Functional and Algebraic Domain Modeling

Algebraic Thinking for Evolution of Pure Functional Domain Models

Debasish Ghosh
@debasishg
Traveling back in time .. more than 40 years ..
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.
Reduction semantics is programming by composition

"normal form program," which is the result (reduction semantics)?

2.1.4 Clarity and conceptual usefulness of programs. Are programs of the model clear expressions of a process or computation? Do they embody concepts that help us to formulate and reason about processes?

2.2 Classification of Models

Using the above criteria we can crudely characterize three classes of models for computing systems—simple operational models, applicative models, and von Neumann models.

2.2.1 Simple operational models. Examples: Turing machines, various automata. Foundations: concise and useful. History sensitivity: have storage, are history sensitive. Semantics: state transition with very simple states. Program clarity: programs unclear and conceptually not helpful.

2.2.2 Applicative models. Examples: Church's lambda calculus [5], Curry's system of combinators [6], pure Lisp [17], functional programming systems described in this paper. Foundations: concise and useful. History sensitivity: no storage, not history sensitive. Semantics: reduction semantics, no states. Program clarity: programs can be clear and conceptually useful.

2.2.3 Von Neumann models. Examples: von Neumann computers, conventional programming languages. Foundations: complex, bulky, not useful. History sensitivity: have storage, are history sensitive. Semantics: state transition with complex states. Program clarity: programs can be moderately clear, are not very useful conceptually.
Von Neumann program for Inner Product

\[
c := 0 \\
\text{for } i := 1 \text{ step } 1 \text{ until } n \text{ do} \\
c := c + a[i] \times b[i]
\]

“It is dynamic and repetitive. One must mentally execute it to understand it”

- John Backus
Def Innerproduct =
(Insert +) o (ApplyToAll X) o Transpose

“IT'S STRUCTURE IS HELPFUL IN UNDERSTANDING IT WITHOUT MENTALLY EXECUTING IT”

- John Backus
“.. programs can be expressed in a language that has an associated algebra. *This algebra can be used to transform programs* and to solve some equations whose "unknowns" are programs, in much the same way one solves equations in high school algebra. Algebraic transformations and proofs use the language of the programs themselves, rather than the language of logic, which talks about programs.”

- John Backus
What is an Algebra?

Algebra is the study of algebraic structures

In mathematics, and more specifically in abstract algebra, an algebraic structure is a set (called carrier set or underlying set) with one or more finitary operations defined on it that satisfies a list of axioms

-Wikipedia

(https://en.wikipedia.org/wiki/Algebraic_structure)
The Algebra of Sets

given

SetA

a binary operation

\[ \phi : A \times A \rightarrow A \]

for specific \( a, b \)

\[ \text{for } (a, b) \in A \]

\[ \phi(a, b) \]

or

\[ a \phi b \]
Algebraic Thinking

• Denotational Semantics

  ✦ programs and the objects they manipulate are symbolic realizations of abstract mathematical objects

  ✦ the purpose of a mathematical semantics is to give a correct and meaningful correspondence between programs and mathematical entities in a way that is entirely independent of an implementation [Scott & Strachey, 1971]
Operational Thinking

• Operational Semantics
  ✦ formalize program implementation and how the various functions must be computed or represented
  ✦ not much of a relevance towards algebraic reasoning
Option[A]

A: Carrier Type of the algebra

Introduction Forms

Option.apply[A](a: A): Option[A]
Option.empty[A]: Option[A]
Option[A]

- A: Carrier Type of the algebra

**Introduction Forms**

- Option.apply[A](a: A): Option[A]
- Option.empty[A]: Option[A]

```scala
def f[A, B](func: A ⇒ B) = ???
optionA.map(f)
// Option[B]
```

**Combinators**

- Option flatMap

```scala
def f[A, B](func: A ⇒ Option[B]) = ???
optionA.flatMap(f)
// Option[B]
```
**Option[A]**

- **A**: Carrier Type of the algebra

### Introduction Forms

- `Option.apply[A](a: A): Option[A]`
- `Option.empty[A]: Option[A]`

```scala
def f[A, B](func: A => B) = ???
optionA.map(f)
// Option[B]
```

### Combinators

- `def f[A, B](func: A => Option[B]) = ???`
- `optionA.flatMap(f)
// Option[B]
```

### Eliminator Forms

- `optionA.getOrElse(default)
// A or B >: A`
Option[A]

- **A**: Carrier Type of the algebra

### Introduction Forms

- `Option.apply[A](a: A): Option[A]`
- `Option.empty[A]: Option[A]`

### Combinators

- `def f[A, B](func: A => B) = ???`
  - `optionA.map(f)`
  - `// Option[B]`
- `def f[A, B](func: A => Option[B]) = ???`
  - `optionA.flatMap(f)`
  - `// Option[B]`

### Eliminator Forms

- `optionA.getOrElse(default)`
  - `// A or B >: A`

### Laws

- `Option.empty[Int].flatMap(...) == Option.empty[Int]`
  - `// res1: Boolean = true`
- `Option.empty[Int].map(...) == Option.empty[Int]`
  - `// res2: Boolean = true`
A: Carrier Type of the algebra

Introduction Forms

Combinators

Eliminator Forms

Laws
• Thinking in terms of combinators (map/fatMap/fold) and their laws is *algebraic thinking*

• Thinking in terms of concrete implementations (pattern match with Some/None) is *operational thinking*
Module with an algebra

```scala
trait Monoid[A] {
  def zero: A
  def combine(l: A, r: => A): A
}

// identity
combine(x, zero) =
  combine(zero, x) = x

// associativity
combine(x, combine(y, z)) =
  combine(combine(x, y), z)
```

Module with an Algebra

trait Foldable[F[_]] {

  def foldl[A, B](as: F[A], z: B, f: (B, A) ⇒ B): B

  def foldMap[A, B](as: F[A], f: A ⇒ B) (implicit m: Monoid[B]): B =
  foldl(as,
    m.zero,
    (b: B, a: A) ⇒ m.combine(b, f(a))
  )

}
def mapReduce[F[_], A, B](as: F[A], f: A ⇒ B) (implicit ff: Foldable[F], m: Monoid[B]) =

    ff.foldMap(as, f)
def mapReduce[F[_], A, B](as: F[A], f: A ⇒ B) (implicit ff: Foldable[F], m: Monoid[B]) =

    ff.foldMap(as, f)

a complete map/reduce program abstracted as a functional form
def mapReduce[F[_], A, B](as: F[A], f: A ⇒ B) (implicit ff: Foldable[F], m: Monoid[B]) =

    ff.foldMap(as, f)

a complete map/reduce program abstracted as a functional form

derived intuitively from the algebras of a fold and a monoid
Building and understanding higher order abstractions is much more intuitive using algebraic than operational thinking.
Building and understanding higher order abstractions is much more intuitive using algebraic than operational thinking.
Healthy recipes for an algebra

(in a statically typed functional programming language)
trait Monoid[A] {
  def zero: A
  def combine(l: A, r: => A): A
}
// identity
combine(x, zero) =
    combine(zero, x) = x

// associativity
combine(x, combine(y, z)) =
    combine(combine(x, y), z)
trait Foldable[F[_]] {
  def foldl[A, B](as: F[A], z: B, f: (B, A) ⇒ B): B

  def foldMap[A, B](as: F[A], f: A ⇒ B) (implicit m: Monoid[B]): B =
    foldl(as, m.zero,
      (b: B, a: A) ⇒ m.combine(b, f(a)))
}
def mapReduce(F[_], A, B)(as: F[A],
     f: A ⇔ B)
    (implicit ff: Foldable[F],
     m: Monoid[B]) =
    ff.foldMap(as, f)
Implementation Independent

\( f : A \rightarrow B \) and \( g : B \rightarrow C \), we should be able to reason that we can compose \( f \) and \( g \) \textit{algebraically} to build a larger function \( h : A \rightarrow C \)
trait Repository[M[_]] {
    def query[A](key: String): M[Option[A]]
    // ..
}
What is a domain model?

A domain model in problem solving and software engineering is a conceptual model of all the topics related to a specific problem. It describes the various entities, their attributes, roles, and relationships, plus the constraints that govern the problem domain. It does not describe the solutions to the problem.

Conference Management System

Bounded Context A
Conference reservations

- Domain model A
  - Ubiquitous language
  - Entities
  - Value objects
  - Services

- Code
- Schemas
- Other artifacts

Bounded Context B
Program management

- Domain model B
  - Ubiquitous language
  - Entities
  - Value objects
  - Services

- Code
- Schemas
- Other artifacts

Bounded Context C
Badge printing

- Domain model C
  - Ubiquitous language
  - Entities
  - Value objects
  - Services

- Code
- Schemas
- Other artifacts

A Bounded Context

• has a consistent vocabulary
• a set of domain behaviors modeled as functions on domain objects implemented as types
• each of the behaviors honor a set of business rules
• related behaviors grouped as modules
Domain Model = \( \bigcup_{(i)} \text{Bounded Context}(i) \)

Bounded Context = \{ \( m[T_1,T_2,..] \) | \( T(i) \in \text{Types} \) \}

Module = \{ \( f(x,y,..) \) | \( p(x,y) \in \text{Domain Rules} \) \}

- domain function
- on an object of types \( x, y, .. \)
- composes with other functions
- closed under composition

- business rules
Domain Model = \( \bigcup (i) \) Bounded-Context\( (i) \)

Bounded-Context = \{ m[T1,T2,..] | T(i) \in \) Types \}

Module = \{ f(x,y,..) | p(x,y) \in \) Domain Rules \}

- domain function
- on an object of types x, y, ..
- composes with other functions
- closed under composition

- business rules
Client places order - flexible format
Client places order - flexible format

Transform to internal domain model entity and place for execution
1. Client places order - flexible format

2. Transform to internal domain model entity and place for execution

3. Trade & Allocate to client accounts
trait Trading[M[_]] {

  def orders(csvOrder: String): M[List[Order]]

  def execute(orders: List[Order],
               market: Market,
               brokerAccountNo: AccountNo)
          : M[List[Execution]]

  def allocate(executions: List[Execution],
               clientAccounts: List[AccountNo])
          : M[List[Trade]]
}

Effect Type that parameterizes the Trading algebra
trait Trading[M[_]] {

  def orders(csvOrder: String): M[NonEmptyList[Order]]

  def execute(orders: NonEmptyList[Order],
               market: Market,
               brokerAccountNo: AccountNo)
  : M[NonEmptyList[Execution]]

  def allocate(executions: NonEmptyList[Execution],
               clientAccounts: NonEmptyList[AccountNo])
  : M[NonEmptyList[Trade]]
}

Effect Type that parameterizes the Trading algebra
Effects

• an algebraic way of handling computational effects like non-determinism, probabilistic non-determinism, exceptions, interactive input-output, side-effects, continuations etc.

• first formalized by Plotkin and Power [2003]
Option[A]
  (partiality)

Either[A,B]
  (disjunction)

List[A]
  (non-determinism)

Reader[E,A]
  (read from environment aka dependency injection)

State[S,A]
  (state management)

IO[A]
  (external side-effects)

Writer[W,A]
  (logging)

.. and there are many many more ..
The answer that the effect computes

The additional stuff modeling the computation
Side-effects

• Error handling?
  • throw / catch exceptions is not RT

• Partiality?
  • partial functions can report runtime exceptions if
    invoked with unhandled arguments (violates RT)

• Reading configuration information from environment?
  • may result in code repetition if not properly handled

• Logging?
Side-effects

- Database writes
- Writing to a message queue
- Reading from stdin / files
- Interacting with any external resource
- Changing state in place
modularity

side-effects don't compose
def orders(csvOrder: String): M[NonEmptyList[Order]]

def execute(orders: NonEmptyList[Order],
market: Market,
brokerAccountNo: AccountNo)
  M[NonEmptyList[Execution]]

def allocate(executions: NonEmptyList[Execution],
clientAccounts: NonEmptyList[AccountNo])
  M[NonEmptyList[Trade]]

trait Trading[M[_]] {
  def orders(csvOrder: String): M[NonEmptyList[Order]]

  def execute(orders: NonEmptyList[Order],
  market: Market,
  brokerAccountNo: AccountNo)
    M[NonEmptyList[Execution]]

  def allocate(executions: NonEmptyList[Execution],
  clientAccounts: NonEmptyList[AccountNo])
    M[NonEmptyList[Trade]]
}

Effect Types offer compositionality even in the presence of side-effects.
• The $M[\_\_]$ that we saw is an opaque type - it has no denotation till we give it one

• The denotation that we give to $M[\_\_]$ depends on the *semantics of compositionality* that we would like to have for our domain model behaviors
def generateTrade[M[_]: Monad](T: Trading[M]) = for {
  orders ← T.orders(csvOrders)
  executions ← T.execute(orders, Market.NewYork, brokerAccountNo)
  trades ← T.allocate(executions, clientAccountNos)
}

yield trades
The Program

```python
def generateTrade[M[_]: Monad](T: Trading[M]) = for {

  orders ← T.orders(csvOrders)
  executions ← T.execute(orders, Market.NewYork, brokerAccountNo)
  trades ← T.allocate(executions, clientAccountNos)

} yield trades
```

Composition of the algebra of a Monad with our domain algebra of trading
Parametricity

• Trading module is polymorphic on \texttt{M[_]}. We could have committed to \texttt{Trading[IO]} upfront - but then we are making decisions on behalf of the call site. This is premature evaluation.

• In implementation we can say \texttt{M[_]}: \texttt{Monad} and suddenly the only operations available to us are \texttt{pure} and \texttt{flatMap}. This reduces the surface area of implementation. With \texttt{IO} we could have done anything in the implementation.
trait Accounting[M[_]] {

}
def generateTradeAndPostBalance[M[_]:Monad](T: Trading[M], A: Accounting[M]) = for {

  orders ← T.orders(csvOrders)
  executions ← T.execute(orders, Market.NewYork, brokerAccountNo)
  trades ← T.allocate(executions, clientAccountNos)
  balances ← A.postBalance(trades)

} yield (trades, balances)
The Program

def generateTradeAndPostBalance[M[[_]:Monad]
    (T:Trading[M], A:Accounting[M]) = for {
      orders ← T.orders(csvOrders)
      executions ← T.execute(orders, Market.NewYork, brokerAccountNo)
      trades ← T.allocate(executions, clientAccountNos)
      balances ← A.postBalance(trades)
    } yield (trades, balances)

Composition of multiple domain algebras
• .. we have intentionally kept the algebra open for interpretation..

• .. there are use cases where you would like to have multiple interpreters for the same algebra ..
Interpreters

class TradingInterpreter[M[+_.]]

(implicit E: MonadError[M, Throwable],
 R: ApplicativeAsk[M, Repository[M]])

extends Trading[M] {

// ..
}

monad with error handling

asks for a repository from the environment
class TradingInterpreter[M[_]]

(implicit E: MonadError[M, Throwable],
 R: ApplicativeAsk[M, Repository[M]])

extends Trading[M] {

// ..
}

monad with error handling

asks for a repository from the environment

InMemoryRepository[M]

DoobieRepository[M]
Finally ..

```scala
implicit val .. = // ..

generateTradeAndPostBalance(
  new TradingInterpreter[IO],
  new AccountingInterpreter[IO]
)
```
Effects ≠ Side-effects
“Effects and side-effects are not the same thing. Effects are good, side-effects are bugs. Their lexical similarity is really unfortunate because people often conflate the two ideas”

- Rob Norris at scale.bythebay.io talk - 2017 (https://www.youtube.com/watch?v=po3wmq4S15A)
Takeaways

• Algebra **scales** from that of one single data type to an entire bounded context

• Algebras **compose** enabling composition of domain behaviors

• Algebras let you focus on the compositionality **without any context of implementation**

• Statically typed functional programming is **programming with algebras**
Takeaways

- Abstract **early**, interpret as **late** as possible
- Abstractions / functions compose **only** when they are **abstract** and **parametric**
- Modularity **in the presence of side-effects** is a challenge
- Effects as algebras are **pure values** that can compose based on laws
- Honor the law of using the **least powerful abstraction** that works
Functional and Reactive Domain Modeling

Debasish Ghosh
Foreword by: Jonas Boner
October 2016 • ISBN 9781617292248 • 320 pages • printed in black & white

"Brings together three different tools—domain-driven design, functional programming, and reactive principles—in a practical way."

From the Foreword by Jonas Boner, Creator of Akka

Functional and Reactive Domain Modeling teaches you how to think of the domain model in terms of pure functions and how to compose them to build larger abstractions.
Questions?
References
