A brief tutorial on the Curry-Howard correspondence

For programmers, with code examples in Scala

Sergei Winitzki

Academy By the Bay

September 19, 2020

What problems does Curry-Howard correspondence solve?

The CH correspondence is a theory that answers these questions:

1. Can a program compute a value of type X given values of some types A, B, C, ...? Example:

```
def f[A, B, C](x: A => Option[B], y: Either[A, C], z: C => B): B = {
   val x: Either[B, C] = ??? // Can we implement ‘x‘ here?
   x.map(z).merge
}
```

2. Can we infer the code of a function from its type signature? Examples:

```
def f[A, B, C]: (A => Either[B, C]) => Either[A, C] => Either[B, C]
def g[A, B, C]: (A => Either[B, C]) => Either[A => B, A => C]
def h[A, B]: ((((A => B) => A) => A) => B) => B
```

Theory gives an algorithm for writing code "guided by the types"

- The curryhoward library generates Scala code from type signatures
  - Often, there is only one "useful" implementation out of many
    - The `curryhoward` library tries to find that implementation

```
def h[A, B]: ((((A => B) => A) => A) => B) => B  =  implement
```

From types to logical propositions I. $\mathcal{CH}$-propositions

- How to *prove* that this function is not implementable?

  `def bad[A, B, C](x: A => Option[B], y: Either[A, C], z: C => B): B`

The idea is to build a system of logical derivation rules and axioms (a **logic**)
The logic should be able to prove rigorously whether any code expression in the body of the function `bad` can compute values of type `B`

- Denote such *propositions* by $\mathcal{CH}(B)$ – "$\mathcal{C}$ode $\mathcal{H}$as a value of type `B`"

How to obtain rules for reasoning about $\mathcal{CH}$-propositions?

- The code of `bad` might contain expressions such as `y.map(z)`
  - This computes a value of type `Either[A, B]` from values of types `Either[A, C]` and `(C => B)`
- Code expressions create *logical relationships* between $\mathcal{CH}$-propositions
  - "Logical relationship": X can be proved true if A, B, C are true
  - In logic, such a proof task is represented by a **sequent**
    - Notation: $A, B, C \vdash X$; the **premises** are A, B, C and the **goal** is X
  - Proofs are achieved via axioms and derivation rules
    - Axioms: sequents that are true without proof
    - Derivation rules: prove a sequent given proofs of some other sequent(s)

From types to logical propositions II. Fully parametric code

To determine the logical relationships between types, we need to know *all possible code snippets*

"Fully parametric" code allows only combinations of these snippets:

- Use an existing value `x` of type `A`. Scala code: `val y:  A = x`
- Tuple type: `(A, B)`
  - Create: `val pair: (A, B) = (a, b)`
  - Use: `val y: B = pair._2`
- Function type: `A => B`
  - Create: `def f: (A => B) = { x:  A => ...  /*`(may use `x` here)`*/ }`
  - Use: `val y: B = f(a)`
- Disjunctive type: `Either[A, B]`
  - Create:
    `val x: Either[A, B] = Left(a); val y: Either[A, B] = Right(b)`
  - Use:
    ```
    val z: C = x match {
     case Left(a) => ...
     case Right(b) => ...
    }
    ```
- Unit type: `Unit`
  - Create: `val x: Unit = ()`

From types to logical propositions III. Sequents

- Sequents correspond to code fragments that have specified types
- A sequent $\mathcal{CH}(X), \mathcal{CH}(Y) \vdash \mathcal{CH}(Z)$ corresponds to an *expression* of type `Z` that uses some previously defined values `x:X` and `y:Y`
  - Sequents only describe the *types* of expressions and their parts
- Each allowed code snippet means that we can compute a value of some type given value(s) of other type(s)

Express this in sequent notation as **derivation rules**:

- Use an existing value: $\mathcal{CH}(A) \vdash \mathcal{CH}(A)$
- Create tuple: $\mathcal{CH}(A), \mathcal{CH}(B) \vdash \mathcal{CH}($`Tuple2(A,B)`$)$
- Use tuple: $\mathcal{CH}($`Tuple2(A,B)`$) \vdash \mathcal{CH}(A)$ and $\mathcal{CH}($`Tuple2(A,B)`$) \vdash \mathcal{CH}(B)$
- Create function: $\emptyset \vdash \mathcal{CH}($`A => B`$)$ if given $\mathcal{CH}(A) \vdash \mathcal{CH}(B)$
  - Function body is an expression of type `B` that uses `x` of type `A`
- Use function: $\mathcal{CH}($`A => B`$), \mathcal{CH}(A) \vdash \mathcal{CH}(B)$
- Create disjunctive value: $\mathcal{CH}(A) \vdash \mathcal{CH}($`Either[A, B]`$)$ and $\mathcal{CH}(B) \vdash \mathcal{CH}($`Either[A, B]`$)$
- Use disjunctive value:
  $\mathcal{CH}($`A => C`$), \mathcal{CH}($`B => C`$), \mathcal{CH}($`Either[A, B]`$) \vdash \mathcal{CH}(C)$
- Create unit value: $\emptyset \vdash \mathcal{CH}($`Unit`$)$

Translating language constructions into the logic I

- If there are no other allowed code snippets, we can summarize the correspondence between type constructors and propositions:

Scala type	Proposition in logic	Short type notation
`A`	$\mathcal{CH}(A)$	A
`(A, B)`	$\mathcal{CH}(A) \wedge \mathcal{CH}(B)$	$A \times B$
`Either[A, B]`	$\mathcal{CH}(A) \vee \mathcal{CH}(B)$	$A + B$
`A => B`	$\mathcal{CH}(A) \Rightarrow \mathcal{CH}(B)$	$A \rightarrow B$
`()`	$True$; $\top$	$\mathbb{1}$
`Nothing`	$False$; $\bot$	$\mathbb{0}$

We can now translate types into logic formulas and back

- Example: `def duplicate[A]: A => (A, A)`
  - The type of this function in the short type notation is $A \rightarrow A \times A$
  - This corresponds to the logical formula $\forall A.\, \mathcal{CH}(A) \Rightarrow \mathcal{CH}(A) \wedge \mathcal{CH}(A)$
- The question about the function `bad` is written in logic as this sequent:

$$\mathcal{CH}(A \rightarrow \mathbb{1} + B), \mathcal{CH}(A + C), \mathcal{CH}(C \rightarrow B) \vdash \mathcal{CH}(B)$$

Translating language constructions into the logic II

What are the axioms and the derivation rules in the logic of types?

The set of *all well-typed fully parametric programs* $\cong$ the set of *all valid derivations in the logic of types*

- Write just A instead of $\mathcal{CH}(A)$ and use short type notation
- Axioms:
  - $\emptyset \vdash \top$ – create the value of unit type
  - $A \vdash A$ – use variable
  - $A, B \vdash (A \times B)$ – create tuple
  - $(A \times B) \vdash A$ – use left part of tuple
  - $(A \times B) \vdash B$ – use right part of tuple
  - $A, (A \rightarrow B) \vdash B$ – apply function to argument
  - $A \vdash (A + B)$ – create left part of `Either`
  - $B \vdash (A + B)$ – create right part of `Either`
  - $(A + B), (A \rightarrow C), (B \rightarrow C) \vdash C$ – use `Either` via match/case
- Derivation rules:
  - "create function": we can prove $\emptyset \vdash (A \rightarrow B)$ given $A \vdash B$
  - "add premise": we can prove $A, ..., C, D \vdash G$ given $A, ..., C \vdash G$
  - "reorder": we can prove $B, A, C, ... \vdash G$ given $A, B, C, ... \vdash G$

The logic of types I

Now we have all the axioms and the derivation rules of the logic of types.

- What theorems can we derive in this logic?
- Example theorem: $\forall A.\ \forall B.\ A \to B \to A$ or $\emptyset \vdash A \to B \to A$
  - Start with an axiom $A \vdash A$; add an unused premise B, get $A, B \vdash A$
  - Use the "create function" rule with B and A, get $A \vdash B \to A$
  - Use the "create function" rule with A and $B \to A$, get the final sequent $\emptyset \vdash A \to B \to A$ showing that $\forall A.\ \forall B.\ A \to B \to A$ is a **theorem** since $\emptyset \vdash A \to B \to A$ was derived from no premises for all A and B
- What code does this describe?
  - The axiom $A \vdash A$ represents the expression `x: A`
  - The unused premise B corresponds to unused variable `y: B`
  - The "create function" rule gives the function `{ y: B => x }`
  - The second "create function" rule gives `{ x: A => y: B => x }`
  - Complete code:

```
def f[A, B]: A => B => A = { (x: A) => (y: B) => x }
```

- Any code expression's type can be translated into a sequent
- A proof of a theorem directly guides us in writing code for that type
- The code can be used to find a proof (every code snippet gives a rule)

Correspondence between programs and proofs

- By construction, any theorem can be implemented in code

Proposition	Scala code
$\forall A.\ A \to A$	`{ x: A => x }`
$\forall A.\ A \to 1$	`{ x: A => () }`
$\forall A.\ \forall B.\ A \to A + B$	`Left.apply`
$\forall A.\ \forall B.\ A \times B \to A$	`_._1`
$\forall A.\ \forall B.\ A \to B \to A$	`{ x: A => y: B => x }`

- "Types are propositions, programs are proofs"
- Also, non-theorems *cannot be implemented* in code
  - Examples of non-theorems:
    $\forall A.\ 1 \to A$; $\forall A.\ \forall B.\ A + B \to A$;
    $\forall A.\ \forall B.\ A \to A \times B$; $\forall A.\ \forall B.\ (A \to B) \to A$
- Given a type's formula, can we implement it in code? Not obvious.
  - Example: $\forall A.\ \forall B.\ ((((A \to B) \to A) \to A) \to B) \to B$
    - ★ Can we write a function with this type? Can we prove this formula?

The logic of types II

What kind of logic is this? What do mathematicians call this logic?

This is called “intuitionistic propositional logic”, IPL (also “constructive”)

- This is a “nonclassical” logic because it is different from Boolean logic
- Disjunction works differently from Boolean logic
  - Example: $(A \to B + C) \vdash (A \to B) + (A \to C)$ does not hold in IPL
  - This is counter-intuitive!
  - We cannot implement a function with this type:

    ```
    def q[A, B, C]: (A => Either[B, C]) => Either[A => B, A => C]
    ```

  - Disjunction is “constructive”: need to supply one of the parts
    - ⋆ ...but cannot compute `A => B` or `A => C` from `A => Either[B, C]`
- Implication works differently
  - Example: $((A \to B) \to A) \to A$ holds in Boolean logic but not in IPL
  - Cannot compute an `x: A` because of insufficient data
- Conjunction works the same as in Boolean logic
  - Example: $(A \to B \times C) \vdash (A \to B) \times (A \to C)$

The logic of types III

How to determine whether a given IPL formula is a theorem?

- In Boolean logic, we can compute the "truth value" of a formula and decide whether the formula is a theorem
- The IPL cannot have a truth table with a fixed number of truth values
  - This was proved by Gödel in 1932 (see Wikipedia page)
- The IPL has a decision procedure (algorithm) that either finds a proof for a given IPL formula, or determines that there is no proof
- There may be several inequivalent proofs of an IPL theorem
- Each proof can be *automatically translated* into code
  - The djinn-ghc compiler plugin and the JustDoIt plugin implement an IPL prover in Haskell, and generate Haskell code from types
  - The curryhoward library implements an IPL prover as a Scala macro, and generates Scala code from types
- All these IPL provers use the same basic algorithm called LJT
  - presented in the paper [Dyckhoff 1992]

Proof search I: looking for an algorithm

Why our initial presentation of IPL does not give a proof search algorithm

The FP type constructions give nine axioms and three derivation rules:

- $\Gamma, A, B \vdash A \times B$
- $\Gamma, A \times B \vdash A$
- $\Gamma, A \times B \vdash B$
- $\Gamma, A \rightarrow B, A \vdash B$
- $\Gamma, A \vdash A + B$
- $\Gamma, B \vdash A + B$
- $\Gamma, A + B, A \rightarrow C, B \rightarrow C \vdash C$
- $\Gamma \vdash 1$
- $\Gamma, A \vdash A$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}$$

$$\frac{\Gamma \vdash G}{\Gamma, D \vdash G}$$

$$\frac{\Gamma, A, B \vdash G}{\Gamma, B, A \vdash G}$$

Can we use these rules to obtain a finite and complete search tree? No.

- Try proving $A, B + C \vdash A \times B + C$: cannot find matching rules
  - Need a better formulation of the logic

Proof search II: Gentzen's calculus LJ (1935)

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

$$(A \text{ is atomic}) \; \frac{}{\Gamma, A \vdash A}\, Id \qquad \frac{}{\Gamma \vdash \top}\, \top$$

$$\frac{\Gamma, A \to B \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C}\, L_{\to} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\, R_{\to}$$

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A + B \vdash C}\, L_{+} \qquad \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2}\, R_{+_i}$$

$$\frac{\Gamma, A_i \vdash C}{\Gamma, A_1 \times A_2 \vdash C}\, L_{\times_i} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B}\, R_{\times}$$

- Two axioms and eight derivation rules
  - Each derivation rule says: The sequent at bottom will be proved if proofs are given for sequent(s) at top
  - The symbol Γ means "any number of premises, or $\emptyset$"
- Use these rules "bottom-up" to perform a proof search
  - Sequents are nodes and proofs are edges in the tree of proof search

Proof search example I

Example: to prove $\forall R. \forall Q. \, ((R \to R) \to Q) \to Q$

- Root sequent $S_0 : \emptyset \vdash ((R \to R) \to Q) \to Q$
- S_0 with rule $R_\to$ yields $S_1 : (R \to R) \to Q \vdash Q$
- S_1 with rule $L_\to$ yields $S_2 : (R \to R) \to Q \vdash R \to R$ and $S_3 : Q \vdash Q$
- Sequent S_3 follows from the *Id* axiom; it remains to prove S_2
- S_2 with rule $L_\to$ yields $S_4 : (R \to R) \to Q \vdash R \to R$ and $S_5 : Q \vdash R \to R$
  - 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 $R_\to$ yields $S_6 : (R \to R) \to Q; R \vdash R$
- Sequent S_6 follows from the *Id* axiom

Therefore we have proved S_0
Since $((R \to R) \to Q) \to Q$ is derived from no premises, it is a theorem
$Q.E.D.$

Proof search III: The calculus LJT

Gentzen-Vorobieff-Hudelmaier-Dyckhoff, 1935–1990

- The Gentzen calculus LJ will loop if rule $L_{\rightarrow}$ is applied ≥ 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 $A \rightarrow B$:

$$\text{(if } A \text{ is atomic)}\ \frac{\Gamma, A, B \vdash D}{\Gamma, A, A \rightarrow B \vdash D}\ L_{\rightarrow_1}$$

$$\frac{\Gamma, A \rightarrow B \rightarrow C \vdash D}{\Gamma, (A \times B) \rightarrow C \vdash D}\ L_{\rightarrow_2}$$

$$\frac{\Gamma, A \rightarrow C, B \rightarrow C \vdash D}{\Gamma, (A + B) \rightarrow C \vdash D}\ L_{\rightarrow_3}$$

$$\frac{\Gamma, B \rightarrow C \vdash A \rightarrow B \qquad \Gamma, C \vdash D}{\Gamma, (A \rightarrow B) \rightarrow C \vdash D}\ L_{\rightarrow_4}$$

- When using LJT rules, the proof tree has no loops and terminates
  - See this paper for an explicit decreasing measure on the proof tree

Proof search IV: From deduction rules to code

- The new rules are equivalent to the old rules, therefore...
  - Proof of a sequent $A, B, C \vdash G \Leftrightarrow$ code snippet $t(a, b, c) : G$
  - Also can be seen as a function t from A, B, C to G
- Sequent in a proof follows from an axiom or from a transforming rule
  - The two axioms are fixed expressions, $x^{:A} \rightarrow x$ and 1
  - Each rule has a *proof transformer* function: $\text{PT}_{R_\rightarrow}$, PT_{L_+} , etc.
- Examples of proof transformer functions:

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A + B \vdash C}\, L_+$$

$$PT_{L_+}(t_1^{:A\rightarrow C}, t_2^{:B\rightarrow C}) = x^{:A+B} \rightarrow\ x \text{ match} \begin{cases} a^{:A} \rightarrow t_1(a) \\ b^{:B} \rightarrow t_2(b) \end{cases} = \begin{array}{|c||c|} & C \\ \hline A & t_1 \\ B & t_2 \end{array}$$

$$\frac{\Gamma, A \rightarrow B \rightarrow C \vdash D}{\Gamma, (A \times B) \rightarrow C \vdash D}\, L_{\rightarrow_2}$$

$$PT_{L_{\rightarrow_2}}(f^{:(A\rightarrow B\rightarrow C)\rightarrow D}) = g^{:A\times B\rightarrow C} \rightarrow f\,(x^{:A} \rightarrow y^{:B} \rightarrow g(x, y))$$

- Verify that we can indeed produce PTs for every rule of LJT

Proof search example II: code inference

Once a proof tree is found, start from leaves and apply PTs

- For each sequent S_i, this will derive a **proof expression** t_i
- Example: to prove S_0, start from S_6 backwards:

$S_6 : (R \to R) \to Q; R \vdash R$	(axiom Id)	`def t6(rrq, r) = r`
$S_2 : (R \to R) \to Q \vdash (R \to R)$	$\text{PT}_{R_\to}(t_6)$	`def t2(rrq) = r => t6(rrq, r)`
$S_3 : Q \vdash Q$	(axiom Id)	`def t3(q) = q`
$S_1 : (R \to R) \to Q \vdash Q$	$\text{PT}_{L_\to}(t_2, t_3)$	`def t1(rrq) = t3(rrq(t2(rrq)))`
$S_0 : \emptyset \vdash ((R \to R) \to Q) \to Q$	$\text{PT}_{R_\to}(t_1)$	`def t0 = rrq => t1(rrq)`

- The proof expression for S_0 is then obtained as

```
def t0 = { rrq => t3(rrq(t2(rrq))) }
       = { rrq => rrq(r => t6(rrq, r)) }
                = { rrq => rrq(r => r) }
```

Simplified final code having the required type:

```
def t0[R, Q]: ((R => R) => Q) => Q = { x => x(y => y) }
```

Using the `curryhoward` library for code inference

Two main use cases:

1. Define a type signature and derive an implementation automatically

```
def map[E, A, B](reader: E => A, f: A => B): E => B = implement
```

1. Automatically build an expression from previously computed values

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

Fixed types (`Int`, `String`, etc.) are treated as type parameters

- This is a practical application of the Curry-Howard correspondence
- The CH correspondence works only for “fully parametric” code

Summary

- The CH correspondence maps the type system of each programming language into a certain system of logical propositions
- Proof of logical propositions corresponds to implementation of the type
- If the logic of types is decidable, we can automatically produce code from type signatures
- Simple fully parametric code corresponds to IPL, which is decidable
- Algorithms exist for proof search (and for disproof search) in IPL
  - See the book by R. Bornat: *Proof and Disproof in Formal Logic* (2005)
- The CH correspondence provides powerful type-directed reasoning about code, as long as we work with fully parametric functions
- Software engineers need to develop intuition for this reasoning
  - The decision procedures are not easy to use
  - Software can help, e.g., the `curryhoward` library