



TARTU ÜLIKOOL

Monaadid ja kõrvalefektid

Danel Ahman

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine
 - mälu kasutus

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine
 - mälu kasutus
 - muudetavad (ingl *mutable*) muutujad

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine
 - mälu kasutus
 - muudetavad (ingl *mutable*) muutujad
 - mitteterminism

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine
 - mälu kasutus
 - muudetavad (ingl *mutable*) muutujad
 - mitteterminism
 - tõenäosuslikud arvutused

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine
 - mälu kasutus
 - muudetavad (ingl *mutable*) muutujad
 - mitteterminism
 - tõenäosuslikud arvutused
 - ...

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine ($f : \text{Nat} \rightarrow ???$)
 - mälu kasutus
 - muudetavad (ingl *mutable*) muutujad
 - mitteterminism
 - tõenäosuslikud arvutused
 - ...

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine ($f : \text{Nat} \rightarrow ???$)
 - mälu kasutus ($f : \text{Nat} \rightarrow ???$)
 - muudetavad (ingl *mutable*) muutujad
 - mitteterminism
 - tõenäosuslikud arvutused
 - ...

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine ($f : \text{Nat} \rightarrow ???$)
 - mälu kasutus ($f : \text{Nat} \rightarrow ???$)
 - muudetavad (ingl *mutable*) muutujad ($f : \text{Nat} \rightarrow ???$)
 - mittedeterminism
 - tõenäosuslikud arvutused
 - ...

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine ($f : \text{Nat} \rightarrow ???$)
 - mälu kasutus ($f : \text{Nat} \rightarrow ???$)
 - muudetavad (ingl *mutable*) muutujad ($f : \text{Nat} \rightarrow ???$)
 - mittedeterminism ($f : \text{Nat} \rightarrow ???$)
 - tõenäosuslikud arvutused
 - ...

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine ($f : \text{Nat} \rightarrow ???$)
 - mälu kasutus ($f : \text{Nat} \rightarrow ???$)
 - muudetavad (ingl *mutable*) muutujad ($f : \text{Nat} \rightarrow ???$)
 - mitteterminism ($f : \text{Nat} \rightarrow ???$)
 - tõenäosuslikud arvutused ($f : \text{Nat} \rightarrow ???$)
 - ...

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine ($f : \text{Nat} \rightarrow ???$)
 - mälu kasutus ($f : \text{Nat} \rightarrow ???$)
 - muudetavad (ingl *mutable*) muutujad ($f : \text{Nat} \rightarrow ???$)
 - mittedeterminism ($f : \text{Nat} \rightarrow ???$)
 - tõenäosuslikud arvutused ($f : \text{Nat} \rightarrow ???$)
 - ...

monaadid

- Vaikimisi on Idrise funktsioonid **puhtad, ilma kõrvalefektideta, matemaatilised**
 - $f : \text{Nat} \rightarrow \text{Bool}$
- Samas on päris programmides aga tihti kasulikud ka **kõrvalefektid**
 - sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
 - erandid (ingl *exceptions*) ja nende töötlemine ($f : \text{Nat} \rightarrow ???$)
 - mälu kasutus ($f : \text{Nat} \rightarrow ???$)
 - muudetavad (ingl *mutable*) muutujad ($f : \text{Nat} \rightarrow ???$)
 - mittedeterminism ($f : \text{Nat} \rightarrow ???$)
 - tõenäosuslikud arvutused ($f : \text{Nat} \rightarrow ???$)
 - ...



monaadid

- Aritmeetilised avaldised

```
infixl 10 :+:, :-:  
infixl 11 :*:
```

```
data Expr = Num Int  
          | ( :+: ) Expr Expr  
          | ( :- ) Expr Expr  
          | ( :* ) Expr Expr
```

- Aritmeetilised avaldised

```
infixl 10 :+:, :-:  
infixl 11 :*:
```

```
data Expr = Num Int  
          | (:+:) Expr Expr  
          | (:-:) Expr Expr  
          | (:*:) Expr Expr
```

- Näited

```
exp1 : Expr  
exp1 = Num 1 :+: Num 2
```

```
exp2 : Expr  
exp2 = Num 2 :* (Num 4 :- Num 1)
```


- Aritmeetiliste avaldiste **väärtustamine** (nende semantika arvutamine)

`eval : Expr -> Int`

`eval (Num i) = i`

`eval (e1 :+: e2) = eval e1 + eval e2`

`eval (e1 :-: e2) = eval e1 - eval e2`

`eval (e1 :*: e2) = eval e1 * eval e2`

- Aritmeetiliste avaldiste **väärtustamine** (nende semantika arvutamine)

`eval : Expr -> Int`

`eval (Num i) = i`

`eval (e1 :+: e2) = eval e1 + eval e2`

`eval (e1 :-: e2) = eval e1 - eval e2`

`eval (e1 :*: e2) = eval e1 * eval e2`

- Näited

`exp1 : Expr`

`exp1 = Num 1 :+: Num 2`

- Aritmeetiliste avaldiste **väärtustamine** (nende semantika arvutamine)

`eval : Expr -> Int`

`eval (Num i) = i`

`eval (e1 :+: e2) = eval e1 + eval e2`

`eval (e1 :-: e2) = eval e1 - eval e2`

`eval (e1 :*: e2) = eval e1 * eval e2`

- Väljund

Loeng15a> eval exp1

3

- Näited

`exp1 : Expr`

`exp1 = Num 1 :+: Num 2`

- Aritmeetiliste avaldiste **väärtustamine** (nende semantika arvutamine)

`eval : Expr -> Int`

`eval (Num i) = i`

`eval (e1 :+: e2) = eval e1 + eval e2`

`eval (e1 :-: e2) = eval e1 - eval e2`

`eval (e1 :*: e2) = eval e1 * eval e2`

- Väljund

Loeng15a> eval exp1

3

- Näited

`exp1 : Expr`

`exp1 = Num 1 :+: Num 2`

`exp2 : Expr`

`exp2 = Num 2 :*: (Num 4 :-: Num 1)`

- Aritmeetiliste avaldiste **väärtustamine** (nende semantika arvutamine)

`eval : Expr -> Int`

`eval (Num i) = i`

`eval (e1 :+: e2) = eval e1 + eval e2`

`eval (e1 :-: e2) = eval e1 - eval e2`

`eval (e1 :*: e2) = eval e1 * eval e2`

- Väljund

Loeng15a> eval exp1

3

Loeng15a> eval exp2

6

- Näited

`exp1 : Expr`

`exp1 = Num 1 :+: Num 2`

`exp2 : Expr`

`exp2 = Num 2 :*: (Num 4 :-: Num 1)`

- Aritmeetilised avaldised (koos jagamisega)

```
infixl 10 :+:, :-:  
infixl 11 :*: , :/:
```

```
data Expr = Num Int  
          | ( :+: ) Expr Expr  
          | ( :- ) Expr Expr  
          | ( :* ) Expr Expr  
          | ( :/ ) Expr Expr
```

- Aritmeetilised avaldised (koos jagamisega)

```
infixl 10 :+:, :-:  
infixl 11 :*:, :/:
```

```
data Expr = Num Int  
          | ( :+: ) Expr Expr  
          | ( :- ) Expr Expr  
          | ( :* ) Expr Expr  
          | ( :/ ) Expr Expr
```

- Näited

```
exp1 : Expr  
exp1 = Num 1 :+: Num 2
```

```
exp2 : Expr  
exp2 = Num 2 :* (Num 4 :- Num 1)
```

- Aritmeetilised avaldised (koos jagamisega)

```
infixl 10 :+:, :-:  
infixl 11 :*, :/:
```

```
data Expr = Num Int  
          | (:+:) Expr Expr  
          | (:~:) Expr Expr  
          | (:*) Expr Expr  
          | (:/) Expr Expr
```

- Näited

```
exp1 : Expr  
exp1 = Num 1 :+: Num 2
```

```
exp2 : Expr  
exp2 = Num 2 :* (Num 4 :- Num 1)
```

```
exp3 : Expr  
exp3 = Num 4 :* Num 3 :/ Num 2
```

```
exp4 : Expr  
exp4 = Num 4 :* Num 3 :/ (Num 2 :- Num 2)
```


- Aritmeetiliste avaldiste **väärtustamine** (koos jagamisega)

```
eval1 : Expr -> Int
```

```
eval1 (Num i) = i
```

```
eval1 (e1 :+: e2) = eval1 e1 + eval1 e2
```

```
eval1 (e1 :-: e2) = eval1 e1 - eval1 e2
```

```
eval1 (e1 :*: e2) = eval1 e1 * eval1 e2
```

```
eval1 (e1 :/: e2) = eval1 e1 `div` eval1 e2
```

- Aritmeetiliste avaldiste **väärtustamine** (koos jagamisega)

```
eval1 : Expr -> Int
```

```
eval1 (Num i) = i
```

```
eval1 (e1 :+: e2) = eval1 e1 + eval1 e2
```

```
eval1 (e1 :-: e2) = eval1 e1 - eval1 e2
```

```
eval1 (e1 :*: e2) = eval1 e1 * eval1 e2
```

```
eval1 (e1 :/: e2) = eval1 e1 `div` eval1 e2
```

- Näited

```
exp3 : Expr
```

```
exp3 = Num 4 :*: Num 3 :/: Num 2
```

- Aritmeetiliste avaldiste **väärtustamine** (koos jagamisega)

```
eval1 : Expr -> Int
```

```
eval1 (Num i) = i
```

```
eval1 (e1 :+: e2) = eval1 e1 + eval1 e2
```

```
eval1 (e1 :-: e2) = eval1 e1 - eval1 e2
```

```
eval1 (e1 :*: e2) = eval1 e1 * eval1 e2
```

```
eval1 (e1 :/: e2) = eval1 e1 `div` eval1 e2
```

- Väljund

```
Loeng15b> eval1 exp3
```

6

- Näited

```
exp3 : Expr
```

```
exp3 = Num 4 :*: Num 3 :/: Num 2
```

- Aritmeetiliste avaldiste **väärtustamine** (koos jagamisega)

```
eval1 : Expr -> Int
```

```
eval1 (Num i) = i
```

```
eval1 (e1 :+: e2) = eval1 e1 + eval1 e2
```

```
eval1 (e1 :-: e2) = eval1 e1 - eval1 e2
```

```
eval1 (e1 **: e2) = eval1 e1 * eval1 e2
```

```
eval1 (e1 :/: e2) = eval1 e1 `div` eval1 e2
```

- Väljund

```
Loeng15b> eval1 exp3
```

6

- Näited

```
exp3 : Expr
```

```
exp3 = Num 4 **: Num 3 :/: Num 2
```

```
exp4 : Expr
```

```
exp4 = Num 4 **: Num 3 :/: (Num 2 :-: Num 2)
```

- Aritmeetiliste avaldiste **väärtustamine** (koos jagamisega)

```
eval1 : Expr -> Int
```

```
eval1 (Num i) = i
```

```
eval1 (e1 :+: e2) = eval1 e1 + eval1 e2
```

```
eval1 (e1 :-: e2) = eval1 e1 - eval1 e2
```

```
eval1 (e1 **: e2) = eval1 e1 * eval1 e2
```

```
eval1 (e1 :/: e2) = eval1 e1 `div` eval1 e2
```

- Väljund

```
Loeng15b> eval1 exp3
```

```
6
```

```
Loeng15b> eval1 exp4
```

```
let False = True in
```

```
prim__div_Int 12 0
```

- Näited

```
exp3 : Expr
```

```
exp3 = Num 4 **: Num 3 :/: Num 2
```

```
exp4 : Expr
```

```
exp4 = Num 4 **: Num 3 :/: (Num 2 :-: Num 2)
```

- Aritmeetiliste avaldiste **väärtustamine** (koos jagamisega)

```
eval1 : Expr -> Int
```

```
eval1 (Num i) = i
```

```
eval1 (e1 :+: e2) = eval1 e1 + eval1 e2
```

```
eval1 (e1 :-: e2) = eval1 e1 - eval1 e2
```

```
eval1 (e1 **: e2) = eval1 e1 * eval1 e2
```

```
eval1 (e1 :/: e2) = eval1 e1 / eval1 e2
```

Viga, mida ei saa edasi töödelda!
(Error, mitte Exception)

- Väljund

```
Loeng15b> eval1 exp3
```

```
6
```

```
Loeng15b> eval1 exp4
```

```
let False = True in  
prim__div_Int 12 0
```

```
exp3 : Expr
```

```
exp3 = Num 4 **: Num 3 :/: Num 2
```

```
exp4 : Expr
```

```
exp4 = Num 4 **: Num 3 :/: (Num 2 :-: Num 2)
```

- Aritmeetiliste avaldiste väärtustamine (koos jagamisega)

```
eval1 : Expr -> Int
```

```
eval1 (Num i) = i
```

```
eval1 (e1 :+: e2) = eval1 e1 + eval1 e2
```

```
eval1 (e1 :-: e2) = eval1 e1 - eval1 e2
```

```
eval1 (e1 **: e2) = eval1 e1 * eval1 e2
```

```
eval1 (e1 :/: e2) = eval1 e1 / eval1 e2
```

```
Integral Integer where
```

```
div x y
```

```
= case y == 0 of
```

```
False => prim_div_Integer x y
```

- Väljund

```
Loeng15b> eval1 exp3
```

```
6
```

```
Loeng15b> eval1 exp4
```

```
let False = True in  
prim_div_Int 12 0
```

Viga, mida ei saa edasi töödelda!
(Error, mitte Exception)

```
exp3 : Expr
```

```
exp3 = Num 4 **: Num 3 :/: Num 2
```

```
exp4 : Expr
```

```
exp4 = Num 4 **: Num 3 :/: (Num 2 :-: Num 2)
```


- **Osaliste väärtuste** andmetüüp (isomorfne Maybe andmetüübiga standardteegis)

```
data Option a = None  
              | Some a
```

- **Osaliste väärtuste** andmetüüp (isomorfne Maybe andmetüübiga standardteegis)

```
data Option a = None
               | Some a
```

- Aritmeetiliste avaldiste **väärtustamine** (koos jagamise ja **eranditega**)

- **Osaliste väärtuste** andmetüüp (isomorfne Maybe andmetüübiga standardteegis)

```
data Option a = None
               | Some a
```

- Aritmeetiliste avaldiste **väärtustamine** (koos jagamise ja **eranditega**)

```
eval2 : Expr -> Option Int
```

- **Osaliste väärtuste** andmetüüp (isomorfne Maybe andmetüübiga standardteegis)

```
data Option a = None  
              | Some a
```

- Aritmeetiliste avaldiste **väärtustamine** (koos jagamise ja **eranditega**)

```
eval2 : Expr -> Option Int
```

```
eval2 (Num i) = Some i
```

- **Osaliste väärtuste** andmetüüp (isomorfne Maybe andmetüübiga standardteegis)

```
data Option a = None
               | Some a
```

- Aritmeetiliste avaldiste **väärtustamine** (koos jagamise ja **eranditega**)

```
eval2 : Expr -> Option Int
```

```
eval2 (Num i) = Some i
```

```
eval2 (e1 :+: e2) = case eval2 e1 of
                     None      => None
                     Some v1 => case eval2 e2 of
                                   None      => None
                                   Some v2 => Some (v1 + v2)
```

```
eval2 (e1 :-: e2) = ...
```

```
eval2 (e1 :*: e2) = ...
```

```
eval2 (e1 :/: e2) = ...
```

- **Osaliste väärtuste** andmetüüp (isomorfne Maybe andmetüübiga standardteegis)

```
data Option a = None
               | Some a
```

- Aritmeetiliste avaldiste **väärtustamine** (koos jagamise ja **eranditega**)

```
eval2 : Expr -> Option Int
```

```
...
```

- **Osaliste väärtuste** andmetüüp (isomorfne Maybe andmetüübiga standardteegis)

```
data Option a = None
               | Some a
```

- Aritmeetiliste avaldiste **väärtustamine** (koos jagamise ja **eranditega**)

```
eval2 : Expr -> Option Int
```

```
...
```

```
eval2 (e1 :/: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
      None    => None
      Some v2 => if v2 == 0
                  then None
                  else Some (v1 `div` v2)
```

- **Osaliste väärtuste** andmetüüp (isomorfne Maybe andmetüübiga standardteegis)

```
data Option a = None
               | Some a
```

- Aritmeetiliste avaldiste **väärtust**

```
eval2 : Expr -> Option Int
```

```
...
```

```
eval2 (e1 :/: e2) =
```

```
  case eval2 e1 of
```

```
    None    => None
```

```
    Some v1 => case eval2 e2 of
```

```
      None    => None
```

```
      Some v2 => if v2 == 0
```

```
        then None
```

```
        else Some (v1 `div` v2)
```

```
exp3 : Expr
```

```
exp3 = Num 4 :*: Num 3 :/: Num 2
```

```
Loeng15b> eval2 exp3
```

```
Some 6
```

```
exp4 : Expr
```

```
exp4 = Num 4 :*: Num 3 :/: (Num 2 :-: Num 2)
```

```
Loeng15b> eval2 exp4
```

```
None
```


- **Ilma eranditeta** väärtustaja

```
eval1 : Expr -> Int
```

```
eval1 (Num i) = i
```

```
eval1 (e1 :+: e2) = eval1 e1 + eval1 e2
```

```
eval1 (e1 :-: e2) = eval1 e1 - eval1 e2
```

```
eval1 (e1 ::: e2) = eval1 e1 * eval1 e2
```

```
eval1 (e1 :/: e2) = eval1 e1 `div` eval1 e2
```

- Ilma eranditeta väärtustaja

```
eval1 : Expr -> Int
```

```
eval1 (Num i) = i
```

```
eval1 (e1 :+: e2) = eval1 e1 + eval1 e2
```

```
eval1 (e1 :-: e2) = eval1 e1 - eval1 e2
```

```
eval1 (e1 :*: e2) = eval1 e1 * eval1 e2
```

```
eval1 (e1 :/: e2) = eval1 e1 `div` eval1 e2
```

- Eranditega väärtustaja

```
eval2 : Expr -> Option Int
```

```
eval2 (Num i) = Some i
```

```
eval2 (e1 :+: e2) =
```

```
  case eval2 e1 of
```

```
    None    => None
```

```
    Some v1 => case eval2 e2 of
```

```
      None    => None
```

```
      Some v2 => Some (v1 + v2)
```

```
eval2 (e1 :-: e2) =
```

```
  case eval2 e1 of
```

```
    None    => None
```

```
    Some v1 => case eval2 e2 of
```

```
      None    => None
```

```
      Some v2 => Some (v1 - v2)
```

```
eval2 (e1 :*: e2) =
```

```
  case eval2 e1 of
```

```
    None    => None
```

```
    Some v1 => case eval2 e2 of
```

```
      None    => None
```

```
      Some v2 => Some (v1 * v2)
```

```
eval2 (e1 :/: e2) =
```

```
  case eval2 e1 of
```

```
    None    => None
```

```
    Some v1 => case eval2 e2 of
```

```
      None    => None
```

```
      Some v2 =>
```

```
        if v2 == 0 then None else Some (v1 `div` v2)
```

- **Eranditega väärtustaja**

```
eval2 : Expr -> Option Int

eval2 (Num i) = Some i

eval2 (e1 :+: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 + v2)

eval2 (e1 :-: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 - v2)

eval2 (e1 :*: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 * v2)

eval2 (e1 :/: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 =>
                    if v2 == 0 then None else Some (v1 `div` v2)
```

- Korduvad mustrid:

- Eranditega väärtustaja

```
eval2 : Expr -> Option Int

eval2 (Num i) = Some i

eval2 (e1 :+: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 + v2)

eval2 (e1 :-: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 - v2)

eval2 (e1 :*: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 * v2)

eval2 (e1 :/: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 =>
                    if v2 == 0 then None else Some (v1 `div` v2)
```

- Korduvad mustrid:

- väärtuse tagastamine ilma erandita

- Eranditega väärtustaja

```
eval2 : Expr -> Option Int
eval2 (Num i) = Some i

eval2 (e1 :+: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 + v2)

eval2 (e1 :-: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 - v2)

eval2 (e1 :*: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 * v2)

eval2 (e1 :/: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 =>
                    if v2 == 0 then None else Some (v1 `div` v2)
```

- Korduvad mustrid:

- väärtuse tagastamine ilma erandita

- Eranditega väärtustaja

```
eval2 : Expr -> Option Int

eval2 (Num i) = Some i

eval2 (e1 :+: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 + v2)

eval2 (e1 :-: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 - v2)

eval2 (e1 :*: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 * v2)

eval2 (e1 :/: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 =>
                    if v2 == 0 then None else Some (v1 `div` v2)
```

- Korduvad mustrid:

- väärtuse tagastamine ilma erandita
- alamavaldiste väärtustamine toimub järjestikku

- Eranditega väärtustaja

```
eval2 : Expr -> Option Int
eval2 (Num i) = Some i

eval2 (e1 :+: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 => Some (v1 + v2)

eval2 (e1 :-: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 => Some (v1 - v2)

eval2 (e1 :*: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 => Some (v1 * v2)

eval2 (e1 :/: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 =>
        if v2 == 0 then None else Some (v1 `div` v2)
```

- Korduvad mustrid:

- väärtuse tagastamine ilma erandita
- alamavaldiste väärtustamine toimub järjestikku

- Eranditega väärtustaja

```
eval2 : Expr -> Option Int

eval2 (Num i) = Some i

eval2 (e1 :+: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 + v2)

eval2 (e1 :-: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 - v2)

eval2 (e1 :*: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 * v2)

eval2 (e1 :/: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 =>
                    if v2 == 0 then None else Some (v1 `div` v2)
```

- Korduvad mustrid:
 - väärtuse tagastamine ilma erandita
 - alamavaldiste väärtustamine toimub järjestikku
 - kui üks ebaõnnestub, siis ebaõnnestub kogu avaldise väärtustamine

- Eranditega väärtustaja

```
eval2 : Expr -> Option Int
eval2 (Num i) = Some i

eval2 (e1 :+: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 => Some (v1 + v2)

eval2 (e1 :-: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 => Some (v1 - v2)

eval2 (e1 :*: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 => Some (v1 * v2)

eval2 (e1 :/: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 =>
        if v2 == 0 then None else Some (v1 `div` v2)
```

- **Korduvad mustrid:**

- väärtuse tagastamine **ilma erandita**
- alamavaldiste **väärtustamine toimub järjestikku**
- kui üks ebaõnnestub, siis **ebaõnnestub kogu avaldise väärtustamine**

- **Eranditega väärtustaja**

```
eval2 : Expr -> Option Int

eval2 (Num i) = Some i

eval2 (e1 :+: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 + v2)

eval2 (e1 :-: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 - v2)

eval2 (e1 :*: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 * v2)

eval2 (e1 :/: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 =>
                    if v2 == 0 then None else Some (v1 `div` v2)
```

- **Korduvad mustrid:**

- väärtuse tagastamine **ilma erandita**
- alamavaldiste **väärtustamine toimub järjestikku**
- kui üks ebaõnnestub, siis **ebaõnnestub kogu avaldise väärtustamine**
- õnnestumise korral **antakse tulemus edasi järgnevale arvutusele**

- **Eranditega väärtustaja**

```
eval2 : Expr -> Option Int
eval2 (Num i) = Some i

eval2 (e1 :+: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 => Some (v1 + v2)

eval2 (e1 :-: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 => Some (v1 - v2)

eval2 (e1 :*: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 => Some (v1 * v2)

eval2 (e1 :/: e2) =
  case eval2 e1 of
    None => None
    Some v1 => case eval2 e2 of
      None => None
      Some v2 =>
        if v2 == 0 then None else Some (v1 `div` v2)
```

- **Korduvad mustrid:**

- väärtuse tagastamine **ilma erandita**
- alamavaldiste **väärtustamine toimub järjestikku**
- kui üks ebaõnnestub, siis **ebaõnnestub kogu avaldise väärtustamine**
- õnnestumise korral **antakse tulemus edasi järgnevale arvutusele**

- **Eranditega väärtustaja**

```
eval2 : Expr -> Option Int

eval2 (Num i) = Some i

eval2 (e1 :+: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 + v2)

eval2 (e1 :-: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 - v2)

eval2 (e1 :*: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 * v2)

eval2 (e1 :/: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 =>
                    if v2 == 0 then None else Some (v1 `div` v2)
```

- **Korduvad mustrid:**

- väärtuse tagastamine **ilma erandita**
- alamavaldiste **väärtustamine toimub järjestikku**
- kui üks ebaõnnestub, siis **ebaõnnestub kogu avaldise väärtustamine**
- õnnestumise korral **antakse tulemus edasi järgnevale arvutusele**
- vajadusel **erandi tõstatamine**

- **Eranditega väärtustaja**

```
eval2 : Expr -> Option Int
eval2 (Num i) = Some i

eval2 (e1 :+: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 + v2)

eval2 (e1 :-: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 - v2)

eval2 (e1 :*: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 => Some (v1 * v2)

eval2 (e1 :/: e2) =
  case eval2 e1 of
    None    => None
    Some v1 => case eval2 e2 of
                  None    => None
                  Some v2 =>
                    if v2 == 0 then None else Some (v1 `div` v2)
```


- **Väärtuse tagastamine** ilma erandit tõstatamata

```
oReturn : a -> Option a  
oReturn x = Some x
```

- **Väärtuse tagastamine ilma erandit tõstatamata**

```
oReturn : a -> Option a  
oReturn x = Some x
```

- Eranditega arvutuste järjestikku jooksumine

```
oBind : Option a -> (a -> Option b) -> Option b
```

```
oBind comp f = case comp of  
    None    => None  
    Some x => f x
```

- **Väärtuse tagastamine ilma erandit tõstatamata**

```
oReturn : a -> Option a  
oReturn x = Some x
```

- Eranditega arvutuste järjestikku jooksumine

```
oBind : Option a -> (a -> Option b) -> Option b
```

```
oBind comp f = case comp of  
    None    => None  
    Some x => f x
```

- Vajadusel erandi tõstatamine

```
oThrow : Option a  
oThrow = None
```

```
oReturn : a -> Option a
```

```
oBind : Option a -> (a -> Option b) -> Option b
```

```
oThrow : Option a
```

Näide 3: Avaldiste väärtustamine (abstraktsemalt)



```
eval3 : Expr -> Option Int
```

```
eval3 (Num i) = oReturn i
```

```
eval3 (e1 :+: e2) = eval3 e1 `oBind` \ v1 =>  
                    eval3 e2 `oBind` \ v2 =>  
                    oReturn (v1 + v2)
```

```
eval3 (e1 :-: e2) = eval3 e1 `oBind` \ v1 =>  
                    eval3 e2 `oBind` \ v2 =>  
                    oReturn (v1 - v2)
```

```
eval3 (e1 :*: e2) = eval3 e1 `oBind` \ v1 =>  
                    eval3 e2 `oBind` \ v2 =>  
                    oReturn (v1 * v2)
```

```
eval3 (e1 :/: e2) = eval3 e1 `oBind` \ v1 =>  
                    eval3 e2 `oBind` \ v2 =>  
                    if v2 == 0 then oThrow else oReturn (v1 `div` v2)
```

```
oReturn : a -> Option a
```

```
oBind : Option a -> (a -> Option b) -> Option b
```

```
oThrow : Option a
```

Option tüüp on monaad!



- **Option tüüp** kirjeldab potentsiaalselt erandeid tõstatavaid arvutusi
 - kus saab ilma erandita **väärtusi tagastada** (**oReturn**)
 - mida saab **järjestikku käivitada** (**oBind**)
 - kus saab **vajadusel erandit tõstatada** (**oThrow**)

- **Option tüüp** kirjeldab potentsiaalselt erandeid tõstatavaid arvutusi
 - kus saab ilma erandita **väärtusi tagastada** (**oReturn**)
 - mida saab **järjestikku käivitada** (**oBind**)
 - kus saab **vajadusel erandit tõstatada** (**oThrow**)
- **Monaadid** kirjeldavad üldisemalt arvutusi
 - kus saab **väärtusi tagastada** (**return**)
 - mida saab **järjestikku käivitada** (**bind**)
 - kus võib saada **käivitada spetsiifilisi operatsioone**

- **Monaadid** üldiselt (Idrise tüübiklassina)

```
interface Monaad (m : Type -> Type) where  
  return : a -> m a  
  bind    : m a -> (a -> m b) -> m b
```

(>>= ja do-notatsioonid)

- **Monaadid** üldiselt (Idrise tüübiklassina)

```
interface Monaad (m : Type -> Type) where
  return : a -> m a
  bind    : m a -> (a -> m b) -> m b
```

(>>= ja do-notatsioonid)

- + **võrduslikud seadused** returni ja bindi kohta (mida FP keeltes tihti ei kontrollita)

- $(\text{return } x) \gg= f \approx f \ x$
- $c \gg= (\lambda x \Rightarrow \text{return } x) \approx c$
- $(c \gg= f) \gg= g \approx c \gg= (\lambda x \Rightarrow (f \ x) \gg= g)$

- **Monaadid** üldiselt (Idrise tüübiklassina)

```
interface Monaad (m : Type -> Type) where
  return : a -> m a
  bind    : m a -> (a -> m b) -> m b
```

(>>= ja do-notatsioonid)

- + **võrduslikud seadused** returni ja bindi kohta (mida FP keeltes tihti ei kontrollita)

- $(\text{return } x) \gg= f \approx f \ x$
- $c \gg= (\lambda x \Rightarrow \text{return } x) \approx c$
- $(c \gg= f) \gg= g \approx c \gg= (\lambda x \Rightarrow (f \ x) \gg= g)$

- Option monaad eranditega programmeerimiseks

```
Monaad Option where
  return = oReturn
  bind    = oBind
```


- Idrises on ka **sisseehitatud monaadide tüübiklass**

```
interface Functor f where  
  map : (a -> b) -> f a -> f b
```

```
interface Functor f => Applicative f where  
  pure : a -> f a  
  (<*>) : f (a -> b) -> f a -> f b
```

```
interface Applicative m => Monad m where  
  (>>=) : m a -> (a -> m b) -> m b
```

- Idrises on ka **sisseehitatud monaadide tüübiklass**

```
interface Functor f where  
  map : (a -> b) -> f a -> f b
```

```
interface Functor f => Applicative f where  
  pure : a -> f a  
  (<*>) : f (a -> b) -> f a -> f b
```

```
interface Applicative m => Monad m where
```

← sisseehitatud monaadide tüübiklass

```
  (>>=) : m a -> (a -> m b) -> m b
```

- Idrises on ka **sisseehitatud monaadide tüübiklass**

```
interface Functor f where  
  map : (a -> b) -> f a -> f b
```

```
interface Functor f => Applicative f where  
  pure : a -> f a  
  (<*>) : f (a -> b) -> f a -> f b
```

```
interface Applicative m => Monad m where
```

← sisseehitatud monaadide tüübiklass

```
  (>>=) : m a -> (a -> m b) -> m b
```

- Niimoodi mitmekihiliselt monaadide konstrueerimine on **tihti liiga kohmakas!**

- Idrises on ka **sisseehitatud monaadide tüübiklass**

```
interface Functor f where
  map : (a -> b) -> f a -> f b
```

```
interface Functor f => Applicative f where
  pure : a -> f a
  (<*>) : f (a -> b) -> f a -> f b
```

```
interface Applicative m => Monad m where ← sisseehitatud monaadide tüübiklass
  (>>=) : m a -> (a -> m b) -> m b
```

- Niimoodi mitmekihiliselt monaadide konstrueerimine on **tihti liiga kohmakas!**
- Selle nädala (boonus)ülesannetes defineerime monaadid **oma tüübiklassi abil**

```
Monaad m => Functor m      where ...
Monaad m => Applicative m  where ...
Monaad m => Monad m        where ...
```

```
interface Monaad (0 m : Type -> Type) where
  return : a -> m a
  bind   : m a -> (a -> m b) -> m b
```


- Kuna
 - näitasime, et Option on Monaad
 - iga Monaad on Monad
 - iga Monad'i puhul saame kasutada **do-notatsiooni!**
 - täpselt nagu IO programmides

```
eval4 : Expr -> Option Int
```

```
eval4 (Num i) = return i
```

```
eval4 (e1 :+: e2) = do
  v1 <- eval4 e1
  v2 <- eval4 e2
  return (v1 + v2)
```

```
eval4 (e1 :-: e2) = do
  v1 <- eval4 e1
  v2 <- eval4 e2
  return (v1 - v2)
```

```
eval4 (e1 :*: e2) = do
  v1 <- eval4 e1
  v2 <- eval4 e2
  return (v1 * v2)
```

```
eval4 (e1 :/: e2) = do
  v1 <- eval4 e1
  v2 <- eval4 e2
  if v2 == 0 then throw else return (v1 `div` v2)
```

- Kuna

- näitasime, et Option on Monaad
- iga Monaad on Monad
- iga Monad'i puhul saame kasutada **do-notatsiooni!**
- täpselt nagu IO programmides

- Saame modelleerida ka **Java/Pythoni-stiilis erandite töötlemist**

```
tryCatch : Option a -> Option a -> Option a
```

```
tryCatch comp errorHandler = case comp of  
    None    => errorHandler  
    Some x => return x
```

- Saame modelleerida ka **Java/Pythoni-stiilis erandite töötlemist**

```
tryCatch : Option a -> Option a -> Option a
```

```
tryCatch comp errorHandler = case comp of  
    None    => errorHandler  
    Some x => return x
```

- Näiteks

```
exp4 : Expr  
exp4 = Num 4 :*: Num 3 :/: (Num 2 :-: Num 2)
```

```
Loeng15b> eval2 exp4
```

```
None
```

```
Loeng15b> tryCatch (eval2 exp4) (return 42)
```

```
Some 42
```

- Saame modelleerida ka **Java/Pythoni-stiilis erandite töötlemist**

```
tryCatch : Option a -> Option a -> Option a
```

```
tryCatch comp errorHandler
```

- Näiteks

```
exp4 : Expr  
exp4 = Num 4 :: Num 3 ::
```

```
Loeng15b> eval2 exp4
```

```
None
```

```
Loeng15b> tryCatch (eval2
```

```
Some 42
```

Option on erijuht **rohkemate eranditega monaadist**

```
data Exc e a = MkExc (Either e a)
```

```
eReturn : a -> Exc e a  
eReturn x = MkExc (Right x)
```

```
eBind : Exc e a -> (a -> Exc e b) -> Exc e b  
eBind comp f = case comp of  
    MkExc (Left e)   => MkExc (Left e)  
    MkExc (Right x) => f x
```


- Vaatame jälle **ilma jagamiseta** (ilma eranditeta) aritmeetilisi avaldisi

```
infixl 10 :+:, :-:  
infixl 11 :*:
```

```
data Expr = Num Int  
          | (:+:) Expr Expr  
          | (:-:) Expr Expr  
          | (:*:) Expr Expr
```

- Vaatame jälle **ilma jagamiseta** (ilma eranditeta) aritmeetilisi avaldisi

```
infixl 10 :+:, :-:  
infixl 11 :*:
```

```
data Expr = Num Int  
          | (:+:) Expr Expr  
          | (:~:) Expr Expr  
          | (:*:) Expr Expr
```

- Eesmärk: defineerida väärtustaja, mis **tagastab **ka** väärtustatud avaldiste arvu**
 - lihtne näide arvutuste/programmide instrumenteerimisest
 - teised sarnased näited: tegevuste logimine, autentimine, dünaamiline verifitseerimine

- Vaatame jälle **ilma jagamiseta** (ilma eranditeta) aritmeetilisi avaldisi

```
infixl 10 :+:, :-:  
infixl 11 :*:
```

```
data Expr = Num Int  
          | (:+:) Expr Expr  
          | (:~:) Expr Expr  
          | (:*:) Expr Expr
```

- Eesmärk: defineerida väärtustaja, mis **tagastab **ka** väärtustatud avaldiste arvu**
 - lihtne näide arvutuste/programmide instrumenteerimisest
 - teised sarnased näited: tegevuste logimine, autentimine, dünaamiline verifitseerimine
- Näitame, et selline väärtustamine on ka **näide monaadilistest arvutustest!**

- Aritmeetiliste avaldiste **väärtustamine (instrumenteeritult)**

Counter = Int

eval1 : Expr -> (Int, Counter)

- Aritmeetiliste avaldiste **väärtustamine** (instrumenteeritult)

```
Counter = Int
```

```
eval1 : Expr -> (Int, Counter)
```

```
eval1 (Num i) = (i, 0)
```

```
eval1 (e1 :+: e2) = case eval1 e1 of  
    (v1, c1) => case eval1 e2 of  
        (v2, c2) => (v1 + v2 , c1 + c2 + 1)
```

```
eval1 (e1 :-: e2) = case eval1 e1 of  
    (v1, c1) => case eval1 e2 of  
        (v2, c2) => (v1 - v2 , c1 + c2 + 1)
```

```
eval1 (e1 :*: e2) = case eval1 e1 of  
    (v1, c1) => case eval1 e2 of  
        (v2, c2) => (v1 * v2 , c1 + c2 + 1)
```

- Aritmeetiliste avaldiste **väärtustamine** (instrumenteeritult)

```
Counter = Int
```

```
eval1 : Expr -> (Int, Counter)
```

```
eval1 (Num i) = (i, 0)
```

```
eval1 (e1 :+: e2) = case eval1 e1 of  
    (v1, c1) => case  
        (v2, c2) =>
```

```
eval1 (e1 :-: e2) = case eval1 e1 of  
    (v1, c1) => case  
        (v2, c2) =>
```

```
eval1 (e1 :*: e2) = case eval1 e1 of  
    (v1, c1) => case eval1 e2 of  
        (v2, c2) => (v1 * v2 , c1 + c2 + 1)
```

```
exp1 : Expr  
exp1 = Num 1 :+: Num 2
```

```
exp2 : Expr  
exp2 = Num 2 :*: (Num 4 :-: Num 1)
```

```
Loeng15c> eval1 exp1  
(3, 1)
```

```
Loeng15c> eval1 exp2  
(6, 2)
```

- Aritmeetiliste avaldiste **väärtustamine (instrumenteeritult)**

```
Counter = Int
```

```
eval1 : Expr -> (Int, Counter)
```

```
eval1 (Num i) = (i, 0)
```

```
eval1 (e1 :+: e2) = case eval1 e1 of  
    (v1, c1) => case eval1 e2 of  
        (v2, c2) => (v1 + v2 , c1 + c2 + 1)
```

```
eval1 (e1 :-: e2) = case eval1 e1 of  
    (v1, c1) => case eval1 e2 of  
        (v2, c2) => (v1 - v2 , c1 + c2 + 1)
```

```
eval1 (e1 :*: e2) = case eval1 e1 of  
    (v1, c1) => case eval1 e2 of  
        (v2, c2) => (v1 * v2 , c1 + c2 + 1)
```

- Aritmeetiliste avaldiste **väärtustamine** (instrumenteeritult)

Counter = Int

```
eval1 : Expr -> (Int, Counter)
```

```
eval1 (Num i) = (i, 0)
```

```
eval1 (e1 :+: e2) = case eval1
                        (v1, c1) =
```

```
eval1 (e1 :-: e2) = case eval1 e1 of
  (v1,c1) => case eval1 e2 of
    (v2,c2) => (v1 - v2 , c1 + c2 + 1)
```

```
eval1 (e1 :: e2) = case eval1 e1 of
    (v1,c1) => case eval1 e2 of
        (v2,c2) => (v1 * v2 , c1 + c2 + 1)
```

- Samasugused **korduvad mustrid** nagu varem:
 - väärtuste tagastamine
 - arvutuste järjestikku käivitamine
 - instrumenteerimine (loendurile + I tegemine)

- Instrumenteeritud arvutuste tüüp

```
data Ctr a = MkCtr (a,Counter)
```

- Instrumenteeritud arvutuste tüüp

```
data Ctr a = MkCtr (a, Counter)
```

- **Väärtuse tagastamine** ilma avaldise väärtustamata (loendur on 0)

```
cReturn : a -> Ctr a  
cReturn x = MkCtr (x, 0)
```

- Instrumenteeritud arvutuste tüüp

```
data Ctr a = MkCtr (a, Counter)
```

- **Väärtuse tagastamine** ilma avaldisi väärtustamata (loendur on 0)

```
cReturn : a -> Ctr a  
cReturn x = MkCtr (x, 0)
```

- Instrumenteeritud **arvutuste järjestikku jooksumine** (loendurid liidetakse)

```
cBind : Ctr a -> (a -> Ctr b) -> Ctr b  
cBind comp f = case comp of  
    MkCtr (x, c1) => case (f x) of  
        MkCtr (y, c2) => MkCtr (y , c1 + c2)
```

- Instrumenteeritud arvutuste tüüp

```
data Ctr a = MkCtr (a, Counter)
```

- **Väärtuse tagastamine** ilma avaldisi väärtustamata (loendur on 0)

```
cReturn : a -> Ctr a  
cReturn x = MkCtr (x, 0)
```

- Instrumenteeritud **arvutuste järjestikku jooksumine** (loendurid liidetakse)

```
cBind : Ctr a -> (a -> Ctr b) -> Ctr b  
cBind comp f = case comp of  
    MkCtr (x, c1) => case (f x) of  
        MkCtr (y, c2) => MkCtr (y , c1 + c2)
```

- Avaldiste **väärtustamise arvu loendamine** (loendur on 1)

```
cCount : Ctr ()  
cCount = MkCtr ((), 1)
```

- Instrumenteeritud arvutuste tüüp

```
data Ctr a = MkCtr (a, Counter)
```

- **Väärtuse tagastamine** ilma avaldisi väärtustamata (loendur on 0)

```
cReturn : a -> Ctr a  
cReturn x = MkCtr (x, 0)
```

- Instrumenteeritud **arvutuste järjestikku jooksumine** (loendurid liidetakse)

```
cBind : Ctr a -> (a -> Ctr b) -> Ctr b  
cBind comp f = case comp of  
    MkCtr (x, c1) => case (f x) of  
        MkCtr (y, c2) => MkCtr (y , c1 + c2)
```

- Avaldiste **väärtustamise arvu loendamine** (loendur on 1)

```
cCount : Ctr ()  
cCount = MkCtr ((), 1)
```

```
Monaad Ctr where  
    return = cReturn  
    bind    = cBind
```

- Instrumenteeritud arvutuste tüüp

```
data Ctr a = MkCtr (a, Counter)
```

- **Väärtuse tagastamine** ilma avaldisi või

```
cReturn : a -> Ctr a  
cReturn x = MkCtr (x, 0)
```

- Instrumenteeritud arvutuste järjestik

```
cBind : Ctr a -> (a -> Ctr b) -> Ctr b  
cBind comp f = case comp of  
    MkCtr (x, c1) => case f x of  
        MkCtr (y, c2) => cBind comp f
```

- Avaldiste väärtustamise arvu loendamine

```
cCount : Ctr ()  
cCount = MkCtr ((), 1)
```

Counter ei pea olema ainult Int

Võib parametrizeerida suvalise Monoid-iga

```
interface Semigroup ty where  
    (<+>) : ty -> ty -> ty
```

```
interface Semigroup ty => Monoid ty where  
    neutral : ty
```

Saame nn üldise **kirjutajamonaadi**
(vt Idrise koodi)

- Instrumenteeritud arvutuste tüüp

```
data Ctr a = MkCtr (a, Counter)
```

- **Väärtuse tagastamine** ilma avaldisi väärtustamata (loendur on 0)

```
cReturn : a -> Ctr a  
cReturn x = MkCtr (x, 0)
```

- Instrumenteeritud **arvutuste järjestikku jooksumine** (loendurid liidetakse)

```
cBind : Ctr a -> (a -> Ctr b) -> Ctr b  
cBind comp f = case comp of  
    MkCtr (x, c1) => case (f x) of  
        MkCtr (y, c2) => MkCtr (y , c1 + c2)
```

- Avaldiste **väärtustamise arvu loendamine** (loendur on 1)

```
cCount : Ctr ()  
cCount = MkCtr ((), 1)
```

<pre>Monaad Ctr where return = cReturn bind = cBind</pre>
--

- Aritmeetiliste avaldiste **väärtustamine** (instrumenteeritult)

```
eval3 : Expr -> Ctr Int
```

```
eval3 (Num i) = return i
```

```
eval3 (e1 :+: e2) = do
  v1 <- eval3 e1
  v2 <- eval3 e2
  count
  return (v1 + v2)
```

```
eval3 (e1 :-: e2) = do
  v1 <- eval3 e1
  v2 <- eval3 e2
  count
  return (v1 - v2)
```

```
eval3 (e1 :*: e2) = do
  v1 <- eval3 e1
  v2 <- eval3 e2
  count
  return (v1 * v2)
```

- Aritmeetiliste avaldiste väärtustamine (instrumenteeritult)

```
eval3 : Expr -> Ctr Int
```

```
eval3 (Num i) = return i
```

```
eval3 (e1 :+: e2) = do
```

```
  v1 <- eval3 e1
```

```
  v2 <- eval3 e2
```

```
  count
```

```
  return (v1 + v2)
```

← avaldise väärtustamise loendamine

```
eval3 (e1 :-: e2) = do
```

```
  v1 <- eval3 e1
```

```
  v2 <- eval3 e2
```

```
  count
```

```
  return (v1 - v2)
```

← avaldise väärtustamise loendamine

```
eval3 (e1 :*: e2) = do
```

```
  v1 <- eval3 e1
```

```
  v2 <- eval3 e2
```

```
  count
```

```
  return (v1 * v2)
```

← avaldise väärtustamise loendamine

- Aritmeetiliste avaldiste **väärtustamine** (instrumenteeritult)

```
eval3 : Expr -> Ctr Int
```

```
eval3 (Num i) = return i
```

```
eval3 (e1 :+: e2) = do  
  v1 <- eval3 e1  
  v2 <- eval3 e2  
  count  
  return (v1 + v2)
```

```
eval3 (e1 :-: e2) = do  
  v1 <- eval3 e1  
  v2 <- eval3 e2  
  count  
  return (v1 - v2)
```

```
eval3 (e1 :*: e2) = do  
  v1 <- eval3 e1  
  v2 <- eval3 e2  
  count  
  return (v1 * v2)
```

- Aritmeetiliste avaldiste väärtustamine (instrumenteeritult)

```
eval3 : Expr -> Ctr Int
```

```
eval3 (Num i) = return i
```

```
eval3 (e1 :+: e2) = do
  v1 <- eval3 e1
  v2 <- eval3 e2
  count
  return (v1 + v2)
```

```
eval3 (e1 :-: e2) = do
  v1 <- eval3 e1
  v2 <- eval3 e2
  count
  return (v1 - v2)
```

```
eval3 (e1 :*: e2) = do
  v1 <- eval3 e1
  v2 <- eval3 e2
  count
  return (v1 * v2)
```

```
exp1 : Expr
exp1 = Num 1 :+: Num 2
```

```
Loeng15c> eval3 exp1
MkCtr (3, 1)
```

```
exp2 : Expr
exp2 = Num 2 :*: (Num 4 :-: Num 1)
```

```
Loeng15c> eval3 exp2
MkCtr (6, 2)
```

- Aritmeetiliste avaldiste **väärtustamine** (instrumenteeritult)

```
eval3 : Expr -> Ctr Int
```

```
eval3 (Num i) = return i
```

```
eval3 (e1 :+: e2) = do  
  v1 <- eval3 e1  
  v2 <- eval3 e2  
  count  
  return (v1 + v2)
```

```
eval3 (e1 :-: e2) = do  
  v1 <- eval3 e1  
  v2 <- eval3 e2  
  count  
  return (v1 - v2)
```

```
eval3 (e1 :*: e2) = do  
  v1 <- eval3 e1  
  v2 <- eval3 e2  
  count  
  return (v1 * v2)
```

Näide 5: Muudetavad muutujad (nagu Javas, ...)



- Idrise programmides esinevad **muutujad pole muudetavad ega ülekirjutatavad**

- Idrise programmides esinevad **muutujad pole muudetavad ega ülekirjutatavad**
- Seetõttu ei saa me Idrises järgmisel kujul olevaid programme esitada

- Idrise programmides esinevad **muutujad pole muudetavad ega ülekirjutatavad**
- Seetõttu ei saa me Idrises järgmisel kujul olevaid programme esitada

```
int x = 1;  
int y = 2;  
int z = 3;
```

```
int prog(int i, int j) {  
    x = x + i;  
    y = x + j;  
    x = 42;  
    return z  
}
```

- Idrise programmides esinevad **muutujad pole muudetavad ega ülekirjutatavad**
- Seetõttu ei saa me Idrises järgmisel kujul olevaid programme esitada

```
int x = 1;  
int y = 2;  
int z = 3;
```

```
int prog(int i, int j) {  
    x = x + i;  
    y = x + j;  
    x = 42;  
    return z  
}
```

- Probleemiks on Idrise **funktsioonide puhtus** ning **mälu mittekasutamine**

Näide 5: Muudetavad muutujad (nagu Javas, ...)



- Näitame, et **muudetavaid muutujaid saab monaadiga modelleerida**

- Näitame, et **muudetavaid muutujaid saab monaadiga modelleerida**
- Selle jaoks vaatleme funktsioonide $f : a \rightarrow b$ asemel funktsioone $f : a \rightarrow St\ b$

- Näitame, et **muudetavaid muutujaid saab monaadiga modelleerida**
- Selle jaoks vaatleme funktsioonide $f : a \rightarrow b$ asemel funktsioone $f : a \rightarrow \text{St } b$

```
St' : Type -> Type  
St' a = State -> (a, State)
```

```
data St : Type -> Type where  
  MkSt : St' a -> St a
```

- Näitame, et **muudetavaid muutujaid saab monaadiga modelleerida**
- Selle jaoks vaatleme funktsioonide $f : a \rightarrow b$ asemel funktsioone $f : a \rightarrow \text{St } b$

```
St' : Type -> Type
St' a = State -> (a, State)

data St : Type -> Type where
  MkSt : St' a -> St a
```

```
data Vars = X | Y | Z
```

```
State : Type
State = Vars -> Int
```

```
lookup : Vars -> State -> Int
lookup x s = s x
```

```
update : Vars -> Int -> State -> State
update x i s y = if x == y then i else s y
```

- Näitame, et **muudetavaid muutujaid saab monaadiga modelleerida**
- Selle jaoks vaatleme funktsioonide $f : a \rightarrow b$ asemel funktsioone $f : a \rightarrow \text{St } b$

```
St' : Type -> Type
St' a = State -> (a, State)

data St : Type -> Type where
  MkSt : St' a -> St a
```

```
data Vars = X | Y | Z
```

```
State : Type
State = Vars -> Int
```

```
lookup : Vars -> State -> Int
lookup x s = s x
```

```
update : Vars -> Int -> State -> State
update x i s y = if x == y then i else s y
```

- Funktsioonid $f : a \rightarrow \text{St } b$ teisendavad **algoleku lõppväärtuseks ja lõppolekuks**

Näide 5: Muudetavad muutujad (nagu Javas, ...)



Näide 5: Muudetavad muutujad (nagu Javas, ...)



```
St' : Type -> Type  
St' a = State -> (a, State)
```

```
data St : Type -> Type where  
  MkSt : St' a -> St a
```

Näide 5: Muudetavad muutujad (nagu Javas, ...)

```
St' : Type -> Type
St' a = State -> (a, State)
```

```
data St : Type -> Type where
  MkSt : St' a -> St a
```

```
sReturn : a -> St a
sReturn x = MkSt (\ s => (x, s))
```

```
sBind : St a -> (a -> St b) -> St b
sBind f g = MkSt (\ s => case f of
  MkSt f' => case f' s of
    (x, s') => case g x of
      MkSt g' => g' s')
```

Näide 5: Muudetavad muutujad (nagu Javas, ...)

```
St' : Type -> Type
St' a = State -> (a, State)
```

```
data St : Type -> Type where
  MkSt : St' a -> St a
```

```
sReturn : a -> St a
sReturn x = MkSt (\ s => (x, s))
```

```
sBind : St a -> (a -> St b) -> St b
sBind f g = MkSt (\ s => case f of
  MkSt f' => case f' s of
    (x, s') => case g x of
      MkSt g' => g' s')
```

```
Monaad St where
  return = sReturn
  bind   = sBind
```

Näide 5: Muudetavad muutujad (nagu Javas, ...)

```
St' : Type -> Type
St' a = State -> (a, State)
```

```
data St : Type -> Type where
  MkSt : St' a -> St a
```

```
sReturn : a -> St a
sReturn x = MkSt (\ s => (x, s))
```

```
sBind : St a -> (a -> St b) ->
sBind f g = MkSt (\ s => case f
                             MkSt
```

```
Monaad St where
  return = sReturn
  bind    = sBind
```

State ei pea olema ainult Vars -> Int

```
ST' : Type -> Type -> Type
ST' s a = s -> (a, s)
```

```
data ST : Type -> Type -> Type where
  MkST : ST' s a -> ST s a
```

```
stReturn : a -> ST s a
stReturn x = ...
```

```
stBind : ST s a -> (a -> ST s b) -> ST s b
stBind f g = ...
```

Näide 5: Muudetavad muutujad (nagu Javas, ...)

```
St' : Type -> Type
St' a = State -> (a, State)
```

```
data St : Type -> Type where
  MkSt : St' a -> St a
```

```
sReturn : a -> St a
sReturn x = MkSt (\ s => (x, s))
```

```
sBind : St a -> (a -> St b) -> St b
sBind f g = MkSt (\ s => case f of
  MkSt f' => case f' s of
    (x, s') => case g x of
      MkSt g' => g' s')
```

```
Monaad St where
  return = sReturn
  bind   = sBind
```

Näide 5: Muudetavad muutujad (nagu Javas, ...)

```
St' : Type -> Type
St' a = State -> (a, State)
```

```
data St : Type -> Type where
  MkSt : St' a -> St a
```

```
sReturn : a -> St a
sReturn x = MkSt (\ s => (x, s))
```

```
sBind : St a -> (a -> St b) -> St b
sBind f g = MkSt (\ s => case f of
  MkSt f' => case f' s of
    (x, s') => case g x of
      MkSt g' => g' s')
```

Näide 5: Muudetavad muutujad (nagu Javas, ...)

```
St' : Type -> Type
St' a = State -> (a, State)
```

```
data St : Type -> Type where
  MkSt : St' a -> St a
```

```
sReturn : a -> St a
sReturn x = MkSt (\ s => (x, s))
```

```
sBind : St a -> (a -> St b) -> St b
sBind f g = MkSt (\ s => case f of
  MkSt f' => case f' s of
    (x, s') => case g x of
      MkSt g' => g' s')
```

```
prefix 8 ^!
(^!) : Vars -> St Int
(^!) x = MkSt (\ s => (lookup x s, s))
```

Näide 5: Muudetavad muutujad (nagu Javas, ...)

```
St' : Type -> Type
St' a = State -> (a, State)
```

```
data St : Type -> Type where
  MkSt : St' a -> St a
```

```
sReturn : a -> St a
sReturn x = MkSt (\ s => (x, s))
```

```
sBind : St a -> (a -> St b) -> St b
sBind f g = MkSt (\ s => case f of
  MkSt f' => case f' s of
    (x, s') => case g x of
      MkSt g' => g' s')
```

```
prefix 8 ^!
(^!) : Vars -> St Int
(^!) x = MkSt (\ s => (lookup x s, s))
```

```
infix 7 ^=
(^=) : Vars -> Int -> St ()
(^=) x i = MkSt (\ s => ((), update x i s))
```

Näide 5: Muudetavad muutujad (nagu Javas, ...)

```
St' : Type -> Type
St' a = State -> (a
```

```
data St : Type ->
  MkSt : St' a ->
```

```
sReturn : a -> St' ()
sReturn x = MkSt (\
```

```
sBind : St a -> (a -> St b) -> St b
sBind f g = MkSt (\ s => case f of
```

```
  MkSt f' => case f' s of
    (x, s') => case g x of
      MkSt g' => g' s')
```

```
prefix 8 ^!
```

```
(^!) : Vars -> St Int
(^!) x = MkSt (\ s => (lookup x s, s))
```

```
infix 7 ^=
```

```
(^=) : Vars -> Int -> St ()
(^=) x i = MkSt (\ s => ((), update x i s))
```

Need muutujate lugemise ja kirjutamise operatsioonid on spetsiifilised olekumonaadile olekutüübiga `Vars -> Int`

- Java programm

```
int x = 1;
int y = 2;
int z = 3;

int prog(int i, int j) {
    x = x + i;

    y = x + j;

    x = 42;

    return z
}
```

- Java programm do-notatsiooniga

`prog1 : Int -> Int -> St Int`

`prog1 i j = do`

`x <- ^! X`
`X ^= x + i`

`x' <- ^! X`
`Y ^= x' + j`

`X ^= 42`

`^! Z`

- Java programm

```
int x = 1;
int y = 2;
int z = 3;
```

```
int prog(int i, int j) {
    x = x + i;
    y = x + j;
    x = 42;
    return z
}
```

Näide 5: Muudetavad muutujad (nagu Javas, ...)



- Muudetavate muutujatega programmide jooksumiseks defineerime **run funktsiooni**

```
run : St a -> State -> (a, State)
run (MkSt f) s = f s
```

```
stateToList : State -> List Int
stateToList s = [s X , s Y , s Z]
```

```
runToList : St a -> State -> (a, List Int)
runToList f s = let (x, s') = run f s in (x, stateToList s')
```

- Muudetavate muutujatega programmide jooksumiseks defineerime **run funktsiooni**

```
run : St a -> State -> (a, State)
run (MkSt f) s = f s
```

```
stateToList : State -> List Int
stateToList s = [s X , s Y , s Z]
```

```
runToList : St a -> State -> (a, List Int)
runToList f s = let (x, s') = run f s in (x, stateToList s')
```

- Näiteks valime oma programmi **muutujate algolekuks** järgmise oleku

```
initialState : State
initialState X = 1
initialState Y = 2
initialState Z = 3
```

- Java programm do-notatsiooniga

```
prog1 : Int -> Int -> St Int
```

```
prog1 i j = do
```

```
  x <- ^! X  
  X ^= x + i
```

```
  x' <- ^! X  
  Y ^= x' + j
```

```
  X ^= 42
```

```
  ^! Z
```

- Algolek

```
initialState : State
```

```
initialState X = 1
```

```
initialState Y = 2
```

```
initialState Z = 3
```

Näide 5: Muudetavad muutujad (nagu Javas, ...)

- Java programm do-notatsiooniga

```
prog1 : Int -> Int -> St Int
```

```
prog1 i j = do
```

```
  x <- ^! X  
  X ^= x + i
```

```
  x' <- ^! X  
  Y ^= x' + j
```

```
  X ^= 42
```

```
  ^! Z
```

- Algolek

```
initialState : State
```

```
initialState X = 1
```

```
initialState Y = 2
```

```
initialState Z = 3
```

```
Loeng15d> run (prog1 4 7) initialState  
(3, update X 42 (update Y 12 (update X 5 initialState)))
```

```
Loeng15d> runToList (prog1 4 7) initialState  
(3, [42, 12, 3])
```

```
Loeng15d> runToList (prog1 4 7) (\ _ => 0)  
(0, [42, 11, 0])
```


- **Puhtad funktsioonid** on samuti näide monaadilistest arvutustest!

- **Puhtad funktsioonid** on samuti näide monaadilistest arvutustest!

```
data Id a = MkId a
```

```
iReturn : a -> Id a  
iReturn x = MkId x
```

```
iBind : Id a -> (a -> Id b) -> Id b  
iBind (MkId x') f = f x'
```

```
Monaad Id where  
  return = iReturn  
  bind    = iBind
```

- **Puhtad funktsioonid** on samuti näide monaadilistest arvutustest!

```
data Id a = MkId a
```

```
iReturn : a -> Id a  
iReturn x = MkId x
```

```
iBind : Id a -> (a -> Id b) -> Id b  
iBind (MkId x') f = f x'
```

```
Monaad Id where  
  return = iReturn  
  bind    = iBind
```

```
pureprog : Int -> Id Int -> Id Int
```

```
pureprog x y =  
  return (x * !y + 7)
```

- **Puhtad funktsioonid** on samuti näide monaadilistest arvutustest!

```
data Id a = MkId a
```

```
iReturn : a -> Id a  
iReturn x = MkId x
```

```
iBind : Id a -> (a -> Id b) -> Id b  
iBind (MkId x') f = f x'
```

```
Monaad Id where  
  return = iReturn  
  bind    = iBind
```

```
pureprog : Int -> Id Int -> Id Int
```

```
pureprog x y =  
  return (x * !y + 7)
```

- $\text{Id } a$ on **isomorfne** tüübiga a

```
i : Id a -> a  
i (MkId x) = x  
j : a -> Id a  
j x = MkId x
```

- **Puhtad funktsioonid** on samuti näide monaadilistest arvutustest!

```
data Id a = MkId a
```

```
iReturn : a -> Id a  
iReturn x = MkId x
```

```
iBind : Id a -> (a -> Id b) -> Id b  
iBind (MkId x') f = f x'
```

```
Monaad Id where  
  return = iReturn  
  bind    = iBind
```

- **Id a on isomorfne** tüübiga a

```
i : Id a -> a  
i (MkId x) = x  
j : a -> Id a  
j x = MkId x
```

```
pureprog : Int -> Id Int -> Id Int
```

```
pureprog x y =  
  return (x * !y + 7)
```

```
ij : (x : a) -> i (j x) = x  
ij x = Refl
```

```
ji : (x : Id a) -> j (i x) = x  
ji (MkId x) = Refl
```


- Algebralised efektid ja nende töötlejad (OCaml, Eff, Koka, Haskell, ...)

- Algebralised efektid ja nende töötlejad (OCaml, Eff, Koka, Haskell, ...)
- Kõrvalefektide sisaldavate keelte tüübikontrollis ja semantikas (OCaml, ...)

- Algebralised efektid ja nende töötlejad (OCaml, Eff, Koka, Haskell, ...)
- Kõrvalefektide sisaldavate keelte tüübikontrollis ja semantikas (OCaml, ...)

```
val incr : ref nat -> nat
```

```
let incr (r:ref nat) =  
  r := !r + 1;  
  !r
```

- Algebralised efektid ja nende töötlejad (OCaml, Eff, Koka, Haskell, ...)
- Kõrvalefektide sisaldavate keelte tüübikontrollis ja semantikas (OCaml, ...)

```
val incr : ref nat -> nat
```



```
let incr (r:ref nat) =  
  r := !r + 1;  
  !r
```

- Algebralised efektid ja nende töötlejad (OCaml, Eff, Koka, Haskell, ...)
- Kõrvalefektide sisaldavate keelte tüübikontrollis ja semantikas (OCaml, ...)

```
val incr : ref nat -> nat
```



```
let incr (r:ref nat) =  
  r := !r + 1;  
  !r
```

```
val incr_expl : ref nat-> St nat
```

```
let incr_expl (r:ref nat) =  
  !r >>= \ v =>  
  (v + 1) >>= \ w =>  
  (r := w) >>= \ _ =>  
  return w
```

- Algebralised efektid ja nende töötlejad (OCaml, Eff, Koka, Haskell, ...)
- Kõrvalefektide sisaldavate keelte tüübikontrollis ja semantikas (OCaml, ...)

```
val incr : ref nat -> nat
```



```
val incr_expl : ref nat -> St nat
```

```
let incr (r:ref nat) =  
  r := !r + 1;  
  !r
```

```
let incr_expl (r:ref nat) =  
  !r >>= \ v =>  
  (v + 1) >>= \ w =>  
  (r := w) >>= \ _ =>  
  return w
```

- Tõestamist toetavates keeltes imperatiivse koodi korrektsuse kontroll (FStar, Rocq, ...)

- Algebralised efektid ja nende töötlejad (OCaml, Eff, Koka, Haskell, ...)
- Kõrvalefektide sisaldavate keelte tüübikontrollis ja semantikas (OCaml, ...)

```
val incr : ref nat -> nat
```



```
val incr_expl : ref nat -> St nat
```

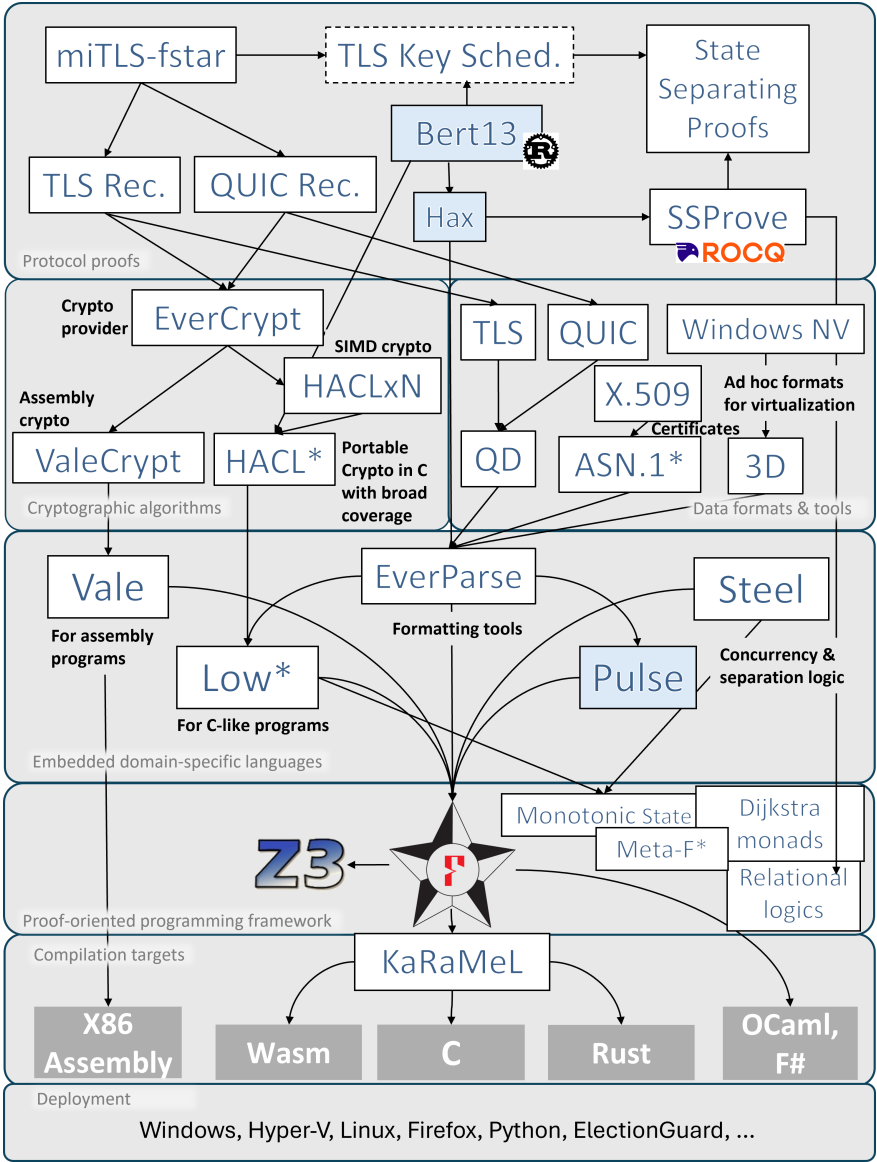
```
let incr (r:ref nat) =  
  r := !r + 1;  
  !r
```

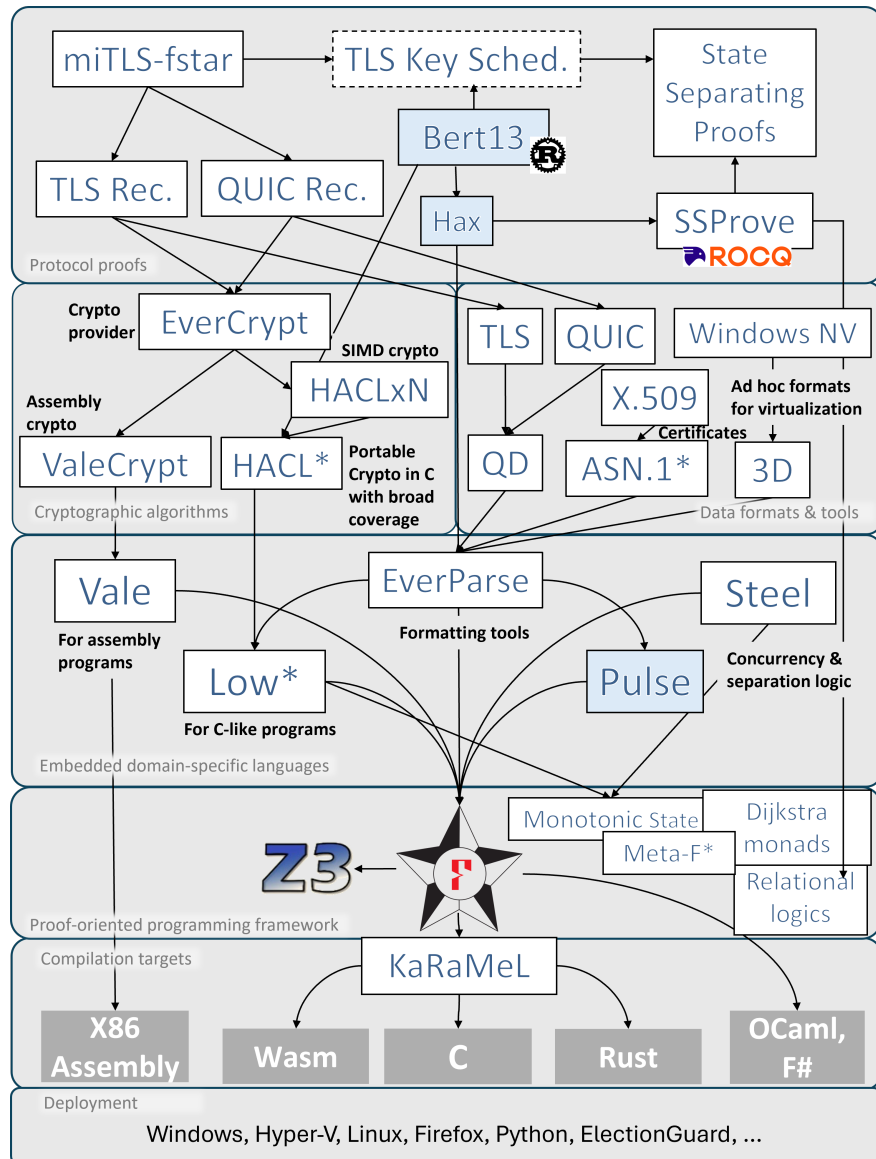
```
let incr_expl (r:ref nat) =  
  !r >>= \ v =>  
  (v + 1) >>= \ w =>  
  (r := w) >>= \ _ =>  
  return w
```

- Tõestamist toetavates keeltes imperatiivse koodi korrektsuse kontroll (FStar, Rocq, ...)

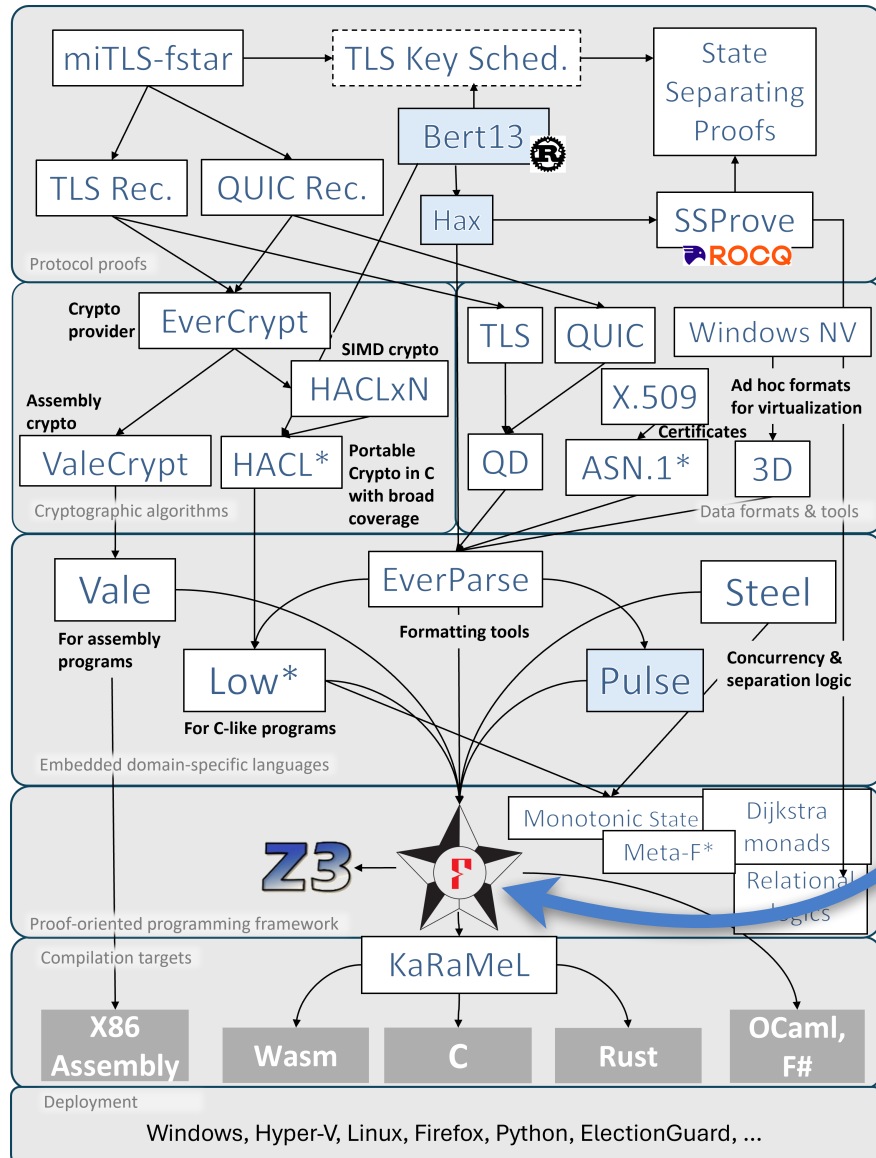
```
val incr : (r:ref nat)  
  -> State nat (requires (\ s => live s r))  
                (ensures (\ s v s' => lookup r s' == lookup r s + 1 /\  
                               v == lookup r s'))
```







```
val incr : (r:ref nat)
-> State nat
  (requires (\ s => live s r))
  (ensures  (\ s v s' =>
    lookup r s' == lookup r s + 1 /\
    v == lookup r s'))
```

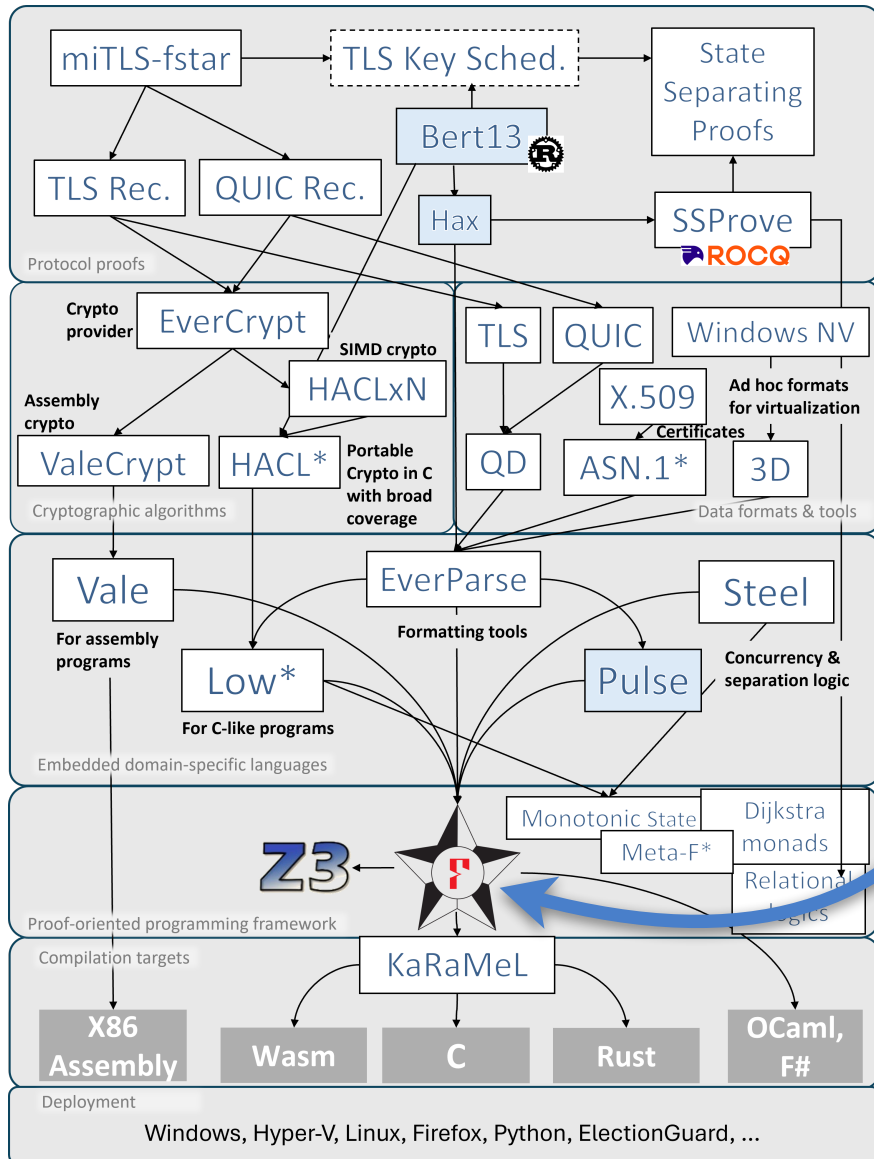


`val incr : (r:ref nat)`

`-> State nat`

`(requires (\ s => live s r))`

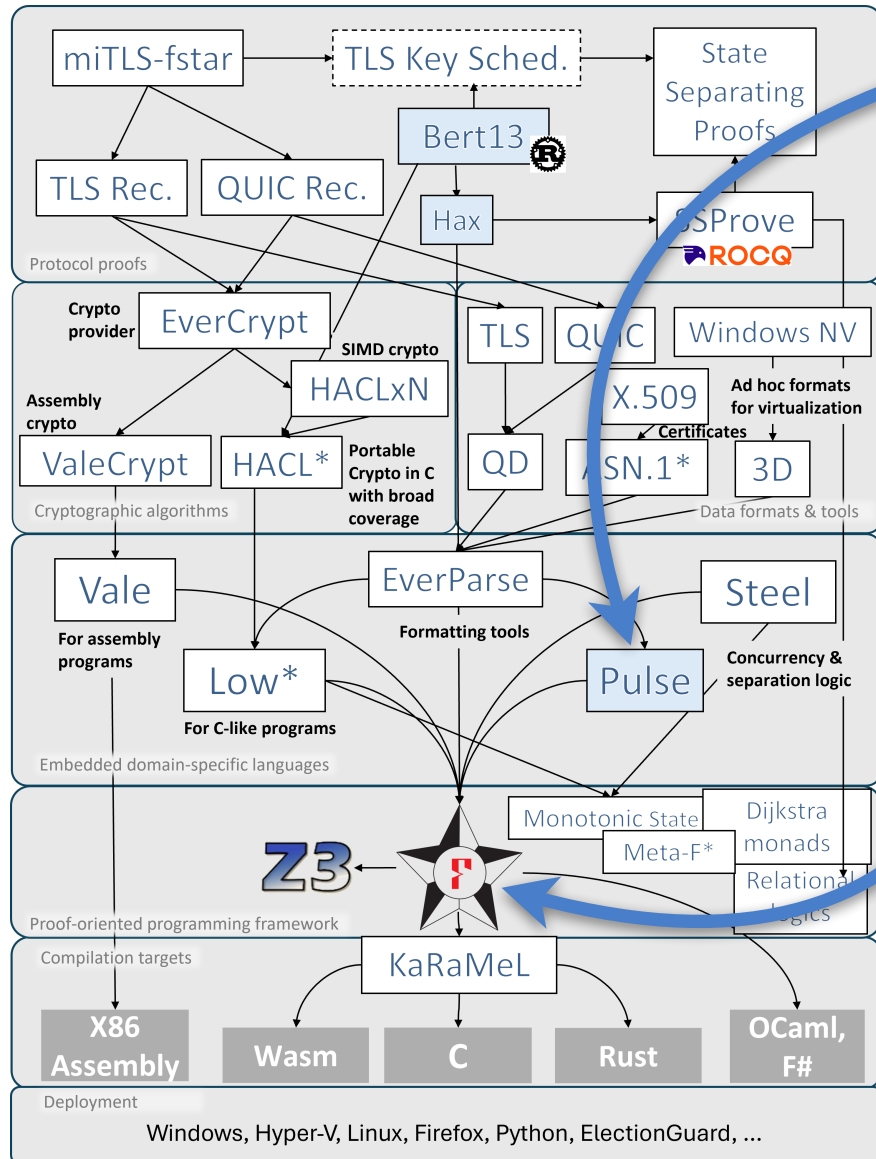
`(ensures (\ s v s' =>
lookup r s' == lookup r s + 1 /\
v == lookup r s'))`



```
fn rec append (x y:l1ist 't) (l1 l2:erased (list 't))
requires is_list x l1 **
         is_list y l2 **
         pure (Some? x)

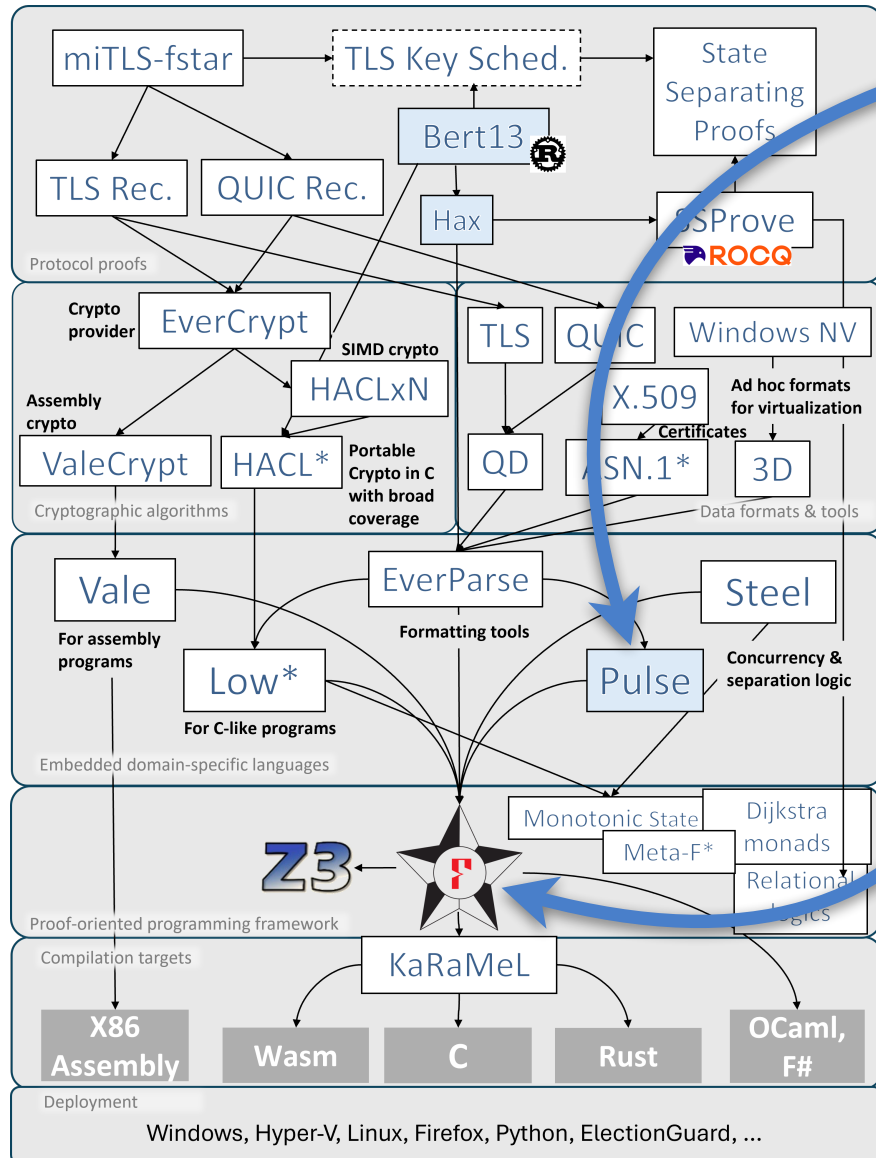
ensures  is_list x (l1 @ l2)
{
  ...
}
```

```
val incr : (r:ref nat)
-> State nat
(requires (\ s => live s r))
(ensures  (\ s v s' =>
  lookup r s' == lookup r s + 1 /\
  v == lookup r s'))
```



```
fn rec append (x y:l1ist 't) (l1 l2:erased (list 't))
  requires is_list x l1 **
           is_list y l2 **
           pure (Some? x)
  ensures  is_list x (l1 @ l2)
{
  ...
}
```

```
val incr : (r:ref nat)
  -> State nat
  (requires (\ s => live s r))
  (ensures  (\ s v s' =>
    lookup r s' == lookup r s + 1 /\
    v == lookup r s'))
```

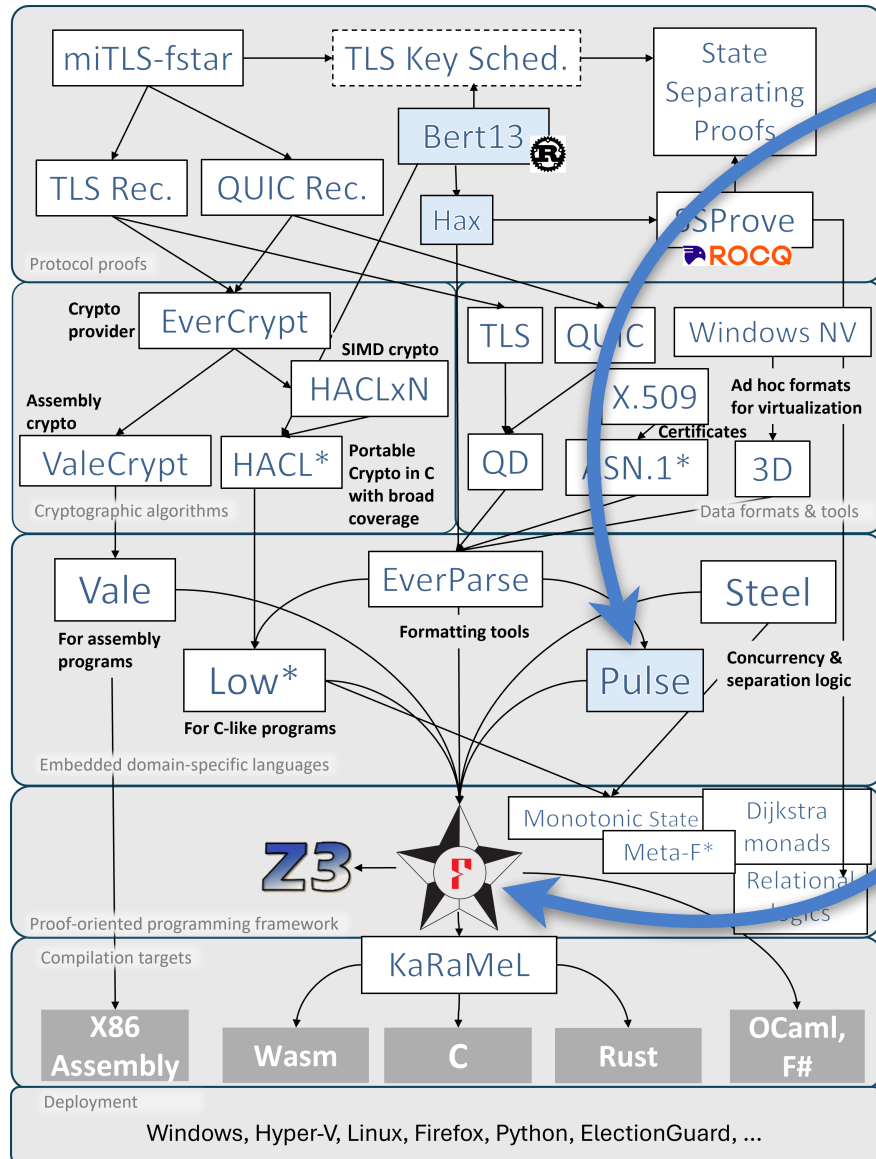


```
fn rec append (x y:l1ist 't) (l1 l2:erased (list 't))
  requires is_list x l1 **
           is_list y l2 **
           pure (Some? x)

  ensures is_list x (l1 @ l2)
{
  ...
}
```

Hoare logic

```
val incr : (r:ref nat)
  -> State nat
  (requires (\ s => live s r))
  (ensures (\ s v s' =>
    lookup r s' == lookup r s + 1 /\
    v == lookup r s'))
```



```
fn rec append (x y:l1list 't) (l1 l2:erased (list 't))
requires is_list x l1 **
        is_list y l2 **
        pure (Some? x)
```

```
ensures is_list x (l1 @ l2)

{
  ...
}
```

Separation logic

```
val incr : (r:ref nat)
-> State nat
(requires (\ s => live s r))
(ensures (\ s v s' =>
  lookup r s' == lookup r s + 1 /\
  v == lookup r s'))
```

Hoare logic

- **Monaad = järjestikuste, kõrvalefekte sisaldavate programmide abstraktsioon**

- **Monaad = järjestikuste, kõrvalefekte sisaldavate programmide abstraktsioon**

- **Palju näiteid**

- puhtad funktsioonid ($f : \text{Nat} \rightarrow \text{Id Bool}$)
- sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
- erandid ($f : \text{Nat} \rightarrow \text{Option Bool}$)
- mälu kasutus ($f : \text{Nat} \rightarrow \text{St Bool}$)
- mitteterminism ($f : \text{Nat} \rightarrow \text{List Bool}$)
- tõenäosused ($f : \text{Nat} \rightarrow \text{Dist Bool}$)
- termipuud ($f : \text{Nat} \rightarrow \text{Term Bool}$)
- ...

monaadid ($f : \text{Nat} \rightarrow m \text{ Bool}$)

- **Monaad = järjestikuste, kõrvalefekte sisaldavate programmide abstraktsioon**

- **Palju näiteid**

- puhtad funktsioonid ($f : \text{Nat} \rightarrow \text{Id Bool}$)
- sisend-väljund ($f : \text{Nat} \rightarrow \text{IO Bool}$)
- erandid ($f : \text{Nat} \rightarrow \text{Option Bool}$)
- mälu kasutus ($f : \text{Nat} \rightarrow \text{St Bool}$)
- mitteterminism ($f : \text{Nat} \rightarrow \text{List Bool}$)
- tõenäosused ($f : \text{Nat} \rightarrow \text{Dist Bool}$)
- termipuud ($f : \text{Nat} \rightarrow \text{Term Bool}$)
- ...

monaadid ($f : \text{Nat} \rightarrow m \text{ Bool}$)

- Veel rohkem näiteid saab **olemasolevate monaadide komponeerimisest**

- **Monaad**

aktsioon

- **Palju nä**

- puhta

- sisene

- erand

- mälu

- mitte

- tõenä

- term

- ...

m Bool)

- **Veel rohk**

est

- **Monaad**

aktsioon

- **Palju nä**

- puhta

- sisene

- erand

- mälu

- mitte

- tõenä

- term

- ...

m Bool)

- **Veel rohk**

est

- **Monaad**
- **Palju nä**
 - puhta
 - sisen
 - erand
 - mälu
 - mitte
 - tõenä
 - term
 - ...
- **Veel rohk**
- **Loogika Arvutiteaduses** (LTAT.03.02 I, BSc, kevaditi)
 - Curry-Howardi isomorphismi loogika-poolne osa
 - erinevad loogika tuletus- ja tõestussüsteemid
 - automaatse tõestamise elemente (resolutsioon)
 - ajaloogikad (ajas muutuvate omaduste kontrollimine)
 - modaalloogikad (mina tean, et sina tead, et tema teab, et ...)
 - programmilooikad (imperatiivsete programmide korrektsus)

- **Monaad**
- **Palju nä**
 - puhta
 - sisen
 - erand
 - mälu
 - mitte
 - tõenä
 - term
 - ...
- **Veel rohk**
- **Loogika Arvutiteaduses** (LTAT.03.02 I, BSc, kevaditi)
 - Curry-Howardi isomorphismi loogika-poolne osa
 - erinevad loogika tuletus- ja tõestussüsteemid
 - automaatse tõestamise elemente (resolutsioon)
 - ajaloogikad (ajas muutuvate omaduste kontrollimine)
 - modaalloogikad (mina tean, et sina tead, et tema teab, et ...)
 - programmi loogikad (imperatiivsete programmide korrektsus)
- **Programmeerimiskeelte Põhimõtted** (LTAT.03.027, MSc, sügiseti)
 - jätkab sealt, kus FP aine (teooriaosa) pooleli jääb
 - arvutuslikud kõrvalefektid (ja muidugi ka monaadid)
 - arvutuslike kõrvalefektide töötlejad (eranditöötlejad+++)
 - alamtüüpimine, polümorfism, efekti-ja-tüübisüsteemid
 - sõltuvate tüüpide teooria edasiarendusi
 - lineaarsed ja modaalsed tüübid, programmi loogikad tüüpide sees

aktsioon

m Bool)

Aitäh kuulamast!



TARTU ÜLIKOOL

GET THE DRAGON!

HOW TO KILL THE DRAGON
USING 9 PROGRAMMING
LANGUAGES

BY toggl
Goon Squad

YOU HAVE PYTHON



EVERYBODY MARVELS AT
YOUR AWESOME DRAGON
SLAYING TOOLS.



YOU DISCOVER YOUR
TOOLS ARE ONLY FOR
SLAYING A SNAKE



YOU HAVE SWIFT



YOU'VE GOT AN NSDRAGON
CLASS, BUT YOU NEED TO
WRITE AN 500 LOC EXTENSION
TO IMPLEMENT SLAYABILITY



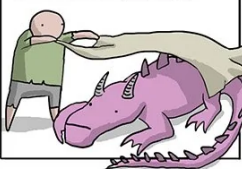
JUST AS YOU'RE ALMOST
DONE, APPLE RELEASES
ANOTHER DRAGON



YOU HAVE
CSS/HTML



YOU TRY TO COVER THE
DRAGON WITH A HIGHLY
FLAMMABLE BLANKET



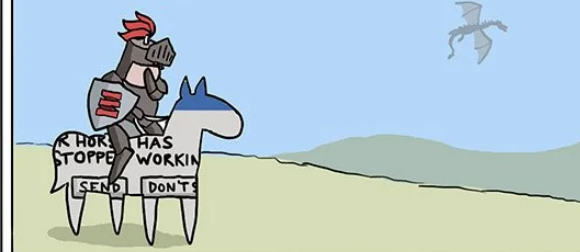
NOW EVERYTHING'S ON
FIRE (ALSO, THE VERTICAL
CENTER HAS FAILED)



YOU HAVE SCALA



YOUR HORSE HAS CRASHED :(



YOU HAVE C#



YOU SET ALL REFERENCES TO
THE DRAGON TO NULL.
THE DRAGON SEEMS GONE.



BUT WITH NON-DETERMINISTIC
GARBAGE COLLECTION, CAN YOU
EVER BE SURE THE DRAGON
IS REALLY GONE?



YOU HAVE COBOL



YOU GO FROM VILLAGE TO
VILLAGE FIGHTING THE SAME
DRAGON OVER AND OVER



YOU MAKE A FORTUNE
IN THE PROCESS



YOU HAVE LUA



IT'S AN INCREDIBLY FAST
AND EFFECTIVE WEAPON
... BUT YOU'RE OUT OF AMMO



MIKE PALL HAS
THE LAST BULLETS,
BUT HE'S GONE & YOU
DON'T KNOW HIS REAL
NAME OR EVEN WHAT
HE LOOKS LIKE



YOU HAVE ~~HASKELL~~
(AND A MONAD)



YOU IMPLEMENT A MONAD TO ENCAPSULATE
THE DRAGON KILLING SIDE-EFFECTS.
THE VILLAGERS ARE CONCERNED AND
URGE YOU TO STOP, BUT YOU SAY IT'S OK
CAUSE A MONAD IS JUST A MONOID IN
THE CATEGORY OF ENDOFUNCTORS, SO
YOU DON'T SEE WHAT
THE PROBLEM IS



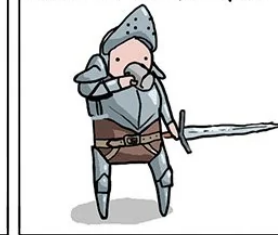
YOU NEED HELP.



YOU HAVE
COFFEESCRIPT



YOU DON'T HAVE DRAGON
SLAYING POWERS SO YOU
DRINK IT TO GET GOOD



YOU DIED



MART VIRKUS '17 TOGGL.COM