Formal Methods and Functional Programming

Isabel Haas (isabel.haas@inf.ethz.ch)

June, July 2022

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.
There is no guarantee for correctness or completeness, please refer to the course material.
Main sources are the course material and material provided by the course TA Max Schlegel on https://n.ethz.ch/ mschlegel/fmfp22/

Contents

Functional Programming

1 Haskell

Credits: Big parts of this section are copied from/inspired by https://n.ethz.ch/ mschlegel/fmfp22/, hence credits go to Max

1.1 Basics

```haskell
-- Basic function
-- Declaration, comparable to int mul(int a, int b){} in Java
mul :: Int -> Int -> Int
mul a b = a + b -- Definition
-- function composition
f (g x) = f.g x
-- dollar sign:
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
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
```

```haskell
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 list
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 x y
```

1.4 Algebraic data types

Define new types

```haskell
-- 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
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)
    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 $\Gamma \equiv (\neg A) \wedge (A \vee B)$

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

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 $\Gamma \equiv \exists x.P \wedge Q, P \wedge Q$

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

** 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 to names not yet used
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:

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

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.
I.H.: `P[n ↦ m]`
Show `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` $\mapsto$ `[]]`

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

Step case: Let `y::a, ys::[a]` be arbitrary.
I.H.: `P[xs` $\mapsto$ `ys]`
Show `P[xs` $\mapsto$ `(y:ys)]`
Assume `foldr (:)  [] ys = 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'` $\mapsto$ `ys]`. We show that ∀`xs P`.
(This implies ∀`ys'`.∀`xs.P'`)
Base case: We show `P[xs` $\mapsto$ `[]]`

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

Step case: Let `z::a, zs::[a]` be arbitrary.
Fix arbitrary `y::a, ys::[a]`.
I.H.: `rev (zs ++ rev ys) = ys ++ rev zs`
We show `P[xs` $\mapsto$ `(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.
I.H.1: `P[t ↦ l]`
I.H.2: `P[t ↦ r]`
We show `P[t ↦ Node x l r]`

```
        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)
   `b = (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`

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

T_2:

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

Finding out τ_0:

$\tau_0 = \tau_2 \to \tau_3$ (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)	$\tau_5 = Bool$	$\tau_5 = Bool$	
$\tau_1 = Bool$ ($True^1$)			

$\tau_0 = (Bool \to Int) \to \tau_3$	$\tau_0 = (Bool \to Int) \to Int$
$\tau_3 = Int$	$\tau_3 = Int$
$\tau_4 = Bool$	$\tau_4 = Bool$
$\tau_5 = Bool$	$\tau_5 = Bool$

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

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

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 Introduction to language semantics

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

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

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

1.4 Free variables

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

1.5 Substitution

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

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 $\forall e'' \sqsubset e\, P(e')$ (**I.H.**)
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 $\forall b'' \sqsubset b\, P(b')$ (**I.H.**)
Case $b \equiv b_1\ \texttt{ or }\ b_2$ for some boolean expressions b_1, b_2:
...
Case $b \equiv b_1\ \texttt{ and }\ b_2$ for some boolean expressions b_1, b_2:
...
Case $b \equiv \texttt{not }\ b'$ for some boolean expression b':

...
Case $b \equiv e_1 \, op \, e_2$ for some arithmetic expressions e_1, e_2:
... $\square$

2 Operational Semantics

2.1 Properties

2.1.1 Big step semantics

The execution of a statement s in state σ

- **terminates successfully** iff $\exists \sigma'$ st $\vdash \langle s, \sigma \rangle \to \sigma'$
- **fails to terminate** iff $\nexists \sigma'$ st $\vdash \langle s, \sigma \rangle \to \sigma'$

Semantic equivalence: s_1 and s_2 are semantically equivalent iff:

$$\forall \sigma, \sigma'. (\vdash \langle s_1, \sigma \rangle \to \sigma' \Leftrightarrow \langle s_2, \sigma \rangle \to \sigma')$$

2.1.2 Small step semantics

The execution of a statement s in state σ

- **terminates successfully** iff $\exists \sigma'$ st $\vdash \langle s, \sigma \rangle \to_1^* \sigma'$
- **fails to terminate** iff $\nexists \sigma'$ st $\vdash \langle s, \sigma \rangle \to_1^* \sigma'$

Semantic equivalence: s_1 and s_2 are semantically equivalent iff for all σ:

- for all stuck or terminal configurations γ: $\langle s_1, \sigma \rangle \to_1^* \gamma$ if and only if $\langle s_2, \sigma \rangle \to_1^* \gamma$
- there is an infinite derivation sequence starting in $\langle s_1, \sigma \rangle$ if and only if there is one starting in $\langle s_2, \sigma \rangle$

Lemma: The small-step semantics of IMP are **deterministic**: $\vdash \langle s, \sigma \rangle \to_1 \gamma \, \wedge \, \langle s, \sigma \rangle \to_1 \gamma' \Rightarrow \gamma = \gamma'$

2.2 Applying big-step semantics

2.2.1 Example

Let $s =$ `if` $x > y$ `then` $(x := y + 1; y := x - 2)$ `else skip end`.
We want to prove $\langle s, \sigma \rangle \to \sigma'$ for σ with $\sigma(x) = 4$, $\sigma(y) = 2$ and $\sigma' = \sigma[x, y \mapsto 3, 1]$

$$\dfrac{\dfrac{\dfrac{}{\langle x := y + 1, \sigma \rangle \to \sigma[x \mapsto 3]}\,\text{ASS}_{\text{NS}} \qquad \dfrac{}{\langle y := x - 2, \sigma[x \mapsto 3] \rangle \to \sigma'}\,\text{ASS}_{\text{NS}}}{\langle (x := y + 1; y := x - 2), \sigma \rangle \to \sigma'}\,\text{SEQ}_{\text{NS}}}{\langle s, \sigma \rangle \to \sigma'}\,\text{IFT}_{\text{NS}}$$

2.3 Applying small-step semantics

2.3.1 Example

Let $s =$ `if` $x > y$ `then` $(x := y + 1; y := x - 2)$ `else skip end`.
We want to prove $\langle s, \sigma \rangle \to_1^* \sigma'$ for σ with $\sigma(x) = 4$, $\sigma(y) = 2$ and $\sigma' = \sigma[x, y \mapsto 3, 1]$ Derivation sequence:

$$\begin{aligned}
& \langle s, \sigma \rangle \\
\to_1^1 \; & \langle (x := y + 1; y := x - 2), \sigma \rangle \\
\to_1^1 \; & \langle y := x - 2, \sigma[x \mapsto 3] \rangle \\
\to_1^1 \; & \sigma[x, y \mapsto 3, 1]
\end{aligned}$$

with the following derivation trees justifying the steps

$$\frac{}{\langle s,\sigma\rangle \to_1 \langle (x := y+1; y := x-2), \sigma\rangle}\ \text{IFT}_{\text{SOS}}$$

$$\frac{\dfrac{}{\langle x := y+1, \sigma\rangle \to_1 \sigma[x \mapsto 3]}\ \text{ASS}_{\text{SOS}}}{\langle (x := y+1; y := x-2), \sigma\rangle \to_1 \langle y := x-2, \sigma[x \mapsto 3]\rangle}\ \text{SEQ1}_{\text{SOS}}$$

$$\frac{}{\langle y := x-2, \sigma[x \mapsto 3]\rangle \to_1 \sigma[x, y \mapsto 3, 1]}\ \text{ASS}_{\text{SOS}}$$

2.4 Induction on shape of derivation tree

General idea:

1. Prove that for all σ, σ',(var. in s_1), if $\vdash \langle s_1, \sigma\rangle \to \sigma'$ then P for some property P
   $\Rightarrow$ Let $P(T) \equiv \forall \sigma, \sigma'$, (var. in s_1) $(root(T) \equiv \langle s_1, \sigma\rangle \to \sigma' \Rightarrow P)$
   Goal: Prove $\forall T.P(T)$ by induction on the shape of a derivation tree
2. Induction hypothesis: For arbitrary T, $\forall T' \sqsubset T.P(T')$
3. Let σ, σ', (var. in s_1) be arbitrary, assume $\langle s_1, \sigma\rangle \to \sigma'$
4. Do case analysis of last rule applied in T
5. Derive subtrees of T and properties about σ, like $\mathcal{B}[\![e]\!]\sigma = \texttt{tt}$
6. Use subtrees, properties of σ and induction hypothesis to prove P

Option: Simply do case distinction, I.H. doesn't have to be applied

2.4.1 Session sheet 11, Ex.4

$$\frac{\langle s,\sigma\rangle \to \sigma'}{\langle \texttt{repeat}\ s\ \texttt{until}\ b, \sigma\rangle \to \sigma'}\ (\text{RepT}_{\text{NS}}) \text{ if } \mathcal{B}[\![b]\!]\sigma' = \texttt{tt}$$

$$\frac{\langle s,\sigma\rangle \to \sigma'' \qquad \langle \texttt{repeat}\ s\ \texttt{until}\ b, \sigma''\rangle \to \sigma'}{\langle \texttt{repeat}\ s\ \texttt{until}\ b, \sigma\rangle \to \sigma'}\ (\text{RepF}_{\text{NS}}) \text{ if } \mathcal{B}[\![e]\!]\sigma'' = \texttt{ff}$$

Prove that for all σ, σ, b, s, if

$$\vdash \langle \texttt{repeat}\ s\ \texttt{until}\ b, \sigma\rangle \to \sigma'$$

then

$$\vdash \langle s; \texttt{while not}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \to \sigma'$$

Proof.

$$P(T) \equiv \forall \sigma, \sigma', b, s\ (root(T) \equiv \texttt{repeat}\ s\ \texttt{until}\ b, \sigma\rangle \to \sigma'$$
$$\Rightarrow \vdash \langle s; \texttt{while not}\ b\ \texttt{do}\ s\ \texttt{end}, \sigma\rangle \to \sigma')$$

We prove $\forall T.P(T)$ by induction on the shape of a derivation tree
Induction hypothesis: For arbitrary T, $\forall T' \sqsubset T.P(T')$
Let σ, σ', b, s be arbitrary, assume $\langle \texttt{repeat}\ s\ \texttt{until}\ b, \sigma\rangle \to \sigma'$.
Case analysis of last rule applied in T:
Case (REPT) Then T has the form:

$$\frac{\dfrac{\mathbf{T_1}}{\langle s,\sigma\rangle \to \sigma'}}{\langle \texttt{repeat}\ s\ \texttt{until}\ b, \sigma\rangle \to \sigma'}\ (\text{RepT}_{\text{NS}})$$

for some derivation tree T_1 and we must have $\mathcal{B}[\![b]\!]\sigma' = \texttt{tt}$, hence $\mathcal{B}[\![\texttt{not}\ b]\!]\sigma' = \texttt{ff}$.
We can construct following tree:

$$\dfrac{\dfrac{\mathbf{T_1}}{\langle s,\sigma\rangle \to \sigma'} \qquad \dfrac{}{\langle \texttt{while not } b \texttt{ do } s \texttt{ end}, \sigma'\rangle \to \sigma'}\ (\text{WHF}_{\text{NS}})}{\langle s; \texttt{while not } b \texttt{ do } s \texttt{ end}, \sigma\rangle \to \sigma'}\ (\text{SEQ}_{\text{NS}})$$

Case (REPF) Then T has the form:

$$\dfrac{\dfrac{\mathbf{T_1}}{\langle s,\sigma\rangle \to \sigma''} \qquad \dfrac{\mathbf{T_2}}{\langle \texttt{repeat } s \texttt{ until } b, \sigma''\rangle \to \sigma'}}{\langle \texttt{repeat } s \texttt{ until } b, \sigma\rangle \to \sigma'}\ (\text{RepF}_{\text{NS}})$$

for some state σ'' and derivation trees T_1, T_2, where $\mathcal{B}[\![e]\!]\sigma'' = \texttt{ff}$.
T_2 is a proper subtree of T, hence $P(T_2)$ holds by I.H.. This implies that there's a derivation tree T_3 with $root(T_3) \equiv \langle s; \texttt{while not } b \texttt{ do } s \texttt{ end}, \sigma''\rangle \to \sigma'$. The last rule applied in T_3 must be SEQ_{NS}, so T_3 has the form:

$$\dfrac{\dfrac{\mathbf{T_4}}{\langle s,\sigma'\rangle \to \sigma'''} \qquad \dfrac{\mathbf{T_5}}{\langle \texttt{while not } b \texttt{ do } s \texttt{ end}, \sigma'''\rangle \to \sigma'}}{\langle s; \texttt{while not } b \texttt{ do } s \texttt{ end}, \sigma''\rangle \to \sigma'}\ (\text{SEQ}_{\text{NS}})$$

for some state σ''' and derivation trees T_4, T_5.
We can now construct following derivation tree

$$\dfrac{\dfrac{\mathbf{T_1}}{\langle s,\sigma\rangle \to \sigma''} \qquad \dfrac{\dfrac{\mathbf{T_4}}{\langle s,\sigma'\rangle \to \sigma'''} \qquad \dfrac{\mathbf{T_5}}{\langle \texttt{while not } b \texttt{ do } s \texttt{ end}, \sigma'''\rangle \to \sigma'}}{\langle \texttt{while not } b \texttt{ do } s \texttt{ end}, \sigma''\rangle \to \sigma'}\ (\text{WHT}_{\text{NS}})}{\langle s; \texttt{while not } b \texttt{ do } s \texttt{ end}, \sigma\rangle \to \sigma'}\ (\text{SEQ}_{\text{NS}})$$

□

2.4.2 Session sheet 12/Sheet 12: Proof of equivalence lemmas

Direction big step to small step semantics:
Use derivation tree of big step semantic to get a derivation sequence for the small step semantic.

Proof.

$$P(T) \equiv \forall \sigma, \sigma', s\ (root(T) \equiv (\langle s,\sigma\rangle \to \sigma') \Rightarrow \langle s,\sigma\rangle \to_1^* \sigma')$$

We prove $\forall T.P(T)$ by induction on the shape of a derivation tree
Induction hypothesis: For arbitrary T, $\forall T' \sqsubset T.P(T')$
Let σ, σ', s be arbitrary, assume $\langle s,\sigma\rangle \to \sigma'$.
Case distinction by last rule applied in T:
Case (WHFNS) Then T has the form:

$$\dfrac{}{\langle \texttt{while } b \texttt{ do } s' \texttt{ end}, \sigma\rangle \to \sigma'}\ (\text{WHF}_{\text{NS}})$$

for some b, s' such that $s \equiv \texttt{while } b \texttt{ do } s' \texttt{ end}$ and $\mathcal{B}[\![e]\!]\sigma = \texttt{ff}$.
We can construct following derivation sequence:

$$\begin{aligned} &\langle \texttt{while } b \texttt{ do } s' \texttt{ end}, \sigma\rangle \\ \to_1^* &\ \ldots \\ \to_1^1 &\ \sigma \end{aligned}$$

Case (IFT_{NS}) Then T has the form:

$$\dfrac{\dfrac{\mathbf{T_1}}{\langle s_1,\sigma\rangle \to \sigma'}}{\langle \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}, \sigma\rangle \to \sigma'}\ (\text{IFT}_{\text{NS}})$$

for some b, s_1, s_2, T_1 such that $s \equiv \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}$ and $\mathcal{B}[\![e]\!]\sigma = \texttt{tt}$.
From $P(T_1)$ we learn $\langle s_1, \sigma\rangle \rightarrow_1^* \sigma'$ We can construct following derivation sequence:

$$\begin{aligned} &\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}, \sigma\rangle \\ \rightarrow_1^1\ &\langle s_1, \sigma\rangle \\ \rightarrow_1^*\ &\sigma \end{aligned}$$

. . . (other cases)

□

2.5 Proving properties of derivation sequences

General idea:

1. Prove $\gamma \rightarrow_1^* \gamma' \Rightarrow P$ for some property P
2. Define $P(k) \equiv (\gamma \rightarrow_1^k \gamma' \Rightarrow P)$ to do a strong induction over k
3. Deal with case $k = 0$ (if applicable)
4. Deal with case $k > 0$ by splitting off first execution step $\sigma \rightarrow_1^1 \delta \rightarrow_1^{k-1} \gamma$
   - Get information by case distinction of first execution step
   - Apply induction hypothesis to remaining steps

2.5.1 Session sheet 12/Sheet 12: Proof of equivalence lemmas

Direction small step to big step semantics:

Proof.

$$Q(k) \equiv (\forall \sigma, \sigma', s\ (\langle s, \sigma\rangle \rightarrow_1^k \sigma') \Rightarrow \vdash \langle s, \sigma\rangle \rightarrow \sigma')$$

Using strong induction, we prove $\forall k\ Q(k)$.
$\mathbf{k = 0}$: Trivially, σ must be and end state.
$\mathbf{k > 0}$: Assume $\langle s, \sigma\rangle \rightarrow_1^k \sigma'$.
The derivation sequence is unrolled to $\langle s, \sigma\rangle \rightarrow_1^1 \gamma \rightarrow_1^{k-1} \sigma'$. Let T be the derivation tree justifying the first transition. We do a case distinction on the last rule applied in T:
Case (ASS_{SOS}): T has the form

$$\frac{}{\langle x := e, \sigma\rangle \rightarrow_1 \sigma'}\ (\text{ASS}_{\text{SOS}})$$

for some x, e such that $s \equiv x := e$ and $\gamma = \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$. Since γ is a final state there is no further derivation sequence (k=1), and hence $\sigma' = \gamma = \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$. We can construct the following derivation tree:

$$\frac{}{\langle x := e, \sigma\rangle \rightarrow \sigma'}\ (\text{ASS}_{\text{NS}})$$

. . . (other cases)

□

3 Axiomatic semantics

Meaning of $\{P\}\, s\, \{Q\}$:

- **If** P evaluates to true in an initial state σ, and **if** the execution of s from σ terminates in a state σ' **then** Q will evaluate to true in σ'.
- This describes **partial correctness**, that is, termination is not an essential property

Two statements s_1 and s_2 are **provably equivalent** if:

$$\forall P, Q ::; \vdash \{P\}\ s_1\ \{Q\} \ \Leftrightarrow\ \{P\}\ s_2\ \{Q\}$$

Meaning of $\{P\}\, s\, \{\Downarrow Q\}$:

- **If** P evaluates to true in an initial state σ, **if** the execution of s from σ terminates in a state σ' **and** Q will evaluate to true in σ'.
- This describes **total correctness**, that is, termination is not an essential property

Termination is proved using **loop variants**

- Loop variant is an expression that evaluates to a value in a well-founded set (for instance, $\mathbb{N}$)
- Each loop iteration must decrease the value of the loop variant
- The loop has to terminate when a minimal value of the well-founded set is reached (or earlier)

Total correctness derivation rule for loops:

$$\frac{\{n \wedge P \wedge e = Z\}\ s\ \{\Downarrow P \wedge e < Z\}}{\{P\}\ \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}\ \{\Downarrow \neg b \wedge P\}}\ (\text{WHTOT}_{\text{Ax}})\text{ if } b \wedge P \vDash 0 \leq e$$

where Z is a fresh logical variable (not used in P)
The derivation system for partical correctness of IMP programs is **sound** or **complete**

- Soundness: if a property can be proved it does indeed hold
- Completeness: if a property holds it can be proved

3.1 Induction on shape of derivation trees

(Not in exercises, but in slides and old exam)
Same technique as with big-step semantics

3.1.1 Example

Statement $\forall P, Q.\ \{P\}\ \texttt{skip}\ \{Q\}\ \Rightarrow\ P \vDash Q$

Proof.

$$P(T) \equiv \forall P, Q.\ root(T) \equiv (\{P\}\ \texttt{skip}\ \{Q\})\ \Rightarrow\ P \vDash Q$$

where T is a derivation tree with axiomatic semantic rules applied.
We want to prove $\forall T.\, P(T)$ by strong structural induction on the shape of a derivation tree.
We do a case distinction by the last rule applied in T:
... □

3.2 Proof outlines

1. Write pre- and postcondition at beginning/start
2. If you have a loop: Write loop invariant before loop, at starts of loop and after loop
3. Add conditions of while and if statements at start of the statement body

3.2.1 Notations

```
$\{P\}$
  skip
$\{P\}$
```

```
$\{P[x \mapsto e]\}$
  $x := e$
$\{P\}$
```

```
$\{P\}$
  $s_1$
$\{Q\}$
  $s_2$
$\{R\}$
```

```
$\{P\}$
  if $b$ then
    $\{b \wedge P\}$
      $s_1$
    $\{Q\}$
  else
    $\{\neg b \wedge P\}$
      $s_2$
    $\{Q\}$
  end
$\{Q\}$
```

```
$\{P\}$
  while $b$ do
    $\{b \wedge P\}$
      $s_1$
    $\{P\}$
$\{\neg b \wedge P\}$
```

3.2.2 Finding loop invariants and variants

Finding an invariant:

- Include variables changed in loop
- Include relationships between input and loop variables
- If the goal of a program is to f.e. compute the gcd or compute a sum, include this in the invariant, i.e. $gcd(a, b) = gcd(x, y)$, $\sum_{i=0}^{k}(...) = ...$
- Implications for if-statements

Finding a variant:

- Find loop variable that decreases
- Find difference/sum of loop variables that decreases

Session sheet 13: GCD algorithm :
IMP program s:

```
b := x
c := y
while b # c do
    if b < c then
        c := c - b
    else
        b := b-c
    end
end
z := b
```

Goal: Prove Hoare Triple $\{x = X \wedge y = Y \wedge X > 0 \wedge Y > 0\}$ s $\{\Downarrow z = gcd(X, Y)\}$
Suitable loop invariant: $gcd(x, y) = gcd(a, b) \wedge b > 0 \wedge c > 0 \wedge x = X \wedge y = Y$
Suitable loop variant: $b + c$

FS10, Ex. 7
IMP program s:

```
while x < y do
    t := x
    x := y
    y := t
end
```

Goal: Prove Hoare Triple $\{x = X \wedge y = Y\}$ s $\{\Downarrow x = \max(X, Y)\}$
Suitable loop invariant: $\max(x, y) = \max(X, Y)$
Suitable loop variant: $y - x$

3.2.3 Sheet 13, Ex.2

IMP program s:

```
y := 0
z := 0
while y * y < n do
    y := y+1
    if y * y <= n then
        z := z+1
    else
        skip
    end
end
```

Goal: Prove Hoare Triple $\{n = N \land n \geq 0\}$ s $\{z^2 \leq N \land N < (z+1)^2\}$

Suitable loop invariant:

$$(y^2 \leq n \Rightarrow y = z) \land (y^2 > n \Rightarrow y = z + 1) \land (z^2 \leq N) \land (n = N) \land (z \geq 0)$$

Suitable loop variant: $n - y^2$

Please disregard the termination part of the proof for now, as I'm not sure where we can write the ⇓ symbol already

Proof outline:

$\{\ n = N \land n \geq 0\}$

$\models$

$\{\ (0^2 \leq n \Rightarrow 0 = 0) \land (0^2 > n \Rightarrow 0 = 0 + 1) \land 0^2 \leq n \land n = N \land 0 \geq 0\ \}$

`y := 0`

$\{\ (y^2 \leq n \Rightarrow y = 0) \land (y^2 > n \Rightarrow y = 0 + 1) \land 0^2 \leq n \land n = N \land 0 \geq 0\ \}$

`z := 0`

$\{\ (y^2 \leq n \Rightarrow y = z) \land (y^2 > n \Rightarrow y = z + 1) \land z^2 \leq n \land n = N \land z \geq 0\ \}$

`while y * y < n do`

$\{\ y^2 < n \land (y^2 \leq n \Rightarrow y = z) \land (y^2 > n \Rightarrow y = z + 1) \land z^2 \leq n \land n = N \land z \geq 0 \land n - y^2 = V\ \}$

$\models$ (1)

$\{(y + 1 - 1)^2 < n \land y + 1 = z + 1 \land z^2 \leq n \land n = N \land z \geq 0 \land n - (y + 1)^2 < V\ \}$

`  y := y+1`

$\{\Downarrow (y - 1)^2 < n \land y = z + 1 \land z^2 \leq n \land n = N \land z \geq 0 \land n - y^2 < V\ \}$

`  if y * y <= n then`

$\{\ y \leq n \land (y - 1)^2 < n \land y = z + 1 \land z^2 \leq n \land n = N \land z \geq 0 \land n - y^2 < V\ \}$

$\models$

$\{\ y \leq n \land (y - 1)^2 < n \land y = z + 1 \land (z + 1 - 1)^2 \leq n \land n = N \land z + 1 \geq 0 \land n - y^2 < V\ \}$

`    z := z+1`

$\{\Downarrow y \leq n \land (y - 1)^2 < n \land y = z \land (z - 1)^2 \leq n \land n = N \land z \geq 0 \land n - y^2 < V\ \}$

$\models$ (2)

$\{\Downarrow (y^2 \leq n \Rightarrow y = z) \land (y^2 > n \Rightarrow y = z + 1) \land z^2 \leq n \land n = N \land z \geq 0 \land n - (y - 1)^2 < V\ \}$

`  else`

$\{\ \neg(y \leq n) \land (y - 1)^2 < n \land y = z + 1 \land z^2 \leq n \land n = N \land z \geq 0 \land n - y^2 < V\ \}$

`    skip`

$\{\Downarrow \neg(y \leq n) \land (y - 1)^2 < n \land y = z + 1 \land z^2 \leq n \land n = N \land z \geq 0 \land n - y^2 < V\ \}$

$\models$ (2)
$\{\Downarrow (y^2 \leq n \Rightarrow y = z) \land (y^2 > n \Rightarrow y = z+1) \land z^2 \leq n \land n = N \land z \geq 0 \land n - y^2 < V \}$

$\boxed{\quad \texttt{end}}$

$\{\Downarrow (y^2 \leq n \Rightarrow y = z) \land (y^2 > n \Rightarrow y = z+1) \land z^2 \leq n \land n = N \land z \geq 0 \land n - y^2 < V \}$

$\boxed{\texttt{end}}$

$\{\Downarrow \neg(y^2 < n) \land (y^2 \leq n \Rightarrow y = z) \land (y^2 > n \Rightarrow y = z+1) \land z^2 \leq N \land n = N \land z \geq 0 \}$
$\models$ (3)
$\{\Downarrow z^2 \leq N \land N < (z+1)^2\}$

(1) $y^2 < n$ and $y^2 \leq n \Rightarrow y = z$ implies $y = z$, $y \leq 0$ must hold for $n - y^2 > n - (y+1)^2$

(2) As either $y^2 > n$ or $y^2 \leq n$ are false, they immediately imply the right hand side

(3) $N < (z+1)^2$ can be shown by case distinction on y:
Case $y^<n$: As $y^2 \geq n$, this case is not possible
Case $y^2 = n$: $y^2 = n \land (y^2 \leq n \implies y = z)$ implies $y^2 = z^2$ and then $N = n = y^2 = z^2 < (z+1)^2$ (last step requires $z \geq 0$)
Case $y^2 > n$: $y^2 > n \land (y^2 > n \Rightarrow y = z+1)$ implies $y = z+1$ and then $N = n < y^2 = (z+1)^2$

4 Model Checking

Process:

1. Modeling phase: Model the system under consideration using the description language of your choice; Formalize the properties to be checked
2. Running phase: Run the model checker to check the validity of the property in the system model
3. Analysis Phase: if the property is satisfied, celebrate and move on; if the property is violated, analyze counterexample; If out of memory, reduce model and try again

Main Purposes of Model checking:

- Model checking is mainly used to analyze system designs
- Typical properties to be analyzed include: Deadlocks, reachability of undesired states, protocol violations

Modeling concurrent systems:

- Systems are modelled as **finite transition systems**
- We model systems as **communicating sequential processes** (agents): Finite number of processes, interleaved process execution
- Processes can communicate via: shared variables, synchronous message passing, asynchronous message passing

4.1 Promela

4.1.1 Syntax

- Constant declarations:

```
#define N 5
mtype = { ack, req };
```

- Structure declarations: `typedef vector  int x; int y;`
- Global channel declarations: `chan buf = [2] of  int ;`
- Global variable declarations: `byte counter;`
- Process declarations: `proctype myProc(int p) ...`
- `skip`: does not change state (except the location counter), always executable
- `timeout`: does not change state (except the location counter), executable if all other statements in the system are blocked
- `assert(e)`: aborts execution if expression E evaluates to zero, otherwise equivalent to skip; always executable
- Sequential compositions: `s1;s2` is executable if `s1` is executable
- Expression statement: evaluates expression `E`, executable if E evaluates to value different from zero, `E` must not change state (no side effects)
- Selection: executable if at least one of its options is executable, chooses an option non-determinisitically and executes it, statement `else` is executable if no other option is executable

  ```
  if
  :: x > 0 -> x = x + 1
  :: x < 0 -> x = x - 1
  :: y > 0 -> y = y + 1
  :: y < 0 -> y = y - 1
  :: color = color + 1
  fi
  ```

- Repetions: executable if at least one of its options is executable, chooses repeatedly an option non-determinisitically and executes it, terminates when a `break` or `goto` is executed

  ```
  do
  :: n > 1 -> r = r*n; n = n-1
  :: else -> break
  od
  ```

- Atomic: `atomic  s` , executable if first statement of s is executable
- Macros: `inline func(x)`, defines replacement text for symbolix name, no new variable scopes, recursion or return values

4.1.2 Examples from session sheet 13

1.2: `x := 1 [] x := 2; x + 2` will result in a state σ where either $\sigma(x) = 1$ or $\sigma(x) = 4$

```
int x
init{
    if
    :: x = 1
    :: x = 2; x = x + 2
    fi
    assert (x == 1 || x == 4)
}
```

1.4*: `y = 1 (x := 1 par (x := 2; x + 2))` will result in a state σ where either $\sigma(x) \in \{1, 3, 4\}$

```
int x,y
proctype left(){
    x = 1
}
proctype right(){
    x = 2
    x = x + 2
}
init{
    y = 1
    atomic{
        run left()
        run right()
    }
    // wait for processes to terminate
    _nr_pr == 1
    assert(x == 1 || x == 3 || x == 4)
}
```

5 Linear temporal logic

5.1 Properties

A **finite transition system** is a tuple $(\Gamma, \Sigma_I, \rightarrow)$

- Γ: a finite set of configurations
- $\sigma_I \in \Gamma$: an initial congiguration
- $\rightarrow \subseteq \Gamma \times \Gamma$: a transition relation
- ommiting terminal configurations

$\gamma \in \Gamma^\omega$ (Γ^ω set of infinite sequences) is a **computation** of a transition system if:

- $\gamma_{[0]} = \sigma_I$
- $\gamma_{[i]} \rightarrow \gamma_{[i+1]}$ (for all $i \geq 0$)

Linear-time property P over Γ: subset of Γ^ω
Atomic proposition AP: proposition containing no logical connectives
Labling function: $L : \Gamma \mapsto \mathcal{P}(AP)$, we call $L(\sigma)$ an **abstract state**
Trace $t \in \mathcal{P}(AP)^\omega$: Abstraction of a computation, $t = L(\gamma_{[0]})L(\gamma_{[1]})L(\gamma_{[2]})...$ for a transition system
Liveness property

- Intuition: if the thing has not happened yet, it could happen in the future
- A liveness property does not rule out any prefix
- Every finite prefix can be extended to an infinte sequence that is in P
- Liveness properties are violated in infinite time

Safety property

- Intuition: Something bad is never allowed to happen (and can't be fixed)
- If in a finite prefix a property is violated, all sequences with this prefix violate the property
- Liveness properties are violated in finite time

5.1.1 Example

maybe will be added some time

5.2 Operators

For a trace $t \in \mathcal{P}(AP)^\omega$

$t \vDash p$	iff $p \in t_{[0]}$	now
$t \vDash \neg\phi$	iff not $\phi \in t_{[0]}$	not now
$t \vDash \phi \wedge \psi$	iff $t \vDash \phi$ and $t \vDash \psi$	and
$t \vDash \phi \,\mathrm{U}\, \psi$	iff $\exists k \geq 0$ with $t_{\geq k} \vDash \psi$ and $t_{\geq j} \vDash \phi$ $\forall 0 \leq j < k$	until
$t \vDash \bigcirc\phi$	iff $t_{[1]} \vDash \phi$	next
$t \vDash \Diamond\phi$	$\equiv true \,\mathrm{U}\, \psi$	eventually
$t \vDash \Box\phi$	iff $\equiv \neg(\Diamond(\neg\phi))$	always (from now)

5.2.1 Example

maybe will be added some time