Introduction to the Curry-Howard correspondence

The logic of types in functional programming languages

Sergei Winitzki

Academy by the Bay

February 17, 2018

Type constructions in functional programming

The common ground between OCaml, Haskell, Scala, Rust, and other languages

Five type constructions are common in FP languages:

- Tuple ("product") type: $\text{Int} \times \text{String}$
- Function type: $\text{Int} \Rightarrow \text{String}$
- Disjunction ("sum") type: $\text{Int} + \text{String}$
- Unit type ("empty tuple"): 1
- Type parameters: T

The syntax is different; the meaning is the same

Type constructions: Scala

- Tuple type: `(Int, String)`
  - Create: `val pair: (Int, String) = (123, "abc")`
  - Use: `val y: String = pair._2`
- Function type: `Int ⇒ String`
  - Create: `val f: (Int ⇒ String) = x ⇒ "Value is " + x.toString`
  - Use: `val y: String = f(123)`
- Disjunction type: `Either[Int, String]`
  - Create:

    ```
    val x: Either[Int, String] = Left(123)
    val y: Either[Int, String] = Right("abc")
    ```
  - Use: `val z: Boolean = x match {`

    ```
     case Left(i) ⇒ i > 0
     case Right(_) ⇒ false
    }
    ```
- Unit type: `Unit`
  - Create: `val x: Unit = ()`

Type constructions: OCaml

- Tuple type: `int * string`
  - Create: `let pair: int * string = (123, "abc")`
  - Use: `let y: string = snd pair`
- Function type: `Int ⇒ String`
  - Create: `let f: int -> string =`
    `fun x -> Printf.sprintf "Value is %d" x`
  - Use: `let y: string = f 123`
- Disjunction type: `type e = Left of int | Right of string`
  - Create:
    ```
    let x: e = Left 123
    let y: e = Right "abc"
    ```
  - Use: `let z: bool = match x with`
    ```
    Left i -> i > 0
    Right _ -> false
    ```
- Unit type: `unit`
  - Create: `let x: unit = ()`

Type constructions: Haskell

- Tuple type: `(Int, String)`
  - Create: `pair = (123, "abc")`
  - Use: `(_, y) = pair`
- Function type: `Int` $\Rightarrow$ `String`
  - Create: `f = \x -> "Value is " ++ show x`
  - Use: `y = f 123`
- Disjunction type: `data E = Left Int | Right String`
  - Create:
    ```
    x = Left 123
    y = Right "abc"
    ```
  - Use: `z = case x of`
    ```
    Left i -> i > 0
    Right _ -> false
    ```
- Unit type: `Unit`
  - Create: `x = ()`

From types to propositions

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 $\mathcal{CH}(T)$ – "$\mathcal{C}$ode $\mathcal{H}$as a value of type `T`"
- Correspondence between types and propositions, for a given program:

Type	Proposition	Short notation
`T`	$\mathcal{CH}(T)$	T
`(A, B)`	$\mathcal{CH}(A)$ *and* $\mathcal{CH}(B)$	$A \wedge B$; $A \times B$
`Either[A, B]`	$\mathcal{CH}(A)$ *or* $\mathcal{CH}(B)$	$A \vee B$; $A + B$
`A ⇒ B`	$\mathcal{CH}(A)$ *implies* $\mathcal{CH}(B)$	$A \Rightarrow B$
`Unit`	*True*	1

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

Translating language constructions into the logic I

What are the derivation rules for the logic of types?

What logical relationships exist between propositions $\mathcal{CH}(...)$?

- Expressions (program code) are represented by **sequents**
  - $A, B \vdash C$ represents an *expression* of type `C` that uses `x: A` and `y: B`
    - Sequents only describe the *types* of expressions and their parts
  - In $A, B, ... \vdash C$ the **premises** are $A, B, ...$ and the **goal** is C
- Some sequents are immediate, others follow from previous ones
  - Tuple type: $A \times B$
    - Create: $A, B \vdash A \times B$
    - Use: $A \times B \vdash A$ and also $A \times B \vdash B$
  - Function type: $A \Rightarrow B$
    - Create: if we have $A \vdash B$ then we will have $\emptyset \vdash A \Rightarrow B$
    - Use: $A \Rightarrow B, A \vdash B$
  - Disjunction type: $A + B$
    - Create: $A \vdash A + B$ and also $B \vdash A + B$
    - Use: $A + B, A \Rightarrow C, B \Rightarrow C \vdash C$
  - Unit type: 1
    - Create: $\emptyset \vdash 1$

Translating language constructions into the logic II

Additional rules for the logic of types

In addition to constructions using types, we have "trivial" constructions:

- a single, unmodified value of type A is a valid expression of type A
  - For any A we have the sequent $A \vdash A$
- if a value can be computed using some given data, it can also be computed if given *more* data
  - If we have $A, ..., C \vdash G$ then also $A, ..., C, D \vdash G$ for any D
  - For brevity, we denote by Γ a sequence of arbitrary premises
- the order in which data is given does not matter, we can still compute all the same things given the same premises in different order
  - If we have $\Gamma, A, B \vdash G$ then we also have $\Gamma, B, A \vdash G$

Syntax conventions:

- the implication operation associates *to the right*
  - $A \Rightarrow B \Rightarrow C$ means $A \Rightarrow (B \Rightarrow C)$
- precedence order: implication, disjunction, conjunction
  - $A + B \times C \Rightarrow D$ means $(A + (B \times C)) \Rightarrow D$

Quantifiers: implicitly, all our type variables are universally quantified

- When we write $A \Rightarrow B \Rightarrow A$, we mean $\forall A : \forall B : A \Rightarrow B \Rightarrow A$

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: $A \Rightarrow B \Rightarrow A$
  - Start with an axiom $A \vdash A$; add an unused extra premise B: $A, B \vdash A$
  - Use the "create function" rule with B and A, get $A \vdash B \Rightarrow A$
  - Use the "create function" rule with A and $B \Rightarrow A$, get the final sequent $\emptyset \vdash A \Rightarrow (B \Rightarrow A)$ showing that $A \Rightarrow B \Rightarrow A$ is a **theorem** since it is derived from no premises
- What code does this describe?
  - The axiom $A \vdash A$ represents the expression x where x is of type A
  - The unused premise B corresponds to unused variable y of type B
  - The "create function" rule gives the function $y \Rightarrow x$
  - The second "create function" rule gives $x \Rightarrow (y \Rightarrow x)$
  - Scala 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

Correspondence between programs and proofs

- By construction, any theorem can be implemented in code

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

- Also, 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: $\forall A \forall B : ((((A \Rightarrow B) \Rightarrow A) \Rightarrow A) \Rightarrow B) \Rightarrow B$
    - ★ Can we write a function with this type?

The logic of types II

What kind of logic is this?

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

- Disjunction works very differently from classical (Boolean) logic
  - Example: $A \Rightarrow B + C \vdash (A \Rightarrow B) + (A \Rightarrow C)$ does not hold in IPL
  - This is counter-intuitive!
  - We cannot implement a function with this type:

```
def q[A,B,C](f: A ⇒ Either[B, C]): Either[A ⇒ B, A ⇒ C] = ???
```

  - Disjunction is "constructive": need to supply one of the parts
- Implication works somewhat differently
  - Example: $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ 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: $A \Rightarrow B \times C \vdash (A \Rightarrow B) \times (A \Rightarrow C)$

The logic of types III

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

- The IPL cannot have a truth table with a fixed number of truth values
- 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 curryhoward library implements an IPL prover as a Scala macro, and generates Scala code from types
  - The djinn-ghc compiler plugin and the JustDoIt plugin implement an IPL prover in Haskell, and generate Haskell code from types
- All these IPL provers use the same basic algorithm called LJT
  - and cite the same paper by Dyckhoff [1992]
  - because most other papers on this subject are incomprehensible to engineers or describe algorithms that are too complicated

Proof search I: looking for an algorithm

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

We have 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?

- 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!) valid theorems of the logic

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

$$\frac{\Gamma, A \Rightarrow B \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \Rightarrow B \vdash C} L\Rightarrow \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} R\Rightarrow$$

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

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

- Two axioms and eight derivation rules
- Each rule says: The sequent at bottom will be proved if proofs are given for sequent(s) at top
- Use these rules "bottom-up" to perform a proof search
  - Sequents are nodes and proofs are edges in the proof search tree
- Example: to prove $\emptyset \vdash ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$

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 III: The calculus LJT

Vorobieff-Hudelmaier-Dyckhoff, 1950-1990

- The Gentzen calculus generates a 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 $A \Rightarrow B$:

$$(X \text{ is atomic})\ \frac{\Gamma, X, B \vdash D}{\Gamma, X, X \Rightarrow B \vdash D}\ L \Rightarrow_1$$

$$\frac{\Gamma, A \Rightarrow (B \Rightarrow C) \vdash D}{\Gamma, (A \times B) \Rightarrow C \vdash D}\ L \Rightarrow_2$$

$$\frac{\Gamma, A \Rightarrow C, B \Rightarrow C \vdash D}{\Gamma, (A + B) \Rightarrow C \vdash D}\ L \Rightarrow_3$$

$$\frac{\Gamma, B \Rightarrow C \vdash A \Rightarrow B \qquad \Gamma, C \vdash D}{\Gamma, (A \Rightarrow B) \Rightarrow C \vdash D}\ L \Rightarrow_4$$

- When using LJT rules, the proof tree has no loops and terminates
  - Apply all rules that fit the sequent, and repeat
- 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. $\vdash_{\mathrm{LJ}} \Gamma, (C \supset D) \supset B \Rightarrow C \supset D$ *iff* $\vdash_{\mathrm{LJ}} \Gamma, D \supset B \Rightarrow C \supset D$.
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): $f^{(A \Rightarrow B) \Rightarrow C} \Rightarrow b^B \Rightarrow f\,(x^A \Rightarrow b)$

- *Details are left as exercise for the reader*

Proof search V: 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/expression $t(a, b, c) : G$
  - Also can be seen as a function t from A, B, C to G
- Sequent in a proof follow from an axiom or from a transforming rule
  - 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:

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, {\color{blue}A + B} \vdash C} \, L+$$

$$PT_{L+}(t_1^{A\Rightarrow C}, t_2^{B\Rightarrow C})(x^{A+B}) = x \text{ match} \begin{cases} a \Rightarrow & t_1(a) \\ b \Rightarrow & t_2(b) \end{cases}$$

$$\frac{\Gamma, A \Rightarrow (B \Rightarrow C) \vdash D}{\Gamma, {\color{blue}(A \times B) \Rightarrow C} \vdash D} \, L \Rightarrow_2$$

$$PT_{L\Rightarrow_2}(f^{(A\Rightarrow B\Rightarrow C)\Rightarrow D})(g^{A\times B\Rightarrow C}) = f\,(x^A \Rightarrow y^B \Rightarrow g(x, y))$$

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

Proof search VI: Example deduction

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

- Example: to prove S_0, start from S_6 backwards:

$$
\begin{array}{rll}
S_6 : (R \Rightarrow R) \Rightarrow Q; R \vdash R & (\text{axiom } Id) & t_6(rrq, r) : R = r \\
S_2 : (R \Rightarrow R) \Rightarrow Q \vdash (R \Rightarrow R) & \mathrm{PT}_{R\Rightarrow}(t_6) & t_2(rrq) : (R \Rightarrow R) = (r \Rightarrow t_6(rrq, r)) \\
S_3 : Q \vdash Q & (\text{axiom } Id) & t_3(q) : Q = q \\
S_1 : (R \Rightarrow R) \Rightarrow Q \vdash Q & \mathrm{PT}_{L\Rightarrow}(t_2, t_3) & t_1(rrq) : Q = t_3(rrq(t_2(rrq))) \\
S_0 : \emptyset \vdash ((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q & \mathrm{PT}_{R\Rightarrow}(t_1) & t_0 = (rrq \Rightarrow t_1(rrq))
\end{array}
$$

- The expression for the proof of S_0 is

$$
\begin{aligned}
t_0 &= rrq \Rightarrow t_3\left(rrq\left(t_2\left(rrq\right)\right)\right) = rrq \Rightarrow rrq(r \Rightarrow t_6\left(rrq, r\right) \\
&= rrq \Rightarrow rrq\left(r \Rightarrow r\right)
\end{aligned}
$$

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

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

Type isomorphisms I: identities

Using known properties of propositional logic and arithmetic

Are $A + B$, $A \times B$ more like logic ($A \vee B$, $A \wedge B$) or like arithmetic?

- Some standard identities in logic ($\forall A \forall B \forall C$ is assumed):

$$\begin{aligned} A \times 1 = A; &\quad A \times B = B \times A \\ A \vee 1 = 1; &\quad A \vee B = B \vee A \\ (A \times B) \times C = A \times (B \times C); &\quad A \vee (B \times C) = (A \vee B) \times (A \vee C) \\ (A \vee B) \vee C = A \vee (B \vee C); &\quad A \times (B \vee C) = (A \times B) \vee (A \times C) \end{aligned}$$

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

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

$$(A \vee B) \Rightarrow C = (A \Rightarrow C) \times (B \Rightarrow C)$$

- Each identity means 2 function types: $X = Y$ is $X \Rightarrow Y$ *and* $Y \Rightarrow X$
  - Do these functions convert values between the types X and Y?

Type isomorphisms II

- Types A and B are isomorphic, $A \equiv B$, if there is a 1-to-1 correspondence between the sets of values of these types
  - Need to find two functions $f : A \Rightarrow B$ and $g : B \Rightarrow A$ such that $f \circ g = id$ and $g \circ f = id$

Example 1: Is $\forall A : A \times 1 \equiv A$? Types in Scala: `(A, Unit)` and `A`

- Two functions with types $\forall A : A \times 1 \Rightarrow A$ and $\forall A : A \Rightarrow A \times 1$:

```
def f1[A]: ((A, Unit)) ⇒ A = { case (a, ()) ⇒ a }
def f2[A]: A ⇒ (A, Unit) = a ⇒ (a, ())
```

- Verify that their compositions equal `identity`

Example 2: Is $\forall A : 1 + A \equiv 1$? (The formula $\forall A : A \vee 1 = 1$ is a theorem!)

- Types in Scala: `Option[A]` and `Unit`
  - These types are obviously *not* equivalent

Some logic identities yield isomorphisms of types

- Which ones *do not* yield isomorphisms, and why?

Type isomorphisms III

Verifying type equivalence by implementing isomorphisms

- Need to verify that $f_1 \circ f_2 = id$ and $f_2 \circ f_1 = id$

Example 3: $\forall A \forall B \forall C : (A \times B) \times C \equiv A \times (B \times C)$

```
def f1[A,B,C]: (((A, B), C)) ⇒ (A, (B, C)) = ???
def f2[A,B,C]: ((A, (B, C))) ⇒ ((A, B), C) = ???
```

Example 4: $\forall A \forall B \forall C : (A + B) \times C \equiv A \times C + B \times C$

```
def f1[A,B,C]: ((Either[A,B], C)) ⇒ Either[(A,C), (B,C)] = ???
def f2[A,B,C]: Either[(A,C), (B,C)] ⇒ (Either[A, B], C) = ???
```

Example 5: $\forall A \forall B \forall C : (A + B) \Rightarrow C \equiv (A \Rightarrow C) \times (B \Rightarrow C)$

```
def f1[A,B,C]: (Either[A, B] ⇒ C) ⇒ (A ⇒ C, B ⇒ C) = ???
def f2[A,B,C]: ((A ⇒ C, B ⇒ C)) ⇒ Either[A, B] ⇒ C = ???
```

Example 6: $\forall A \forall B \forall C : A + B \times C \not\equiv (A + B) \times (A + C)$ – "information loss"

```
def f1[A,B,C]: Either[A,(B,C)]⇒(Either[A,B],Either[A,C]) = ???
def f2[A,B,C]: ((Either[A,B],Either[A,C])) ⇒ Either[A,(B,C)] = ???
```

Type isomorphisms III

Logical CH vs. arithmetical CH

- WLOG, consider types $A, B, ...$ that have *finite* sets of possible values
  - Sum type $A + B$ (size $|A| + |B|$) provides a disjoint union of sets
  - Product type $A \times B$ (size $|A| \cdot |B|$) provides a Cartesian product of sets
  - Function type $A \Rightarrow B$ provides the set of all maps between sets
    - The size of $A \Rightarrow B$ is $|B|^{|A|}$
    - Note the identities $a^c b^c = (ab)^c$, $a^{b+c} = a^b a^c$, $a^{bc} = \left(a^b\right)^c$
- If the set size (**cardinality**) differs, A and B cannot be equivalent
  - Logic identities give only the "equal implementability" of types

The meaning of the type/logic/arithmetic correspondence:

- Arithmetical identities are related to type equivalence (isomorphism)
- Logic identities are related to implementability

Reasoning about types is *school-level algebra* with polynomials and powers

- **Exp-polynomial** expressions: constants, sums, products, exponentials
  - exp-poly types: primitive types, disjunctions, tuples, functions
  - polynomial types are commonly called "algebraic types"

Making practical use of the CH correspondence I

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 the computer fill in the code when it is "trivial" to do so
  - This is often (not always) the case for fully type-parametric functions
- Decide type isomorphisms using the "arithmetical CH"
- Isomorphically transform types using school-level algebra

What problems cannot be solved with these tools?

- Automatically generate code satisfying properties (e.g. isomorphism)
- Express complicated conditions via types (e.g. "array is sorted")
- Generate code using type constructors with properties (e.g. `map`)
  - Scala type signature: `(x: List[A]).map[B](f: A ⇒ B): List[B]`
  - This formula has a quantifier *inside*: $\text{List}^A \Rightarrow (\forall B : f^{A\Rightarrow B} \Rightarrow \text{List}^B)$
  - This requires **first-order logic**, which is generally *undecidable* (no algorithm can guarantee finding a proof)

Some caveats

- The CH correspondence becomes informative only with parameterized types. For concrete types, e.g. `Array[Int]`, we can always produce *some* value even with no previous data, so $\mathcal{CH}(\text{Int})$ is always true.
- Functions such as `(x: Int) ⇒ x + 1` have type `Int⇒Int`, so the type signature is insufficient to specify the code. Only for fully type-parametric functions the type signature can be, in some cases, informative enough for deriving the code automatically.
- Having an arithmetic identity does not guarantee that we have a type equivalence via CH (it is a necessary but not a sufficient condition); but it does yield a type equivalence in all cases I looked at so far.

Making practical use of the CH correspondence II

Implications for designing new programming languages

- The CH correspondence maps the type system of each programming language into a certain system of logical propositions
- Scala, Haskell, OCaml, F#, Swift, Rust, etc. are mapped into the full constructive logic (all logical operations are available)
  - C, C++, Java, C#, etc. are mapped to *incomplete logics* – without "or" and without "true" / "false"
  - Python, JavaScript, Ruby, Clojure, etc. have only one type ("any value") and are mapped to logics with only one proposition
- The CH correspondence is a principle for designing type systems:
  - Choose a complete logic, free of inconsistency
    - Mathematicians have studied all kinds of logics and determined which ones are interesting, and found the minimal sets of axioms for them
    - Modal logic, temporal logic, linear logic, etc.
  - Provide a type constructor for each basic operation (e.g. "*or*", "*and*")