Abstract

This document should give an overview over the types of exercises in the FMFP course and how to solve them. It also contains parts of theory and an overview of Haskell.
Main sources are the course material and material provided by the course TA Max Schlegel on https://n.ethz.ch/ mschlegel/fmfp22/fmfp.html.

Contents

Functional Programming

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
-- £
f $ x = f x
f $ map g xs = f (map g xs) -- to avoid parentheses
-- 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 <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
```

```haskell
f x = if (prime x) then "PRIME" else "NOT"
```

1.2 Lists

```haskell
[] -- 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
f [] = 0
f (x:xs) = 2 + f xs
-- [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 comb. of characters in "abc" & "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

```haskell
-- 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
-- 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..]
-- 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 . toList
foldl f z (a:b:c:[]) = f (f a (f b)) c
-- returns longest prefix of elements satisfying p and corresponding remainder of li
span :: (a -> Bool) -> [a] -> ([a], [a]) -- span p xs
span (< 3) [1,2,3,4,1,2,3,4] -- ([1,2],[3,4,1,2,3,4])
curry :: ((a,b)->c) -> a -> b -> c
curry f a b = f (a,b)
uncurry :: (a->b->c) -> (a,b) -> c
uncurry f (x,y) == f a b
```

1.4 Algebraic data types

Define new types

```
-- Structure: on the right side are value constructors
-- data type can have one of those different values
data keyword = constr1 | constr2 | ... | constrn
-- Option can be simple types
data Bool = False | True
-- New value constructors can be defined
-- Circle takes three floats as fields, rectangle 4
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
-- ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
-- functions for data types
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
-- has argument of type a or b
```

```haskell
data myType a b = myConstr a | myOtherConstructor b
-- definitions can  be recursive
data myList a = Empty | Cons a (MyList a)
-- tree
data Tree t = Leaf | Node t (Tree t) (Tree t)

-- deriving keyword
-- typeclasses like Eq, Ord, Enum, Bounded, Show, Read can function as "interfaces"
-- Example: == and /= and can now be used to compare values
data Vector = Vector Int Int Int deriving (Eq, Show)

-- instance keyword
data TrafficLight = Red | Yellow | Green
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"

-- fold for data types
-- data type:
data DType = C1 ... | C2 ... | ... | CN ...
-- fold
foldDType :: foldC1 -> foldC2 -> ... -> foldCN -> DType -> b
-- example
data Prop a = Var a | Not (Prop a) | And (Prop a) (Prop a) | Or (Prop a) (Prop a)
foldProp :: (a->b) -> (b->b) -> (b->b->b) -> (b->b->b) -> (Prop a) -> b
foldProp fVar fNot fAnd fOr prop  = go prop
    where
        go (Var v) = fVar v
        go (Not v) = fNot (go v)
        go (And v w) = fAnd (go v) (go w)
        go (Or v w) = fOr (go v) (go w)
```

2 Evaluation strategies

Lazy evaluation strategy of application `t1 t2`

1. Evaluate `t1`

2. The argument `t2` is substituted in `t1` without being evaluated

3. No evaluation inside lambda abstractions. In other words, in an abstraction `\... -> f t`, then `f t` is not evaluated

Eager evaluation strategy of application `t1 t2`

1. Evaluate `t1`

2. `t2` is evaluated prior to substitution in `t1`

3. Evaluation is carried out inside lambda abstractions

2.1 Lazy evaluation in Haskell

Haskell: Lazy Evaluation

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

2.1.1 Sheet 1, Ex. 1

```
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))))
...
-- pattern (0+1) is repeated
fst ((0+1)+(1+(0+1)), (1+(0+1))+((0+1)+(1+(0+1)))) =
(0+1)+(1+(0+1)) =
1 + (1 + 1) =
3
```

3 Natural Deduction

3.1 Parenthesizing formulas

- $\wedge$ binds stronger than $\vee$ stronger than $\to$
- $\to$ associates to right; $\wedge$ and $\vee$ to the left
- Negation binds stronger than binary operators
- Quantifiers extend to the right as far as possible: end of line or)

$p \vee q \wedge \neg r \to p \vee q$	$(p \vee (q \wedge (\neg r))) \to (p \vee q)$
$p \to q \vee p \to r$	$p \to ((q \vee p) \to r)$
$p \wedge \forall x.q(x) \vee r$	$p \wedge (\forall x.(q(x) \vee r))$
$\neg \forall x.p(x) \wedge \forall x.q(x) \wedge r(x) \wedge s$	$\neg(\forall x.(p(x) \wedge (\forall x.((q(x) \wedge r(x)) \wedge s))))$

3.2 Natural Deduction without quantifiers

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

3.2.1 Example

Exercise: $P = (\neg A) \wedge (A \vee B) \to B$ is a tautology
First step: Parenthesizing $\Rightarrow$ $P \equiv ((\neg A) \wedge (A \vee B)) \to B$
Let $\Gamma \equiv (\neg A) \wedge (A \vee B)$

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

3.3 Natural Deduction with quantifiers

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

3.3.1 Sheet 2, Ex. 3b

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

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

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

4 Binding and α-conversion

Bound: Each occurrence of a variable is bound or free: A variable occurrence x in a formula A is **bound** if x occurs within a sub formula 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

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

5.1 Induction on natural numbers

Induction scheme:

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

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))
```

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))
```

□

5.2 Induction on lists

Induction scheme:

$$\frac{\Gamma \vdash P[xs \mapsto []] \qquad \Gamma, P[xs \mapsto ys] \vdash P[xs \mapsto (y : ys)]}{\Gamma \vdash \forall xs :: [a].P} \; y, ys \text{ not free in P}$$

5.2.1 Sheet 3, Ex. 2b

(Important parts/"framework" of proof)
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 (:) [] [] = []
```

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

(Important parts/"framework" of proof)
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)
```

□

5.3 Induction on Trees

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

Induction scheme:

$$\frac{\Gamma \vdash P[x \mapsto \text{Leaf}] \qquad \Gamma, P[x \mapsto l], P[x \mapsto r] \vdash P[x \mapsto \text{Node}\; a\, l\, r]}{\Gamma \vdash \forall xs :: \text{Tree}\; t.P}\; a, l, r \text{ not free in P}$$

5.3.1 Sheet 6, Ex. 1

(Important parts/"framework" of proof)

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

For arbitrary `f :: a -> b` and `g :: b -> c`
`∀t :: Tree a. mapTree g (mapTree f t) = mapTree (g . f) t`

Proof. Let `f :: a -> b` and `g :: b -> c` be arbitrary functions.
Let `P := mapTree g (mapTree f t) = mapTree (g . f) t`, and we prove by induction that `∀t :: (Tree a).P`
Base Case: Show `P[t ↦ Leaf]`

```
mapTree g (mapTree f Leaf) = ..
        = mapTree (g . f) Leaf
```

Step case: Let `x::a, l::Tree a, r::Tree a` be arbitrary.
Assume `P[t ↦ l]` and `P[t ↦ r]`. (IH)
We know show that then `P[t ↦ Node x l r]` holds

```
mapTree g (mapTree f (Node x l r)) = ..
        = mapTree (g . f) (Node x l r)
```

□

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, x::a`
- 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 end boss)

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`

6.2 Typing proof and Inference

Solving type inference constraints

1. Remove trivial equations like $t = t$

2. Transform equations of form $\{f(s_0, ..., s_k) = g(t_0, ..., s_m)\}$ into $\{s_0 = t_0, ..., s_k = t_k\}$ if $f = g$ and $k = m$, else there is no solution

3. Substitute one equation into the others

6.2.1 Sheet 5, Ex. 3

a Proof λ`x.(x 1 True, x 0) ::  (Int -> Bool -> a) -> (a, Bool -> a)`:
Try to match left and right side with typing rule and apply it, should be straight forward
b Infer the type of (λ`x.`λ`y.(y (iszero (y x))) True`

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x:\tau_1, y:\tau_2 \vdash y :: \tau_4 \to \tau_3}\,Var^1 \quad \mathbf{T_2}}{x:\tau_1, y:\tau_2 \vdash y\ (\mathbf{iszero}\ (y\ x)) :: \tau_3}\,App}{x:\tau_1 \vdash \lambda y.(y\ (\mathbf{iszero}\ (y\ x))) :: \tau_0}\,Abs^1}{\vdash \lambda x.\lambda y.(y\ (\mathbf{iszero}\ (y\ x))) :: \tau_1 \to \tau_0}\,Abs \quad \dfrac{}{\vdash True :: \tau_1}\,True^1}{\vdash (\lambda x.\lambda y.(y\ (\mathbf{iszero}\ (y\ x))))\ True :: \tau_0}\,App$$

T_2:

$$\dfrac{\dfrac{\dfrac{}{x:\tau_1, y:\tau_2 \vdash y :: \tau_5 \to Int}\,Var^2 \quad \dfrac{}{x:\tau_1, y:\tau_2 \vdash x :: \tau_5}\,Var^3}{x:\tau_1, y:\tau_2 \vdash y\ x :: Int}\,App}{x:\tau_1, y:\tau_2 \vdash \mathbf{iszero}\ (y\ x) :: \tau_4}\,iszero^1$$

Finding out τ_0:

$$\begin{array}{l|l|l|l}
\tau_0 = \tau_2 \to \tau_3\ (Abs^1) & \tau_0 = \tau_2 \to \tau_3 & \tau_0 = \tau_2 \to \tau_3 & \tau_0 = (Bool \to Int) \to \tau_3 \\
\tau_2 = \tau_4 \to \tau_3\ (Var^1) & \tau_2 = \tau_4 \to \tau_3 & \tau_2 = \tau_4 \to \tau_3 & Bool \to Int = \tau_4 \to \tau_3 \\
\tau_4 = Bool\ (iszero^1) & \tau_4 = Bool & \tau_4 = Bool & \tau_4 = Bool \\
\tau_2 = \tau_5 \to Int\ (Var^2) & \tau_2 = \tau_5 \to Int & \tau_2 = Bool \to Int & \tau_5 = Bool \\
\tau_1 = \tau_5\ (Var^3) & \tau_5 = Bool & \tau_5 = Bool & \\
\tau_1 = Bool\ (True^1) & & &
\end{array}$$

$$\begin{array}{l|l}
\tau_0 = (Bool \to Int) \to \tau_3 & \tau_0 = (Bool \to Int) \to Int \\
\tau_3 = Int & \tau_3 = Int \\
\tau_4 = Bool & \tau_4 = Bool \\
\tau_5 = Bool & \tau_5 = Bool
\end{array}$$

d Infer type of `iszero(fst (3+5))`

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash 3 :: Int}\,Int \quad \dfrac{}{\vdash 5 :: Int}\,Int}{\vdash (3+5) :: (Int, \tau_1)}\,BinOp}{\vdash \mathbf{fst}(3+5) :: Int}\,fst}{\vdash \mathbf{iszero}(\mathbf{fst}(3+5)) :: \tau_0}\,iszero$$

Collected type constraints: $\tau_0 = Bool$ from $iszero$, $(Int = (Int, \tau_1))$ from $BinOp$, second constraint does not unify, meaning this doesn't type

Formal Methods

1 States and Expressions

1.1 States

State as a function:

$$\text{State: Var} \to \text{Val}$$

Zero state:

$$\sigma_{zero}(x) = 0 \text{ for all } x$$

Updating states:

$$(\sigma[y \mapsto v](x)) = \begin{cases} v & \text{if } x \equiv y \\ \sigma(x) & x \not\equiv y \end{cases}$$

Two states are equal:

$$\sigma_1 = sigma_2 \Leftrightarrow \forall x.(\sigma_1(x) = \sigma_2(x))$$

1.2 Semantics of arithmetic expression

Semantic function:

$$\mathcal{A} : \text{Aexp} \to \text{State} \to \text{Val}$$

Mapping

$$\begin{array}{ll} \mathcal{A}[\![x]\!]\sigma & = \sigma(x) \\ \mathcal{A}[\![n]\!]\sigma & = \mathcal{N}[\![n]\!] \\ \mathcal{A}[\![e_1\, op\, e_2]\!]\sigma & = \mathcal{A}[\![e_1]\!]\, \overline{op}\, \mathcal{A}[\![e_2]\!] \end{array}$$

with $\overline{op}$ the relation Val $\times$ Val corresponding to op

1.3 Semantics of boolean expression

Semantic function:

$$\mathcal{B} : \text{Bexp} \to \text{State} \to \text{Val}$$

Mapping

$$
\begin{aligned}
\mathcal{B}[\![e_1\, op\, e_2]\!]\sigma &= \begin{cases} \texttt{tt} & \text{if } \mathcal{A}[\![e_1]\!]\sigma\, \overline{op}\, \mathcal{A}[\![e_2]\!]\sigma \\ \texttt{ff} & \text{otherwise} \end{cases} \\
\mathcal{B}[\![b_1 \texttt{ or } b_2]\!]\sigma &= \begin{cases} \texttt{tt} & \text{if } \mathcal{B}[\![b_1]\!]\sigma = \texttt{tt} \text{ or } \mathcal{B}[\![b_2]\!]\sigma = \texttt{tt} \\ \texttt{ff} & \text{otherwise} \end{cases} \\
\mathcal{B}[\![b_1 \texttt{ and } b_2]\!]\sigma &= \begin{cases} \texttt{tt} & \text{if } \mathcal{B}[\![b_1]\!]\sigma = \texttt{tt} \text{ and } \mathcal{B}[\![b_2]\!]\sigma = \texttt{tt} \\ \texttt{ff} & \text{otherwise} \end{cases} \\
\mathcal{B}[\![\texttt{not } b]\!]\sigma &= \begin{cases} \texttt{tt} & \text{if } \mathcal{B}[\![b]\!]\sigma = \texttt{ff} \\ \texttt{ff} & \text{otherwise} \end{cases}
\end{aligned}
$$

with $\overline{op}$ the relation Val $\times$ Val corresponding to op

1.4 Free variables

$$
\begin{aligned}
&FV(e_1\, op\, e_2) &&= FV(e_1) \cup FV(e_2) \\
&FV(n) &&= \emptyset \\
&FV(x) &&= \{x\} \\
&FV(\texttt{not } b) &&= FV(b) \\
&FV(b_1 \texttt{ or } b_2) &&= FV(b_1) \cup FV(b_2) \\
&FV(b_1 \texttt{ and } b_2) &&= FV(b_1) \cup FV(b_2) \\
&FV(\texttt{skip}) &&= \emptyset \\
&FV(x := e) &&= \{x\} \cup FV(e) \\
&FV(s_1; s_2) &&= FV(s_1) \cup FV(e_2) \\
&FV(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}) &&= FV(b) \cup FV(s_1) \cup FV(s_2) \\
&FV(\texttt{while } b \texttt{ do } s \texttt{ end}) &&= FV(b) \cup FV(s)
\end{aligned}
$$

1.5 Substitution

$$
\begin{aligned}
&(e_1\, op\, e_2)[x \mapsto e] && \equiv (e_1[x \mapsto e]) \\
&n[x \mapsto e] && \equiv n \\
&y[x \mapsto e] && \equiv \begin{cases} e & \text{if } x \equiv y \\ y & \text{otherwise} \end{cases} \\
&(\texttt{not } b)[x \mapsto e] && \texttt{not } (b[x \mapsto e]) \\
&(b_1 \texttt{ or } b_2)[x \mapsto e] && (b_1[x \mapsto e] \texttt{ or } b_2[x \mapsto e]) \\
&(b_1 \texttt{ and } b_2)[x \mapsto e] && (b_1[x \mapsto e] \texttt{ and } b_2[x \mapsto e])
\end{aligned}
$$

Substitution Lemma:

$$
\mathcal{B}[\![b[x \mapsto e]]\!]\sigma = \mathcal{B}[\![b]\!](\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma])
$$

1.6 Structural induction on arithmetic and boolean expressions

1.6.1 Session sheet 10, Ex. 2

Statement: $\forall \sigma, e, e', x \, \mathcal{A}[\![e[x \mapsto e']]\!]\sigma = \mathcal{A}[\![e]\!](\sigma[x \mapsto \mathcal{A}[\![e']\!]\sigma])$

Proof. Let σ, x, e' be arbitrary.
Let $P(e) \equiv (\mathcal{A}[\![e[x \mapsto e']]\!]\sigma = \mathcal{A}[\![e]\!](\sigma[x \mapsto \mathcal{A}[\![e']\!]\sigma]))$.
We prove $\forall e.P(e)$ by strong structural induction on e.
We want to show $P(e)$ for some arbitrary e and assume $\forall e'' \sqsubset e \, P(e')$
Case $e \equiv n$ for some numerical value n:
...
Case $e \equiv y$ for some variable y:
...
Case $e \equiv e_1 \, op \, e_2$ for some arithmetic expressions e_1, e_2:
...

□

1.6.2 Sheet 10, Ex. 2

Statement: $\forall \sigma, e, e', x \, (\mathcal{B}[\![b[x \mapsto e]]\!]\sigma = \mathcal{B}[\![b]\!](\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]))$

Proof. Let σ, x, e be arbitrary.
Let $P(b) \equiv (\mathcal{B}[\![b[x \mapsto e]]\!]\sigma = \mathcal{B}[\![b]\!](\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]))$.
We prove $\forall b.P(b)$ by strong structural induction on e.
We want to show $P(e)$ for some arbitrary b and assume $\forall b'' \sqsubset b \, P(b')$
Case $b \equiv b_1$ `or` b_2 for some boolean expressions b_1, b_2:
...
Case $b \equiv b_1$ `and` b_2 for some boolean expressions b_1, b_2:
...
Case $b \equiv$ `not` b' for some boolean expression b':
...
Case $b \equiv e_1 \, op \, e_2$ for some arithmetic expressions e_1, e_2:
...

□