following statements are valid in intuitionistic predicate logic.

(a) $((\exists x.\, P(x)) \to Q) \to \forall x.\, P(x) \to Q$, where x does not occur free in Q.

(b) $(\exists x.\, P(x) \wedge Q(x)) \to (\exists x.\, P(x)) \wedge (\exists y.\, Q(y))$

(c) $(\forall x.\, P(x) \to Q(x)) \to \forall x.\, \neg Q(x) \to \neg P(x)$

Assignment 2:

For each of the following formulas, find two structures with universe $\{a, b, c\}$ and nonempty relations. One that satisfies the formula and another one that does not satisfy it.

(a) $(\exists x.\, P(x)) \wedge (\exists y.\, Q(y)) \rightarrow (\exists x.\, P(x) \wedge Q(x))$

(b) $\forall x.\, (\exists y.\, R(x, y) \wedge Q(y)) \rightarrow (\forall y.\, R(x, y) \rightarrow Q(y))$

(c) $\forall x, y.\, R(x, y) \rightarrow R(y, x) \rightarrow x = y$

Hint: all the structures $\mathcal{A}_i$ are of the form $\mathcal{A}_i = (\mathcal{U}_{\mathcal{A}_i}, I_{\mathcal{A}_i})$ for $\mathcal{U}_{\mathcal{A}_i} = \{a, b, c\}$ and $I_{\mathcal{A}_i}(S_l) = \{\ldots\}$ for all relations S_l occurring in the formula.

Assignment 3:

In this assignment you will develop a Haskell program based on Newton's method for calculating the square root of a nonnegative `Float`. Since the type `Float` is of limited precision, your square root function `mySqrt :: Float -> Float` will compute the square root up to some suitable small error `eps`.

(a) Write a function `improve :: Float -> Float -> Float` that improves your approximation. The first argument is the number from which you want to calculate the square root and the second argument is the approximation you have calculated so far. Newton's method says that if y_n is an approximation of $\sqrt{x}$ then

$$y_{n+1} = \frac{y_n + x/y_n}{2}$$

is a better approximation.

(b) Write a function `goodEnough :: Float -> Float -> Bool` that checks whether your approximation is in the error bound `eps :: Float`. More precisely, the approximation y_n is good enough if

$$\left|y_n^2 - x\right| < \texttt{eps}$$

You can chose, e.g., `eps` as 0.001.

(c) Use the functions `improve` and `goodEnough` for writing the function `mySqrt`. As the first approximation y_0 you can use, e.g., $y_0 = 1$.

Assignment 4

Write a Haskell function `cntChange :: Int -> Int` that computes the number of ways to change any given amount of money by using CHF coins.
Hint: Think recursively. The number of ways to change amount a using n different kinds of coins is equal to the sum of

- the number of ways to change a using all but the first kind of coin, and
- the number of ways to change amount $a - d$ using the n kinds of coins, where d is the denomination of the first kind of coin.

Test your program thoroughly!