Monad-Based Programming WS 2021

Assignment 1

Deadline for solutions: 10.11.2021

Exercise 1 Small-step v.s. Big-step (6 Points)

Consider the following rules for the small-step and big-step call-by-value 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 $t \to_{\mathsf{cbv}}^{\star} v$. 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.

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 for any closed λ-term p, $p \to_{\mathsf{cbv}}^{\star} q$ with q being a value iff $p \Downarrow_{\mathsf{cbv}} q$. To this end, 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 rule from S of the form

$$\frac{}{t}$$

(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.

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

Exercise 2 Lazy Lists (7 Points)

(a) Complete the untyped λ-calculus with constructors for lists (i.e. a zero-ary constructor *nil* for forming the empty list and a binary constructor *cons* for forming a list from a head and a tail) and with the *head* and *tail* destructors.

(b) Design a call-by-name (lazy) small-step and big-step semantics for the obtained extension in such a way that the observable behaviour of terms is analogous to the corresponding behaviour of Haskell programs.

(c) Recall the Haskell program for generating Fibonacci numbers

```
fib = 1 : 1 : [a + b | (a, b) <- zip fib (tail fib) ]
```

from the lecture. How can this program be implemented in the lazy untyped λ-calculus with lists? Justify your answer.

Hint: Recall that natural numbers can be modelled with Church numerals.

Exercise 3 Getting Real (7 Points)

Consider a notion of number which includes all natural numbers and supports the operations of summation and multiplication. Let us denote by $\mathcal{S}$ the set of such numbers. We can extend $\mathcal{S}$ to the numbers of the form

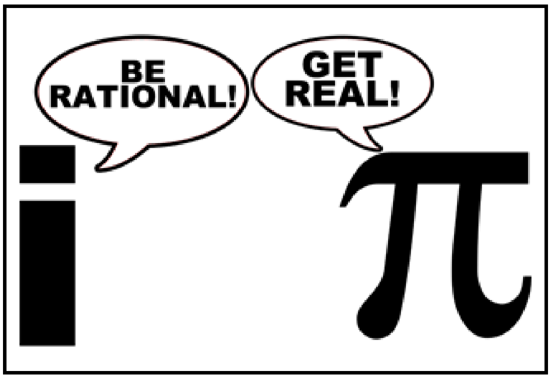
$$a + \sqrt{2} \cdot b \tag{$*$}$$

with $a, b \in \mathcal{S}$ and denote the extended numbers as $\mathcal{S}[\sqrt{2}]$. Note that depending on $\mathcal{S}$, $\mathcal{S}$ may be equi-expressive with $\mathcal{S}[\sqrt{2}]$ (e.g. if $\mathcal{S}$ are all real numbers) or properly less expressive (e.g. if $\mathcal{S}$ are all rational numbers).

Implement the numbers ($*$) in Haskell as an algebraic data type

`Sq2Num a`

where `a` is the type capturing the elements of $\mathcal{S}$. Ensure that `Sq2Num a` (under suitable assumptions) is an instance of the following type classes: `Eq`, `Ord`, `Show`, `Num`, `Fractional`, e.g. by completing the following declarations:




```
instance (Num a, Eq a) => Eq (Sq2Num a)
instance (Num a, Eq a) => Num (Sq2Num a)
instance (Num a, Eq a, Ord a) => Ord (Sq2Num a)
instance (Fractional a, Eq a) => Fractional (Sq2Num a)
```

Additionally, provide a conversion function

```
getReal :: Floating a => Sq2Num a -> a
```

reducing from $\mathcal{S}[\sqrt{2}]$ to $\mathcal{S}$ in such a way that real numbers are converted to themselves.

Hint: For inspiration, you can use the standard implementation of complex numbers in Haskell [1]. Like in the case of complex numbers, you need to prove (!) and implement the mathematical fact that the numbers ($*$) are closed under summation and multiplication and additionally under division, provided that so are the numbers from $\mathcal{S}$.

References

[1] https://www.haskell.org/onlinereport/complex.html.