Monad-Based Programming SS 2023

Assignment 4

Deadline for solutions: 22.06.2023

Exercise 1 Soundness of (Lazy) PCF (8 Points)

Recall that the soundness property states that for every closed program $p$ and every value $v$ of the same type, $p \downarrow v$ entails $\llbracket p \rrbracket = \llbracket v \rrbracket$. Using the equivalence of the big-step and the small-step semantics, this property follows from the other one, which is also called soundness of the individual rules: for every operational semantics rule

$$
\frac{p_1 \rightarrow q_1 \quad \ldots \quad p_n \rightarrow q_n}{p \rightarrow q}
$$

if $\llbracket p_1 \rrbracket = \llbracket q_1 \rrbracket, \ldots, \llbracket p_n \rrbracket = \llbracket q_n \rrbracket$ then $\llbracket p \rrbracket = \llbracket q \rrbracket$.

(a) Why?

(b) Prove soundness of the following rules

$$
\frac{p \rightarrow p'}{pq \rightarrow p'q} \quad \frac{Yp \rightarrow p(Yp)}{(\lambda x.p)q \rightarrow q[p/x]}
$$

using Substitution Lemma from the script: Given $\Gamma \vdash q : A$, $\Gamma, x : A \vdash p : B$ and $\rho \in \llbracket \Gamma \rrbracket$

$$
\llbracket \Gamma \vdash p[q/x] : B \rrbracket_\rho = \llbracket \Gamma, x : A \vdash p : B \rrbracket(\rho, \llbracket \Gamma \vdash q : A \rrbracket_\rho)
$$

(c) Prove that $\llbracket \Gamma \vdash f : A \rightarrow B \rrbracket = \llbracket \Gamma \vdash \lambda x. f x : A \rightarrow B \rrbracket$.

Exercise 2 Iteration from Recursion (7 Points)

Consider the following term formation rule for while:

$$
\frac{\Gamma \vdash p : S \quad \Gamma, x : S \vdash b : Bool \quad \Gamma, x : S \vdash q : S}{\Gamma \vdash \text{let } x = p \text{ while } b \text{ do } q : S}
$$

Here,

- $S$ is regarded as a type of states,
- $p$ is a program that initializes the state,
- $b$ is a test, depending on the current state, and
- $q$ is a loop body, which transforms a given state to a new one.
(a) Express \((\text{let } x \leftarrow p \text{ while } b \text{ do } q)\) in PCF as a term over \(p, b\) and \(q\) in such a way that
\[
[\Gamma \vdash \text{let } x \leftarrow p \text{ while } b \text{ do } q : S] = [\Gamma \vdash \text{if } b[p/x] \text{ then } (\text{let } x \leftarrow q[p/x] \text{ while } b \text{ do } q) \text{ else } p : S].
\] (*)

**Hint:** Some exploration will be needed. You should think of that, how substitution
\[
(\text{let } x \leftarrow p \text{ while } b \text{ do } q)[r/y] \]
must be computed. Then note that if \(x \neq y\) and \(y\) is not free either in \(b\) or in \(q\),
\[
[\Gamma \vdash \text{let } x \leftarrow y \text{ while } b \text{ do } q : S] = [\Gamma \vdash \text{if } b[y/x] \text{ then } (\lambda y. \text{let } x \leftarrow y \text{ while } b \text{ do } q)(q[y/x]) \text{ else } y : S].
\]
This gives an idea how to recursively express \((\text{let } x \leftarrow y \text{ while } b \text{ do } q)\) through itself, and thus, how to express the general case using the \(Y\)-combinator.

(b) Prove (*)

(c) The simplest implementation of Fibonacci numbers, corresponding to the following Haskell defletion
\[
\text{fib 0} = 1 \\
\text{fib 1} = 1 \\
\text{fib } n = \text{fib } (n - 1) + \text{fib } (n - 2)
\]
can be extremely inefficient, for every step recursively calls \text{fib} twice, so the number of recursive calls grows exponentially. Because of that, it makes sense to use the following variant of \text{fib}, which simultaneously calculates the \(n\)-th and the \((n + 1)\)-th Fibonacci number:
\[
\text{fib2 0} = (1, 1) \\
\text{fib2 } n = \text{let } (\text{fn}_1, \text{fn}) = \text{fib2 } (n - 1) \text{ in } (\text{fn}, \text{fn}_1 + \text{fn})
\]
It can be shown that \text{fib} is equivalent to \(\text{fst } \$ \text{fib2}\).
This example demonstrates that inefficiency of unconstrained recursion cannot generally be prevented, which is sometimes considered as a drawback of functional programming. The reason why in the second program recursion is harmless is because it is essentially an iterative program: the state is a pair of numbers, which are updated in a while-loop, as long as the counter \(n\) is non-zero. Formalize this by reformulating \text{fib2} using the above while-operator.

(d) Use the encoding of while in PCF from (a), and run the corresponding program in your interpreter from Exercise 2, Assignment 3, to compute \text{fib2} 3.

**Exercise 3**  **Dinaturality**  

(5 Points)

The least fixpoint operator \(\mu\), figuring in the Kleene fixpoint theorem, features many properties. One such property is the so-called **dinaturality law**: given two continuous functions \(f : A \rightarrow B\) and \(g : B \rightarrow A\) between domains,
\[
f(\mu(gf)) = \mu(fg).
\]
Prove it. **Hint:** prove mutual inequality. You need not use any knowledge, except the very definition of \(\mu\) – that it defines a fixpoint, and this fixpoint is the least one.