

# Chapter 3: The Logic of Types, Part III

## The Curry-Howard correspondence

Sergei Winitzki

Academy by the Bay

December 16, 2017

# 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 $\mathcal{CH}(B)$	$A \times B$
<code>Either[A, B]</code>	$\mathcal{CH}(A)$ or $\mathcal{CH}(B)$	$A + B, A \vee B$
<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) proposition  $\forall A : A \Rightarrow A \times A$

# Working with the CH correspondence I

Convert Scala types to short notation and back

## Example 1: A disjunction type

```
sealed trait UserAction
case class SetName(first: String, last: String) extends UserAction
case class SetEmail(email: String) extends UserAction
case class SetUserId(id: Long) extends UserAction
```

- Short notation:  $\text{UserAction} \equiv \text{String} \times \text{String} + \text{String} + \text{Long}$

## Example 2: A parameterized disjunction type

```
sealed trait Either3[A, B, C]
case class Left[A, B, C](x: A) extends Either3[A, B, C]
case class Middle[A, B, C](x: B) extends Either3[A, B, C]
case class Right[A, B, C](x: C) extends Either3[A, B, C]
```

- Short notation:  $\text{Either3}^{A,B,C} \equiv A + B + C$

# Working with the CH correspondence II

- Any valid formula 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⇒A = (y:B)⇒x</code>

- Invalid formulas *cannot be implemented* in code
  - Examples of invalid formulas:  
 $\forall A : 1 \Rightarrow A$ ;  $\forall A \forall B : A \vee B \Rightarrow A$ ;  
 $\forall A \forall B : A \Rightarrow A \times B$ ;  $\forall A \forall B : (A \Rightarrow B) \Rightarrow A$
- 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 propositional logic has a decision algorithm
- See code examples using the **curryhoward** library

# Working with the CH correspondence III

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):

$$A \times 1 = A; \quad A \times B = B \times A$$

$$A \vee 1 = 1; \quad A \vee B = B \vee A$$

$$(A \times B) \times C = A \times (B \times C); \quad A \vee (B \times C) = (A \vee B) \times (A \vee C)$$

$$(A \vee B) \vee C = A \vee (B \vee C); \quad A \times (B \vee C) = (A \times B) \vee (A \times C)$$

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

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

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

- 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 I

- 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` (see test code)

Example 2: Is  $\forall A : A + 1 \equiv 1$ ? (The logic formula  $\forall A : A \vee 1 = 1$  is valid.)

- 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 II

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)] = ???
```

- See example code for methods of testing these properties

# Type isomorphisms III

Logic CH vs. arithmetic CH for elementary (“algebraic”) types

- WLOG, consider types  $A, B, \dots$  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  $a^c b^c = (ab)^c$ ,  $a^{b+c} = a^b a^c$ ,  $a^{bc} = (a^b)^c$
- If the set size (cardinality) differs,  $A$  and  $B$  cannot be equivalent
  - ▶ Logic identities give only the “equal implementability”

The meaning of the types/logic/arithmetic correspondence:

- Arithmetic formulas are related to type equivalence
- Logic formulas 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”



# Algebraic computation with recursive types

Using school algebra to understand types

Recursive type: “list of integers”

```
sealed trait IntList
final case object Empty extends IntList
final case class Nonempty(head: Int, tail: IntList) extends IntList
```

$$\text{IntList} \equiv 1 + \text{Int} \times \text{IntList}$$

Parameterized recursive type: “list of  $A$ ”, short notation:  $\text{List}^A$

```
sealed trait List[A]
final case object Nil extends List[Nothing]
final case class ::(head: A, tail: List[A]) extends List[A]
```

Short notation: (the sign “ $\equiv$ ” means type isomorphism)

$$\begin{aligned}\text{List}^A &\equiv 1 + A \times \text{List}^A \equiv 1 + A \times (1 + A \times (1 + A \times (\dots) \dots)) \\ &\equiv 1 + A + A \times A + A \times A \times A + \dots\end{aligned}$$

A curious analogy with calculus:  $\text{List}(t) = 1 + t \cdot \text{List}(t)$ ; “solve” this as

$$\text{List}(t) = \frac{1}{1-t} = 1 + t + t^2 + t^3 + \dots$$

# Worked examples

- 1 Define a parameterized type `MyT[T]` for the short type notation  $\text{Boolean} \Rightarrow (1 + T + \text{Int} \times T + (\text{String} \Rightarrow T))$
- 2 Transform `(Either[A,B], Either[C,D])` into an equivalent sum type
- 3 Show that  $A + A \not\equiv A$  and  $A \times A \not\equiv A$ , although these hold in logic
- 4 Show that  $(A \times B) \Rightarrow C \not\equiv (A \Rightarrow C) + (B \Rightarrow C)$  in logic
- 5 Denote  $\text{Reader}^{E,T} \equiv E \Rightarrow T$  and implement functions with types  $A \Rightarrow \text{Reader}^{E,A}$  and  $\text{Reader}^{E,A} \Rightarrow (A \Rightarrow B) \Rightarrow \text{Reader}^{E,B}$
- 6 Show that one cannot implement `Reader[A,T]  $\Rightarrow$  (A $\Rightarrow$ B)  $\Rightarrow$  Reader[B,T]`
- 7 Implement  $\text{map}^{A,B} : 1 + A \Rightarrow (A \Rightarrow B) \Rightarrow 1 + B$  with no “information loss”, that is, `map(opt)(x  $\Rightarrow$  x) = opt`
  - 1 Implement `map` and `flatMap` for `Either[L,R]` by preferring `R` over `L`
- 8 Denoting  $\text{State}^{S,T} \equiv S \Rightarrow T \times S$ , implement the functions:
  - 1  $\text{pure}^{S,A} : A \Rightarrow \text{State}^{S,A}$
  - 2  $\text{map}^{S,A,B} : \text{State}^{S,A} \Rightarrow (A \Rightarrow B) \Rightarrow \text{State}^{S,B}$
  - 3  $\text{flatMap}^{S,A,B} : \text{State}^{S,A} \Rightarrow (A \Rightarrow \text{State}^{S,B}) \Rightarrow \text{State}^{S,B}$
- 9 Define recursive type `NEList[A]` by  $\text{NEList}^A \equiv A + A \times \text{NEList}^A$ 
  - 1 Implement `map` and `concat` for `NEList` (tail recursion not necessary)

# Exercises III

- ① Define type `MyTU[T,U]` for  $1 + T \times U + \text{Int} \times T + \text{String} \times U$
- ② Show that  $A \Rightarrow (B + C) \neq (A \Rightarrow B) + (A \Rightarrow C)$  in logic
- ③ Transform `(Either[A,Int],Either[A,Char],Either[A,Float])` into an equivalent type of the form  $A \times (\dots)$
- ④ Define type  $\text{OptEither}^{A,B} = 1 + A + B$  and implement `map` and `flatMap` for it, without information loss, preferring  $B$  over  $A$ . Get the same result using the equivalent type  $(1 + A) + B$ , i.e. `Either[Option[A], B]`
- ⑤ Implement `map` for `MyT[T]` (see worked example 1) and for `MyTU[T,U]`
- ⑥ Implement type-parametric functions with the following types:
  - ①  $\text{State}^{S,A} \Rightarrow (S \times A \Rightarrow S \times B) \Rightarrow \text{State}^{S,B}$
  - ②  $A + Z \Rightarrow (A \Rightarrow B) \Rightarrow B + Z$  and  $A + Z \Rightarrow B + Z \Rightarrow (A \Rightarrow B \Rightarrow C) \Rightarrow C + Z$
  - ③  $\text{flatMap}^{E,A,B} : \text{Reader}^{E,A} \Rightarrow (A \Rightarrow \text{Reader}^{E,B}) \Rightarrow \text{Reader}^{E,B}$
- ⑦ \* Denoting `Density[Z,T] = (T⇒Z)⇒T`, implement the functions:
  - ①  $\text{map}^{Z,A,B} : \text{Density}^{Z,A} \Rightarrow (A \Rightarrow B) \Rightarrow \text{Density}^{Z,B}$
  - ②  $\text{flatMap}^{Z,A,B} : \text{Density}^{Z,A} \Rightarrow (A \Rightarrow \text{Density}^{Z,B}) \Rightarrow \text{Density}^{Z,B}$
- ⑧ \* Denote `Cont[R,T] = (T⇒R)⇒R` and implement the functions:
  - ①  $\text{map}^{R,T,U} : \text{Cont}^{R,T} \Rightarrow (T \Rightarrow U) \Rightarrow \text{Cont}^{R,U}$
  - ②  $\text{flatMap}^{R,T,U} : \text{Cont}^{R,T} \Rightarrow (T \Rightarrow \text{Cont}^{R,U}) \Rightarrow \text{Cont}^{R,U}$
- ⑨ Define recursive type  $\text{Tr3}^A \equiv 1 + A \times A \times A \times \text{Tr3}^A$ ; implement `map` for it.

# Working with the CH correspondence IV

## 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”)

# Working with the CH correspondence V

## Implications for actually writing code

What problems can we solve now?

- Use the short type notation for reasoning about types
- Given a fully parametric type, decide whether it can be implemented in code (“type is inhabited”); if so, *generate* the code
  - ▶ The **Gentzen-Vorobiev-Hudelmaier algorithm** and its generalizations
  - ▶ See also the **curryhoward** project
- Given some expression, infer the most general type it can have
  - ▶ The **Damas-Hindley-Milner algorithm** (**Scala code**) and generalizations
- Decide type isomorphism, simplify type formulas (the “arithmetic CH”)
- Compute the necessary types before starting to write code

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”)
  - ▶ Need dependent types for that (Coq, Agda, Idris, ...)