Generating code from type signatures using the Curry-Howard correspondence

With implementations in Haskell and Scala

Sergei Winitzki

Bay Area Haskell Users' Group

March 22, 2018

Type-directed coding

How to implement functions given their type?

We write code "guided by the types"

1. Implement `fmap` for the Reader monad,
$$\text{fmap} :: (a \to b) \to (e \to a) \to (e \to b)$$
2. Show that one cannot implement $(e \to f) \to (e \to a) \to (f \to a)$
3. Implement $\text{fmap} :: (a \to b) \to (e \to \text{Maybe}\, a) \to (e \to \text{Maybe}\, b)$
4. Implement the distributive law: $(A + B) \times C \equiv A \times C + B \times C$ or equivalently $(\text{Either}\, a\, b, c) \equiv \text{Either}\, (a, c)\, (b, c)$
5. Implement $((((a \to b) \to a) \to a) \to b) \to b$?

Can we generate the code automatically?

- The djinn-ghc compiler plugin and the JustDoIt plugin generate Haskell code from type (need tooling to use)
- The curryhoward library generates Scala code from type signatures

Often, there is only one "useful" implementation out of many

The `djinn` and `curryhoward` libraries try to generate that implementation

Haskell: Using the `djinn` tool

Demo time

Features:

- Haskell syntax, supports algebraic data types and type classes
- Constant types (`Int`, `String`, etc.) are treated as type parameters
- If several implementations are available, chooses "intelligently"
- Can output several implementations if desired

Examples:

```
Djinn> f1 ? (a -> b) -> (e -> a) -> (e -> b)
f1 :: (a -> b) -> (e -> a) -> e -> b
f1 a b c = a (b c)
Djinn> f2 ? (a, a, a) -> Maybe (a, a, a)
f :: (a, a, a) -> Maybe (a, a, a)
f (a, b, c) = Just (c, b, a)
```

Scala: Using the `curryhoward` library

Two main use cases:

1. Define a method and provide an automatic implementation

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

2. Automatically build an expression from previously computed values

```
val f: String ⇒ Boolean ⇒ 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"
- Lambda-calculus evaluator available for symbolic law checking

Benefits and limitations of this method

Benefits:

- Save time implementing "trivial" functions
- With some more work, can verify algebraic laws
- In many practical use cases, supports type class derivation

Limitations:

- Heuristics often fail with certain kinds of data (repeated types)
- Cannot generate recursive code
- Cannot depend on existing type class instances (`Functor f ⇒ ...`)

Type constructions in functional programming

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

Type constructions 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: List^T

Up to differences in syntax, the FP languages share all these features

Type constructions: Haskell syntax

- Tuple type: `(Int, String)`
  - Create: `pair = (123, "abc")`
  - Use: `(_, y) = pair`
- Function type: `Int -> 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 = ()`

Type constructions: Scala syntax

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

From types to propositions

The code `x:: t; x = ...` 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* $\mathcal{CH}(b)$	$a \wedge b$; $a \times b$
`A a | B b`	$\mathcal{CH}(a)$ *or* $\mathcal{CH}(b)$	$a \vee b$; $a + b$
`a → b`	$\mathcal{CH}(a)$ *implies* $\mathcal{CH}(b)$	$a \Rightarrow b$
`()`	*True*	1

- Type parameter in a function type means $\forall t$
- Example: `dupl:: a → (a, a)`. The type of this function, $a \Rightarrow a \times a$, corresponds to the theorem $\forall a : a \Rightarrow a \wedge a$

Translating language constructions into the logic I

How to represent logical relationships between $\mathcal{CH}(...)$ propositions?

Code expressions create *logical relationships* between propositions $\mathcal{CH}(...)$

- "Logical relationships" = what will be true if something given is true
- The elementary proof task is represented by a **sequent**
  - Notation: $A, B, C \vdash G$; the **premises** are A, B, C and the **goal** is G
- Proofs are achieved via axioms and derivation rules
  - Axioms: such and such sequents are already true
  - Derivation rules: this sequent is true if such and such sequents are true
- To make connection with logic, represent code fragments as **sequents**
- $a, b \vdash c$ represents an *expression* of type `c` that uses `x :: a` and `y :: b`
- Examples in Haskell (assume `x :: Int`):
  - `show x ++ "abc"` is an expression of type `String` that uses an `x:: Int`, and is represented by the sequent $\text{Int} \vdash \text{String}$
  - `\x → show x + "abc"` is an expression of type `Int → String` and is represented by the sequent $\emptyset \vdash \text{Int} \Rightarrow \text{String}$
- Sequents only describe the *types* of expressions and their parts

Translating language constructions into the logic II

What are the derivation rules for the logic of types?

Write all the constructions in FP languages as sequents

- This will give all the derivation rules for the logic of types
  - Each type construction has an expression for creating it and an expression for using it
- 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 III

Additional rules for the logic of types

In addition to constructions that use 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 *additional* 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^A where x is of type A
  - The unused premise B corresponds to unused variable y^B of type B
  - The "create function" rule gives the function $y^B \Rightarrow x^A$
  - The second "create function" rule gives $x^A \Rightarrow \left(y^B \Rightarrow x\right)$
  - Haskell code:

```
f :: A → B → A
f = \x → \y → 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
$\forall A : A \Rightarrow A$	`identity x = x`
$\forall A : A \Rightarrow 1$	`toUnit x = ()`
$\forall A \forall B : A \Rightarrow A + B$	`Left :: a → Either a b`
$\forall A \forall B : A \times B \Rightarrow A$	`fst :: (a, b) → a`
$\forall A \forall B : A \Rightarrow B \Rightarrow A$	`const = \x → \y → x`

- Also, non-theorems *cannot be implemented* in code
  - Examples of non-theorems:
    $\forall A : 1 \Rightarrow A; \qquad \forall A \forall B : A + B \Rightarrow A;$
    $\forall A \forall B : A \Rightarrow A \times B; \qquad \forall A \forall B : (A \Rightarrow B) \Rightarrow A$
- Given a type's formula, can we implement it in code? Not obvious.
  - Example: $\forall A \forall B : ((((A \Rightarrow B) \Rightarrow A) \Rightarrow A) \Rightarrow B) \Rightarrow B$
    - 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 very differently from 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:

    ```
    q :: (a → Either b c) → Either (a → b) (a → c)
    ```

  - Disjunction is "constructive": need to supply one of the parts
    - ...but `Either (a → b) (a → c)` is not a function of `a`
- 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
  - This was shown 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
  - and all cite the same paper [Dyckhoff 1992]
  - because most other papers on this subject are incomprehensible to non-specialists, 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

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

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

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

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

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

- Two axioms and eight derivation rules
  - Each derivation 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

Proof search example I

Example: to prove $((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$

- 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

Since $((R \Rightarrow R) \Rightarrow Q) \Rightarrow Q$ is derived from no premises, it is a theorem

$Q.E.D.$

Proof search III: The calculus LJT

Vorobieff-Hudelmaier-Dyckhoff, 1950-1990

- 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 $A \Rightarrow B$:

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

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

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

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

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

Proof search IV: The calculus LJT

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

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

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

- 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 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: $\mathrm{PT}_{R\Rightarrow}$, PT_{L+} , etc.
- Examples of proof transformer functions:

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

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

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

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

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

Proof search example II: deriving code

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:

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

- The proof expression for S_0 is then obtained as

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

Simplified final code having the required type:

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

Summary

- The CH correspondence maps the type system of each programming language into a certain system of logical propositions
- If that logic is decidable, we can automatically produce code from type signatures
- Simply-typed Lambda Calculus corresponds to IPL, which is decidable
- In practice, many types have more than one implementation
- To make this into a practical tool, need heuristics or algebraic laws
- Implementations available in Scala and Haskell