FMFP

isabel.haas@inf.ethz.ch

June 16, 2022

1 Haskell

1.1 Basics

```haskell
-- Basic function
-- Declaration, comparable to int add(int a, int b){} in Java
add :: Int -> Int -> Int
add a b = a + b -- Definition
-- function composition
f (g x) = f.g x
-- functions can also be arguments
filter :: (a->Bool) -> [a] -> [a] -- first arg: function taking a returning Bool
-- Pattern matching
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
-- Guards
myAbs :: Int -> Int
myAbs x
    | x < 0 = -x
    | otherwise = x
-- where
f :: Int -> Int
f x = 1 + magic
    where magic = sqrt x
-- let in
-- let <def> in <expr> equal to <expr> where <def>
f :: Int -> Int
f x = (let magic = sqrt x in 1 + magic)
-- case expression (pattern matching)
case expression of pattern1 -> result1
```

```
                    pattern2 -> result2
div1byx :: Double -> Double
div1byx = case x of 0 -> 0.0
                     n -> 1/n
-- if else
if b then x else y -- returns either x or y
f x = if (prime x) then "PRIME" else "NOT"
```

1.2 Lists

```
[] -- empty list
x:xs -- first element is x, xs is rest of list
[a,b,c] -- syntactic sugar for a:b:c:[]
-- Basic pattern matching
-- [1..x]
[1..4] -- [1,2,3,4]
[1,3..10] -- [1,3,5,7,9]
[5, 4..1] -- [5,4,3,2,1]
[5..1] -- []
[1,2...] -- [1,2,...], used with lazy evaluation
-- List comprehensions
[f x | x <- list , guard_1, ..., guard_n]
[2*x | x <- [1..20], x `mod` 2 == 1] -- [2,6,10,..38]
[(l,r)| l <- "abc", r <- "xyz"] -- all combinations of characters in "abc" and "xyz"
-- Quick sort, very pretty
q (p:xs) = q [x | x <- xs, x <= p] ++ [p] ++ q [x | x <- xs, x > p]
```

1.3 Prelude functions

```
-- Basics
head [1,2,3] -- 1 :: Int
tail [1,2,3] -- [2,3] :: [Int]
last [1,2,3] -- 3 :: Int
init [1,2,3] -- [1,2] :: [Int]
length [1,2,3] -- 3 :: Int
take 3 [1,2,3,4,5] -- [1,2,3] :: [Int]
drop 3 [1,2,3,4,5] -- [4,5] :: [Int]
reverse [1,2,3] -- [3,2,1] :: [Int]
maximum [1,2,3] -- 1 :: Int
minimum [1,2,3] -- 3 :: Int
sum [1,2,3,4] -- 10 :: Int
product [1,2,3,4] -- 24 :: Int
4 `elem` [1,2,3] -- False
```

```haskell
-- More interesting
zip :: [a] -> [b] -> [(a,b)]
zip [1, 2] ['a', 'b'] -- [(1,'a'),(2,'b')]
filter :: (a->Bool) -> [a] -> [a]
filter odd [1, 2, 3] -- [1,3]
map :: (a -> b) -> [a] -> [b]
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f [x1,x2,x3..] [y1,y2,y3..] == [f x1 y1, f x2 y2, f x3 y3..]
```

1.4 Fold

```haskell
-- right associative
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
foldr f z (a:b:c:[]) = f a (f b (f c (f z [])))
foldr (+) 0 [1..4] =
-- left associative
foldl :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldl f z xs = foldl f z . zoList
foldl f z (a:b:c:[]) = f (f a (f b)) c
```

2 Evaluation strategies

Haskell: Lazy Evaluation

- argument only evaluaed when no other steps possible
- left term is evaluated first
- argument made to fit pattern

2.1 Lazy evalutation

2.1.1 Sheet 1, Ex. 1

```haskell
fibLouis :: Int -> Int
fibLouis 0 = 1
fibLouis 1 = 1
fibLouis n = fibLouis (n - 1) + fibLouis (n - 2)
fibEva :: Int -> Int
fibEva n = fst (aux n) where
    aux 0 = (0, 1)B
```

```
    aux n = next (aux (n - 1))
    next (a, b) = (b, a + b)
```

Lazy Evaluation of fibLouis 4

```
fibLouis 4 =
fibLouis (4-1) + fibLouis (4-2) =
-- most left term is evaluated first
fibLouis 3 + fibLouis (4-2) =
(fibLouis (3-1) + fibLouis (3-2)) + fibLouis (4-2)
...
((fibLouis 1 + fibLouis (2-2)) + fibLouis (3-2)) + fibLouis (4-2) =
((1 + fibLouis (2-2)) + fibLouis (3-2)) + fibLouis (4-2) =
...
2 + fibLouis 2 =
2 + (fibLouis (2-1) + fibLouis (2-2))
... = 3
```

Lazy Evaluation of fibEva 4

```
fibEva 4 =
fst (aux 4) =
fst (next (aux (4-1))) =
fst (next (aux 3)) =
fst (next (next (aux (3-1)))) =
fst (next (next (aux 2))) =
...
fst (next (next (next (next (0, 1))))) =
fst (next (next (next (1, 0+1)))) =
fst (next (next (0+1, 1+(0+1)))) =
fst (next (1+(0+1), (0+1)+(1+(0+1))))
...
fst ((0+1)+(1+(0+1)), (1+(0+1))+((0+1)+(1+(0+1)))) =
(0+1)+(1+(0+1)) =
-- pattern (0+1) is repeated
1 + (1 + 1) =
3
```

3 Natural Deduction

3.1 Paranthesizing formulas

- $\wedge$ binds stronger than $\vee$ stronger than $\rightarrow$
- $\rightarrow$ associates to right; $\wedge$ and $\vee$ to the left

- Negation extend to the right as far as possible: end of line or)
- Quantifiers extend to the right as far as possible: end of line or)

$p \lor q \land \neg r \to p \lor q$	$(p \lor (q \land (\neg r))) \to (p \lor q)$
$p \to q \lor p \to r$	$p \to ((q \lor p) \to r)$
$p \land \forall x.q(x) \lor r$	$p \land (\forall x.(q(x) \lor r))$
$\neg \forall x.p(x) \land \forall x.q(x) \land r(x) \land s$	$\neg(\forall x.(p(x) \land (\forall x.((q(x) \land r(x)) \land s))))$

3.2 Natural Deduction without quantifiers

If you cannot continue, try to add assumptions by using $\lor E$

3.2.1 Example

Exercise: P $(\neg A) \land (A \lor B) \to B$ is a tautology
First step: Paranthesizing $\Rightarrow$ $P \equiv ((\neg A) \land (A \lor B)) \to B$
Let $\Gamma \equiv (\neg A) \land (A \lor B)$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, A \vdash (\neg A) \land (A \lor B)}\,ax}{\Gamma \vdash A \lor B}\,\land ER \qquad \dfrac{\dfrac{}{\Gamma, A \vdash A}\,ax \qquad \dfrac{\dfrac{}{\Gamma, A \vdash (\neg A) \land (A \lor B)}\,ax}{\Gamma, A \vdash \neg A}\,\land EL}{\Gamma, A \vdash B}\,\neg E \qquad \dfrac{}{\Gamma, B \vdash B}\,ax}{\Gamma \vdash B}\,\lor E}{\vdash (\neg A) \land (A \lor B)}\,\to I$$

3.3 Natural Deduction with quantifiers

If you cannot continue, try to add assumptions by using $\exists E$
Always check side conditions and write it down

3.3.1 Sheet 2, Ex. 3b

Exercise: Proof $(\exists x.P \land Q) \to ((\exists x.P) \lor (\exists x.Q))$
Let $\Gamma \equiv \exists x.P \land Q, P \land Q$

$$\dfrac{\dfrac{\dfrac{}{(\exists x.P \land Q) \vdash (\exists x.P \land Q)}\,ax \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash P \land Q}\,ax}{\Gamma \vdash P}\,\land EL}{\Gamma \vdash \exists x.P}\,\exists I \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash P \land Q}\,ax}{\Gamma \vdash Q}\,\land ER}{\Gamma \vdash \exists x.Q}\,\exists I}{\Gamma \vdash (\exists x.P) \lor (\exists x.Q)}\,\land I}{(\exists x.P \land Q) \vdash (\exists x.P) \lor (\exists x.Q)}\,\exists E^{**}}{\vdash (\exists x.P \land Q) \to ((\exists x.P) \lor (\exists x.Q))}\,\to I$$

** side condition OK: x not free in $\exists x.P \land Q$ nor $(\exists x.P) \lor (\exists x.Q)$

4 Binding and α-conversion

Bound: Each occurrence of a variable is bound or free: A variable occurence x in a formula A is **bound** if x occurs within a subformula B of A of the form $\exists x.B$ or $\forall x.B$.
Alpha-conversion: bound variables can be renamed
Examples

		α-convertible
$\forall x.\exists y.p(x,y)$	$\forall y.\exists x.p(y,x)$	yes
$\exists z.\forall y.p(z,f(y))$	$\exists y.\forall y.p(y,f(y))$	no
$(\forall x.p(x)) \vee (\exists x.q(x))$	$(\forall z.p(z)) \vee (\exists y.q(y))$	yes
$p(x) \to \forall x.p(x)$	$p(y) \to \forall y.p(y)$	no

5 Induction

5.1 Induction for natural numbers

Induction scheme:

$$\frac{\Gamma \vdash P[n \mapsto 0] \qquad \Gamma \vdash \forall m : Nat.P[n \mapsto m] \to P[n \mapsto m+1]}{\Gamma \vdash \forall n : Nat.P} \text{ m not free in P}$$

5.1.1 Sheet 3, Ex. 1b

(Important parts/"framework" of proof)
Lemma: $\forall n. : Nat$ `aux n = (fibLouis n, fibLouis (n+1))`

Proof. Let `P:=`(aux n = (fibLouis n, fibLouis (n+1)))
Base case. Show `P[n ↦ 0]`

```
aux 0 = ...
    = (fibLouis 0, fibLouis (0+1)) -- arithmetic
```

Step case. Let `m:Nat` be arbitrary.
Show that `P[n↦ m]` implies `P[n↦ m+1]`.
Assume `aux m = (fibLouis m, fibLouis (m+1))`

```
aux (m+1) = ...
        = (fibLouis (m+1), fibLouis ((m+1)+1)) -- arithmetic
```

□

5.2 Induction over lists

For proofs with `[]` or `0`, you may first have to proof a generalised statement with induction and then simply plug in your values.

5.2.1 Sheet 3, Ex. 2b

Lemma: `foldr (:)  [] xs = xs`

Proof. Let`P:= (foldr (:) [] xs = xs)`.
We prove by induction over lists that $\forall xs :: [a]$. `P` holds.
Base case. Show `P[xs ↦ []]`

```
foldr (:) [] [] = [] -- foldr.1
```

Step case. Let `y::a ys::[a]` be arbitrary.
Show that `P[xs↦ ys]` implies `P[xs↦ (y:ys)]`
Assume `foldr (:) [] ys = ys` and we show that `foldr (:) [] (y:ys) = y:ys`

```
foldr (:) [] (y:ys) =
    = ...
    = (y:ys)
```

□

5.2.2 Sheet 4, Ex. 1

Lemma: `rev (xs ++ rev ys) = ys ++ rev xs`

Proof. Let `P' := rev (xs ++ rev ys') = ys' ++ rev xs`. We show that ∀ `ys'`.∀ `xs`..
Fix an arbitrary `ys` and let `P := [ys' ↦ ys]`. We show that ∀`xs P`.
(This implies ∀`ys'`.∀`xs.  P`)
Base case: We show `P[xs ↦ []]`

```
rev ([] ++ rev ys) = ...
    = ys ++ rev []
```

Step case: We need to show ∀`z, zs P[xs ↦ zs] → P[xs ↦ (z:zs)]`.
Fix arbitrary `y::a, ys::[a]`.
We assume `IH: rev (zs ++ rev ys) = ys ++ rev zs`
and show that `rev ((z:zs) ++ rev ys) = ys ++ rev (z:zs)`

```
rev ((z:zs) ++ rev ys)
    = ...
    = ys ++ rev (z:zs)
```

□

6 Types and typing inference

`f :: a -> b -> c -> d`:

- same as `f :: a -> (b -> (c -> d))` (parentheses are right associative)
- `f x y z` implies `x :: a, y :: b, z :: c`
- f.e. `f x :: b -> c -> d`

6.1 Types

- Detect function applications, f.e. `f x` ⇒ `f :: a -> b, f :: x`
- Detect prelude functions such as map, filter, foldr etc.
- "Match" types of different function, f.e. `f :: (a->b) -> [a] -> b` for `f x` ⇒ `x :: (a->b)`
- Don't forget things like `Num a, Eq b => ...`

6.1.1 Sheet 5

1a `\x y z -> (x y) z`

1. Three arguments, one return value
2. `(x y) :: a -> b` and `z :: a`
3. `x :: c -> (a->b)` and `y :: c`
4. `\x y z -> (x y) z :: (c -> a -> b) -> c -> a -> b`

2a.4 `(.).(.)` (the endboss)

1. `(.) :: (b->c) -> (a->b) -> a -> c`
2. Rewrite: `(.).(.) = .(.)(.) = f g h`
3. Definition of `(.)`:
   `f :: (b->c) -> ((a->b) -> a -> c)`
   `g :: (n->o) -> ((m->n) -> m -> o)`
   `h :: (q->r) -> ((p->q) -> p -> r)`
4. `g` is first argument of `f`:
   `g :: b -> c`
   ⇒ `b = n -> o` (I) and `c = (m->n) -> m -> o` (II)
5. `h` is first argument of `f g`:
   `f g :: (a->b) -> a -> c`
   ⇒ `h :: a -> b`
   ⇒ `a = q -> r` (III)
   `(p->q) -> p -> r` (IV)
6. (I) and (IV) ⇒ `n = p -> q` (V) and `o = p -> r`
7. After"taking" two arguments, we have the following type
   `f g h :: a -> c`
   `= (q->r) -> (m->n) -> m -> o`
   `= (q->r) -> (m->p->q) -> m -> p -> r`