Haskell als Beweis-Checker

Thorsten Wißmann

FSI Lightning-Talk

27. Januar 2020

Logik
(siehe GLoIn, GTI)

Funktionale
Programmierung
(siehe PFP)

Haskell

Funktionstyp `a -> b` in Haskell:
Typ der Funktionen von `a` nach `b`.
(ähnliches in Scala, C, Java, ...)

Mathematik

Die Menge $A \to B$
enthält alle Funktionen
von A nach B.

```haskell
f :: (Int,a) -> (a,Int)
f (n,x) = (x, n + 3)
```

Mathematische Notation:

$f\colon \mathbb{Z} \times A \to A \times \mathbb{Z}$
$f(n, x) = (x, n + 3)$

Haskell

Funktionstyp `a -> b` in Haskell:
Typ der Funktionen von `a` nach `b`.
(ähnliches in Scala, C, Java, ...)

Mathematik

Die Menge $A \to B$
enthält alle Funktionen
von A nach B.

```
f :: (Int,a) -> (a,Int)
f (n,x) = (x, n + 3)

tuple :: a -> (b -> (a,b))
tuple x y = (x,y)
--  (λx.λy.(x,y))
```

Mathematische Notation:

$$f\colon \mathbb{Z} \times A \to A \times \mathbb{Z}$$
$$f(n, x) = (x, n + 3)$$
$$t\colon A \to (B \to A \times B)$$
$$t(x)(y) = (x, y)$$

Curry-Howard-Korrespondenz

1-zu-1-Korrespondenz zwischen . . .

Intuitionistisch!

Logischen Formeln	und	Typen in Programmen
Formel	=	Typ
Beweis	=	(Terminierendes) Programm
	=	
	=	

Curry-Howard-Korrespondenz

1-zu-1-Korrespondenz zwischen . . .

Intuitio-
nistisch!

Logischen Formeln	und	Typen in Programmen
Formel	=	Typ
Beweis	=	(Terminierendes) Programm
Beweisprüfung	=	Typ-Checking
	=	

Curry-Howard-Korrespondenz

1-zu-1-Korrespondenz zwischen ...

Intuitio-nistisch!

Logischen Formeln	und	Typen in Programmen
Formel	=	Typ
Beweis	=	(Terminierendes) Programm
Beweisprüfung	=	Typ-Checking
Beweisassistenten	=	Programmiersprachen

z.B. Coq, Agda

Curry-Howard-Korrespondenz

1-zu-1-Korrespondenz zwischen ...

Intuitio-
nistisch!

Logischen Formeln	und	Typen in Programmen
Formel	=	Typ
Beweis	=	(Terminierendes) Programm
Beweisprüfung	=	Typ-Checking
Beweisassistenten	=	Programmiersprachen

z.B. Coq,
Agda

Beweisen einer Formel mittels Haskell:

1. Formel als Typ in Haskell schreiben
2. Ein (terminierendes) Programm diesen Typs schreiben
3. Den Compiler aufrufen.

Vokabelheft

Logik „Formel a gilt“		Haskell „Typ `a` ist bewohnt“	
Name	Syntax	Syntax	Name
Implikation	$a \to b$	`a -> b`	Funktions-Typ
und	$a \wedge b$	`(a,b)`	Tupel / struct
oder	$a \vee b$		
wahr	$\top$		
falsch	$\bot$		

```
modus_ponens :: (a -> b, a) -> b
modus_ponens (f,x) = f(x)
```

Vokabelheft

Logik „Formel a gilt“		Haskell „Typ `a` ist bewohnt“	
Name	Syntax	Syntax	Name
Implikation	$a \to b$	`a -> b`	Funktions-Typ
und	$a \wedge b$	`(a,b)`	Tupel / struct
oder	$a \vee b$	`Either a b`	Tagged union
wahr	$\top$		
falsch	$\bot$		

```haskell
modus_ponens :: (a -> b, a) -> b
modus_ponens (f,x) = f(x)
```

Vokabelheft

Logik „Formel a gilt“		Haskell „Typ `a` ist bewohnt“	
Name	Syntax	Syntax	Name
Implikation	$a \to b$	`a -> b`	Funktions-Typ
und	$a \wedge b$	`(a,b)`	Tupel / struct
oder	$a \vee b$	`Either a b`	Tagged union
wahr	$\top$	`()`	Unit
falsch	$\bot$		

```
modus_ponens :: (a -> b, a) -> b
modus_ponens (f,x) = f(x)
```

Vokabelheft

Logik „Formel a gilt“		Haskell „Typ `a` ist bewohnt“	
Name	Syntax	Syntax	Name
Implikation	$a \to b$	`a -> b`	Funktions-Typ
und	$a \wedge b$	`(a,b)`	Tupel / struct
oder	$a \vee b$	`Either a b`	Tagged union
wahr	$\top$	`()`	Unit
falsch	$\bot$	`Void`	Leerer Typ, $\varnothing$

```
modus_ponens :: (a -> b, a) -> b
modus_ponens (f,x) = f(x)
```

Vokabelheft

Logik „Formel a gilt“		Haskell „Typ `a` ist bewohnt“	
Name	Syntax	Syntax	Name
Implikation	$a \to b$	`a -> b`	Funktions-Typ
und	$a \wedge b$	`(a,b)`	Tupel / struct
oder	$a \vee b$	`Either a b`	Tagged union
wahr	$\top$	`()`	Unit
falsch	$\bot$	`Void`	Leerer Typ, $\varnothing$
Abgeleitet:			

```
modus_ponens :: (a -> b, a) -> b
modus_ponens (f,x) = f(x)
```

Vokabelheft

Logik „Formel a gilt“		Haskell „Typ `a` ist bewohnt“	
Name	Syntax	Syntax	Name
Implikation	$a \to b$	`a -> b`	Funktions-Typ
und	$a \wedge b$	`(a,b)`	Tupel / struct
oder	$a \vee b$	`Either a b`	Tagged union
wahr	$\top$	`()`	Unit
falsch	$\bot$	`Void`	Leerer Typ, $\varnothing$
Abgeleitet:			
Negation	$\neg a$		

```
modus_ponens :: (a -> b, a) -> b
modus_ponens (f,x) = f(x)
```

Vokabelheft

Logik „Formel *a* gilt“		Haskell „Typ `a` ist bewohnt“	
Name	Syntax	Syntax	Name
Implikation	$a \to b$	`a -> b`	Funktions-Typ
und	$a \wedge b$	`(a,b)`	Tupel / struct
oder	$a \vee b$	`Either a b`	Tagged union
wahr	$\top$	`()`	Unit
falsch	$\bot$	`Void`	Leerer Typ, $\varnothing$
Abgeleitet:			
Negation	$\neg a$	`a -> Void`	

```haskell
modus_ponens :: (a -> b, a) -> b
modus_ponens (f,x) = f(x)
```

Vokabelheft

Logik „Formel a gilt“		Haskell „Typ `a` ist bewohnt“	
Name	Syntax	Syntax	Name
Implikation	$a \to b$	`a -> b`	Funktions-Typ
und	$a \wedge b$	`(a,b)`	Tupel / struct
oder	$a \vee b$	`Either a b`	Tagged union
wahr	$\top$	`()`	Unit
falsch	$\bot$	`Void`	Leerer Typ, $\varnothing$
Abgeleitet:			
Negation	$\neg a$	`a -> Void`	
Äquivalenz	$a \leftrightarrow b$		

```
modus_ponens :: (a -> b, a) -> b
modus_ponens (f,x) = f(x)
```

Vokabelheft

Logik „Formel a gilt“		Haskell „Typ `a` ist bewohnt“	
Name	Syntax	Syntax	Name
Implikation	$a \to b$	`a -> b`	Funktions-Typ
und	$a \wedge b$	`(a,b)`	Tupel / struct
oder	$a \vee b$	`Either a b`	Tagged union
wahr	$\top$	`()`	Unit
falsch	$\bot$	`Void`	Leerer Typ, $\varnothing$
Abgeleitet:			
Negation	$\neg a$	`a -> Void`	
Äquivalenz	$a \leftrightarrow b$	`(a -> b, b -> a)`	

```
modus_ponens :: (a -> b, a) -> b
modus_ponens (f,x) = f(x)
```

```
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
import Data.Void -- absurd :: Void -> x

type Not a = a -> Void

-- (a ∨ b) → (¬a ∨ ¬c) → (c → b)
```

```haskell
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
import Data.Void -- absurd :: Void -> x

type Not a = a -> Void

-- (a ∨ b) → (¬a ∨ ¬c) → (c → b)
blatt4aufg5a :: Either a b
             -> Either (Not a) (Not c)
             -> (c -> b)
```

```haskell
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
import Data.Void -- absurd :: Void -> x

type Not a = a -> Void

-- (a ∨ b) → (¬a ∨ ¬c) → (c → b)
blatt4aufg5a :: Either a b
             -> Either (Not a) (Not c)
             -> (c -> b)
blatt4aufg5a a_or_b na_or_nc c =
```

```
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
import Data.Void -- absurd :: Void -> x

type Not a = a -> Void

-- (a ∨ b) → (¬a ∨ ¬c) → (c → b)
blatt4aufg5a :: Either a b
             -> Either (Not a) (Not c)
             -> (c -> b)
blatt4aufg5a a_or_b na_or_nc c =
  case a_or_b of
    Right b -> b
    Left a ->
```

```haskell
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
import Data.Void -- absurd :: Void -> x

type Not a = a -> Void

-- (a ∨ b) → (¬a ∨ ¬c) → (c → b)
blatt4aufg5a :: Either a b
             -> Either (Not a) (Not c)
             -> (c -> b)
blatt4aufg5a a_or_b na_or_nc c =
  case a_or_b of
    Right b -> b
    Left a ->
      case na_or_nc of
        Left not_a -> absurd (not_a(a))
        Right not_c -> absurd (not_c(c))
```

$$a \cdot b \qquad\qquad a \wedge b$$
$$a + b \quad \rightsquigarrow \quad a \vee b$$

$$\begin{array}{ccc} a \cdot b & & a \wedge b \\ a + b & \rightsquigarrow & a \vee b \\ b^a & & a \to b \end{array}$$

$$a \cdot b \qquad\qquad a \wedge b$$
$$a + b \quad \rightsquigarrow \quad a \vee b$$
$$b^a \qquad\qquad a \to b$$

Potenzgesetze

Für Zahlen a, b, c:

$$c^{a \cdot b} = (c^b)^a$$
$$(b \cdot c)^a = b^a \cdot c^a$$
$$c^{a+b} = c^a \cdot c^b$$
$$a^1 = a$$
$$a^0 = 1$$

$\rightsquigarrow$

$$a \cdot b \quad \rightsquigarrow \quad a \wedge b$$
$$a + b \quad \rightsquigarrow \quad a \vee b$$
$$b^a \quad \rightsquigarrow \quad a \to b$$

Potenzgesetze

Für Zahlen a, b, c:

$$\begin{aligned} c^{a \cdot b} &= (c^b)^a \\ (b \cdot c)^a &= b^a \cdot c^a \\ c^{a+b} &= c^a \cdot c^b \\ a^1 &= a \\ a^0 &= 1 \end{aligned}$$

$\rightsquigarrow$

Logische Interpretation

$$\begin{aligned} (a \wedge b \to c) &\leftrightarrow (a \to (b \to c)) \\ (a \to b \wedge c) &\leftrightarrow ((a \to b) \wedge (a \to c)) \\ (a \vee b \to c) &\leftrightarrow ((a \to c) \wedge (b \to c)) \\ (\top \to a) &\leftrightarrow a \\ (\bot \to a) &\leftrightarrow \top \end{aligned}$$

```
type Aequivalent a b = (a -> b, b -> a)

-- Arithmetik: c ^ (a * b) = (c ^ b) ^ a
-- Logik: ((a ∧ b) → c) ↔ (a → (b → c))
un_currying :: Aequivalent ((a,b) -> c) (a -> (b -> c))
un_currying = (curry, uncurry)
```

```haskell
type Aequivalent a b = (a -> b, b -> a)

-- Arithmetik: c ^ (a * b) = (c ^ b) ^ a
-- Logik: ((a ∧ b) → c) ↔ (a → (b → c))
un_currying :: Aequivalent ((a,b) -> c) (a -> (b -> c))
un_currying = (curry, uncurry)

-- Universelle Eigenschaft des Produkts
-- Arithmetik: (a * b)^c = a^c * a^b
-- Logik: (a → (b ∧ c)) ↔ ((a → b) ∧ (a → c))
univ_eig_prod :: Aequivalent (a -> (b,c)) (a -> b, a -> c)
univ_eig_prod = (\f -> (fst . f, snd . f), \(f,g) -> \a -> (f(a),g(a)))
```

```
type Aequivalent a b = (a -> b, b -> a)

-- Arithmetik: c ^ (a * b) = (c ^ b) ^ a
-- Logik: ((a ∧ b) → c) ↔ (a → (b → c))
un_currying :: Aequivalent ((a,b) -> c) (a -> (b -> c))
un_currying = (curry, uncurry)

-- Universelle Eigenschaft des Produkts
-- Arithmetik: (a * b)^c = a^c * a^b
-- Logik: (a → (b ∧ c)) ↔ ((a → b) ∧ (a → c))
univ_eig_prod :: Aequivalent (a -> (b,c)) (a -> b, a -> c)
univ_eig_prod = (\f -> (fst . f, snd . f), \(f,g) -> \a -> (f(a),g(a)))

-- Universelle Eigenschaft der disjunkten Vereinigung
-- Arithmetik: c^(a+b) = c^a * c^b
-- Logik: ((a ∨ b) → c) ↔ ((a → c) ∧ (b → c))
univ_eig_coprod :: Aequivalent (Either a b -> c) (a -> c, b -> c)
univ_eig_coprod = (\f -> (f . Left, f . Right), uncurry either)
```

```haskell
type Aequivalent a b = (a -> b, b -> a)

-- Arithmetik: c ^ (a * b) = (c ^ b) ^ a
-- Logik: ((a ∧ b) → c) ↔ (a → (b → c))
un_currying :: Aequivalent ((a,b) -> c) (a -> (b -> c))
un_currying = (curry, uncurry)

-- Universelle Eigenschaft des Produkts
-- Arithmetik: (a * b)^c = a^c * a^b
-- Logik: (a → (b ∧ c)) ↔ ((a → b) ∧ (a → c))
univ_eig_prod :: Aequivalent (a -> (b,c)) (a -> b, a -> c)
univ_eig_prod = (\f -> (fst . f, snd . f), \(f,g) -> \a -> (f(a),g(a)))

-- Universelle Eigenschaft der disjunkten Vereinigung
-- Arithmetik: c^(a+b) = c^a * c^b
-- Logik: ((a ∨ b) → c) ↔ ((a → c) ∧ (b → c))
univ_eig_coprod :: Aequivalent (Either a b -> c) (a -> c, b -> c)
univ_eig_coprod = (\f -> (f . Left, f . Right), uncurry either)

-- Arithmetik: a^1 = a
-- Logik: (⊤ → a) ↔ a
hoch_eins :: Aequivalent (() -> a) a
hoch_eins = (\f -> f(), const)
```

```haskell
import Data.Void
type Aequivalent a b = (a -> b, b -> a)

-- Arithmetik: c ^ (a * b) = (c ^ b) ^ a
-- Logik: ((a ∧ b) → c) ↔ (a → (b → c))
un_currying :: Aequivalent ((a,b) -> c) (a -> (b -> c))
un_currying = (curry, uncurry)

-- Universelle Eigenschaft des Produkts
-- Arithmetik: (a * b)^c = a^c * a^b
-- Logik: (a → (b ∧ c)) ↔ ((a → b) ∧ (a → c))
univ_eig_prod :: Aequivalent (a -> (b,c)) (a -> b, a -> c)
univ_eig_prod = (\f -> (fst . f, snd . f), \(f,g) -> \a -> (f(a),g(a)))

-- Universelle Eigenschaft der disjunkten Vereinigung
-- Arithmetik: c^(a+b) = c^a * c^b
-- Logik: ((a ∨ b) → c) ↔ ((a → c) ∧ (b → c))
univ_eig_coprod :: Aequivalent (Either a b -> c) (a -> c, b -> c)
univ_eig_coprod = (\f -> (f . Left, f . Right), uncurry either)

-- Arithmetik: a^1 = a
-- Logik: (⊤ → a) ↔ a
hoch_eins :: Aequivalent (() -> a) a
hoch_eins = (\f -> f(), const)

-- Arithmetik: a^0 = 1
-- Logik: (⊥ → a) ↔ ⊤
ex_falso :: Aequivalent (Void -> a) ()
ex_falso = (const (), const absurd)
```

Lust auf mehr?

- For the fun: `tinyurl.com/hsqed`
- For the ects: Vorlesung zu Beweis-Assistenten

Lust auf mehr?

- For the fun: `tinyurl.com/hsqed`
- For the ects: Vorlesung zu Beweis-Assistenten

Botschaft

- Jedes (terminierende) (Haskell-)Programm beweist eine (intuitionistische) logische Formel
- Arithmetik, Logik, Typen haben ähnliche Eigenschaften

Lust auf mehr?

- For the fun: `tinyurl.com/hsqed`
- For the ects: Vorlesung zu Beweis-Assistenten

Botschaft

- Jedes (terminierende) (Haskell-)Programm beweist eine (intuitionistische) logische Formel
- Arithmetik, Logik, Typen haben ähnliche Eigenschaften

Unterschiedliche Namen für das selbe

- (Un-)Currying: `(a,b) -> c` $\leftrightarrows$ `a -> (b -> c)`
- Adjungiertheit von $\wedge b$ zu $b \rightarrow$: $a \wedge b \rightarrow c \leftrightarrow a \rightarrow (b \rightarrow c)$
- Potenzgesetz $c^{a \cdot b} = (c^b)^a$

Ende der Bildschirmpräsentation. Zum Beenden klicken.

Negation

In der Logik: $\neg a \; := a \to \bot$

Intuitionistische Logik: „Tertium non datur“ kann nicht verwendet werden.

Es gibt keinen intuitionistischen Beweis für $a \vee \neg a$, $\neg\neg a \to a$, …

Negation

In der Logik: $\neg a \ := a \to \bot$
Intuitionistische Logik: „Tertium non datur“ kann nicht verwendet werden.
Es gibt keinen intuitionistischen Beweis für $a \vee \neg a$, $\neg\neg a \to a$, …

Haskell

```haskell
type Not a = (a -> Void)
```

Es gibt kein Haskell-Programm des Typs `Either a (Not a)`.

Negation

In der Logik: $\neg a \ := a \to \bot$
Intuitionistische Logik: „Tertium non datur“ kann nicht verwendet werden.
Es gibt keinen intuitionistischen Beweis für $a \vee \neg a$, $\neg\neg a \to a$, …

Haskell

```
type Not a = (a -> Void)
```

Es gibt kein Haskell-Programm des Typs `Either a (Not a)`.

```
type Not a = a -> Void

consist :: Not (a, Not a)
consist (a,not_a) = not_a a

dneg :: (Either a (Not a)) -> (Not (Not a) -> a)
dneg tnd not_not_a = either id (absurd . not_not_a) tnd
```

Intuitionistisch = eine Konnektive mehr

Klassisches *a* oder *b*

$$\neg(\neg a \wedge \neg b)$$

„Es gilt *a* oder *b*, aber ich weiß nicht welches von beiden.“

```
type Oder a b = (a -> Void, b -> Void) -> Void
```

Intutionistisches *a* oder *b*

$$a \vee b$$

„Es gilt a oder b, und ich weiß welches von beiden“

```
data Either a b = Left a | Right b
```