Bevis og programmering er sammme sak?

Rask intro til Haskellsyntaks

```haskell
data MinDatatype a = Venstre a | Hoyre a

eksempel :: MinDatatype Int
eksempel = Venstre 1

isVenstre :: MinDatatype a -> String
isVenstre minDatatype = case minDatatype of
  Venstre a -> "Yes"
  Hoyre a -> "No"
```

Hva er bevis?

- Et argument som viser at antagelsene garanterer konklusjonen
- Sammenheng med programmering?

Curry-Howard-korrespondansen

Teorem i logikk $\leftrightarrow$ Typen har en verdi

Recap : Utsagnslogikk

- Verdier : $\top$(true), $\perp$(false)
- Operatorer : &, |, $\rightarrow$, not
- Formler
  - Bygget opp av verdier og operatorer
  - Bruker A,B osv for å snakke om vilkårlige formler
  - $\perp \rightarrow \top$, $A \rightarrow$ (B & not B) osv
- Bevis

Typer er utsagn og programmer er bevis

- Du kan bevise utsagnet `Int` ved å gi en eksempelverdi av typen `Int`

```haskell
num :: Int
num = 42
```

- En implementasjon av en verdi av en type blir da et bevis for utsagnet typen representer er sann

Implikasjon

- $A \rightarrow B$
- I programmering : funksjoner
  - parametertype A
  - returtype B
  - Haskell : `a -> b`
- Eksempel

```
impliesSelf :: a -> a
impliesSelf a = a
```

Implikasjon - eksempler

```haskell
const :: a -> b -> a
const a b = a
```

$\top$: Sant - True

- En verdi av typen A beviser utsagnet A
- `True` er da typen som er trivielt sant.

```
data True = True
```

& : And

- Logikk : A & B er et teorem kun hvis både A og B er teoremer

```
data And a b = And a b
```

And : eksempler

```haskell
ae1 :: And True True
ae1 = And True True

ae2 :: And a b -> a
ae2 (And a b) = a

ae3 :: And a b -> And b a
ae3 (And a b) = (And b a)
```

| : Or

- Logikk : $A \mid B$ er et teorem hvis A er et teorem eller B er et teorem

```
data Or a b = OrLeft a | OrRight  b
```

Or : eksempler

```
oe1 :: Or True b
oe1 = OrLeft True

oe2 :: Or a a -> a
oe2 orAA = case orAA of
    OrLeft a -> a
    OrRight a -> a
```

$\perp$: False - en type uten verdier

- En verdi av typen A beviser utsagnet A
- $\perp$ - usant - skal aldri kunne bevises

```
data False
```

- Er faktisk praktisk nyttig
  - Kotlin : Nothing
  - Rust : ! (never)
  - Haskell : Void

False - absurd

Fra noe usant kan man utlede hva som helst!

```haskell
absurd :: False -> b
absurd false = case false of {}

orFalse :: Or a False -> a
orFalse or_a_False = case or_a_False of
    OrLeft a -> a
    OrRight false -> absurd false
```

Not - Negasjon

- Typer med verdier til typer uten verdier

```
type Not a = a -> False
```

Not Not - dobbelnegering

- not (not a) $\longrightarrow$ a
- a $\longrightarrow$ not (not a)

```haskell
impliesNotNot :: a -> Not (Not a)
--            :: a -> (Not (a -> False)
--            :: a -> ((a -> False) -> False)
--            :: a -> (a -> False) -> False
impliesNotNot a a2false = a2false a
```

Not Not - andre veien

```haskell
notnotImplies :: Not (Not a) -> a
--            :: (Not (a -> False)) -> a
--            :: ((a -> False) -> False) -> a
notnotImplies a2false_false = umulig
```

Not Not - andre veien

- Hva går galt her?

Logikk - klassisk

- Det finnes flere logikksystemer
- Den mest vanlige blir kalt *klassisk logikk*
- Med `Or a (Not a)`, `Not (Not a) -> a` og motsigelsesbevis
- Kan føre til litt "*fjerne*" bevis

Konstruktiv logikk

- Alle bevis demonstrerer eksistens
  - Å bevise A er å demonstrere at A eksisterer, med et eksempel
- En "svakere" logikk
- Uten `Or a (Not a)`, `Not (Not a) -> a` og motsigelsesbevis

De Morgans lover

- not (A | B) = (not A) & (not B)

```
law2 :: And (Not a) (Not b) -> Not (Or a b)
--    :: And (a -> False) (b -> False) -> Or a b -> False
law2 (And a2False b2False) orAB = case orAB of
  OrLeft a -> a2False a
  OrRight b -> b2False b
```

- not (A & B) = (not A) | (not B)

```
law3 :: Not (And a b) -> Or (Not a) (Not b)
--(And a b -> False) -> Or (a -> False) (b -> False)
law3 andAB2False = umulig
```

Oppsummering

- Typer er utsagn, programmer er bevis
- Dyp korrespondanse : gjelder for mange logikk og typesystemer
- Bevisene blir maskinsjekket!
    - All koden i denne talken kompilerer*
- Grunnlaget for theorem provers som Coq, Agda, Lean
- *Gjelder kun hvis alt vi implementerer terminerer og ikke bruker errors/exceptions osv. Hvis ikke kan vi fort få motsigelser og inkonsistent logikk. Ref `umulig`