THE ORIGINS OF FREE

Adam Warski, SoftwareMill
10/2/2017, LambdaDays

THE PLAN

- Why bother?
- Universal algebra
- Free algebras
- The meaning of ~~life~~ free
- Monads & free monads
- Free monads in Haskell, Cats and Scalaz

- It's simpler than it sounds

WHY BOTHER WITH FREE?

- Define a composable program
- Using high-level custom instructions
- Run it in context later, given instruction interpretation

```
def issueCreditCard(u: UserId): Free[BankOps, CreditLimit] = for {
  user <- LookupUserData(u).liftFree
  otherCredits <- FetchOtherCredits(user).liftFree
  val limit = calculateLimit(user, otherCredits)
  creditCard <- IssueNewCard(user, limit).liftFree
  _ <- SendEmail(user.email, cardIssuedEmail(user, limit)).liftFree
} yield limit
```

WHY BOTHER WITH FREE?

```
def issueCreditCard(u: UserId): Free[BankOps, CreditLimit] = for {
  user <- LookupUserData(u).liftFree
  otherCredits <- FetchOtherCredits(user).liftFree
  val limit = calculateLimit(user, otherCredits)
  creditCard <- IssueNewCard(user, limit).liftFree
  _ <- SendEmail(user.email, cardIssuedEmail(user, limit)).liftFree
} yield limit

val productionInterpreter: BankOp ~> Future = {
  override def apply[A](bo: BankOp[A]): Future[A] = bo match {
    case LookupUserData(u) => oracleDB2dao.lookupUser(u)
    case FetchOtherCredits(user) =>
        legacySoapSystem.fetchCredits(user)
    // …
  }
}
```

WHY BOTHER WITH FREE?

```scala
def issueCreditCard(u: UserId): Free[BankOps, CreditLimit] = for {
  user <- LookupUserData(u).liftFree
  otherCredits <- FetchOtherCredits(user).liftFree
  val limit = calculateLimit(user, otherCredits)
  creditCard <- IssueNewCard(user, limit).liftFree
  _ <- SendEmail(user.email, cardIssuedEmail(user, limit)).liftFree
} yield limit

val testInterpreter: BankOp ~> Id = {
  override def apply[A](bo: BankOp[A]): Id[A] = bo match {
    case LookupUserData(u) => new User(…)
    case FetchOtherCredits(user) => List(Credit(1000000.usd))
    // …
  }
}
```

WHY BOTHER WITH FREE?

```
def issueCreditCard(u: UserId): Free[BankOps, CreditLimit] = for {
  user <- LookupUserData(u).liftFree
  otherCredits <- FetchOtherCredits(user).liftFree
  val limit = calculateLimit(user, otherCredits)
  creditCard <- IssueNewCard(user, limit).liftFree
  _ <- SendEmail(user.email, cardIssuedEmail(user, limit)).liftFree
} yield limit
```

```
val result: CreditLimit = issueCreditCard(UserId(42))
                            .foldMap(testInterpreter)
```

ABOUT ME

- Software Engineer, co-founder @ SOFTWAREMILL
- Mainly Scala
- Open-source: Quicklens, MacWire, ElasticMQ, ScalaClippy, …
- Long time ago: student of Category Theory

WHAT IS AN ALGEBRA?

"algebra is the study of mathematical symbols and the rules for manipulating these symbols"

Wikipedia, 2017

"the part of mathematics in which letters and other general symbols are used to represent numbers and quantities in formulae and equations."

Google Search, 2017

$y = ax + b$

$E = mc^2$

```
f(10◊x) = K(‣9)
def sum(l: List[L]) = l.fold(_ + _)
main = getCurrentTime >>= print
```

UNIVERSAL ALGEBRA: SIGNATURE

- Goal: Model programs as algebras
- Let's generalise!
- Studies algebraic structures, rather than concrete models
- Syntax: algebraic signature $\Sigma = (S, \Omega)$
  - type names: set S
  - operation names: family Ω of sets indexed by $S^* \times S$

UNIVERSAL ALGEBRA: SIGNATURE EXAMPLE

$$S = \{int, str\}$$
$$\Omega_{\epsilon,int} = \{0\}$$
$$\Omega_{int,int} = \{succ\}$$
$$\Omega_{(int,int),int} = \{+\}$$
$$\Omega_{(str,str),str} = \{++\}$$
$$\Omega_{int,str} = \{toString\}$$

$$toString(succ(0) + succ(succ(0)))$$

UNIVERSAL ALGEBRA: ALGEBRA

- A specific interpretation of the **signature**
  - for each type, a set
  - for each operation, a function between appropriate sets

$\Sigma = (S, \Omega)$, $S = \{int, str\}$ and $\Omega = \{0, succ, +, ++, toString\}$

We can define a Σ-algebra A:

$$|A|_{int} = \{0, 1, 2, ...\} = \mathbb{N}$$
$$|A|_{str} = \{"a", "aa", ..., "b", "ab", ...\}$$

$$succ_A = \lambda x.x + 1$$
$$+_A = \lambda xy.x + y$$
$$\ldots$$

TERM ALGEBRA

- Can we build an algebra out of pure syntax?
- Expressions (terms) that can be built from the signature
- Rather boring, no interpretation at all

$\Sigma = (S, \Omega)$, $S = \{int, str\}$ and $\Omega = \{0, succ, +, ++, toString\}$

We define the term algebra T_Σ:

$$|T_\Sigma|_{int} = \{0, succ(0), succ(succ(0)), ..., 0 + 0, 0 + succ(0), ...\}$$
$$|T_\Sigma|_{str} = \{toString(0), toString(succ(0)), ..., toString(0){+}{+}toString(0), ...\}$$

$$succ_{T_\Sigma}(t) = succ(t), \text{ e.g. } succ_{T_\Sigma}(succ(0)) = succ(succ(0))$$
$$+_{T_\Sigma}(t_1, t_2) = t_1 + t_2$$
$$\ldots$$

TERM ALGEBRA

$\Sigma = (S, \Omega),\ S = \{int, str\}$ and $\Omega = \{0, succ, +, ++, toString\}$

- Defined **inductively**
  - base: all constants are terms
  - step: any functions we can apply on previous terms

$$\{0\}$$
$$\{0, 0 + 0, succ(0), toString(0)\}$$
$$\{0, 0 + 0, succ(0), 0 + succ(0), succ(0) + 0, succ(0) + succ(0),$$
$$toString(0), toString(succ(0)), toString(0) + +toString(0)\}$$

HOMOMORPHISM

- Homomorphism is a function between algebras
  - For each type, functions between type interpretations
  - Such that operations are preserved

$\Sigma = (S, \Omega)$, $S = \{int, str\}$ and $\Omega = \{0, succ, +, ++, toString\}$

When A and B are Σ-algebras, $f : A \to B$ is a homomorphism when:

$f_{int} : |A|_{int} \to |B|_{int}$

$f_{str} : |A|_{str} \to |B|_{str}$

$\forall_{x \in |A|_{int}} f_{int}(succ_A(x)) = succ_B(f_{int}(x))$

$\forall_{xy \in |A|_{int}} f(x +_A y) = f(x) +_B f(y)$

$\forall_{x \in |A|_{int}} f_{str}(toString_A(x)) = toString_B(f_{int}(x))$

INITIAL ALGEBRA

Σ-algebra I is **initial** when for *any other* Σ-algebra A there is **exactly one** homomorphism between them.

Theorem 1 T_Σ *is initial*

INITIAL ALGEBRA

Theorem 1 *T_Σ is initial*

$\Sigma = (S, \Omega)$, $S = \{int, str\}$ and $\Omega = \{0, succ, +, ++, toString\}$

We can define a Σ-algebra A:

$|A|_{int} = \{0, 1, 2, ...\} = \mathbb{N}$

$|A|_{str} = \{"a", "aa", ..., "b", "ab", ...\}$

$succ_A = \lambda x.x + 1$

$+_A = \lambda xy.x + y$

$\cdots$

$$f : T_\Sigma \to A$$

$$f(0_{T_\Sigma}) = 0_A$$
$$f(succ_{T_\Sigma}(t)) = succ_A(f(t))$$
$$...$$

INITIAL ALGEBRA

Σ-algebra I is **initial** when for *any other* Σ-algebra A there is **exactly one** homomorphism between them.

Theorem 1 *T_Σ is initial*

- Only one way to interpret a term
- *no junk*: term algebra contains only what's absolutely necessary
- *no confusion*: no two values are combined if they don't need to be
- There's only one initial algebra (up to isomorphism)

INITIAL ALGEBRA

- This algebra is definitely **not initial**:

  $\Sigma = (S, \Omega),\ S = \{int, str\}$ and $\Omega = \{0, succ, +, ++, toString\}$

  We can define a Σ-algebra A:

  $$|A|_{int} = \{0, 1, 2, ...\} = \mathbb{N}$$
  $$|A|_{str} = \{"a", "aa", ..., "b", "ab", ...\}$$

- *Junk*: strings *“a”*, *“b”*, …
- *Confusion: 0+succ(0)* is same as *succ(0)+0*

FREE ALGEBRA

For any set X, $T_\Sigma(X)$ is the term algebra with X added as "constants" (but called variables)

$\Sigma = (S, \Omega)$, $S = \{int, str\}$ and $\Omega = \{0, succ, +, ++, toString\}$

$$X_{int} = \{i, j, k\}$$
$$X_{str} = \{s_1, s_2\}$$

$$succ(i) + j + succ(succ(k))$$
$$s_1 + +toString(0)$$
$$toString(succ(0) + k) + +s_2$$

FREE ALGEBRA

For any set X, $T_\Sigma(X)$ is the term algebra with X added as "constants" (but called variables)

Σ-algebra I is **free over X** ($X \subset I$) when for *any other* Σ-algebra A, *any function* $f : X \to |A|$ **extends uniquely** to a homomorphism $f^\# : I \to A$ between them.

Theorem 1 *For any variable set* X, $T_\Sigma(X)$ *is free*

- An interpretation of the variables determines an interpretation of any term

FREE ALGEBRA EXAMPLE

$\Sigma = (S, \Omega)$, $S = \{int, str\}$ and $\Omega = \{0, succ, +, ++, toString\}$

$$X_{int} = \{i, j, k\}$$
$$X_{str} = \{s_1, s_2\}$$
$$|A|_{int} = \mathbb{N},\ |A|_{str} = \{"a", "aa", ..., "b", "ab", ...\}$$

$$succ_A = \lambda x.x + 1$$
$$+_A = \lambda xy.x + y$$
$$\ldots$$

$$f : X \to |A|$$
$$f(i) = 10, f(j) = 5, f(k) = 42$$
$$f(s_1) = "lambda", f(s_2) = "days"$$

$$f^{\#} : T_\Sigma(X) \to A$$
$$f^{\#}(toString(succ(j) + succ(0)) + +s_1) = "7lambda"$$
$$f^{\#}(s_2 + +toString(k) + +s_2) = "lambda42days"$$

MEANING OF FREE

- **Free to interpret in any way**
  - no constraints
- **Free of additional structure**
  - only what's absolutely necessary
- *No junk, no confusion*

FREE RECAP

- Algebraic signature: $\Sigma = (S, \Omega)$
- All possible interpretations: algebras
- For any variable set X
- The term algebra $T_\Sigma(X)$ is **free**
  - any interpretation of the variables $f : X \to |A|$
  - determines an interpretation of any term $f^\# : T_\Sigma(X) \to A$

- A general construction

MODELLING SEQUENTIAL PROGRAMS: MONADS

- A sequential program can:
  - return a value (`pure`)
  - compute what to do next basing on previous result (`flatMap`)
- People decided to call an object with such operations a *Monad*
- Hence, we'll use *Monads* to represent programs as data
  - + sanity laws

FREE MONAD

- *Signature* ~ `pure` + `flatMap`
- *Variables* ~ operations (our `DSL`)
- *Free Monad* ~ terms built out of `pure`, `flatMap`, our `DSL`
  - modulo monad laws!
  - e.g. `flatMap(pure(x), f) = f(x)`

Interpretation of the DSL determines the interpretation of the whole program

FREE MONAD

- Our “world” (category) are Scala/Haskell/… monads (not algebras)
- The “world” (category) of the variables are generic Scala/Haskell/… terms (not sets)
- Signature:
  - types: `M[_]`
  - operations:
    - `pure[A]: A => M[A]`
    - `flatMap[A, B](ma: M[A], f: A => M[B]): M[B]`
- Modulo monad laws

FREE IN CATS/SCALAZ

```scala
trait Free[F[_], A]

object Free {
  case class Pure[F[_], A](a: A)          extends Free[F, A]
  case class Suspend[F[_], A](a: F[A])  extends Free[F, A]
  case class FlatMapped[F[_], B, C](
      c: Free[F, C], f: C => Free[F, B] extends Free[F, B]
}
```

FREE IN HASKELL

```
data Free f r = Free (f (Free f r)) | Pure r
```

```
trait Free[F[_], A]

object Free {
  case class Pure[F[_], A](a: A)                extends Free[F, A]
  case class Join[F[_], A](f: F[Free[F, A]]) extends Free[S, A]
}
```

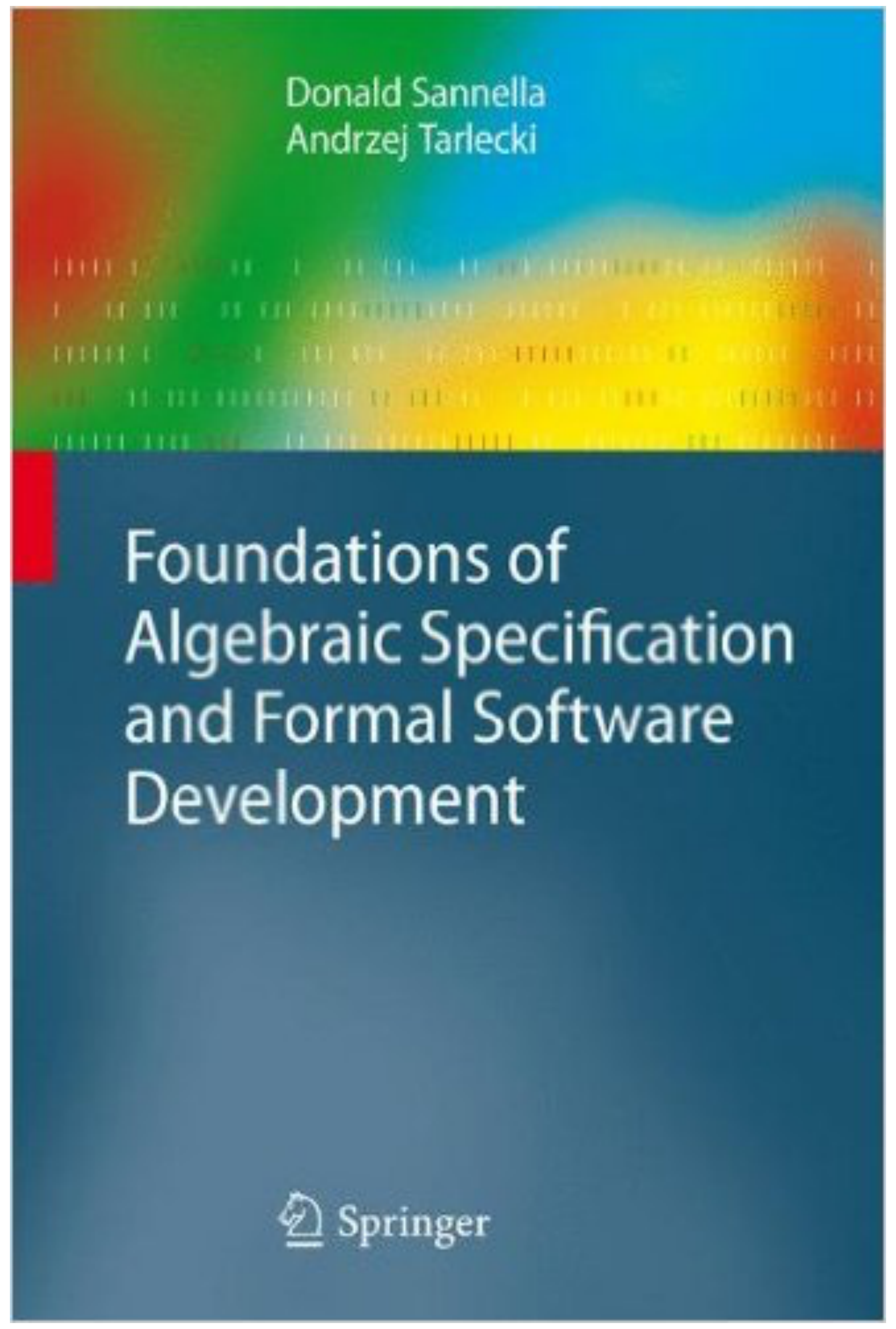
f/F[_] *must be a functor!*

SUMMING UP

- Direct construction of free algebras
- Hand-wavy construction of free monad
- Free
  - free to interpret in any way
  - free of constraints
  - *no junk, no confusion*
- Free in Haskell is the same free as in Scala

FURTHER READING

- “Foundations of Algebraic Specification and Formal Software Development” by Donald Sannella and Andrzej Tarlecki
- The Internet



THANK YOU!