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`"
- Correspondence between types and propositions, for a given program:

Type	Proposition	Short notation
`T`	$\mathcal{CH}(T)$	T
`(A, B)`	$\mathcal{CH}(A)$ *and* $\mathcal{CH}(B)$	$A \times B$
`Either[A, B]`	$\mathcal{CH}(A)$ *or* $\mathcal{CH}(B)$	$A + B$, $A \vee B$
`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 $\forall T$
- Example: `def dupl[A]: A ⇒ (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
$\forall A : A \Rightarrow A$	`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:
    $\forall A : 1 \Rightarrow A$; $\forall A \forall B : A \vee B \Rightarrow A$;
    $\forall A \forall B : A \Rightarrow A \times B$; $\quad \forall A \forall B : (A \Rightarrow B) \Rightarrow A$
- Given a type's formula, can we implement it in code?
  - Example: $\forall A \forall B : ((((A \Rightarrow B) \Rightarrow A) \Rightarrow A) \Rightarrow B) \Rightarrow B$
- 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):

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

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

- 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 $A, B, ...$ 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"

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"

Using school-level algebra to reason about 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: 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 equivalence)

$$\texttt{List}^A \equiv 1 + A \times \texttt{List}^A \equiv 1 + A \times (1 + A \times (1 + A \times (...)...)$$

$$\equiv 1 + A + A \times A + A \times A \times A + ... + A \times ... \times A \times \texttt{List}^A$$

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

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

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 isomorphic 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 $(A \times B) \Rightarrow C \neq (A \Rightarrow C) + (B \Rightarrow C)$ 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]⇒(A⇒B)⇒Reader[B,T]`
7. Implement $map^{A,B} : 1 + A \Rightarrow (A \Rightarrow B) \Rightarrow 1 + B$ with no "information loss", that is, `map(opt)(x ⇒ 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), (A, Float)]]` into an isomorphic type of the form $A \times (...)$; write the equivalence tests
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, ...)

Addendum

Random remarks regarding the topics of this section

- 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 information is insufficient to specify the code. It is only the fully type-parametric functions that have types informative enough for deriving the code automatically from the type.
- 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.
- When using a type-parametric `sealed trait` in Scala, there is a difference between representing a "named `Unit` type" via `case object A` vs. via `case class A[T]()`. Because a `case object` cannot have type parameters, some further features of Scala (covariance annotations) need to be used to get this working. May prefer `case class A[T]()`