ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

D. Basin and P. Müller

# Formal Methods and Functional Programming

## Exercise Sheet 5: Typing and Algebraic Data Types

Submission deadline: March 25th, 2012

## Assignment 1:

(a) Recall the following functions from the Prelude.

```
map     :: (a -> b) -> [a] -> [b]
elem    :: Eq a => a -> [a] -> Bool
0       :: Num a => a
(<)     :: Ord a => a -> a -> Bool
```

Give the most general type for each of the following functions.

1. `(\x y z -> x (y z))`
2. `(\f -> (\h -> (f,h)))`
3. `map (elem 0)`
4. `(\x -> x (<))`

(b) We extend mini-Haskell with the **let** construct that binds a term $t_1$ locally to a variable $x$ in term $t_2$. The extended term language is as follows, where $\mathcal{V}$ is a set of variables and $\mathcal{Z}$ is the set of integers:

$$\begin{array}{rcl} t & ::= & \mathcal{V} \mid (\lambda x.\, t) \mid (t_1\, t_2) \mid \\ & & \mathit{True} \mid \mathit{False} \mid (\textbf{iszero}\ t) \mid \\ & & \mathcal{Z} \mid (t_1 + t_2) \mid (t_1 \times t_2) \mid (\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2) \mid \\ & & (t_1, t_2) \mid (\textbf{fst}\ t) \mid (\textbf{snd}\ t) \mid \\ & & (\textbf{let}\ x = t_1\ \textbf{in}\ t_2) \end{array}$$

Give the typing rule for the **let** construct.

# Assignment 2:

Use the typing rules for mini-Haskell to prove the following statements.

(a) $\lambda x.\ (x\ 1\ \mathit{True}, x\ 0) :: (\mathit{Int} \to \mathit{Bool} \to a) \to (a, \mathit{Bool} \to a)$

(b) $(\lambda x.\ \lambda y.\ (y\ (\mathbf{iszero}\ (y\ x))))\ \mathit{True} :: (\mathit{Bool} \to \mathit{Int}) \to \mathit{Int}$

(c) $\lambda x.\ \lambda y.\ \mathbf{if}\ y\ x\ \mathbf{then}\ (\mathbf{fst}\ x)\ \mathbf{else}\ (\mathbf{snd}\ (\mathbf{snd}\ x)) :: (a, (b, a)) \to ((a, (b, a)) \to \mathit{Bool}) \to a$

# Assignment 3:

Formulas in propositional logic are built from variables of type `a`, conjunctions, disjunctions, and negations.

(a) Specify a Haskell data type `Prop a` to represent formulas in propositional logic.

(b) Implement a Haskell function `foldProp` that folds a proposition using separate folding functions for variables, negations, conjunctions, and disjunctions.

(c) Implement a Haskell function `evalProp :: (a -> Bool) -> Prop a -> Bool` using the `foldProp` function from (b) such that `evalProp v p` evaluates the formula `p` under the variable assignment `v`.

(d) Implement a function `propVars :: Prop a -> [a]` using the `foldProp` function from (b) such that `propVars p` computes the list of variables occuring in formula `p`.

(e) Implement a function `satProp :: Eq a => Prop a -> Bool` that checks whether a given formula is satisfiable, i.e., whether there is a variable assignment under which the formula evaluates to `True`.
**Note:** Your implementation does not need to be efficient.

# Assignment 4:

Recall the algebraic data type `Tree a` from the lecture:

```
data Tree t = Leaf | Node t (Tree t) (Tree t)
```

(a) Implement a *breadth-first* traversal on trees of the data type `Tree t`. Your function should traverse a given tree in a breadth-first manner and return a list with the elements that are stored in the tree.

(b) Recall the function `mapTree` from the lecture:

```
mapTree f Leaf = Leaf
mapTree f (Node x t1 t2) = Node (f x) (mapTree f t1) (mapTree f t2)
```

Prove that

$$\forall x :: \texttt{Tree}\ a.\, \texttt{mapTree}\ g\ (\texttt{mapTree}\ f\ x) = \texttt{mapTree}\ (g.f)\ x$$

for arbitrary $f :: a \texttt{->} b$ and $g :: b \texttt{->} c$ by structural induction over `Tree`.