LC50 The Curry-Howard homeomorphism

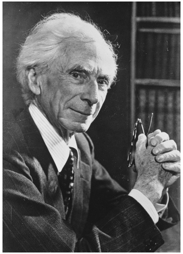
# GADTs and Dependently Typed Programming

FP2 Lecture 2 | Philipp Schröppel | May 3, 2024

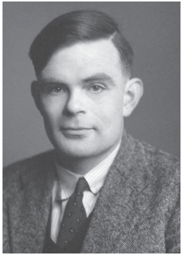
# 1. A Brief History of Types and Proofs

# Computability, Paradoxes and Types

David Hilbert



Bertrand Russell



Alan Turing

Alonzo Church

# Revisiting the Simply Typed Lambda Calculus

**Syntax of Simply Typed Lambda Calculus**

Recall the definition of the Lambda Calculus:

$$e ::= x \mid \lambda x.e \mid e\, e \tag{1}$$

Given a set of base types $B = \{b_1, b_2, ..., b_N\}$ we define the type syntax

$$\tau ::= \tau \to \tau | T, \text{ where } T \in B \tag{2}$$

# Revisiting the Simply Typed Lambda Calculus

## Syntax of Simply Typed Lambda Calculus

Recall the definition of the Lambda Calculus:

$$e ::= x \mid \lambda x.e \mid e\ e \tag{1}$$

Given a set of base types $B = \{b_1, b_2, ..., b_N\}$ we define the type syntax

$$\tau ::= \tau \rightarrow \tau | T, \text{ where } T \in B \tag{2}$$

## Examples of Typed Lambda Terms

$$\lambda x : b_1.x \text{ has type } b_1 \rightarrow b_1$$

$$\lambda x : b_1.\lambda f : b_1 \rightarrow b_2.fx \text{ has type } b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2$$

# Revisiting the Simply Typed Lambda Calculus

## Typing Rules in the Simply Typed Lambda Calculus

We want to specify which lambda terms are well-typed by defining a relation between terms and types.

- A typing assumption has the form $x : b$, meaning variable $x$ has type $b$
- A typing context $\Gamma$ is a set of typing assumptions
- A typing relation $\Gamma \vdash e : b$ indicates that $e$ has type $b$ in the typing context $\Gamma$
- Typing assumptions in the typing context $\Gamma$ are assumed to be well-typed
- Type assumptions outside the typing context are well-typed if they can be derived from the assumptions in the typing context $\Gamma$.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad (3) \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma . e) : (\sigma \to \tau)} \quad (4) \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \ , \ \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 \, e_2 : \tau} \quad (5)$$

# The Curry-Howard Correspondance

## Empty and Nonempty Types

A type is called nonempty if there is a closed term of that type.

## Types from our examples our nonempty

The types

$$b_1 \to b_1, \; b_1 \to (b_1 \to b_2) \to b_2$$

are nonempty.

## Not all types are nonempty

Consider the type

$$(b_1 \to b_2) \to b_1$$

Is it empty?

# The Curry-Howard Correspondance

## A Subset of Propositional Logic

We can construct a subset of propositional logic terms based on the following grammar:

$$p ::= b \mid p \Rightarrow p \tag{6}$$

$$\frac{p \in \Gamma}{\Gamma \vdash p} \quad (7) \qquad \frac{\Gamma, p_1 \vdash p_2}{\Gamma \vdash p_1 \rightarrow p_2} \quad (8) \qquad \frac{\Gamma \vdash p_1,\ \Gamma \vdash p_1 \rightarrow p_2}{\Gamma \vdash p_2} \quad (9)$$

# The Curry-Howard Correspondance

## A Subset of Propositional Logic

We can construct a subset of propositional logic terms based on the following grammar:

$$p ::= b \mid p \Rightarrow p \tag{6}$$

$$\frac{p \in \Gamma}{\Gamma \vdash p} \quad (7) \qquad \frac{\Gamma, p_1 \vdash p_2}{\Gamma \vdash p_1 \rightarrow p_2} \quad (8) \qquad \frac{\Gamma \vdash p_1,\ \Gamma \vdash p_1 \rightarrow p_2}{\Gamma \vdash p_2} \quad (9)$$

## Comparison to Typing Rules

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma.e) : (\sigma \rightarrow \tau)} \qquad \frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau\ ,\ \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}$$

# The Curry-Howard Correspondance

## Curry-Howard Correspondance

| Algebra | Logic | (Haskell) Types |
| --- | --- | --- |
| $a + b$ | $a \vee b$ | Either a b |
| $a \cdot b$ | $a \wedge b$ | $(a, b)$ |
| $a^b$ | $a \Rightarrow b$ | $a \to b$ |
| $a = b$ | $a \Leftrightarrow b$ | $a$ isomorphic to $b$ |
| 0 | $\top$ | Void |
| 1 | $\perp$ | () |

# The Curry-Howard Correspondance

## Curry-Howard Correspondance

| Algebra | Logic | (Haskell) Types |
|---|---|---|
| $a + b$ | $a \vee b$ | Either a b |
| $a \cdot b$ | $a \wedge b$ | $(a, b)$ |
| $a^b$ | $a \Rightarrow b$ | $a \rightarrow b$ |
| $a = b$ | $a \Leftrightarrow b$ | $a$ isomorphic to $b$ |
| 0 | $\top$ | Void |
| 1 | $\bot$ | () |

## Limitations of the Example

This is a simplified and incomplete representation of the correspondance between types in simply typed Lambda calculus and terms in propositional logic.

# Dependent Types

### Implications of the Curry-Howard Correspondance

- The Curry-Howard Correspondance describes a correspondence between a given logic and a type system.
- For each proposition in the logic there is a corresponding non-empty type in the type system.
- For each proof of a given proposition, there is a program of the corresponding type.
- The correspondance applies not only to the simply-typed lambda calculus and propositional logic, but extends to more sophisticated logical calculi and type sytems (Girard-Reynolds, Hindley-Milner, ...).

### Dependent Types

A dependent type is a type whose definition depends on a value.

# Dependent Types

## $\Pi$ Type

The $\Pi$ Type is the type of function whose type of return value depends on the value of its argument.

Notation: $\Pi_{x:A}B(x)$, where $A$ is a family of types and $B : A \to U$ is a family of types.

$\Pi$ Types encode universal quantification: A function of type $\Pi_{x:A}B(x)$ assigns a type $B(a)$ to every $a \in A$

## $\Pi$ Type

A function which maps natural numbers $n \in \mathbb{N}$ to $n$-tuples of floats has type $\Pi_{n:\mathbb{N}}\mathrm{Vec}(\mathbb{R}, n)$.

# Dependent Types

**$\Sigma$ Type**

The $\Sigma$ Type is the type of a tuple in which the type of the second term depends on the value of the first term.

Notation: $\Sigma_{x:A} B(x)$, where $A$ is a family of types and $B : A \to U$ is a family of types.

$\Sigma$ Types encode existential quantification: There is a $a \in A$ such that $B(a)$ is inhabited

**$\Sigma$ Type**

A tuple consisting where the first term is a natural number and second term contains natural numbers greater than the value of the first term has type $\Sigma_{n:\mathbb{N}} \mathbb{N}_{>n}$.

# 2. GADTs

# From ADTs to GADTs

**Warning: Compiler Extensions Required**

```haskell
{-# LANGUAGE DataKinds       #-}
{-# LANGUAGE GADTs           #-}
{-# LANGUAGE RankNTypes      #-}
{-# LANGUAGE TypeFamilies    #-}
{-# LANGUAGE TypeOperators   #-}
```

# From ADTs to GADTs

**An Expression for Adding Integers**

```haskell
data ExprV1 = ExprV1 Int | AddV1 ExprV1 ExprV1
```

# From ADTs to GADTs

### An Expression for Adding Integers

```haskell
data ExprV1 = ExprV1 Int | AddV1 ExprV1 ExprV1
```

### Kinds & Types of Type and Data Constructors

```haskell
:k ExprV1
-- ExprV1 :: *
:t ExprV1
-- ExprV1 :: Int -> ExprV1
:t AddV1
-- AddV1 :: ExprV1 -> ExprV1 -> ExprV1
```

# From ADTs to GADTs

### An Expression for Adding Integers

```haskell
data ExprV1 = ExprV1 Int | AddV1 ExprV1 ExprV1
```

### Kinds & Types of Type and Data Constructors

```haskell
:k ExprV1
-- ExprV1 :: *
:t ExprV1
-- ExprV1 :: Int -> ExprV1
:t AddV1
-- AddV1 :: ExprV1 -> ExprV1 -> ExprV1
```

### Evaluating our Expressions

```haskell
evalExprV1 :: ExprV1 -> Int
evalExprV1 (ExprV1 k) = k
evalExprV1 (AddV1 e1 e2) = evalExprV1 e1 + evalExprV1 e2
```

# From ADTs to GADTs

**Extending Our Expression with Conditionals**

```haskell
data ExprV2 a where
  IntV2  :: Int -> ExprV2 Int
  BoolV2 :: Bool -> ExprV2 Bool
  AddV2 :: ExprV2 Int -> ExprV2 Int -> ExprV2 Int
  NotV2 :: ExprV2 Bool -> ExprV2 Bool
  IfV2 :: ExprV2 Bool -> ExprV2 a -> ExprV2 a -> ExprV2 a
```

# From ADTs to GADTs

## Extending Our Expression with Conditionals

```haskell
data ExprV2 a where
  IntV2  :: Int -> ExprV2 Int
  BoolV2 :: Bool -> ExprV2 Bool
  AddV2 :: ExprV2 Int -> ExprV2 Int -> ExprV2 Int
  NotV2 :: ExprV2 Bool -> ExprV2 Bool
  IfV2 :: ExprV2 Bool -> ExprV2 a -> ExprV2 a -> ExprV2 a
```

## Kinds & Types of Type and Data Constructors

```haskell
:k ExprV2
-- ExprV2 :: * -> *
:t IntV2
-- IntV2 :: Int -> ExprV2 Int
:t AddV2
-- AddV2 :: ExprV2 Int -> ExprV2 Int
```

# GADTs: Pattern Matching Leads to Type Refinement

**Evaluating our Extended Expressions**

```
evalExprV2 :: ExprV2 a -> a
evalExprV2 (IntV2 i) = i
evalExprV2 (BoolV2 b) = b
evalExprV2 (AddV2 x y) = evalExprV2 x + evalExprV2 y
evalExprV2 (NotV2 x) = not $ evalExprV2 x
evalExprV2 (IfV2 c x y) = if evalExprV2 c
                          then evalExprV2 x
                          else evalExprV2 y
```

# 3. Dependently Typed Programming in Haskell

# Lists are Unsafe

**Run-Time Errors**

```haskell
head []
-- *** Exception: Prelude.head: empty list
maximum []
-- *** Exception: Prelude.head: empty list
["foo"] !! 1
-- *** Exception: Prelude.!!: index too large
```

# GADTs Can Alleviate Some Problems

**Nonempty Lists with GADTs**

```haskell
data ContainerStatus = Empty | NonEmpty

data SafeList :: * -> ContainerStatus -> * where
  Nil :: SafeList a Empty
  Cons :: a -> SafeList a b -> SafeList a NonEmpty

safeHead :: SafeList a NonEmpty -> a
safeHead (Cons x _) = x
```

# GADTs Can Alleviate Some Problems

## Nonempty Lists with GADTs

```haskell
data ContainerStatus = Empty | NonEmpty

data SafeList :: * -> ContainerStatus -> * where
  Nil :: SafeList a Empty
  Cons :: a -> SafeList a b -> SafeList a NonEmpty

safeHead :: SafeList a NonEmpty -> a
safeHead (Cons x _) = x
```

## Turning Run-Time into Compile-Time Errors

```haskell
safeHead $ Cons "bar" Nil
-- "bar"
safeHead $ Nil
-- error: Couldn't match type ‘Empty’ with ‘NonEmpty’
```

# Fixed Length Vectors with Singleton Types

## Natural Numbers and Fixed Length Lists

```haskell
data Nat where
  Zero :: Nat
  Succ :: Nat -> Nat

data Vec :: * -> Nat -> * where
  VNil :: Vec a 'Zero
  VCons :: a -> Vec a n -> Vec a ('Succ n)
```

## Constructing Nats and Vecs

```haskell
Succ (Succ Zero) -- 2
VCons 2 (VCons 1 Nil) -- the vector [2, 1]
:t VCons 2 (VCons 1 VNil)
-- VCons 2 (VCons 1 VNil) :: Num a => Vec a ('Succ ('Succ 'Zero))
```

# Fixed Length Vectors with Singleton Types

## A Safe Head Function

```haskell
safeHead' :: Vec a ('Succ n) -> a
safeHead' (VCons x _) = x
```

## Compile-Time Error

```haskell
safeHead' (VCons "foo" VNil)
-- "foo"
safeHead' VNil
-- error: Couldn't match type ‘'Zero’ with ‘'Succ n0’
```

# Fixed Length Vectors with Singleton Types

**Comparing Nats (at Compile-Time!)**

```
type family (m::Nat) :< (n::Nat) :: Bool
type instance m :< 'Zero = 'False
type instance Zero :< (Succ n) = 'True
type instance ('Succ m) :< (Succ n) = m :< n
```

# Fixed Length Vectors with Singleton Types

## Safe Access (with Gaps)

```
nth :: (m:<n) ~ 'True => --?-- -> Vec a n -> a
nth --?-- (VCons a _)         = a
nth --?-- (VCons _ as) = nth sm' as
```

# Fixed Length Vectors with Singleton Types

## Safe Access (with Gaps)

```haskell
nth :: (m:<n) ~ 'True => --?-- -> Vec a n -> a
nth --?-- (VCons a _)         = a
nth --?-- (VCons _ as) = nth sm' as
```

## Singleton Nats

```haskell
data SNat :: Nat -> * where
  SZero :: SNat 'Zero
  -- SSucc :: forall (n::Nat).SNat n -> SNat ('Succ n)
  SSucc :: SNat n -> SNat ('Succ n)
```

# Fixed Length Vectors with Singleton Types

## Safe Access

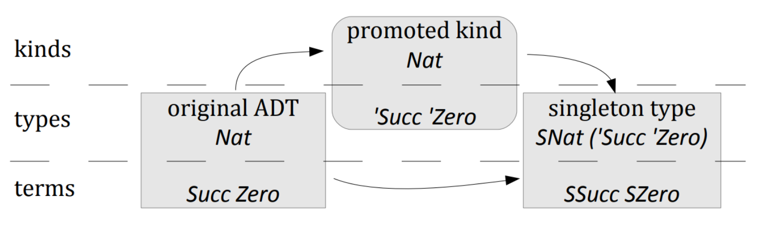
```haskell
nth :: (m:<n) ~ 'True => SNat m -> Vec a n -> a
nth SZero (VCons a _)        = a
nth (SSucc sm') (VCons _ as) = nth sm' as
```

## Compile-Time Error

```haskell
nth SZero (VCons "foo" VNil)
-- "foo"
nth (SSucc SZero) (VCons "foo" VNil)
-- error: Couldn't match type ''False' with ''True'
```

# Fixed Length Vectors with Singleton Types

## Singleton Type Generation




*Eisenberg, R., & Weirich, S. (2012). Dependently typed programming with singletons. In Proceedings of the 2012 Haskell Symposium (pp. 117–130). Association for Computing Machinery.*

# Singleton Types - a Poor Man's Substitute for Dependent Types?

**What is a dependently typed language?**

In a dependently typed language, types depend on run-time values.

**Is Haskell a dependently typed language?**

Elements of a dependently typed language are supported by language extensions:

- There is a notion of type-level data with typing provided by DataKinds.
- We can use GADTs to create run-time representation of type-level data (using Singletons)

Using Singletons oftentimes requires duplicate code for term- and type-level data. The *Singletons* library can help to create this boilerplate code using *Template Haskell*.

LC90 The Curry-Howard homeomorphism

# GADTs and Dependently Typed Programming

FP2 Lecture 2 | Philipp Schröppel | May 3, 2024