

# Generating code with the Curry-Howard correspondence

## Type inhabitation at compile time

Sergei Winitzki

SF Types, Theorems, and Programming Languages

February 12, 2018

# Types and propositional logic

## The Curry-Howard correspondence

The code `val x: T = ...` shows that *we can compute a value* of type `T` as part of our program expression

- Let's denote this *proposition* by  $\mathcal{CH}(T)$  – “Code  $\mathcal{H}$  has a value of type `T`”
- Correspondence between types and propositions, for a given program:

Type	Proposition	Short notation
<code>T</code>	$\mathcal{CH}(T)$	$T$
<code>(A, B)</code>	$\mathcal{CH}(A)$ and $\mathcal{CH}(B)$	$A \wedge B$ ; $A \times B$
<code>Either[A, B]</code>	$\mathcal{CH}(A)$ or $\mathcal{CH}(B)$	$A \vee B$ ; $A + B$
<code>A <math>\Rightarrow</math> B</code>	$\mathcal{CH}(A)$ implies $\mathcal{CH}(B)$	$A \Rightarrow B$
<code>Unit</code>	<i>True</i>	1
<code>Nothing</code>	<i>False</i>	0

- Type parameter `[T]` in a function type means  $\forall T$
- Example: `def dupl[A]: A  $\Rightarrow$  (A, A)`. The type of this function corresponds to the (valid) theorem  $\forall A : A \Rightarrow A \wedge A$

# The CH correspondence: proposition $\rightarrow$ type / proof $\rightarrow$ code

- Any valid theorem can be implemented in code

Proposition	Code
$\forall A : A \Rightarrow A$	<code>def identity[A](x: A): A = x</code>
$\forall A : A \Rightarrow 1$	<code>def toUnit[A](x: A): Unit = ()</code>
$\forall A \forall B : A \Rightarrow A + B$	<code>def inLeft[A,B](x:A): Either[A,B] = Left(x)</code>
$\forall A \forall B : A \times B \Rightarrow A$	<code>def first[A,B](p: (A,B)): A = p._1</code>
$\forall A \forall B : A \Rightarrow (B \Rightarrow A)$	<code>def const[A,B](x: A): B <math>\Rightarrow</math> A = (y:B) <math>\Rightarrow</math> x</code>

- Non-theorems *cannot be implemented* in code

- Examples of non-theorems:

$$\forall A : 1 \Rightarrow A; \quad \forall A \forall B : A + B \Rightarrow A;$$

$$\forall A \forall B : A \Rightarrow A \times B; \quad \forall A \forall B : (A \Rightarrow B) \Rightarrow A$$

- Given a type's formula, can we implement it in code?

- Example:  $\forall A \forall B : (((A \Rightarrow B) \Rightarrow A) \Rightarrow A) \Rightarrow B$

- Constructive (intuitionistic) propositional logic has a decision algorithm
- The **curryhoward** library implements the IPL prover in a Scala macro

# Worked examples I

- 1 Implement `map` for the Reader monad,

$$\text{map} : (E \Rightarrow A) \Rightarrow (A \Rightarrow B) \Rightarrow (E \Rightarrow B)$$

- 2 Show that one cannot implement  $(E \Rightarrow A) \Rightarrow (E \Rightarrow F) \Rightarrow (F \Rightarrow A)$
- 3 Implement `map[A,B]: Option[A] => (A => B) => Option[B]`

Often, there is only one useful implementation

The `curryhoward` library tries to generate that implementation automatically

# Using the curryhoward library

Two main use cases:

- 1 Define a method and provide an automatic implementation

```
def map[E, A, B](readerA: E  $\Rightarrow$  A, f: A  $\Rightarrow$  B): E  $\Rightarrow$  B = implement
```

- 2 Automatically build an expression from previously computed values

```
val f: String  $\Rightarrow$  Boolean  $\Rightarrow$  Int = {...}  
case class Result(x: Int, name: String)  
val result = ofType[Result]("abc", f, true)
```

Features:

- Compile-time code generation via Scala macros
- Supports functions, tuples, sealed trait / case classes / case objects
- Constant types (`Int`, `String`, etc.) are treated as type parameters
- If several implementations are available, chooses “intelligently”

# Worked examples II

Demo time

- 1 Implement `map: Option[A] ⇒ (A ⇒ B) ⇒ Option[B]` that satisfies the identity law: `map(opt)(x ⇒ x) = opt`
- 2 Show that one cannot implement  $(E \Rightarrow A) \Rightarrow (E \Rightarrow F) \Rightarrow (F \Rightarrow A)$
- 3 Implement the distributive law

$$(A + B) \times C \Leftrightarrow A \times C + B \times C$$

In Scala: `(Either[A, B], C) ⇔ Either[(A, C), (B, C)]`

- 4 Implement `point`, `map` and `flatMap` for the Reader and State monads

See test code

# Proof search I: Gentzen's calculus LJ (1935)

- A “complete and sound calculus” is a set of axioms and derivation rules that will yield all (and only!) valid theorems of the logic

$$\begin{array}{c}
 (X \text{ is atomic}) \frac{}{\Gamma, X \vdash X} Id \\
 \frac{\Gamma, A \Rightarrow B \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C} L \Rightarrow \\
 \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C} L+ \\
 \frac{\Gamma, A_i \vdash C}{\Gamma, A_1 \wedge A_2 \vdash C} L \times_i \\
 \frac{}{\Gamma \vdash \top} \top \\
 \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} R \Rightarrow \\
 \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 \vee A_2} R+_i \\
 \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} R \times
 \end{array}$$

- Sequents are nodes in the proof search tree
- Use these rules “bottom-up” to perform a proof search
- Example:  $\emptyset \vdash ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$

# Proof search example I

Root sequent  $S_0 : \emptyset \vdash ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$

- $S_0$  with rule  $R \Rightarrow$  yields  $S_1 : (R \Rightarrow R) \Rightarrow Q \vdash Q$
- $S_1$  with rule  $L \Rightarrow$  yields  $S_2 : (R \Rightarrow R) \Rightarrow Q \vdash R \Rightarrow R$  and  $S_3 : Q \vdash Q$
- Sequent  $S_3$  follows from the *Id* axiom; it remains to prove  $S_2$
- $S_2$  with rule  $L \Rightarrow$  yields  $S_4 : (R \Rightarrow R) \Rightarrow Q \vdash R \Rightarrow R$  and  $S_5 : Q \vdash R \Rightarrow R$ 
  - ▶ We are stuck here because  $S_4 = S_2$  (we are in a loop)
  - ▶ We can prove  $S_5$ , but that will not help
  - ▶ So we backtrack (erase  $S_4, S_5$ ) and apply another rule to  $S_2$
- $S_2$  with rule  $R \Rightarrow$  yields  $S_6 : (R \Rightarrow R) \Rightarrow Q; R \vdash R$
- Sequent  $S_6$  follows from the *Id* axiom

Therefore we have proved  $S_0$ .

*Q.E.D.*



# Proof search II: From deduction rules to code

- Proofs are the  $\lambda$ -calculus terms arising from deduction rules
- Proof of a sequent  $A, B, C \vdash G \Leftrightarrow \text{code/expression } g(a, b, c) : G$
- Each rule has a *proof transformer* function:  $\text{PT}_{R\Rightarrow}$ ,  $\text{PT}_{L+}$ , etc.
- Example: to prove  $S_0$ , start from  $S_6$  backwards:

$$S_6 : (R \Rightarrow R) \Rightarrow Q; R \vdash R \quad (\text{axiom } Id) \quad t_6(rrq, r) : R = r$$

$$S_2 : (R \Rightarrow R) \Rightarrow Q \vdash (R \Rightarrow R) \quad \text{PT}_{R\Rightarrow}(t_6) \quad t_2(rrq) : (R \Rightarrow R) = (r \Rightarrow t_6(rrq, r))$$

$$S_3 : Q \vdash Q \quad (\text{axiom } Id) \quad t_3(q) : Q = q$$

$$S_1 : (R \Rightarrow R) \Rightarrow Q \vdash Q \quad \text{PT}_{L\Rightarrow}(t_2, t_3) \quad t_1(rrq) : Q = t_3(rrq(t_2(rrq)))$$

$$S_0 : \emptyset \vdash ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q \quad \text{PT}_{R\Rightarrow}(t_1) \quad t_0 = (rrq \Rightarrow t_1(rrq))$$

- Simplified final code (proof term) having the required type:

$$t_0 : ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q = (rrq \Rightarrow rrq(r \Rightarrow r))$$

# Proof search III: The calculus LJT

Vorobieff-Hudelmaier-Dyckhoff, 1950-1990

- The Gentzen calculus generates a loop if rule  $L \Rightarrow$  is applied  $\geq 2$  times
- The calculus LJT keeps all rules of LJ except rule  $L \Rightarrow$
- Replace rule  $L \Rightarrow$  by pattern-matching on  $A$  in the premise  $A \Rightarrow B$ :

$$\begin{array}{c} (X \text{ is atomic}) \frac{\Gamma, X, B \vdash D}{\Gamma, X, X \Rightarrow B \vdash D} L \Rightarrow_1 \\ \frac{\Gamma, A \Rightarrow (B \Rightarrow C) \vdash D}{\Gamma, (A \wedge B) \Rightarrow C \vdash D} L \Rightarrow_2 \\ \frac{\Gamma, A \Rightarrow C, B \Rightarrow C \vdash D}{\Gamma, (A \vee B) \Rightarrow C \vdash D} L \Rightarrow_3 \\ \frac{\Gamma, B \Rightarrow C \vdash A \Rightarrow B \quad \Gamma, C \vdash D}{\Gamma, (A \Rightarrow B) \Rightarrow C \vdash D} L \Rightarrow_4 \end{array}$$

- Rule  $L \Rightarrow_4$  is based on the key theorem:

$$((A \Rightarrow B) \Rightarrow C) \Rightarrow (A \Rightarrow B) \iff (B \Rightarrow C) \Rightarrow (A \Rightarrow B)$$

# Proof search IV: The calculus LJT

“*It is obvious that it is obvious*” – a mathematician after thinking for a half-hour

- The key theorem for rule  $L \Rightarrow_4$  is attributed to Vorobieff (1958):

be extracted from Lemma 7 in [22]. One could also go further and make subproofs sensible.

LEMMA 2.  $\vdash_{LJ} \Gamma, (C \supset D) \supset B \Rightarrow C \supset D$  iff  $\vdash_{LJ} \Gamma, D \supset B \Rightarrow C \supset D$ .

PROOF. Trivial [34].

THEOREM 1. *The systems LJ and LJT are equivalent.*

PROOF. As noted earlier, it is routine to show that any sequent provable

[R. Dyckhoff, *Contraction-Free Sequent Calculi for Intuitionistic Logic*, 1992]

- A stepping stone to this theorem:

$$((A \Rightarrow B) \Rightarrow C) \Rightarrow B \Rightarrow C$$

Proof (*obviously* trivial):  $f \Rightarrow b \Rightarrow f (\_ \Rightarrow b)$

# Making practical use of the CH correspondence

## Implications for actually writing code

What can we do now?

- Given a fully parametric type, decide whether it can be implemented in code (“type is inhabited”); if so, *generate* the code
- Let `curryhoward` fill in the code when it is “trivial” to do so
- Use symbolic terms to check results

What problems cannot be solved with these tools?

- Automatically generate code satisfying properties (e.g. isomorphism)
  - ▶ The heuristics will help in some cases
  - ▶ One of the implementations can be chosen at run time
- Express complicated conditions via types (e.g. “array is sorted”)
  - ▶ Need dependent types for that (Coq, Agda, Idris, ...)
- Generate code using type constructors with properties (e.g. `map`)
  - ▶ First-order logic is undecidable

## *Generating code with the Curry-Howard correspondence: Type inhabitation at compile time*

I implemented a library for compile-time code generation from Scala type signatures. The library uses (compile-time) reflection, the Curry-Howard correspondence, and a theorem prover for the constructive propositional logic. Using this library, I illustrate how the Curry-Howard correspondence maps types into propositions and proofs into code. I will also explain some details of the algorithm I used for automatic code generation from type signatures. As an illustration of using this library for automatic code generation, I demonstrate working examples such as implementing `map` and `flatMap` for the Reader and State monads.

- **D. Galmiche, D. Larchey-Wendling** – *Formulae-as-Resources Management for an Intuitionistic Theorem Prover* (1998). In 5th Workshop on Logic, Language, Information and Computation, WoLLIC'98, Sao Paulo.
- **R. Dyckhoff** – *Contraction-free sequent calculi for intuitionistic logic* (1992), The Journal of Symbolic Logic, Vol. 57, No. 3, (Sep., 1992), pp. 795-807.
- **R. Dyckhoff** – *Intuitionistic decision procedures since Gentzen* (2013), talk slides