Reasoning about types and code: an overview

Sergei Winitzki

Academy by the Bay

2019-08-17

A new book: *The Science of Functional Programming*

With examples in Scala

Practitioners of functional programming need to know how to:

- reason about types:
    - design the required custom data types for the given application
    - derive an equivalent simpler type when possible
    - use type constructions to create data types with required properties
- reason about code:
    - verify that given implementations satisfy the required laws (e.g. monad)
    - derive lawful custom implementations of important typeclasses
    - verify that certain functions are computationally equivalent
    - derive an equivalent simpler code when possible

This requires a *very limited* amount of mathematics (polynomials, monoids)

The book will explain (with examples and exercises):

- known techniques of reasoning about types and type constructors
- known techniques for symbolic calculations with code
- deriving and verifying laws symbolically (as equations for functions)
- real-life motivations for (and applications of) these techniques

Examples of reasoning tasks

1. Can we compute a value of type `Either[Z, R => A]` given a value of type `R => Either[Z, A]`? And conversely? (`A`, `R`, `Z` are type parameters.)
2. How to use `for`/`yield` with `Option[A]` and `Future[A]` together?

```
val result = for {
  a <- Future(...) // A computation that takes time and may fail.
  b <- Option(...) // A computation whose result may be unavailable.
  c <- Future(...) // A computation that takes time and may fail.
} yield ??? // Continue computations when results are available.
```

   Should `result` have type `Option[Future[A]]` or `Future[Option[A]]`?
3. Can we implement `flatMap` for the type constructor `Option[(A, A, A)]`?
4. What type describes a chain of `Future[A]` operations that, on any failure, will automatically execute specified cleanups in reverse order?
5. Different people define a "free monad" via different sets of case classes. Are these definitions equivalent? What is the difference?
6. How to define a free monad generated by a `Pointed` functor (i.e. when the functor already has the `pure` method)? Will that type have better performance than the standard free monad generated by a functor?

Notation for types

- A concise mathematical notation for types and type constructors:

Scala syntax	Type notation
type parameter `A`	A
tuple type `(A, B)`, or `case class C(a: A, b: B)`	$A \times B$
disjunctive type `Either[A, B]` or `trait/case classes`	$A + B$
function type `A => B`	$A \Rightarrow B$
`Unit` or a "named" unit type	$\mathbb{1}$
primitive type (`Int`, `String`, etc.)	Int, String, ...
`Nothing` (the void type)	$\mathbb{0}$
type constructor `type P[A] = Option[(A, A, A)]`	$P^A \triangleq \mathbb{1} + A \times A \times A$
parameterized method `def f[A]: A => (A, A)`	$f^A : A \Rightarrow A \times A$
parameterized value (Scala 3) `[A] => P[A]`	$\forall A.\, P^A$

This notation proved convenient for reasoning about equivalence of types:

$$(A + B) \times C \cong A \times C + B \times C \quad , \qquad A \Rightarrow B \Rightarrow C \cong A \times B \Rightarrow C \quad ,$$
$$\mathbb{0} + A \cong A \quad , \qquad A + B \Rightarrow C \cong (A \Rightarrow C) \times (B \Rightarrow C)$$

Notation for code

- A concise mathematical notation for code:

Scala syntax	Type notation
variable, or function argument `x: A`	$x^{:A}$
tuple value `(a, b)`	$a \times b$ or $a^{:A} \times b^{:B}$
value of disjunctive type `Left[A, B](x)`	$x^{:A} + \mathbb{0}^{:B}$
nameless function `{x: A => expr }`	$x^{:A} \Rightarrow \text{expr}$
`Unit` value, `()`	1
`def f[A, B]: A => B => A = x => _ => x`	$f^{:A\Rightarrow B\Rightarrow A} \triangleq x^{:A} \Rightarrow \_^{:B} \Rightarrow x$
`identity[A]` (the identity function)	id^{A} or $\text{id}^{:A\Rightarrow A}$
forward composition, `f andThen g`	$f \fatsemi g$ or $f^{:A\Rightarrow B} \fatsemi g^{:B\Rightarrow C}$
argument piping, `x.pipe(f)` (Scala 2.13)	$x \triangleright f$ where $x^{:A}$ and $f^{:A\Rightarrow B}$
lifted function, `p.map(f)` where `p: P[A]`	$p \triangleright f^{\uparrow P}$ where $p^{:P^{A}}$

This notation proved convenient for reasoning about equational laws:

$$f^{\uparrow P} \fatsemi g^{\uparrow P} \fatsemi h^{\uparrow P} = (f \fatsemi g \fatsemi h)^{\uparrow P}$$

Discussion

1. Show