RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

LEHR- UND FORSCHUNGSGEBIET INFORMATIK II
RWTH Aachen · D-52056 Aachen · GERMANY
http://www-i2.informatik.rwth-aachen.de/lufgi2.html

Prof. Dr. Jürgen Giesl
Darius Dlugosz

Exercises *Functional Programming* – Sheet 7

Solutions will be collected until Thursday, July 5, 2001.

Exercise 1 (2 points)

(a) Let D be a domain. Show that if $\sqsubseteq_D$ is a cpo on D then $\sqsubseteq_{D_\perp}$ is a cpo on $D_\perp$. ($D_\perp$ denotes the lift of domain D.)

(b) Let $D_1, \ldots, D_n$ be domains. Show that if $\sqsubseteq_{D_1}, \ldots, \sqsubseteq_{D_n}$ are cpo's then $\sqsubseteq_{D_1 \oplus \ldots \oplus D_n}$ is a cpo on $D_1 \oplus \ldots \oplus D_n$.

Exercise 2 (3 points)

Let $D_1, \ldots, D_n$ be domains. The domain constructors $\otimes$ and $+$ are defined as follows:

- $D_1 \otimes \ldots \otimes D_n = \{(d_1, \ldots, d_n) \mid \; d_1 \in D_1 - \{\perp_{D_1}\}, \ldots, d_n \in D_n - \{\perp_{D_n}\}\} \cup \{(\perp_{D_1}, \ldots, \perp_{D_n})\}$
  The relation $\sqsubseteq_{D_1 \otimes \ldots \otimes D_n}$ is defined in the same way as $\sqsubseteq_{D_1 \times \ldots \times D_n}$.
- $D_1 + \ldots + D_n = \{d^{D_1} \mid d \in D_1\} \cup \ldots \cup \{d^{D_n} \mid d \in D_n\} \cup \{\perp_{D_1 + \ldots + D_n}\}$
  The relation $\sqsubseteq_{D_1 + \ldots + D_n}$ is defined by $e \sqsubseteq_{D_1 + \ldots + D_n} e'$ iff $e = \perp_{D_1 + \ldots + D_n}$ or $e = d^{D_i}$, $e' = d'^{D_i}$ and $d \sqsubseteq_{D_i} d'$ for some $i \in \{1, \ldots, n\}$.

Represent the domain $D_1 + D_2$ graphically.

Prove or disprove

(a) $D_1 + D_2 = D_{1\perp} \oplus D_{2\perp}$

(b) $(D_1 \times D_2)_\perp = D_{1\perp} \otimes D_{2\perp}$ (without consideration of the labels)

(c) $(\{\underline{\mathsf{constr}}_1\} \times D)_\perp \oplus (\{\underline{\mathsf{constr}}_2\} \times D)_\perp = (\{\underline{\mathsf{constr}}_1, \underline{\mathsf{constr}}_2\} \times D)_\perp$ (without consideration of the labels)

Exercise 3 (2 points)

For the following data type declaration

```
data Expr = Nil | Atom Int | Cons Expr
```

give the corresponding domain and present it graphically.

Exercise 4 (4 points)

For the following Haskell declarations

```
exp1 :=  let not' = \x -> if x == True then False else True in not' y

exp2 :=  let even' = \x -> if x == 0 then True else not (even' (x - 1))
                                            in even' 4

exp3 :=  let even'' = \x -> if x == 0 then True else (if x == 1 then False
                                            else even'' (x - 2)) in even'' 5

exp4 :=  let half = \x -> if x <= 1 then 0 else half (x - 1) in half 3

exp5 :=  let double = \x -> if x == 0 then 0 else (2 + double (x - 1))
                                            in double 2
```

give the value of

(a) $\mathcal{V}al[\![\texttt{exp1}]\!]\,\rho$

(b) $\mathcal{V}al[\![\texttt{exp2}]\!]\,\rho$

(c) $\mathcal{V}al[\![\texttt{exp3}]\!]\,\rho$

(d) $\mathcal{V}al[\![\texttt{exp4}]\!]\,\rho$

(e) $\mathcal{V}al[\![\texttt{exp5}]\!]\,\rho$

where $\rho = \omega + \rho'$ and $\rho'(y) = \texttt{True}$, $\rho'(x) = 3$.

Describe your computation in detail and for each higher-order function $f : \texttt{Dom} \to \texttt{Dom}$ that occurs in the calculation determine what the function $f^i(\bot)$, $i \in \mathbb{N}$, computes.

Exercise 5 (3 points)

Extend the set of transformation rules with additional rules which transform Haskell-expressions with `where` declarations, conditional equations (usage of `|`) and patterns using `@` to a simple Haskell-expressions.

Remark: The pattern `var@pat` matches if `pat` matches and binds the whole value of `pat` to the variable `var`. Example:

```
tails :: [a] -> [[a]]
tails [] = []
tails xxs@(_: xs) = xxs: tails xs
```

Exercise 6 (3 points)

Transform the following Haskell-expressions to a simple Haskell-expressions.

(a)
```
data List a = Nil | Cons a (List a)

let rev Nil l = l
    rev (Cons x k) l = rev k (Cons x l)  in rev (Cons 1 (Cons 2 Nil))
```

(b)
```
data Nats = Zero | Succ Nats

let times Zero y = Zero
    times (Succ x) y = plus y (times x y)

    plus Zero y = y
    plus (Succ x) y = Succ (plus x y)  in times (Succ Zero) (Succ Zero)
```