Effectful functional programming

Release 1.0

Pierre-Évariste Dagand

Jan 25, 2018

CONTENTS

Didier Rémy's lecture:

- *extend* programming language with effects (mutable references, exceptions, etc.)
- *operational* semantics: mimic the behavior of a machine

What if:

- we want a denotational model/semantics? (compositionality!)
- we want a *pure* language? (eg. Haskell)
- we cannot afford an impure language? (eg. a proof assistant)

Today, we shall:

- give a denotational model for a monomorphic memory cell
- present a general framework for modelling *algebraic* effects
- meet a few monads (reader, writer, non-determinism, random, CPS, etc.)

Notions of computations determine monads:

1. syntax of effectful operations
2. equational theory of effectful operations
3. denotational semantics = quotiented syntax
4. interpreter / monad!

CHAPTER

ONE

HISTORICAL BACKGROUND

Notions of computations and monads, Eugenio Moggi (1991):

- “generic” syntax for writing effectful programs
- denotational model based on a categorical notion
- key idea: interpretation in the Kleisli category of a monad
- compositionality: that’s what a category is for!

The essence of functional programming, Phil Wadler (1992):

- bring monads to “mainstream” programming
- key idea : program *in* the denotational model
- Scottish cottage industry: the state monad, exception monad, list monad, CPS monad, etc.

Longstanding open problem:

- combining effects
- example: stateful *and* exceptionful program
- what is the “composition” of the state and exception monad?
- how to handle their interaction?
- example: upon an exception, do we roll-back the state, or not?

Notions of computations determine monads, Power & Plotkin (2002):

- describe syntax through an algebraic signature
- specify semantics through an equational theory
- obtain (some) monads from a more primitive presentation: a Lawvere theory
- key idea: syntax can be combined (trivially), leave the user pick an appropriate semantics

Segal condition meets computational effects, Melliès (2010):

- abstract non-sense does not run programs
- concretely, how do we get the state monad from its equations?
- key idea: use good ol’ rewriting theory
- good idea: sitting next to Tarmo Uustalu in a conference

Nowadays:

- algebraic effects are all the rage

- Even OCaml has some
- Open problems (probably): comparing expressive power & modular compilation

CHAPTER

TWO

STATEFUL OPERATIONS

Let S be a Set. In this section, we are interested in programs manipulating a single reference cell of monomorphic type S.

Such stateful programs are built from two state-manipulating operations:

- get, which returns the current state
- set, which updates the current state

We formalize this intuition with the following signature:

```
data ΣState (X : Set) : Set where
  `get : ⊤ × (S → X) → ΣState X
  `set : S × (⊤ → X) → ΣState X
```

Exercise (difficulty: 1) Note that this type defines an endofunctor on Set:

```
ΣState-map : ∀{V W} → (V → W) → ΣState V → ΣState W
ΣState-map f s = {!!}

ΣState-map-id : ∀{V} (x : ΣState V) →
                 ΣState-map (λ xs → xs) x ≡ x
ΣState-map-id = {!!}

ΣState-map-∘ : ∀{U V W} (f : U → V)(g : V → W)(x : ΣState U) →
               ΣState-map (g ∘ f) x ≡ ΣState-map g (ΣState-map f x)
ΣState-map-∘ = {!!}
```

Remark: an equivalent (but more modular) way to obtain the same signature is through the following constructs:

```
data ΣGet (X : Set) : Set where
  `get : ⊤ × (S → X) → ΣGet X
data ΣSet (X : Set) : Set where
  `set : S × (⊤ → X) → ΣSet X
data _⊕_ (F G : Set → Set)(X : Set) : Set where
  inl : F X → (F ⊕ G) X
  inr : G X → (F ⊕ G) X
```

which gives ΣState ≡ ΣGet ⊕ ΣSet.

It is too early to follow that path but we shall bear in mind this more elementary decomposition.

Exercise (difficulty: 5) What is the minimal language for describing such signatures? We shall make sure that these signature are always functorial, for reasons that will become evident in the following. Generalize the constructions presented in these lecture notes using that language.

2.1 Free term algebra for State

From a signature, we can build a *syntax* for writing stateful programs: we just need to combine ‘get’s, ‘set’s and pure computations (‘return’). The resulting syntactic object is easily described by an inductive type:

```
data StateF (V : Set) : Set where
  return : V → StateF V
  op : ΣState (StateF V) → StateF V
```

In this (very small) language, we have two smart constructors, `get` and `set`, whose definition can be automatically derived from the signature:

```
get : ⊤ → StateF S
get tt = op (`get (tt , λ s → return s))
```

Exercise (difficulty: 1) Implement *set*:

```
set : S → StateF ⊤
set s = {!!}
```

Note that the type of these operations is exactly what we expect in, say, OCaml modulo the presence of `StateF`. It is useful to think of `StateF` as a modality on the arrow type, documenting what effects the function may perform (aside from computing).

Exercise (difficulty: 3) thinking of `V` as a set of variables, `StateF V` denotes stateful computations with variables in `V`. By exploiting the functoriality of `ΣState`, we can implement a form of *composition* (some may say *sequencing*!) of stateful programs. Formally, we have that `StateF` is a monad (the free monad):

```
{-# TERMINATING #-}
_>>=_ : ∀{V W} → StateF V → (V → StateF W) → StateF W
sv >>= mf = {!!}
```

If one thinks of `V` and `W` as sets of variables, then `>>=` (pronounced **bind**) can be thought as implementing a simultaneous substitution. One can also think of these objects as trees (*ie.* syntax trees) terminated by pure values of type `V`, to which one grafts trees terminated by pure values of type `W`. Both intuitions are useful.

Exercise (difficulty: 3): Rewrite `>>=` in such a way that Agda is able to check that it is indeed terminating. Hint: use a pair of mutually recursive functions.

Remark there is nothing special about `StateF`: given any (well-behaved) endofunctor `F : Set → Set`, we can build another functor `Free F : Set → Set` which happens to be a monad: this is the free monad construction which provides, for free, the substitution `>>=`. Free monads seem to provide an infinite source of blog posts and Haskell packages, here are a few:

- https://www.fpcomplete.com/user/dolio/many-roads-to-free-monads
- http://blog.sigfpe.com/2014/04/the-monad-called-free.html
- http://hackage.haskell.org/package/free-operational

Remark from a categorical perspective, it is a bit improper to call `StateF` the “free monad”: as we shall see, category theorists expect some form of quotienting over the terms with have built. Here, we just have a lump of syntax. Rather than “free monad”, we shoud favor the notion of “free term algebra”.

At this stage, we can write (but not execute!) stateful programs, such as:

```
test0 : StateF S
test0 = get tt >>= λ s →
        set s >>= λ _ →
        get tt >>= λ s' →
        return s'

test1 : StateF S
test1 = get tt >>= λ s' →
        return  s'

test2 : StateF S
test2 = get tt >>= λ s →
        set s >>= λ _ →
        return s

random : StateF ℕ
random = get tt >>= λ seed →
         let n = toℕ ((seed *ℕ 25173 + 1725) mod 65536) in
         set n >>= λ _ →
         return n
```

2.2 Monad laws

We have equipped the datatype `StateF` with quite a bit of *structure*. Before delving further into the the specifics of stateful computations, we are going to prove 3 general results, the *monad laws*, which we expect to hold for any such structure, irrespectively of its particular semantics.

The monadic laws specify the interaction between `return` – which brings pure values into stateful programs – and _>>=_ – which applies stateful functions.

Exercise (difficulty: 1) the first law states that applying a stateful program to a pure value amounts to performing a standard function application or, put otherwise, `return` is a left unit for _>>=_:

```
bind-left-unit : ∀ {X Y} → (x : X)(k : X -> StateF Y) →
  (return x >>= k) ≡ k x
bind-left-unit x k = {!!}
```

Exercise (difficulty: 4) the second law states that returning a stateful value amounts to giving the stateful computation itself or, put otherwise, `return` is a right unit for _>>=_:

```
{-# TERMINATING #-}
bind-right-unit : ∀ {X} → (mx : StateF X) →
              mx >>= return ≡ mx
bind-right-unit = {!!}
  where postulate ext : Extensionality Level.zero Level.zero
```

This exercise is artificially difficult because of the need to convince Agda's termination checker. One should feel free to convince oneself of the termination of the straightforward definition instead of fighting the termination checker. We will also need to postulate functional extensionality.

Exercise (difficult: 2) finally, the third law states that we can always parenthesize _>>=_ from left to right or, put otherwise, _>>= is associative:

```
{-# TERMINATING #-}
bind-compose : ∀ {X Y Z} → (mx : StateF X)(f : X -> StateF Y)(g : Y -> StateF Z) →
  ((mx >>= f) >>= g) ≡ (mx >>= λ x → (f x >>= g))
bind-compose = {!!}
  where postulate ext : Extensionality Level.zero Level.zero
```

There is a familiar object that offers a similar interface: (pure) function! For which `_>>=_` amounts to composition and `return` is the identity function. Monads can be understood as offering "enhanced" functions, presenting a suitable notion of composition and identity *as well as* effectful operations. For the programmer, this means that we have `let _ = _ in _` for pure functions and `_>>=_` for effectful functions, both subject to (morally) the same laws of function composition.

2.3 Equational theory of State

Intuitively, `test0`, `test1` and `test2` denote the same program. This section aims at stating this formally.

To do so, we equip our syntax with an equational theory. That is, we need to specify which kind of identities should hold on stateful programs. Or, put otherwise and following an operational approach, we relationally specify the reduction behavior of `StateF`, seen as an embedded language. We want:

```
data _↝_ {V : Set} : StateF V → StateF V → Set where


  get-get : ∀{k : S → S → StateF V} →
          (get tt >>= (λ s → get tt >>= λ s' → k s s' )) ↝ (get tt >>= λ s → k s s )

  set-set : ∀{k s₁ s₂} →
          (set s₁ >>= (λ _ → set s₂ >>= λ _ → k)) ↝ (set s₂ >>= λ _ → k)

  get-set : ∀{k} →
          (get tt >>= λ s → set s >>= λ _ → k) ↝ k

  set-get : ∀{k s} →
          (set s >>= (λ _ → get tt >>= k)) ↝ (set s >>= λ _ → k s)
```

In English, this amounts to the following rules:

- **rule `get_get`: getting the current state twice is equivalent to getting it** only once
- **rule `set_set`: setting the state twice is equivalent to performing only the** last 'set'
- **rule `get-set`: getting the current state and setting it back in is equivalent to** doing nothing
- **rule `set-get`: setting the state then getting its value is equivalent to setting** the state and directly moving on with that value

Remark where do these equations come from? Quite frankly, I took them from Matija Pretnar's PhD thesis. Paul-André Melliès would start from a minimal set of equations and run Knuth-Bendix completion algorithm to find a confluent equational theory/term rewriting system.

Remark coming from a mathematical background, one may understand this formalism as a generalization of algebraic structures such as monoids, groups, etc.:

- we start with a signature of operations, such as "there is a unary symbol `1` and a binary symbol `.`".
- then, we give a set of axioms equating open terms, such as `(a . b) . c = a . (b . c)`, `1 . a = a`, and `a . 1 = a`.

From local equations, we easily build its congruence closure (includes ↝, transitive, reflexive, symmetric, and lift from subterms to terms):

```
data _~_ {V : Set} : StateF V → StateF V → Set₁ where
  inc : ∀{p q} → p ↝ q → p ~ q

  trans : ∀{p q r} → p ~ q → q ~ r → p ~ r
  refl : ∀{p} → p ~ p
  sym : ∀{p q} → p ~ q → q ~ p

  cong : ∀{W}(tm : StateF W){ps qs : W → StateF V}  →
         (∀ w → ps w ~ qs w) →
         (tm >>= ps) ~ (tm >>= qs)
```

To reason up to this equivalence relation, we can state that elements of a set `V` should be considered up to `~`: this defines a so-called (and dreaded) setoid:

```
setoid : Set → Setoid _ _
setoid V = record
  { Carrier       = StateF V
  ; _≈_           = _~_
  ; isEquivalence = isEquivalence
  }
  where  isEquivalence : ∀ {V : Set} → IsEquivalence (_~_ {V = V})
         isEquivalence = record
           { refl  = refl
           ; sym   = sym
           ; trans = trans
           }
```

Exercise (difficulty: 1 or 5) we can now formally reason about the equivalence of programs. This is not only of formal interest, this is also at the heart of compiler optimizations, code refactoring, etc.:

```
prog1 : StateF ℕ
prog1 =
  get tt >>= λ x →
  set (1 + x) >>= λ _ →
  get tt >>= λ y →
  set (2 + x) >>= λ _ →
  get tt >>= λ z →
  set (3 + y) >>= λ _ →
  return y

prog2 : StateF ℕ
prog2 =
  get tt >>= λ x →
  set (4 + x) >>= λ _ →
  return (1 + x)

prog-equiv : prog1 ~ prog2
prog-equiv = {!!}
```

CHAPTER

THREE

SEMANTICS: STATE ≡ STATEF/~

Lawvere theory tells us that if we were to *quotient* the term algebra `StateF` with the equivalence relation `~`, we would obtain a monad, the `State` monad. If you are familiar with Haskell, you already know a State monad, which is usually defined as `S → S × V` to represent stateful computations using a single memory reference of sort `S` and returning a result of sort `V`.

However, in type theory (and in programming in general) quotienting must be engineered. After thinking very hard, one realizes that every term of `StateF` quotiented by `~` will start with a `get`, followed by a `set`, concluded with a `return`. We thus expect the following normal form:

```
State : Set → Set
State V = ΣGet (ΣSet V)
```

Unfolding the definition of `ΣGet` and `ΣSet`, we realize that this type is in fact isomorphic to `S → S × V`: we have recovered Haskell's `State` monad:

```
STATE : Set → Set
STATE V = S → S × V
```

It remains to substantiate this claim that *every* stateful program is equivalent to a `get` followed by a `set`. In the great tradition of constructive mathematics, we should do so computationally, thus inheriting a program computing these normal forms (also known as an evaluator) as well as a proof that this program is correct. We eschew to a technique called normalization-by-evaluation, with is spicy hot Curry-Howard in action.

Exercise (difficulty: 2) the first step is to interpret stateful terms into a suitable semantic domain which is **extensionally** quotiented by the theory of State:

```
eval : ∀{A} → StateF A → STATE A
eval = {!!}
```

This function should satisfy the following unit-proofs:

```
test-eval-get : ∀ {A} tt (k : S → StateF A) s →
            eval (get tt >>= k) s ≡ eval (k s) s
test-eval-get = {!!}

test-eval-set : ∀ {A} (k : ⊤ → StateF A) s s' →
            eval (set s' >>= k) s ≡ eval (k tt) s'
test-eval-set = {!!}
```

Exercise (difficulty: 1) the second step consists in *reifying* the semantic objects into the desired normal forms:

```
reify : ∀{A} → STATE A → State A
reify f = {!!}
```

The normalization procedure thus genuinely computes the normal form:

```
norm : ∀{A} → StateF A → State A
norm p = reify (eval p)
```

and these normal forms are indeed a subset of terms:

```
⌈_⌉ : ∀{A} → State A → StateF A
⌈ `get (tt , k) ⌉ = get tt >>= λ s → help (k s)
  where help : ∀ {A} → ΣSet A → StateF A
        help (`set (s , k)) = set s >>= λ _ → return (k tt)
```

Interpreting the statement *"for every stateful program, there exists a normal form"* constructively means that we have a procedure for computing this normal form. This is precisely the `norm` function.

3.1 Monads strike back

Looking closely at the `eval` function, we notice that we *map* syntactic objects – of type `StateF A` – to semantics objects – of type `STATE A`. The natural question to ask is whether all the structure defined over `StateF A` carries over to `STATE A`, ie. is there a semantical counterpart to `return`, `get`, `set` and `_>>=_`?

Exercise (difficult: 1) guided by `eval`, implement the semantical counterparts of `return`, `get` and `set`:

```
sem-return : ∀{A} → A → STATE A
sem-return a = {!!}

sem-get : ⊤ → STATE S
sem-get tt = {!!}

sem-set : S → STATE ⊤
sem-set s = {!!}
```

Unit-proof your definition with respect to their syntactic specifications:

```
test-sem-return : ∀ {X}{x : X} → eval (return x) ≡ sem-return x
test-sem-return = {!!}

test-sem-get : ∀{s} → eval (get tt) s ≡ sem-get tt s
test-sem-get = {!!}

test-sem-set : ∀{s s'} → eval (set s') s ≡ sem-set s' s
test-sem-set = {!!}
```

Exercise (difficulty: 2) similarly, there is a `_>>=_` over semantical states:

```
_sem->>=_ : ∀ {X Y} → (mx : STATE X)(k : X -> STATE Y) → STATE Y
_sem->>=_ mx k = {!!}
```

whose unit-proof is:

```
test-eval-compose : ∀ {X Y} (mx : StateF X)(k : X -> StateF Y) (s : S) →
  eval (mx >>= k) s ≡ (eval mx sem->>= λ x → eval (k x)) s
test-eval-compose = {!!}
```

In conclusion, we have been able to transport *all* the syntactic structure of `StateF X` to `STATE X`. In fact, we could be so bold as to directly work in `STATE X`, ignoring `StateF` altogether: this is what most purely functional programmers do currently.

3.2 Soundness & Completeness

Now, we must prove that a term thus computed is indeed a normal form. This is captured by two statement, a *soundness* result and a *completeness* result.

Exercise (difficulty: 4) at first, we assume the following two lemmas (whose proof is left as an exercise):

```
pf-sound : ∀{A} → (p : StateF A) → p ~ ⌈ norm p ⌉
pf-sound = {!!}

pf-complete : ∀ {A} {p q : StateF A} → p ~ q → ∀{s} → eval p s ≡ eval q s
pf-complete = {!!}
```

so as to focus on the overall architecture of the proof.

First, `norm` is sound: if two terms have the same normal form, they belong to the same congruence class:

```
sound : ∀ {V} (p q : StateF V) → ⌈ norm p ⌉ ≡ ⌈ norm q ⌉ → p ~ q
sound {V} p q r =
    begin
      p
     ≈⟨ pf-sound p ⟩
      ⌈ norm p ⌉
     ≡⟨ r ⟩
      ⌈ norm q ⌉
     ≈⟨ sym (pf-sound q) ⟩
      q
     ∎
       where open import Relation.Binary.EqReasoning (setoid V)
```

Second, `norm` is complete: if two terms belong to the same congruence class, they have the same normal form:

```
complete : ∀ {A} {p q : StateF A} → p ~ q → ⌈ norm p ⌉ ≡ ⌈ norm q ⌉
complete {p = p} {q} r =
    begin
      ⌈ norm p ⌉
    ≡⟨ refl ⟩
      ⌈ reify (eval p) ⌉
    ≡⟨ cong≡ (λ x → ⌈ reify x ⌉) (ext (λ z → pf-complete r)) ⟩
      ⌈ reify (eval q) ⌉
    ≡⟨ refl ⟩
      ⌈ norm q ⌉
    ∎
    where open ≡-Reasoning
          postulate ext : Extensionality Level.zero Level.zero
```

Note that this last proof needs functional extensionality (which, in Agda, is an axiom that does not compute). This is not a problem here since we are building a proof, whose computational content is void (it is entirely contained in the `norm` function).

3.3 Examples

From a programming perspective, `norm` gives us an interpreter for stateful computation, which is useful in and of itself: this is the foundation for effect handlers. The above proof establish the correctness of our definitions.

However, being in type theory, we can also consider the above proofs as providing us a reflexive decision procedure for equality of stateful programs. For instance we can "prove" (by a trivial reasoning) that our earlier programs `test0`, `test1` and `test2` are all equivalent:

```
test01 : test0 ~ test1
test01 = sound test0 test1 refl

test12 : test1 ~ test2
test12 = sound test1 test2 refl
```

The trick here is to rely on the soundness of normalization and compare the norm forms for (propositional!) equality. This proof technique is called proof by (computational) reflection and it is one of the workhorse of dependently-typed theory.

We can also do some abstract reasoning. For instance, we may be tempted to generalize the `cong` rule, which is restrictively right-leaning (we can only substitute for subterms `ps` and `qs` under a common `tm`) while one might want to have a more general version:

```
cong₂ : ∀{V W}(tm tm' : StateF W){ps qs : W → StateF V}  →
        (tm ~ tm') →
        (∀ w → ps w ~ qs w) →
        (tm >>= ps) ~ (tm' >>= qs)
cong₂ = {!!}
```

We prove this more general statement by working over the normal forms.

Exercise (difficulty: 3) we must first establish a technical lemma relating normalization with monadic composition:

```
norm-compose : ∀{V W}(tm : StateF W)(ps : W → StateF V) →
  ⌈ norm (tm >>= ps) ⌉ ≡ ⌈ norm (⌈ norm tm ⌉ >>= λ w → ⌈ norm (ps  w) ⌉) ⌉
norm-compose = {!!}
```

Exercise (difficulty: 2) Deduce the proof of generalized congruence `cong₂`.

CHAPTER

FOUR

APPLICATION: THE TICK MONAD

I have hinted at the fact that:

1. We could generalize much of the algebraic machinery (free monad, congruence, etc.), and
2. There is a general principle at play when going from signature & equations to some normal form representation

To provide another datapoint (from which to start generalizing), we now breeze through the `tick` monad. It is also sometimes called the complexity monad.

Let `M` be a monoid. We call `R` its carrier set:

```
module Tick (M : Monoid Level.zero Level.zero) where

  open Monoid M

  R : Set
  R = Carrier
```

4.1 Signature

The `Tick` monad has a single operation, `tick` which lets us add some amount `r : R` to a global accumulator:

```
data ΣTick (X : Set) : Set where
  `tick : R × (⊤ → X) → ΣTick X
```

4.2 Free term algebra

Exercise (difficulty: 2) Define the syntax for tickful programs using the free term algebra:

```
data TickF (V : Set) : Set where

tick : R → TickF ⊤
tick r = {!!}

mutual
  _>>=_ : ∀{A B} → TickF A → (A → TickF B) → TickF B
  s >>= mf = {!!}

  ΣTickmap : ∀{A B} → (A → TickF B) → ΣTick (TickF A) → ΣTick (TickF B)
  ΣTickmap mf s = {!!}
```

4.3 Equational theory

The equational theory, once again taken from Matija Pretnar's PhD thesis, is defined as follows:

```
data _⤳_ {V : Set} : TickF V → TickF V → Set where
  -- 1. Counting ϵ ticks amounts to doing nothing:
  tick-eps : ∀{k : TickF V} →
    (tick ϵ >>= λ _ → k) ⤳ k

  -- 2. Counting r₁ ticks followed by r₂ ticks amounts to counting
  --    r₁ • r₂ ticks:
  tick-com : ∀{k : TickF V}{r₁ r₂} →
    (tick r₁ >>= λ _ →
     tick r₂ >>= λ _ → k) ⤳ (tick (r₁ • r₂) >>= λ _ → k)
```

4.4 Normal forms

We realize that every 'TickF' program amounts to a single tick accumulating the sum of all sub-ticks:

```
Tick : Set → Set
Tick X = ΣTick X
```

This type being isomorphic to:

```
TICK : Set → Set
TICK A = R × A
```

Exercise (difficulty: 3) Establish this *a posteriori* by normalization-by-evaluation.

CHAPTER

FIVE

MORE MONADS

In the 2000's, inventing a new monad was a sure way to get an ICFP paper. As a result, whatever side-effects you are interesting in, there is (probably) a monad for that. Let's consider a few, focusing on the semantical presentations:

5.1 Exception/Error monad

We model exceptions (of type `E`) with the following monad:

```
Exn : Set → Set
Exn X = X ⊎ E
```

This monad provides two operations, one for raising an exception and another one for catching them (adopting an Exceptional Syntax):

```
raise : E → Exn ⊤
raise e = inj₂ e

_>>=[_|_] : ∀ {A B} → Exn A → (A → Exn B) → (E → Exn B) → Exn B
inj₁ x >>=[ k | ek ] = k x
inj₂ y >>=[ k | ek ] = ek y
```

Exercise (difficulty: 1) implement `return`, `_>>=_` and prove the monad laws.

Exercise (difficulty: 3) give an algebraic presentation of this monad, ignoring exception handling.

5.2 Reader/environment monad

The *Reader monad threads a read-only environment ‘‘Env‘*:

```
Reader : Set → Set
Reader X = Env → X
```

This environment may be `read` from or locally shadowed:

```
get-env : ⊤ → Reader Env
get-env tt = λ e → e

local : ∀{A} → Env → Reader A → Reader A
local e' f = λ e → f e'
```

Exercise (difficulty: 1) Implement `return`, `_>>=_` and prove the monad laws.

Exercise (difficulty: 3) give an algebraic presentation of this monad.

5.3 Counting/complexity monad

Specializing the `Tick` monad to the monoid `(ℕ, _+_, 0)`, we obtain a way to count the number of times a certain operation is performed at run-time:

```
Count : Set → Set
Count X = ℕ × X

count : ∀{X} → X → Count X
count x = 1 , x
```

Exercise (difficulty: 1) Implement `return`, `_>>=_` and prove the monad laws.

The `Count` monad can be used to **instrument** program, over which we can later perform formal reasoning to establish complexity results (see A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq).

Exercise (difficulty: 2) implement a function inserting an element `x` in a sorted list, counting the number of comparisons performed:

```
insert : ℕ → List ℕ → Count (List ℕ)
insert n l = {!!}
```

5.4 Writer/logging monad

Another interesting monoid structure is `(List Info, _++_, [])`, for which constructio the `Tick` monad essentially gives a model of `syslog`:

```
Log : Set → Set
Log X = List Info × X

log : Info → Log ⊤
log s = (s ∷ []) , tt
```

Exercise (difficulty: 1) Implement `return`, `_>>=_` and prove the monad laws.

5.5 Non-determinism monad

We can model non-deterministic choice (including failure) using the following monad:

```
Nondet : Set → Set
Nondet A = List A

fail : ∀{X} → Nondet X
fail = []

_|_ : ∀{X} → Nondet X → Nondet X → Nondet X
mx₁ | mx₂ = mx₁ ++ mx₂
```

Exercise (difficulty: 1) Implement `return`, `_>>=_` and prove the monad laws.

Exercise (difficulty: 3) give an algebraic presentation of this monad.

Exercise (difficulty: 2, courtesy of X. Leroy) insert an element `x` at a non-deterministic position of a list `l`:

```
insert : ∀{X} → X → List X → Nondet (List X)
insert x l = {!!}
```

Exercise (difficulty: 2, courtesy of X. Leroy) compute (non-deterministically) a permutation of a list `l`:

```
permut : ∀{X} → List X → Nondet (List X)
permut l = {!!}
```

5.6 Random monad

Some (many, in fact) monads have several equally valid semantics for a given signature, making yet another argument for following the algebraic approach. We would thus specify the `Random` monad through the following signature:

```
data ΣRand (X : Set) : Set where
  `rand : ℕ × (ℕ → X) → ΣRand X
  `choose : Float × (Bool → X) → ΣRand X

data RandF (V : Set) : Set where
  return : V → RandF V
  op : ΣRand (RandF V) → RandF V
```

The operation `rand` ought to return an integer uniformly distributed in `[0, n[`:

```
rand : ℕ → RandF ℕ
rand n = op (`rand (n , λ m → return m))
```

The operation `choose` runs `k₁` with probability `p` and `k₂` with probably `1 - p`:

```
choose : ∀{X} → Float → RandF X → RandF X → RandF X
choose p k₁ k₂ = op (`choose (p , λ { false → k₂ ; true → k₁ }))
```

Exercise (difficulty: 1) implement `return` and `_>>=_`.

Exercise (difficulty: 1) implement a dice of 6 sides:

```
dice : RandF ℕ
dice = {!!}
```

Exercise (difficulty: 1) compute the sum of 3 rolls of the dice defined above:

```
sum : RandF ℕ
sum = {!!}
```

One semantics for this monad is to run it with a particular pseudo-random number generator. This amounts to a state monad storing a particular seed and generating a new one every time `rand` is called. In OCaml (courtesy of X. Leroy), this amounts to:

```
module Random_Simulation = struct
  type α mon = int → α × int
  let ret a = fun s → (a, s)
  let bind m f = fun s → match m s with (x, s) → f x s
  let next_state s = s * 25173 + 1725
  let rand n = fun s → ((abs s) mod n, next_state s)
  let choose p a b = fun s →
    if float (abs s) <= p *. float max_int
    then a (next_state s) else b (next_state s)
end
```

However, this approach is a bit unsatisfactory since we hardcode a particular execution of the program, effectively determinizing it. Another (genuinely randomized) semantics consists in solely manipulating the probability distribution (see Probabilistic functional programming in Haskell but also Proofs of randomized algorithms in Coq):

```
Distr : Set → Set
Distr A = List (A × Float)

rand : ℕ → Distr ℕ
rand n = Data.List.map (λ k → (k , 1.0 / ( n ))) (downFrom n)

choose : ∀{X} → Float → Distr X → Distr X → Distr X
choose {X} p k₁ k₂ = prod p k₁ ++ prod (1.0 - p) k₂
  where prod : Float → Distr X → Distr X
        prod p xs = Data.List.map (λ { (k , pk) → (k , p * pk) }) xs
```

Exercise (difficulty: 1) Implement `return`, `_>>=_` and prove the monad laws.

5.7 CPS monad

We have seen a great deal of monads. A natural question to ask is whether there exists *"a mother of all monads"* (quoting Peter Hancock), that is a monad that would be sufficiently expressive to encode all the other ("reasonable") monads. The answer is an emphatic "yes", whose practical and theoretical implication is far beyond the scope of this single lecture. In the subsequent lectures, we shall come back to this fascinating observation. For now, we merely define the CPS monad, guided by the type of the CPS transform presented by François Pottier:

```
CPS : Set → Set
CPS X = (X → ⊥) → ⊥

call-cc : ∀{X} → ((X → ⊥) → CPS X) → CPS X
call-cc f = λ k → f k k

throw : ∀ {X} → (X → ⊥) → CPS X → CPS X
throw k' mx = λ k → mx k'
```

Exercise (difficulty: 2) Implement `return`, `_>>=_` and prove the monad laws.

Remark For those who cannot wait to encode their monads with CPS, one should look into Representing Monads (if you enjoy reading the Classics) and Kan Extensions for Program Optimisation (if you enjoy spicing things up with string diagrams).

CHAPTER

SIX

CONCLUSION

We have recovered the usual State (and Tick) monad from an algebraic presentation based on an equational theory. The key idea was to consider the equational theory as a rewriting system and look for its normal forms. We have justified this equivalence through a normalization-by-evaluation procedure, which we then abused to get proofs by reflection.

Exercises (difficulty: open ended):

1. Implement a generic "free monad construction", equipped with its operators (return, map, and bind).
2. Recast the State and Tick monads in that mold.
3. Implement another monad in that framework. Careful, you're probably thinking about the Exception monad with a `catch` operator: handling exceptions is not an algebraic effect, so it will not work. If you restrict yourself to `throw` (ignoring `catch`), that will work.

Going further:

- If you look up a category theory textbook (such as Categories for the Working Mathematician), the formal definition of a monad will differ from the one we have used: we have looked at monads through their associated Kleisli category. One should prove that both presentations are equivalent.
- I have left aside the question of *combining* theories: what about combining state and tick, for example? This has been studied categorically in Combining effects: sum and tensor (Plotkin, Power and Hyland, 2006), where it is shown that, in most cases, it all boils down to syntactically combine the signature (easy, always possible) and then deciding on a combined semantics (in a potentially non-trivial way). However, Tarmo Uustalu's work on Container Combinatorics seems to suggest that there is more to it than tensor and sums. We have deliberately ignored monad transformers, based on the (personal) opinion that they offer an unsatisfactory solution to the wrong problem.
- Algebraic effects do not capture all monads: the Exception monad (the one with a *throw and* a *catch*) is such a monad. Understanding effect handling and its relationship with delimited continuation (and, therefore, exceptions as a simpler case) is the topic of ongoing work, such as On the expressive power of user-defined effects (Pretnar et al., 2017).