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 $\mathcal{CH}(T)$ – "$\mathcal{C}$ode $\mathcal{H}$as a value of type `T`"
- We have the following correspondence between types and propositions:

Type	Proposition	Short notation
`T`	$\mathcal{CH}(T)$	T
`(A, B)`	$\mathcal{CH}(A)$ *and* $\mathcal{CH}(B)$	$A \times B$
`Either[A, B]`	$\mathcal{CH}(A)$ *or* $\mathcal{CH}(B)$	$A + B$
`A => B`	$\mathcal{CH}(A)$ *implies* $\mathcal{CH}(B)$	$A \Rightarrow B$
`Unit`	*true*	1
`Nothing`	*false*	0

- type parameter `[T]` in a function type means $\forall T$, for example the type of the function `def dupl[A]: A => (A, A)` 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: `UserAction` $=$ `String` $\times$ `String` $+$ `String` $+$ `Long`
- Example 2: A parameterized disjunction type

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

- Short notation: $\forall A \forall B \forall C : (A \Rightarrow B) + B + (C \Rightarrow A)$

Working with the CH correspondence II

- Any valid formula 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`

- Invalid formulas *cannot be implemented* in code
  - Examples of invalid formulas:
    $\forall A : 1 \Rightarrow A$; $\forall A \forall B : A + B \Rightarrow A$;
    $\forall A \forall B : A \Rightarrow A \times B$; $\forall A \forall B : (A \Rightarrow B) \Rightarrow A$
- Given a type's formula, can we implement it in code?
  - Example: $\forall A \forall B : ((((A \Rightarrow B) \Rightarrow A) \Rightarrow A) \Rightarrow B) \Rightarrow B$
- 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 or like arithmetic?

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

$$A \times 1 = A; \quad A + 1 = 1$$
$$(A \times B) \times C = A \times (B \times C)$$
$$(A + B) + C = A + (B + C)$$
$$A \times (B + C) = (A \times B) + (A \times C)$$
$$A + (B \times C) = (A + B) \times (A + C)$$
$$(A \times B) \Rightarrow C = A \Rightarrow (B \Rightarrow C)$$
$$A \Rightarrow (B \times C) = (A \Rightarrow B) \times (A \Rightarrow C)$$
$$(A + B) \Rightarrow C = (A \Rightarrow C) \times (B \Rightarrow C)$$

- Each identity means 2 function types: "$X = Y$" is $X \Rightarrow Y$, $Y \Rightarrow X$
  - 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 all values of these types. Formally, this requires us 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 *id* (see test code)

Example 2: Is $\forall A : A + 1 \equiv 1$? Types in Scala: `Option[A]` and `Unit`

- These types are *not* isomorhic

Some of the logic identities yield isomorphisms of types

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

Type isomorphisms II

Verifying a type isomorphism

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

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

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

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

```
def g1[A,B,C]: (((A, B), C)) ⇒ (A, (B, C)) = ...
def g2[A,B,C]: ((A, (B, C))) ⇒ ((A, B), C) = ...
```

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

```
def h1[A,B,C]: ((A, Either[B, C])) ⇒ Either[(A, B), (A, C)] = ...
def h2[A,B,C]: Either[(A, B), (A, C)] ⇒ (A, Either[B, C]) = ...
```

Example 6: (This is not $\equiv$!) $\forall A \forall B \forall C : A + (B \times C) = (A + B) \times (A + C)$

```
def j1[A,B,C]: Either[A, (B, C)] ⇒ (Either[A,B], Either[A,C]) = ...
def j2[A,B,C]:((Either[A,B], Either[A,C])) ⇒ Either[A, (B,C)] = ...
```

Type isomorphisms III

Logic CH vs. arithmetic CH for elementary ("algebraic") types

- WLOG, consider types $A, B, \ldots$ that have *finite* sets of possible values
  - Disjunction type $A + B$ (size $|A| + |B|$) provides a disjoint union of sets
  - Tuple 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 $|B|^{|A|}$
- If the set size (cardinality) differs, A and B cannot be isomorphic
  - Only the arithmetic identities yield type isomorphisms
  - Logic identities yield only the "equal implementability"

The meaning of the types/logic/arithmetic correspondence:

- Arithmetic formulas show isomorphism
- Logic formulas show implementability

Reasoning about types is just like doing school algebra

- **Elementary types**: constants, sums, products, exponentials
- **Polynomial types**: constants, sums, products

Algebraic computation with recursive 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
```

Parameterized recursive type: "list of `T`"

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

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

$$\begin{aligned}\text{List}(t) &\equiv 1 + t \times \text{List}(t) \equiv 1 + t \times (1 + t \times (1 + t \times (...)...) \\ &\equiv 1 + t + (t \times t) + (t \times t \times t) + ...\end{aligned}$$

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

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

Worked examples

- a
- a

Exercises III

1. a

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, ...)