Utrecht University

Master's Degree Thesis

Modular Semantics for Algebraic Effects

Niek Mulleners

Supervisor

dr. W.S. Swierstra

Second examiner

dr. M.I.L. Vákár

July 2020

Abstract

An important part of writing programs is to ensure that these programs behave as expected. Oftentimes, unit tests are written to show that parts of a program behave correctly in specific cases. Rather than resort to testing only some cases, we can use formal verification to prove that our program behaves correctly in all possible cases. Purely functional programming languages allow us to reason about programs as mathematical functions, but they do not contain side effects (impure operations). Using free monads, we can introduce the syntax of impure operations into a pure programming language, allowing one to syntactically define programs containing side effects. To execute and reason about such programs, we define semantics for the free monad by interpreting their impure operations within a target monad. By defining semantics in terms of monad transformers, combining them comes down to building up a monad transformer stack, and the choice of base monad gives rise to different kinds of semantics. This lets us write and reason about programs containing a variety of side effects in a modular way.

Contents

Chapter 1

Introduction

Almost any real-world application, whether written in an imperative or functional programming language, is bound to have side-effects, ranging from exceptions and mutable state to probabilistic choice and general recursion. Take, for example, the following function, that non-deterministically finds an element in a binary tree for which a predicate `p` holds. The function `amb` represents non-deterministic (ambivalent) choice between two arguments.

```
fun find(p, tree):
  if tree.isLeaf:
    return fail
  else:
    if p(tree.value):
      return tree.value
    else:
      return amb(find(p , tree.left), find(p , tree.right))
```

We would like to show that `find` behaves as expected. That is, it will only return values for which `p` holds, and, if the tree contains any values for which `p` holds, it will find (at least) one of them. Purely functional programming languages are well suited for this kind of formal verification, as they allow us to apply equational reasoning (Wadler [1987]), but, by definition, such languages do not contain side effects.

We can however try and model `find` in a pure language, by making the internal interpretation of ambivalent choice explicit. But what should this interpretation be? A computation of ambivalent choice might return a list of all possible outcomes, or use a random number generator to choose one outcome, or perhaps perform a parallel search. Rather than settle on one interpretation, we would like to allow the programmer to choose any interpretation as they see fit. To do this, we will separate the pure parts of our functions from the impure parts (the side effects), by defining a monad `Amb`, which syntactically extends otherwise pure code with the impure operations `amb` and `fail`.

We will model this in the dependent programming language Agda, in which we can both write programs and reason about their correctness. The rest of this thesis will be presented as a literate Agda file, ensuring that all code typechecks. Any pseudocode or code that does not typecheck will have a grey background to avoid confusion.

```
data Amb where
  pure : a → Amb a
  amb  : Amb a → Amb a → Amb a
  fail : Amb a
```

We define `find` in our pure setting as a function returning `Amb a`.

```
find : (a → Bool) → Tree a → Amb a
find P Leaf = fail
find P (Node l x r) = if P x then pure x
                      else amb (find P l) (find P r)
```

The programmer can then choose a handler function, which assigns an interpretation (semantics) to both the pure values and the impure operations. For example, we define a handler function `collect` that returns all possible outcomes in a list.

```
collect : Amb a → List a
collect (pure x)  = [ x ]
collect (amb a b) = collect a ++ collect b
collect fail      = []
```

Alternatively, we define a handler function `parallel` that performs a parallel search within the monad `Par`.

```
parallel : Amb a → Par a
parallel (pure x)  = return x
parallel (amb a b) = fork (parallel a) (parallel b)
parallel fail      = yield
```

1.1 Reasoning about effects

One way to give a specification for a value `v : t` is to define a predicate `P : t → Set` that should hold on `v`. The type `Amb a` represents an impure computation returning a value of type `a`. As such, we give a specification for the computation `c : Amb a` as a predicate `P : a → Set`. To show that `P` holds for `c`, we have to define what it means for a predicate to hold on a non-deterministic value. Following Swierstra and Baanen [2019], we will transform predicates on `a` to predicates on `Amb a` using predicate transformers of type `(a → Set) → (Amb a → Set)`. There are two canonical predicate transformers for ambivalent choice: `pt∀`, which requires the predicate to hold for every possible outcome (demonic non-determinism), and `pt∃`, which requires the predicate to hold on at least one possible outcome (angelic non-determinism).

```
pt∀ : (P : a → Set) → Amb a → Set
pt∀ P (pure x)  = P x
pt∀ P (amb a b) = pt∀ P a ∧ pt∀ P b
pt∀ P fail      = ⊤

pt∃ : (P : a → Set) → Amb a → Set
pt∃ P (pure x)  = P x
pt∃ P (amb a b) = pt∃ P a ∨ pt∃ P b
pt∃ P fail      = ⊥
```

We can use both `pt∀` and `pt∃` to show that `find` behaves as expected. We use the function `isTrue` to turn a predicate of type `a → Bool` into a predicate of type `a → Set`.

```
isTrue : Bool → Set
isTrue false = ⊥
isTrue true  = ⊤
```

The type of `find∀` states that, for every tree `t`, the predicate `P` holds for every possible result of `find P t`. It is easily proven by induction on the branches of `t`.

```
find∀ : (P : a → Bool) (t : Tree a)
  → pt∀ (isTrue ∘ P) (find P t)
```

Note that `find∀` alone is not sufficient to show that `find` behaves correctly. For example, `find∀` also trivially holds on the degenerate function `find'` which always fails.

```
find' : (a → Bool) → Tree a → Amb a
find' _ _ = fail

find∀' : (P : a → Bool) (t : Tree a)
  → pt∀ (isTrue ∘ P) (find' P t)
find∀' _ _ = tt
```

To show that `find` will actually find values for which `P` holds, provided they exist, we use the inductively defined relation `Exist`.

```
data Exist (P : a → Set) : Tree a → Set where
  here  : P x       → Exist P (Node l x r)
  left  : Exist P l → Exist P (Node l x r)
  right : Exist P r → Exist P (Node l x r)
```

The function `find∃` states that, for every tree `t`, if it contains any values for which `P` holds, there is at least one possible outcome of `find P t` for which `P` holds. The proof of `find∃` follows by induction on `Exist`.

```
find∃ : (P : a → Bool) (t : Tree a) → Exist (isTrue ∘ P) t
  → pt∃ (isTrue ∘ P) (find P t)
```

Together, `find∀` and `find∃` ensure that `find` is syntactically correct, but they do not prevent us from using a degenerate handler function like `collect'`, which defeats the purpose of `find∃`.

```
collect' : Amb a → List a
collect' _ = []
```

To show that the handler function `collect` respects the predicate transformers `pt∀` and `pt∃`, we relate them to the predicate transformers `All` and `Any` respectively, each of which have type `(a → Set) → (List a → Set)`. We say that `pt∀` is sound with respect to `collect` if, for every computation `c : Amb a` on which `P` holds according to `pt∀`, the predicate `P` also holds on all values returned by applying the handler `collect` to `c`. We show that both `pt∀` and `pt∃` are sound with respect to `collect` using some helper functions over `All` and `Any`.

```
sound∀ : ∀ P (c : Amb a) → pt∀ P c → All P (collect c)
sound∀ P (pure x) Px = Px :: []
sound∀ P (amb a b) (Pa , Pb) =
  All++ (sound∀ P a Pa) (sound∀ P b Pb)
sound∀ P fail tt = []

sound∃ : ∀ P (c : Amb a) → pt∃ P c → Any P (collect c)
sound∃ P (pure x) Px = here Px
sound∃ P (amb a b) (inj₁ Pa) = Any++ (sound∃ P a Pa)
sound∃ P (amb a b) (inj₂ Pb) = ++Any (sound∃ P b Pb)
```

1.2 Mutable state

A more common side effect in imperative programming is that of mutable state. Similarly to ambivalent choice, we will model mutable state using the impure operations `read` and `write`.

```
data St where
  pure : a → St s a
  read : (s → St s a) → St s a
  write : s → St s a → St s a
```

The more familiar functions `get` and `put` can be defined in terms of `read` and `write`, which we can then use to define stateful computations like `incr`.

```
get : St s s
get = read pure

put : s → St s 𝟙
put s = write s (pure tt)

incr : St ℕ 𝟙
incr = get >>= put ∘ suc
```

To execute a stateful computation, we use the handler function `run`, which maps a stateful computation to the state monad `s → a × s`.

```
run : St s a → s → a × s
run (pure x)    s = x , s
run (read k)    s = run (k s) s
run (write s k) _ = run k s
```

To give a specification for stateful computations, rather than just using a predicate on its result, we define a relation between the result and the initial and final state values. For example, the specification for `incr` ensures that the final state value is equal to the successor of the initial state value.

```
incrSpec : ℕ → a → ℕ → Set
incrSpec init result final = final ≡ suc (init)
```

Note how, given an initial state `n`, `incrSpec n` is a predicate on the output values of `incr`. To show that this predicate holds on a stateful computation, we define the predicate transformer `ptSt` that, given an initial state, transforms a predicate of type `(a → s → Set)` to a predicate of type `St s a → Set`.

```
ptSt : s → (P : a → s → Set) → St s a → Set
ptSt s P (pure x)    = P x s
ptSt s P (read k)    = ptSt s P (k s)
ptSt _ P (write s k) = ptSt s P k
```

We can then prove that `incr` adheres to the given specification.

```
incrCorrect : ∀ n → ptSt n (incrSpec n) incr
incrCorrect _ = refl
```

To prove that `ptSt` is sound with respect to `run`, we define the predicate transformer `StPT` which, given an initial state, transforms a predicate of type `a → s → Set` to a predicate over the state monad.

```
StPT : s → (a → s → Set) → ((s → a × s) → Set)
StPT s P x = uncurry P (x s)
```

```
soundSt : ∀ (i : s) (P : a → s → Set) (c : St s a)
  → ptSt i P c → StPT i P (run c)
soundSt _ P (pure x)    p = p
soundSt s P (read k)    p = soundSt _ P (k s) p
soundSt _ P (write s k) p = soundSt _ P k p
```

1.3 Research objectives

So far, we have seen how to syntactically formulate simple effects and how to define semantics for them in terms of handler functions and predicate transformers. Oftentimes, we want to write code containing a variety of different effects. When writing parser combinators, for example, we want our computations to be both stateful and non-deterministic. Syntactically, we can define the combination of two effects using coproducts in the style of Swierstra [2008]. Defining semantics for combinations of effects turns out to be less straightforward. The objective of this thesis is to extend and generalize the study of algebraic effects and semantics defined over them. In particular,

- We formalise effectful computations and semantics defined on them using algebraic effects and free monads (chapter 2).
- We adapt the techniques from Schrijvers et al. [2019] for defining modular semantics to use monad transformers (chapter 3) and generalise them to work on predicate transformers and other specificational semantics (chapter 5) while showing that they are monad homomorphisms (chapter 4).
- We define an alternative technique for combining continuation-style semantics (chapter 7).
- We show how the effect of general recursion can be combined with other effects in a modular fashion by adapting the petrol-driven semantics as defined by McBride [2015] to work for combinations of effects (chapter 8).
- We formalise the notion of soundness in terms of a refinement relation and lay the groundwork for proving the soundness of modular semantics (chapter 9).
- We show how predicate transformers give rise to Dijkstra monads (Ahman et al. [2017], Maillard et al. [2019]) and how handler functions give rise to morphisms between Dijkstra monads (section 9.3).

Chapter 2

Algebraic effects

So far, we have defined our effectful computations in terms of a set of characteristic operations. This approach to effectful computations is known as algebraic effects (Plotkin and Power [2002, 2003]). We can formulate an algebraic effect as a monad M, characterised by a set of algebraic operations, each of which is an n-ary function of the form

$$op : \forall a.(M a)^n \to M a$$

for which the algebraicity property holds:

$$op\ (x_1, \ldots, x_n) \ggg k = op\ (x_1 \ggg k, \ldots, x_n \ggg k) \tag{2.1}$$

For example, the effect of ambivalent choice, as we saw before, is characterised by the binary operation $amb : \forall a.\ M a \to M a \to M a$ and the nullary operation $fail : \forall a.\ M a$. It is easy to see that amb and $fail$ are isomorphic to the algebraic operations $branch : \forall a.\ (M a)^2 \to M a$ and $abort : \forall a.\ (M a)^0 \to M a$ respectively. The algebraicity property states that $branch(x, y) \ggg k = branch(x \ggg k, y \ggg k)$ and $abort() \ggg k = abort()$, i.e. that branching results in independent computations and that $abort$ short-circuits the computation.

The effect of mutable state, identified by the operations $read : \forall a.\ (M a)^s \to M a$ and $write : s \to M a \to M a$, also forms an algebraic effect. It is easy to see that $read$ has the correct form, but $write$ has an extra argument s. To alleviate this, for every value $x : s$, we introduce an operation $write_x : \forall a.\ M a \to M a$.

Rather than prove that the monad instances for ambivalent choice and stateful computations adhere to the algebraicity property, we will reformulate our effects such that they are algebraic by construction. Whereas op describes a single algebraic operation, we can use a dependent pair to model a set of algebraic operations as

$$ops : \forall a.\ \Sigma_{(c:C)} (M a)^{R_c} \to M a$$

where C represents the choice of operation and R_c the arity of the operation $c : C$. We call the combination of C and R_c the signature of an algebraic effect and define the corresponding effectful monad M, in the style of Hancock and Setzer [2000a,b], as

$$M = \lambda a.\ \mu x.\ a + \Sigma_{(c:C)} x^{R_c}$$

It is fairly straightforward to show that M is indeed a monad, by defining $\gg\!=$ in terms of the algebraicity property! In particular, M is the free monad on the container functor with shapes C and positions R_c. This shows us that there is a corresponding container functor for every algebraic effect and, vice-versa, that every container functor induces an algebraic effect.

Whereas Plotkin and Power [2002, 2003] specify the intended semantics of algebraic operations using equations, stating that, for example, $branch(x, abort())$ is equal to x, we will assign semantics to algebraic effects by implementing their algebraic operations in the target monad of our choice. For example, we can assign a semantics to ambivalent choice that collects all possible results in the list monad. We implement $branch : \forall a.\ List\ a \to List\ a \to List\ a$ as list concatenation and $abort : \forall a.\ List\ a$ as the empty list. In general, we can assign semantics to an algebraic effect in terms of a fold over the free monad:

$$fold : \forall a\ b.\ (gen : a \to b) \to (alg : \Sigma_{(c:C)} b^{R_c} \to b) \to M a \to b$$

For semantics returning a monad, we choose the generator function gen to be $return$, so that a semantics can be specified entirely in terms of the algebra alg.

2.1 Modeling algebraic effects

We formulate the signature of an effect as the `Sig` data type, which is equivalent to a container (Abbott et al. [2003, 2004]). It consists of a type of commands and an indexed type of responses to those commands. A signature can be constructed using ▷ (or ▶, if the responses are independent of the command).

```
record Sig where
  constructor _▷_
  field
    Cmd : Set
    Res : Cmd → Set

_▶_ : Set → Set → Sig
C ▶ R = C ▷ λ _ → R
```

The effect of ambivalent choice can be defined as having two commands, one of which, `branch`, has two possible continuations and the other, `abort`, has none.

```
data AmbCmd where
  branch abort : AmbCmd
```

```
Amb : Sig
Amb = AmbCmd ▷ λ where
  branch → 𝟚
  abort  → 𝟘
```

For stateful computations on a state of type `s`, the signature `St s` has a command `read` with `|s|` possible continuations and a family of commands $\texttt{write}_x$, each of which has only a single continuation.

```
data StCmd s where
  read : StCmd s
  write : s → StCmd s

St : Set → Sig
St s = StCmd s ▷ λ where
  read       → s
  (write s)  → 𝟙
```

Next, we model the free monad as being either a pure computation or a call to a command `c` (corresponding to an algebraic operation from the signature f) along with a continuation `k`, describing how to continue for each of the possible responses to that command.

```
data _⋆_ f a where
  pure : a → f ⋆ a
  call : (c : Cmd f) (k : Res f c → f ⋆ a) → f ⋆ a
```

The type f ⋆ `a` can be read as a computation that may call any number of operations from f before returning a value of type `a`.

2.2 Generic effects

The free monad defines the syntax of effects and allows us to write programs in terms of their algebraic operations. Like in chapter 1, however, we prefer to define our programs not in terms of the algebraic operations, but rather in terms of the more familiar smart constructors such as `amb`, `fail`, `get` and `put`.

```
amb : Amb ⋆ a → Amb ⋆ a → Amb ⋆ a
amb a b = call branch λ where
  false → a
  true  → b

fail : Amb ⋆ a
fail = call abort λ ()
```

```
get : St s ⋆ s
get = call read return

put : s → St s ⋆ 𝟙
put s = call (write s) return
```

These smart constructors are often referred to as generic effects.

2.3 Semantics

As mentioned before, we will assign semantics to an algebraic effect by assigning semantics to each of its algebraic operations. We do this by using an algebra on the corresponding signatures.

```
_-alg_ : Sig → Set → Set
(C ▷ R) -alg a = (c : C) (k : R c → a) → a
```

The fold over the free monad is defined in terms of an algebra on its signature and a generator function.

```
fold : (gen : a → b) (alg : f -alg b) → f ⋆ a → b
fold gen alg (pure x)   = gen x
fold gen alg (call c k) = alg c (fold gen alg ∘ k)
```

2.3.1 Handler functions

To implement the handler function `collect`, we define an algebra for ambivalent choice to the list monad.

```
collectAlg : Amb -alg List a
collectAlg branch k = k false ++ k true
collectAlg abort  k = []
```

We then apply the semantics defined in this algebra to the free monad using the `fold` function. We choose `return` as the generator function, to lift pure values to the target computational monad.

```
collect : Amb ⋆ a → List a
collect = fold return collectAlg
```

For stateful computations, we define the handler function `run` in terms of the algebra `stAlg`.

```
stAlg : St s -alg (s → t)
stAlg read      k s = k s s
stAlg (write s) k _ = k tt s

run : St s ⋆ a → s → a × s
run = fold return stAlg
```

In general, for a signature f and a monad `m` with algebra `alg :` f `-alg (m a)`, we define a handler function as follows:

```
handler : f ⋆ a → m a
handler = fold return alg
```

2.3.2 Predicate transformers

We can also define predicate transformers, like `pt∀`, as a fold over the free monad.

```
∀-alg : Amb -alg Set
∀-alg branch k = k false ∧ k true
∀-alg abort  k = ⊤

pt∀ : (a → Set) → Amb ⋆ a → Set
pt∀ P = fold P ∀-alg
```

Since the carrier type of our fold is `Set`, we can use the predicate `P : a → Set` as the generator function. Doing the same for stateful computations is not so straightforward, since we have a predicate of the form `P : a → s → Set`. To alleviate this, we rewrite `ptSt` using an isomorphic type signature:

```
ptSt : (a → s → Set) → St s ⋆ a → s → Set
```

This way, the carrier type of our fold is `s → Set`, allowing us to use the predicate of type `a → s → Set` as the generator. This conveniently also allows us to reuse `stAlg`, since we defined it generically over any carrier type of the form `s → t`.

```
ptSt P = fold P stAlg
```

This construction works in this specific case, but how do we define predicate transformer semantics in general?

2.3.3 Specificational semantics

An interesting observation is that we can reorder `pt∀` such that its return type is the continuation monad `(a → Set) → Set`. Not only can we now use `return` as the generator function, but the carrier type `(a → Set) → Set` allows us more freedom in the definition of our algebra, by giving access to the to be transformed predicate.

```
pt∀' : Amb ⋆ a → (a → Set) → Set
pt∀' = fold return λ where
  branch k → λ P → k false P ∧ k true P
  abort  k → λ P → 𝟙
```

This alternative interpretation makes clear that, whereas handler functions map an algebraic effect to a *computational* monad, our predicate transformers map an algebraic effect to a *specificational* monad. Therefore, we speak of handler functions as computational semantics and predicate transformers as specificational semantics. We have taken this idea from Maillard et al. [2019], which we will come back to in section 9.3.

In a similar fashion, the type of `ptSt` can be reordered to get the return type `s → (a × s → Set) → Set`. We might interpret this type as the continuation monad on `a × s`, given some initial state value. In fact, it is equal to the state monad transformer applied to the continuation monad, and thus a valid specificational monad!

```
ptSt' : St s ⋆ a → s → (a × s → Set) → Set
ptSt' = fold return stAlg
```

It is easy to show that both `pt∀'` and `ptSt'` are pointwise equivalent to `pt∀` and `ptSt` respectively.

```
∀≡ : ∀ (c : Amb ⋆ a) (P : a → Set)
  → pt∀' c P ≡ pt∀ P c

st≡ : (c : St s ⋆ a) (P : a × s → Set) (x : s)
  → ptSt' c x P ≡ ptSt (curry P) c x
```

In general, for a signature f and a monad transformer `t`, we define a semantics of the form f `⋆ a → t (cont Set) a` as `fold return alg`, where `alg` is an algebra of type f `-alg (t (cont Set) a)` and `cont Set` is the continuation monad on `Set`. We will see that both interpretations, that is, that of a predicate transformer and of a semantics to the transformed continuation monad, will prove useful.

2.4 Exceptions and modal operators

Another well-known algebraic effect is that of exceptional behaviour. We model it as having a command for every possible value of the exception type `e`, each of which has zero responses, accompanied with the generic effect `throw`.

```
Exc : Set → Sig
Exc e = e ▶ 𝟘

throw : e → Exc e ⋆ a
throw e = call e λ ()

sqrt : Float → Exc String ⋆ Float
sqrt f = if f < 0.0
         then throw "Error: square root of negative!"
         else return (primFloatSqrt f)
```

The handler `try` attempts to execute a computation, but fails when an exception is thrown.

```
try : Exc e ⋆ a → Maybe a
try = fold return λ _ _ → nothing
```

In most languages, exceptions show that the program has reached some state from which cannot be recovered while notifying the programmer what went wrong. Ideally, programs should never reach such a state. To encapsulate this, we can write the predicate transformer `ptSafe`, which guarantees that no exception is thrown. Alternatively, we might be fine with our code containing some exceptional behavior, as long as the result satisfies `P` when no exceptions are thrown. This behavior is captured in the predicate transformer `ptUnsafe`.

```
ptSafe : (a → Set) → Exc e ⋆ a → Set
ptSafe P = fold P λ _ _ → ⊥

ptUnsafe : (a → Set) → Exc e ⋆ a → Set
ptUnsafe P = fold P λ _ _ → ⊤
```

Note how this duality in interpretation is similar to the choice between `pt∀` and `pt∃` for ambivalent choice. This duality can be captured using the modal operators □ and ◇. The algebra `□-alg` states that a predicate should hold for every possible response, whereas `◇-alg` states that it should hold for at least one response.

```
□-alg ◇-alg : f -alg Set
□-alg c k = ∀ r → k r
◇-alg c k = ∃ r → k r

pt□ pt◇ : (a → Set) → f ⋆ a → Set
pt□ P = fold P □-alg
pt◇ P = fold P ◇-alg
```

It is straight-forward to show that `pt□` and `pt◇` are pointwise isomorphic to the respective predicate transformers `pt∀` and `pt∃`, and similarly `ptUnsafe` and `ptSafe`.

Chapter 3

Modular effects

Oftentimes, we would like to work with combinations of multiple different effects. Rather than define a new signature and new semantics specifically for every possible combination of effects, we would like to reuse the signatures and semantics of their constituent effects to assign meaning to combinations of effects.

3.1 Modular syntax

In the style of parser combinators (Hutton and Meijer [1996]), we will define a parser as a non-deterministic stateful computation with backtracking. We can define a signature for this effect as `St String ⊕ Amb`, using the `_⊕_` combinator. The resulting command is simply the coproduct of the commands from `St String` and `Amb`. The corresponding response is constructed by combining their respective responses using the `_▽_` combinator, which combines two dependent functions if their return type is dependent on the coproduct of their input types.

```
_▽_ : {r : a ⊎ b → Set}
    → ((x : a) → r (inj₁ x))
    → ((y : b) → r (inj₂ y))
    → (z : a ⊎ b) → r z
(f ▽ g) (inj₁ x) = f x
(f ▽ g) (inj₂ y) = g y

_⊕_ : Sig → Sig → Sig
(C₁ ▷ R₁) ⊕ (C₂ ▷ R₂) = (C₁ ⊎ C₂) ▷ (R₁ ▽ R₂)
```

This construction allows us to use $\texttt{inj}_1$ and $\texttt{inj}_2$ to access the algebraic operations of `St String` and `Amb` respectively. Unfortunately, however, we cannot define parser combinators in terms of the generic effects of `St` and `Amb`, because their types do not match the combined signature `St String ⊕ Amb`. To remedy this, we might redefine, for example, the generic effects `fail` and `get` for parsers as follows:

```
fail' : (St String ⊕ Amb) ⋆ a
fail' = call (inj₂ abort) λ ()

get' : (St String ⊕ Amb) ⋆ String
get' = call (inj₁ read) return
```

Not only does this approach require us to redefine our generic effects for every combination of effects, but we also have to redefine any helper functions. For example, we can define the function `guard` for ambivalent choice, which aborts a computation if a boolean value returns false.

```
guard : Bool → Amb ⋆ 𝟙
guard false = fail
guard true  = return tt
```

To use `guard` to define parser combinators, we would have to redefine it using `fail'`. A more scalable approach would be to define all effectful computations generically over any list of effects containing the required effects in the style of Baanen [2019], Baanen and Swierstra [2020]. To do so, we will formulate our modular effects in terms of a list of effects. Given a list of effects, we compute their coproduct using `foldr`, with the empty effect ø as the base case.

```
ø : Sig
ø = 𝟘 ▷ λ ()

∐ : List Sig → Sig
∐ = foldr _⊕_ ø
```

We can use the membership relation ∈ to formulate which effects are represented within a list of effects.

```
data _∈_ : a → List a → Set where
  here  : x ∈ (x ∷ xs)
  there : x ∈ xs → x ∈ (y ∷ xs)
```

Using this relation, generic effects can be formulated in terms of a list of effects containing the required signature. Rather than requiring the programmer to explicitly show that a list of effects contains a signature, we use Agda's instance arguments, represented by the double brackets, which can be inferred automatically, provided that there exists a unique value of that type.

```
fail⁺ : {{_ : Amb ∈ f⁺}} → (∐ f⁺) ⋆ a
amb⁺ : {{_ : Amb ∈ f⁺}} → (∐ f⁺) ⋆ a → (∐ f⁺) ⋆ a → (∐ f⁺) ⋆ a
get⁺ : {{_ : St s ∈ f⁺}} → (∐ f⁺) ⋆ s
put⁺ : {{_ : St s ∈ f⁺}} → s → (∐ f⁺) ⋆ 𝟙
```

But how do we implement these? Note how the combinations of injections required to define `fail'` and `get'` are determined by the positions of `Amb` and `St String` in `St String` ⊕ `Amb` ⊕ ø. We define the helper function `alg∈`, that looks up an algebra of type f within an algebra of type $\amalg\ f^+$, by induction on the membership relation.

```
alg∈ : {{_ : f ∈ f⁺}} → (∐ f⁺) -alg a → f -alg a
alg∈ {{here}}     alg = alg ∘ inj₁
alg∈ {{there f∈}} alg = alg∈ {{f∈}} (alg ∘ inj₂)
```

Using `alg∈`, we define `call⁺` as a modular alternative to `call`.

```
call⁺ : {{_ : f ∈ f⁺}} → f -alg (∐ f⁺ ⋆ a)
call⁺ = alg∈ call
```

The implementation of our generic effects now follows immediately from their original implementations, but using `call⁺` in place of `call`.

```
fail⁺ = call⁺ abort λ ()

amb⁺ a b = call⁺ branch λ where
  false → a
  true  → b

get⁺ = call⁺ read return

put⁺ s = call⁺ (write s) return
```

Using these modular generic effects, we can define effectful computations, like `guard`, generically over a list of effects containing the required signature.

```
guard⁺ : {{_ : Amb ∈ f⁺}} → Bool → (∐ f⁺) ⋆ 𝟙
guard⁺ false = fail⁺
guard⁺ true  = return tt
```

For convenience, the type `Parser⊆` f^+ captures that f^+ contains both `Amb` and `St String`. We define the classic parser combinator `token`, which reads characters from the state and matches them to the input string.

```
token : {{_ : Parser⊆ f⁺}} → String → (∐ f⁺) ⋆ 𝟙
token [] = return tt
token (x :: xs) = do
  y :: ys ← get⁺
    where [] → fail⁺
  guard⁺ (x == y)
  put⁺ ys
  token xs
```

In order to define a full-fledged parser library, we would like to define the classical parser combinators `_<$>_`, `_<*>_` and `_<|>_` and several variations on them, like `_<$_`, `_<*_` and `_*>_`. Most of these follow directly from the monad instance of the free monad, except for `_<|>_`, which we define simply as `amb⁺`.

```
_<|>_ : {{_ : Amb ∈ f⁺}} → Π f⁺ ⋆ a → Π f⁺ ⋆ a → Π f⁺ ⋆ a
_<|>_ = amb⁺
```

Using these parser combinators, we can define many parsers like `parseBool`.

```
parseBool : {{_ : Parser⊆ f⁺}} → (Π f⁺) ⋆ Bool
parseBool = token "true"  *> return true
        <|> token "false" *> return false
```

3.2 Modular handlers

To run a modular computation like `parseBool`, we would like to reuse the handler functions `collect` and `run`. Since we defined these handler functions in terms of algebras on their signatures, a straightforward approach to defining handlers for modular effects is to try and combine their algebras. Since algebras are essentially dependent functions, we can combine them using the `_▽_` combinator, but this requires them to have the same carrier type, i.e. it requires our handler functions to map to the same monad.

Schrijvers et al. [2019] show how we can define modular handlers by handling effects one at a time. The partial handler function $\texttt{partial}_a$ executes the effect of ambivalent choice, but keeps the other effect(s) intact, essentially assigning semantics to only the algebraic operations corresponding to `Amb`.

```
partial_a : (Amb ⊕ g) ⋆ a → g ⋆ List a
```

We can define $\texttt{partial}_a$ by defining algebras for `Amb` and g, both of which have g ⋆ `List a` as their carrier type, and then use `_▽_` to combine them.
To define the algebra on `Amb`, we will generalize `collectAlg` to be generic over any monad `m`.

```
alg_a : Amb -alg (m (List a))
alg_a branch k = do
  a ← k false
  b ← k true
  return (a ++ b)
alg_a abort  k = return []
```

We then have the freedom to choose a different monad for $\texttt{alg}_a$: we can choose the free monad over g to define $\texttt{partial}_a$, or, for example, choose the identity monad to recover `collectAlg`.

The algebra on g (called the forwarding algebra) is simply equal to `call`, the identity algebra, because the algebraic operations of g are defined polymorphically. For example, for g `= St s`, we can always update the state using $\texttt{write}_x$, for some `x : s`, regardless of whether our computation is of type `St s ⋆ a` or `St s ⋆ List a`, so the algebraic operation $\texttt{write}_x$ is mapped to itself.

```
fwd_a : g -alg (g ⋆ List a)
fwd_a = call
```

We can then implement $\texttt{partial}_a$ by combining these two algebras with the `_▽_` combinator and choosing `pure ∘ return` as the generator function.

```
gen_a : a → g ⋆ List a
gen_a = pure ∘ return

partial_a = fold gen_a (alg_a ▽ fwd_a)
```

A combined semantics `runCollect` can now be defined as simply the function composition of `run` and $\texttt{partial}_a$.

```
runCollect : (Amb ⊕ St s) ⋆ a → s → (List a) × s
runCollect = run ∘ partial_a
```

This pattern is easily expanded to any amount of effects, by applying partial handlers until only a single effect is left, to which we then apply the regular handler function. For lists of effects, as described in the previous section, we always handle the empty effect ∅ last using the handler function `escape`, which essentially escapes the monadic context.

```
escape : ∅ ⋆ a → a
escape = fold id λ ()
```

For a list of n effects with signatures f_x, each equipped with a partial handler function

```
partial_x : (f_x ⊕ g) ⋆ a → g ⋆ (m_x a)
partial_x = fold gen_x (alg_x ▽ fwd_x)
```

we define a modular handler function on the coproducts of these effects of the form

```
modular : (f_1 ⊕ ... ⊕ f_n ⊕ ∅) ⋆ a → (m_n ∘ ... ∘ m_1) a
modular = escape ∘ partial_n ∘ ... ∘ partial_1
```

Unfortunately, not all algebraic effects have a partial handler function of the form `(f ⊕ g) ⋆ a → g ⋆ (m a)`. For stateful computations, for example, we expect the partial handler function to accept an initial state value.

```
partial_s : (St s ⊕ g) ⋆ a → s → g ⋆ (a × s)
```

To accommodate this, we will interpret the target computational monad of an effect as $\mathtt{m}_x$ = $\varphi_x \circ \psi_x$, the functor composition of an outer functor φ_x and an inner functor ψ_x. In the case of mutable state, we choose `s →_` as the outer functor and `_× s` as the inner functor. For simple monads $\mathtt{m}_x$ (like `List` and `Maybe`), we choose φ_x = `id` and ψ_x = $\mathtt{m}_x$. We can now formulate the partial handler function of the corresponding effects as having the type $(f_x \oplus g) \star \mathtt{a} \rightarrow \varphi_x\ (g \star (\psi_x\ \mathtt{a}))$.

Using this alternative representation, $\mathtt{gen}_x$ and $\mathtt{fwd}_x$ can no longer be defined as simply `pure ∘ return` and `call` respectively. For stateful computations, for example, $\mathtt{gen}_s$ and $\mathtt{fwd}_s$ have to correctly pass the state value along.

```
gen_s : a → s → g ⋆ (a × s)
gen_s x s = pure (x , s)

fwd_s : g -alg (s → g ⋆ (a × s))
fwd_s c k s = call c λ r → k r s

partial_s = fold gen_s (stAlg ▽ fwd_s)
```

Furthermore, we can no longer simply compose partial handlers, since the result of a partial handler is not the free monad on g, but rather it is wrapped in the outer functor φ_x. To address this, we will introduce a finally construct $\mathtt{fin}_x : \varphi_x\ \mathtt{a} \rightarrow \mathtt{a}$, that strips away the outer functor φ_x. We will apply it to the result of $\mathtt{partial}_x$, such that the next partial handler can be applied. Using this finally construct, we can define a handler function for a list of effects with signature f_x of the form

```
modular : (f_1 ⊕ ... ⊕ f_n ⊕ ∅) ⋆ a → (ψ_n ∘ ... ∘ ψ_1) a
modular = escape ∘ fin_n ∘ partial_n ∘ ... ∘ fin_1 ∘ partial_1
```

For example, in the case of stateful computations, we implement $\mathtt{fin}_s$ by applying a value of type `s`.

```
fin_a : id a → a
fin_a = id

fin_s : s → (s → a) → a
fin_s s f = f s

runCollect' : s → (Amb ⊕ St s ⊕ ∅) ⋆ a → (List a) × s
runCollect' s = escape ∘ fin_s s ∘ partial_s ∘ fin_a ∘ partial_a
```

The only downside to this approach is that, whereas $\mathtt{fin}_a$ can be defined generically, $\mathtt{fin}_s$ requires an initial state value as argument, which we have to introduce manually. Note, however, how we can represent such an argument using the outer functor `s →_`. By wrapping a computation in the outer functor `s →_`, we essentially bring the finally construct $\mathtt{fin}_s$ into scope. We capture this behaviour by introducing an initialisation function $\mathtt{init}_s$ that separates the introduction of the argument `s` from its application.

```
initₛ : ((fin : ∀ {a} → (s → a) → a) → b) → s → b
initₛ f s = f (finₛ s)
```

We can trivially do the same for ambivalent choice.

```
initₐ : ((fin : ∀ {a} → id a → a) → b) → id b
initₐ x = x finₐ
```

This allows us to define `runCollect` without having to manually introduce `s` into scope.

```
runCollect'' : (Amb ⊕ St s ⊕ ∅) ⋆ a → s → (List a) × s
runCollect'' c = initₐ λ finₐ → initₛ λ finₛ →
  (escape ∘ finₛ ∘ partialₛ ∘ finₐ ∘ partialₐ) c
```

We can generalise initialisation functions for a functor φ as follows:

```
init : ((fin : ∀ {a} → φ a → a) → b) → φ b
```

To conclude, for a list of signatures f_x we define a modular handler of the form

```
modular : (f₁ ⊕ ... ⊕ fₙ ⊕ ∅) ⋆ a
        → (φ₁ ∘ ... ∘ φₙ ∘ ψₙ ∘ ... ∘ ψ₁) a
modular c = init₁ λ fin₁ → ... → initₙ λ finₙ →
  (escape ∘ finₙ ∘ partialₙ ∘ ... ∘ fin₁ ∘ partial₁) c
```

We can now define a handler function `runParser` using these techniques:

```
Parser : List Sig
Parser = St String :: Amb :: []

runParser : (Π Parser) ⋆ a → String → List (a × String)
runParser c = initₛ λ finₛ → initₐ λ finₐ →
  (escape ∘ finₐ ∘ partialₐ ∘ finₛ ∘ partialₛ) c
```

To use `runParser`, let us look at a simple parser `parseConjunction`, which parses two boolean values and returns their conjunction.

```
parseConjunction : {{_ : Parser⊆ f⁺}} → (Π f⁺) ⋆ Bool
parseConjunction = return conjunction <*> parseBool <*> parseBool
```

When we run `parseConjunction` by applying it to the string "truefalsetrue", the result is a single parse, returning the conjunction of `true` and `false` along with the remaining string "true".

```
> runParser parseConjunction "truefalsetrue"
[ false , "true" ]
```

If we run `parseConjunction` again on the remaining string "true", it cannot parse two boolean values and will return the empty list.

```
> runParser parseConjunction "true"
[]
```

Chapter 4

Monad transformers and homomorphisms

In the previous section, we have shown how to define handler functions for combinations of effects by handling effects one at a time. Being able to apply handler functions one after the other is one of the advantages of algebraic effects. For example, if we want to compose two computations with different effects, we can handle some of their effects until their signature matches. On the flip side, we have to be careful that our resulting computations make sense. Firstly, we want our final computation to be monadic. Secondly, if we compose multiple computations, it should not matter whether we compose them before or after applying handler functions. In this chapter, we will address these concerns and, in doing so, give a more succinct definition of modular handler functions.

4.1 Monad transformers

The order in which we handle effects determines the interpretation of the combined effect. For example, in the case of non-deterministic stateful computations, we have the choice between local versus global state. This choice is reminiscent of the different ways in which we can compose monad transformers. In fact, we can interpret the target monads of our handler functions as monad transformers to ensure that their composition is itself a monad!

We defined our handler functions to return some computational monad $m_x = \varphi_x \circ \psi_x$, such that the modular composition of n handler functions returns a functor $m' = \varphi_1 \circ \ldots \circ \varphi_n \circ \psi_n \circ \ldots \circ \psi_1$. We will define the composition of φ_x and ψ_x with a monad m as $t_x = \lambda\ m \to \varphi_x \circ m \circ \psi_x$, such that $m' = (t_1 \circ \ldots \circ t_n)\ id$. If we show that, for every effect, t_x is a monad transformer, m' is a monad transformer stack applied to the identity monad and therefore also a monad.

4.2 Monad homomorphisms

Not only do we want our semantics to return monads, we want them to be monad homomorphisms. A monad homomorphism is a function morph : $\forall a.\ m\ a \to n\ a$ between two monads m and n such that the following laws hold:

$$\text{morph}\ (\text{return}_m\ x) = \text{return}_n\ x \tag{4.1}$$

$$\text{morph} \circ (f \ggg_m g) = (\text{morph} \circ f) \ggg_n (\text{morph} \circ g) \tag{4.2}$$

where $\ggg$ represents Kleisli composition. Intuitively, these laws state that it should not matter whether we apply a handler function before or after composing two effectful computations. In the rest of this thesis, we will refer to monad homomorphisms as simply morphisms, unless we want to specifically mention that they do, in fact, abide by the previous laws.

As shown by McBride [2015], the monad homomorphism laws enforce that any monad homomorphism from the free monad on `C ▷ R` to a monad `m` is exactly given by `fold return (_>>=_ ∘ h)`, for some characteristic function `h` of type `(c : Cmd) → m (R c)`.
We will refer to `h` as a morphism of type `(C ▷ R) ⇒ m`.

```
_⇒_ : Sig → (Set → Set) → Set
(C ▷ R) ⇒ m = (c : C) → m (R c)
```

Handler functions from the free monad on f to a target monad `m` can then be defined in terms of a morphism of type f `⇒ m` using morphism application.

```
⟦_⟧ : f ⇒ m → f ⋆ a → m a
⟦ morph ⟧ = fold return (_>>=_ ∘ morph)
```

For example, we can define `collect` as the following morphism:

```
collect : Amb ⇒ List
collect branch = false :: true :: []
collect abort  = []
```

This definition might seem unintuitive. What does it mean that the `branch` case returns a list of booleans? To understand these morphisms better, let us reconstruct `collectAlg` from `collect`. The morphism application ⟦_⟧ is defined using the algebra `(_>>=_ ∘ morph)`. Note how `_>>=_` here is the bind operation of the target monad, which, in the case of the list monad, is defined as `flip concatMap`. As such, the resulting algebra for the `collect` handler is `λ c k → concatMap k (morph c)`. For the morphism `collect`, unfolding the definition of `concatMap` gives us the original definition of `collectAlg`.

```
collectAlg : Amb -alg List a
collectAlg branch k = k false ++ k true ++ []
collectAlg abort  k = []
```

In short, we supply the arguments to which the continuation `k` should be applied in such a way that the bind operation of the target monad can combine the results. This prevents us from defining nonsensical handler functions, like, for example, a variant of `collect` which combines `k false` and `k true` by interleaving their elements rather than concatenating them. Such a definition would not result in a valid monad homomorphism, unless, of course, we define our list monad using the same interleaving operation.
Similarly, we might define `pt∀` as a morphism to the continuation monad:

```
pt∀ : Amb ⇒ cont Set
pt∀ branch P = P false ∧ P true
pt∀ abort  P = ⊤
```

It is interesting to note that a morphism to the continuation monad is equal to `(c : C) → (R c → Set) → Set`. Apart from the dependency of `R` on `c : C`, we can reorder this as `(R → Set) → (C → Set)`, which reveals its predicate transformer nature.

4.3 Modular monad morphisms

Assuming that the modular handler construction in chapter 3 gives rise to correct monad morphisms, we should be able to define modular handler functions by composing their morphisms. To do this, we define the semantics for an effect in terms of a polymorphic morphism `morph` whose target monad is a monad transformer applied to a polymorphic base monad.

```
morph : {{M : Monad m}} → f ⇒ t m
```

The partial morphism `partial` is then computed by using `_▽_` to combine `morph` with a forwarding morphism, which lifts the identity morphism `idMorph` over the monad transformer `t`.

```
idMorph : g ⇒ (g ⋆_)
idMorph c = call c return

partial : (f ⊕ g) ⇒ t (g ⋆_)
partial = morph ▽ (lift ∘ idMorph)
```

We can easily define polymorphic morphisms for the effects we described so far.

```
morphCollect : Amb ⇒ listT m
morphCollect branch = return (false :: true :: [])
morphCollect abort  = return []

morphState : St s ⇒ stateT s m
morphState read      s = return (s , s)
morphState (write s) _ = return (tt , s)
```

```
morphTry : Exc e ⇒ maybeT m
morphTry e = return nothing
```

If the target monad of a monad morphism is again a free monad, we can compose it with another morphism using morphism application and function composition, for which we define the operator `_•_`:

```
_•_ : g ⇒ m → f ⇒ (g ⋆_) → f ⇒ m
morph₁ • morph₂ = ⟦ morph₁ ⟧ ∘ morph₂
```

Along with a morphism for the empty effect, we can define handlers for lists of effects, like, for example, a variant of `collect` defined over the singleton list.

```
escape : ∅ ⇒ id
escape ()

collect' : ∐ [ Amb ] ⇒ List
collect' = escape • partialCollect
```

Like we saw before, we have to do some more work for morphisms whose target monad is not simply the free monad. For example, for stateful computations, the target monad is wrapped in the outer functor `s →_`. We generalize this morphism composition to be polymorphic in the monad transformer `t` by lifting the applied morphism using the function `liftMorph`, which can be defined in terms of `init`.

```
liftMorph : (∀ {a} → m₁ a → m₂ a)
  → φ (m₁ (ψ a)) → φ (m₂ (ψ a))
liftMorph f x = init λ fin → f (fin x)

_•_ : g ⇒ m → f ⇒ t (g ⋆_) → f ⇒ t m
morph₁ • morph₂ = liftMorph ⟦ morph₁ ⟧ ∘ morph₂
```

Using this more general composition, we can define the same modular handlers as in section 3.2, but in a more succinct way, that is guaranteed to return monad homomorphisms. For example, we can now define `runParser` as

```
runParser : ∐ Parser ⇒ stateT String List
runParser = escape
          • partialCollect
          • partialState
```

Chapter 5

Modular predicate transformer semantics

Now that we know how to define modular handler functions, we would like to apply these techniques to predicate transformer semantics. In short, handler functions map an effectful computation to a computational monad, which is defined as a monad transformer applied to the identity monad. Our technique combines multiple effects by combining their respective monad transformers into a monad transformer stack and applying it to the identity monad. In this chapter, we show how we can adapt these techniques to work on predicate transformer semantics by choosing another base monad.

5.1 Specificational base monad

As shown in section 2.3.3, predicate transformer semantics map an effectful computation to a specificational monad, which is defined as a monad transformer applied to the continuation monad. To define modular predicate transformer semantics, we will also build up a monad transformer stack, but then apply it to the continuation monad instead of the identity monad. Firstly, we generalise escape to return any base monad.

```
escape : ∅ ⇒ m
escape ()
```

Secondly, we have to define our predicate transformer semantics as a monad morphism to a monad transformer applied to a polymorphic base monad. For stateful computations, this is trivial.

```
morphSt : St s ⇒ stateT s m
morphSt read      s = return (s , s)
morphSt (write s) _ = return (tt , s)
```

Unfortunately, however, we cannot define such a polymorphic monad morphism for many other predicate transformer semantics. For example, `pt∀` is defined by assigning `amb = _∧_` and `fail = ⊤`. The functions `_∧_` and `⊤` are defined in `Set`. In order to use them, we need access to the base monad `cont Set`.

Fortunately, for a single effect, there can be many different predicate transformer semantics. Ahman et al. [2017], Maillard et al. [2019] show that, for any computational monad defined as a monad transformer applied to the identity monad, there is an accompanying specificational monad defined as that monad transformer applied to the continuation monad. This implies that there is an alternative predicate transformer semantics `ptList`:

```
ptList : Amb ⇒ listT (cont Set)
```

We can define `ptList` to be polymorphic over the base monad. In fact, ignoring the base monad, `ptList` has the same type as `collect`. We can thus define `ptList` as `morphCollect`:

```
ptList = morphCollect
```

Originally, we used the predicate transformers `pt∀` and `pt∃` to apply a specification of type `a → Set` to a computation of ambivalent choice. This alternate predicate transformer `ptList`, however, does not accept a specification of type `a → Set`, but rather one of type `List a → Set`. To remedy this, we use the predicate transformers `All` and `Any` of type `(a → Set) → (List a → Set)` to recover the semantics of demonic and angelic non-determinism respectively.

```
ptAll ptAny : Amb ⇒ (cont Set)
ptAll c = ptList c ∘ All
ptAny c = ptList c ∘ Any
```

To define a modular predicate transformer, however, we can only apply `All` and `Any` after the composition. In a sense, we can use the predicate transformer `ptList` in a modular fashion by postponing the choice between demonic and angelic non-determinism. For example, we define a predicate transformer semantics for parsers without choosing the interpretation of the non-determinism, to which we then apply `All` to choose a demonic interpretation.

```
ptParser : ∐ Parser ⇒ (stateT String ∘ listT) (cont Set)
ptParser = escape • partialCollect • partialState

ptParserAll : ∐ Parser ⇒ stateT String (cont Set)
ptParserAll c s = ptParser c s ∘ All
```

Note, however, that this only works because we defined parser with local state, so that we can use `All` to transform a predicate of type `a × s → Set` to a predicate of type `List (a × s) → Set`. By combining ambivalent choice and mutable state in the reverse order, we get a global state. We define `ptGlobal` to be the predicate transformer semantics for non-deterministic computations with global state.

```
ptGlobal : (Amb ⊕ St s ⊕ ∅) ⇒ (listT ∘ stateT s) (cont Set)
ptGlobal = escape • partialState • partialCollect
```

Unlike `ptParser`, `ptGlobal` requires a predicate of type `(List a) × s → Set`, so we cannot give it a demonic interpretation by applying `All` to a predicate of type `a × s → Set`. For this purpose, we define a predicate transformer `globalAll`, which assigns a demonic interpretation to predicates of global state.

```
globalAll : (a × s → Set) → (List a) × s → Set
globalAll P (x , s) = All (λ y → P (y , s)) x

ptGlobalAll : (Amb ⊕ St s ⊕ ∅) ⇒ stateT s (cont Set)
ptGlobalAll c s = ptGlobal c s ∘ globalAll
```

5.2 Predicate transformer transformers

Rather than define `globalAll` (or `globalAny`) by hand, we would like to derive it automatically from `All` (or `Any`). Note how we can easily substitute `All` in the definition of `globalAll` with `Any`. In fact, we can generalise `globalAll` to accept any predicate transformer of type `(a → Set) → (b → Set)` and return a predicate transformer of type `(a × s → Set) → (b × s → Set)`. We will call this generalised function `pttSt` a predicate transformer transformer for stateful computations.

```
pttSt : ((a → Set) → b → Set) → (a × s → Set) → (b × s → Set)
pttSt pt P (x , s) = pt (λ y → P (y , s)) x
```

Rather than directly apply `pttSt` to `All`, we also define a predicate transformer transformer for demonic non-determinism.

```
pttDem : ((a → Set) → b → Set) → (a → Set) → (List b → Set)
pttDem pt P = All (pt P)
```

We can now define a combined predicate transformer transformer as the composition of `pttSt` and `pttAll`, which we will apply to the identity function to obtain the predicate transformer. Depending on the order in which we compose them, we can define a predicate transformer for both global and local state.

```
globalDemonic : (a × s → Set) → ((List a) × s → Set)
globalDemonic = (pttSt ∘ pttDem) id

localDemonic : (a × s → Set) → (List (a × s) → Set)
localDemonic = (pttDem ∘ pttSt) id
```

We might interpret these predicate transformer transformers as possibly assigning semantics to the inner functors of the respective monad transformers: `pttDem` assigns demonic semantics to the inner functor `List` of the list monad transformer, whereas `pttSt` leaves the inner functor `_× s` of the state monad transformer intact. In general, we define a predicate transformer transformer $\texttt{ptt}_x$ as follows, where ψ_x is the inner functor of the corresponding monad transformer and ψ'_x represents the possibly new interpretation of ψ_x.

```
pttₓ : ((a → Set) → b → Set) → (ψ'ₓ a → Set) → ψₓ b → Set
```

The predicate transformer for a list of effects is then defined as the function composition of all its predicate transformer transformers applied to the identity predicate transformer.

```
pt : ((ψ'ₙ ∘ ... ∘ ψ'₁) a → Set) → (ψₙ ∘ ... ∘ ψ₁) a → Set
pt = (pttₙ ∘ ... ∘ ptt₁) id
```

To apply such a predicate transformer to our semantics, we use the initialisation functions of φ_x. Note that, if we do not need access to every `fin` function separately, we can compose initialisation functions as follows:

```
composeInit : ((fin : ∀ {a} → φ (ψ a) → a) → b) → φ (ψ b)
composeInit f = init₁ λ fin₁ → init₂ λ fin₂ → f (fin₂ ∘ fin₁)
```

This allows us to use a single call to `init` to apply the predicate transformer `pt` to our predicate transformer semantics `pts` to get the final predicate transformer semantics `pts'`.

```
pts' : Π f⁺ ⇒ (φ₁ ∘ ... ∘ φₙ ∘ cont Set ∘ ψₙ ∘ ... ∘ ψ₁)
     → Π f⁺ ⇒ (φ₁ ∘ ... ∘ φₙ ∘ cont Set ∘ ψₙ' ∘ ... ∘ ψ₁')
pts' pts c = init λ fin → fin (pts c) ∘ pt
```

Note that we have to make sure that the resulting predicate transformer semantics is still monadic, by showing that every $t_x' = \lambda\ m \rightarrow \varphi_x \circ m \circ \psi_x'$ is still a monad transformer. In a sense, for every effect, we swap out its monad transformer for another more specific monad transformer. For effects such as `Amb` and `Exc`, the resulting monad transformer is simply the identity monad transformer.

5.3 Recap: combining algebraic effects

Syntactically, we combine algebraic effects by taking their coproduct (_⊕_). We give semantics for algebraic effects as monad homomorphisms. By defining these semantics in terms of monad transformers, we can compose partial semantics using morphism composition (`_•_`), effectively handling effects one at a time. This builds up a monad transformer stack, for which we are then free to choose any base monad, which determines the nature of the resulting semantics. Most notably, the identity monad gives rise to computational semantics, whereas the continuation monad on `Set` gives rise to specificational semantics. For a list of effects, a modular semantics is defined as

```
semantics : (f₁ ⊕ ... ⊕ fₙ ⊕ ∅) ⇒ (t₁ ∘ ... ∘ tₙ) m
semantics = escape • partialₙ • ... • partial₁
```

In the case of modular predicate transformer semantics, we postpone the exact choice of semantics (e.g. demonic versus angelic) until after the composition. For each effect, this choice is then made by choosing the right predicate transformer transformer.

```
pts : (f₁ ⊕ ... ⊕ fₙ ⊕ ∅) ⇒ (t₁' ∘ ... ∘ tₙ') (cont Set)
pts c = init λ fin → fin (semantics c) ∘ (pttₙ ∘ ... ∘ ptt₁) id
```

Chapter 6

Free monad transformers

6.1 Pretty-print semantics

Thus far, we have only concerned ourselves with the execution and verification of effectful computations. Sometimes the programmer would just like to gain some intuition about a part of a program. An important tool in this regard is that of a pretty-printer, a function that outputs a stylistic formatting of a value that is aimed at legibility for humans rather than compilers.

We would like to define a pretty-printer for algebraic effects of the form $f \star$ `a` $\rightarrow$ `String` that grants the programmer insight in the use of algebraic operations. For example, pretty-printing the value `amb (return true) fail` should return "branch(true,abort())". Evidently, to define this pretty-printer, we need to know how to print boolean values. Because an effectful computation can contain any type `a`, our pretty-printer requires an extra argument of type `a` $\rightarrow$ `String`. For example, we define a pretty-printer for ambivalent choice as:

```
ppAmb : Amb ⋆ a → (a → String) → String
ppAmb (pure x)        pp = pp x
ppAmb (call branch k) pp =
  "branch(" ++ ppAmb (k false) pp
     ++ "," ++ ppAmb (k true ) pp
     ++ ")"
ppAmb (call abort  k) pp = "abort()"
```

Note how the return type of `ppAmb` is actually the continuation monad on `String`. In fact, we can define `ppAmb` as a monad morphism.

```
ppAmb : Amb ⇒ cont String
ppAmb branch pp =
  "branch(" ++ pp false
     ++ "," ++ pp true
     ++ ")"
ppAmb abort  pp = "abort()"
```

We get the expected result by applying `ppAmb` to `amb (return true) fail` and providing a pretty-printer for booleans:

```
> ⟦ ppAmb ⟧ (amb (return true) fail) ppBool
"branch(true,abort())"
```

For stateful computations, we would like to be able to inspect the current state value whenever `read` is called as well as the final state value when the computation is finished. For example, when printing `incr` with an initial state of 4 we expect our pretty-printer to return "read()[4];write(5);tt[5]".

```
incr : St ℕ ⋆ 𝟙
incr = get >>= put ∘ suc
```

First of all, we require an extra argument of type `ppS : s → String`, for stateful type `s`. Note that, unlike the argument of type `a → String`, the function `ppS` is not polymorphic in `s` and is therefore not part of the return monad, but rather a parameter of the pretty-printer. Furthermore, we need to pass along the state value to get the correct results. Like with other semantics for mutable state, we use the state monad transformer to pass along the state.

```
ppSt : (s → String) → St s ⇒ stateT s (cont String)
ppSt ppS read      s pp = "read()[" ++ ppS s ++ "];" ++ pp (s , s)
ppSt ppS (write s) _ pp = "write(" ++ ppS s ++ ");" ++ pp (tt , s)
```

By applying the pretty-printer `pp𝟙×ℕ` we get the expected result:

```
pp𝟙×ℕ : 𝟙 × ℕ → String
pp𝟙×ℕ (t , n) = pp𝟙 t ++ "[" ++ ppℕ n ++ "]"
```

```
> ⟦ ppSt ppℕ ⟧ incr 4 pp𝟙×ℕ
"read()[4];write(5);tt[5]"
```

6.2 Modular pretty-printers

When trying to define pretty-printers for combinations of effects using the techniques described in the previous chapters, we run into the same problem as with `pt∀`. That is, we cannot define `ppAmb` and `ppSt` polymorphic in the base monad, because we need access to the `String` type, which is provided by the base monad `cont String`. The solution we used for `pt∀` was to postpone the choice of demonic non-determinism by using the alternate predicate transformer semantics `ptList` (defined using the list monad transformer) and, after composition, applying the predicate transformer `All` to choose a demonic interpretation. Unfortunately, this approach does not work for pretty-printers. For example, we might define the alternative pretty-printer semantics `ppList`:

```
ppList : Amb ⇒ listT (cont String)
ppList = morphCollect
```

Similar to how we used the predicate transformer `All` to give an interpretation to `ptList`, to recover the semantics of `ppAmb` from `ppList`, we require a pretty-print transformer of type `(a → String) → (List a → String)`. The problem here is that we cannot recover the used algebraic operators from `List`. For example, the list `x :: y :: []` might, among others, correspond to "branch(x, y)" or "branch(x, branch(abort(), y))".

6.3 Free monad transformers

By using a list monad transformer, we essentially flatten the tree-like structure of ambivalent choice, and erase all memory of the algebraic operations. A more fitting monad transformer for pretty-printing purposes is the free monad transformer with signature `Amb`.

```
freeT : Sig → (Set → Set) → Set → Set
freeT f m a = m (f ⋆ a)

ambT : (Set → Set) → Set → Set
ambT = freeT Amb
```

Before we can define a morphism for `Amb` in terms of `ambT`, we have to show that `ambT` is in fact a monad transformer. We will show that, for every signature f, `freeT` f is a monad transformer if f is a finitary container (Abbott et al. [2003]); i.e. each of its possible responses is isomorphic to a finite type.

```
finitary : Sig → Set
finitary f = ∀ c → ∃ n → Res f c ≅ Fin n
```

We will use the insight that, for any traversable monad `t`, `λ m a → m (t a)` is a monad transformer. That is, given a function `traverse` and a monad `m`, we implement `return` and `_>>=_` for the transformed monad `m ∘ t`.

```
traverse : (a → m b) → t a → m (t b)

trans : Monad (m ∘ t)
trans = record
  { return = return ∘ return
  ; _>>=_  = λ mx k → mx >>= λ x → join <$> traverse k x
  }
```

Furthermore, we define `lift : m a → m (t a)` by mapping `return` over `m`.

```
lift = λ x → return <$> x
```

As shown by Jaskelioff and O'Connor [2015], the extension of every finitary container is traversable. Although we cannot define `traverse` generically over any signature in Agda (without using reflection), the following construction shows how to define it for a specific finitary signature.

```
traverse : (a → m b) → f ⋆ a → m (f ⋆ b)
traverse f = fold (λ x → pure <$> f x) λ where
  c₀ k → do
    x₀ ← k 0
    ...
    xₙ ← k n
    return $ call c₀ λ where
      0 → x₀
      ...
      n → xₙ
...
```

For example, in the case of ambivalent choice:

```
traverseAmb : (a → m b) → Amb ⋆ a → m (Amb ⋆ b)
traverseAmb f = fold (λ x → pure <$> f x) λ where
  branch k → do
    x₀ ← k false
    x₁ ← k true
    return $ call branch λ where
      false → x₀
      true  → x₁
  abort k → return $ call abort λ ()
```

Now that we know that `ambT` is indeed a monad transformer, the corresponding morphism is defined by simply lifting the identity morphism `call c return`.

```
morphAmb : Amb ⇒ ambT m
morphAmb c = return (call c return)
```

In fact, we can generalise this definition to any free monad transformer.

```
morphT : f ⇒ freeT f m
morphT c = return (call c return)
```

We can define a pretty-print semantics for ambivalent choice in terms of `morphAmb` by choosing `cont String` as the base monad.

```
ppAmbT : Amb ⇒ ambT (cont String)
ppAmbT = morphAmb
```

Similarly to how we did for predicate transformer semantics in chapter 5, we apply a pretty-print transformer to `ppAmbT` to reclaim the behaviour of `ppAmb`. Interestingly, this pretty-print transformer is defined in terms of `ppAmb`.

```
ppT : (a → String) → (Amb ⋆ a → String)
ppT pp x = ⟦ ppAmb ⟧ x pp

ppAmb' : Amb ⇒ cont String
ppAmb' c = ppAmbT c ∘ ppT
```

Like in section 5.2, the pretty-printing of combinations of effects is done by building a monad transformer stack applied to the continuation monad on `String`, to which we then apply a pretty-print transformer, which is constructed as the composition of a list of pretty-print transformer transformers.

Unfortunately, this technique only works for effects that have finitary signatures. For stateful computations, this means the state type `s` should be finitary, which is not the case for many useful state types like $\mathbb{N}$ and `String`. For example, this would imply that we are not able to define pretty-print semantics for parsers in this way. It is, however, straightforward to define such a pretty-printer by hand.

The problem of defining `ppParser` in a modular way does not lie with its complexity, but rather with the limitations of our approach for defining modular semantics, which relies on all our semantics being defined in terms of monad transformers. In the next chapter, we will introduce a different approach for defining modular semantics without building up a monad transformer stack, and show how we can use it to implement `ppParser`.

6.4 Back to predicate transformer semantics

Despite our inability to define `ppParser` in a modular fashion, this does not mean that free monad transformers are not useful. In the case of predicate transformer semantics, for example, we can use the free monad transformer where possible. For example, for ambivalent choice, we might use its free monad transformer instead of the list monad transformer to define a modular predicate transformer semantics.

```
ptAmb : Amb ⇒ ambT (cont Set)
ptAmb = morphAmb
```

The demonic predicate transformer transformer can then be defined in terms of `pt∀`!

```
pttDem : ((a → Set) → b → Set) → (a → Set) → Amb ⋆ b → Set
pttDem pt P x = ⟦ pt∀ ⟧ x (pt P)
```

This approach has several advantages over using `ptList`. Firstly, `ptList` assumes that we will use `collect` semantics to execute our computation of ambivalent choice, whereas `ptAmb` does not discriminate between different computational semantics, like parallel search, random choice, etc. Secondly, we are no longer reliant on predicate transformers like `All` and `Any` to choose between

different interpretations. Instead, we can use our previously defined predicate transformers like `pt∀` and `pt∃`. Even better, we can use the more general modal predicate transformers `pt□` and `pt◇`.
We can, for example, redefine `ptParser` using the free monad transformer on `Amb` rather than the list monad transformer.

```
ptParser : Π Parser ⇒ (stateT String ∘ ambT) (cont Set)
ptParser = escape • partialAmb • partialState
```

Instead of the demonic predicate transformer transformer `pttDem`, we can use the more general modal predicate transformer transformer `ptt□`, which is defined in terms of `pt□`, to assign a demonic semantics to `ptParser`.

```
ptt□ : ((a → Set) → b → Set) → (a → Set) → f ⋆ b → Set
ptt□ pt P x = ⟦ pt□ ⟧ x (pt P)

ptParser□ : Π Parser ⇒ stateT String (cont Set)
ptParser□ c = init λ fin → fin (ptParser c) ∘ (ptt□ ∘ pttSt) id
```

Chapter 7

Continuation monad transformer

As we have seen multiple times, when we use the continuation monad as the base monad for our semantics, we often need access to the return type of that continuation monad to define these semantics. For example, `pt∀` needs access to the return type `Set` and `ppAmb` needs access to the return type `String`. This prevented us from defining `pt∀` and `ppAmb` polymorphic in their base monad. The solution to this problem was to postpone the actual logic (the choice of a demonic interpretation for `pt∀` and the pretty-printing for `ppAmb`) until after building the monad transformer stack. For some semantics, however, like `ppSt`, there exists no suitable monad transformer.
In this chapter, we will briefly explore an alternative way of composing semantics using the continuation monad transformer.

7.1 Algebras as morphisms

In chapter 2, before we interpreted specificational semantics as monad transformers applied to the continuation monad, we defined `pt∀` in terms of an algebra on `Set`. Upon further inspection, the type f `-alg Set` is equal to `(c : C) (k : R c → Set) → Set`, which is exactly a monad morphism to the continuation monad on `Set`! In general, f `-alg r` is equal to $f \Rightarrow$ `cont r`.

We defined `pt∀` in terms of a fold, with the predicate `P : a → Set` as the generator and `∀-alg` as the algebra. Using the knowledge that `∀-alg` is a monad homomorphism, we can redefine `pt∀` using morphism application.

```
pt∀ : Amb ⋆ a → (a → Set) → Set
pt∀ = ⟦ ∀-alg ⟧
```

Assuming the extensionality of functions, we can prove, by induction on the free monad, that these ways of constructing `pt∀` from `∀-alg` are equivalent.

```
prf : ∀ (P : a → r) (x : f ⋆ a) (m : f ⇒ cont r)
  → fold P m x ≡ ⟦ m ⟧ x P
```

Similarly, the algebra `stAlg` has type `St s -alg (s → Set)`, which is equal to `St s ⇒ cont (s → Set)`. Rather than define specificational semantics as some monad transformer applied to the continuation monad, we can define them as the continuation monad transformer `contT` applied to some base monad!

```
contT : Set → (Set → Set) → Set → Set
contT r m a = (a → m r) → m r
```

For example, the specificational monad for ambivalent choice is defined as the continuation monad transformer on `Set` applied to the identity monad and the specificational monad for stateful computations is defined as the continuation monad transformer on `Set` applied to the reader monad `s →_`.

In the case of ambivalent choice, it is easy to see that both interpretations are equivalent. That is, both `contT r id` and `idT (cont r)` are equal to `cont r`. In the case of stateful computations, these interpretations are not equivalent, since `stateT s (cont r)` expands to `λ a → s → (a × s → r) → r` and `contT r (s →_)` expands to `λ a → (a → s → r) → s → r`. They are, however, isomorphic, by swapping their arguments and currying/uncurrying the predicate.

7.2 Modular continuation-style semantics

To be able to combine semantics defined in terms of the continuation monad transformer applied to some base monad, we have to be able to combine their base monads. To do so, we require that these base monads are defined in terms of their initialisation function. First, we show that any functor φ with an initialisation function is, in fact, a monad.

```
InitM : Monad φ
InitM = record
  { return = λ x → init λ _ → x
  ; _>>=_  = λ x k → init λ fin → fin (k (fin x))
  }
```

As shown in section 5.2, we can compose initialisation functions. As such, the composition of two monads with initialisation functions is also a monad! We will define the combinator `_▼_`, which combines two continuation-style semantics if their base monads have initialisation functions.

```
_▼_ : f ⇒ contT r φ → g ⇒ contT r ψ
    → (f ⊕ g) ⇒ contT r (φ ∘ ψ)
```

To do so, we define two functions $\texttt{lift}_1$ and $\texttt{lift}_2$, each of which lifts continuation-style morphisms by composing their base monad with another monad equipped with an initialisation function.

```
lift₁ : f ⇒ contT r φ → f ⇒ contT r (φ ∘ ψ)
lift₁ m c k =
  init₁ λ fin₁ →
  init₂ λ fin₂ →
    fin₁ (m c (map fin₂ ∘ k))

lift₂ : f ⇒ contT r ψ → f ⇒ contT r (φ ∘ ψ)
lift₂ m c k =
  init₁ λ fin₁ →
  init₂ λ fin₂ →
    fin₂ (m c (fin₁ ∘ k))
```

The combinator `_▼_` is then defined by combining both morphisms using `_▽_` after lifting them to the same return type.

```
(f ▼ g) = lift₁ f ▽ lift₂ g
```

Note how, unlike the techniques described in previous chapters, this technique does not handle effects one by one, but rather combines all handlers in one go.

7.3 Modular pretty-printers

To give a correct definition of `ppParser`, we first use the isomorphism between `contT r (s →_)` and `stateT s (cont r)` to redefine `ppSt` in terms of the continuation monad transformer.

```
ppSt' : (s → String) → St s ⇒ contT String (s →_)
ppSt' ppS c pp s = ppSt ppS c s (uncurry pp)

ppString : String → String
ppString s = "’" ++ s ++ "’"
```

We can then define `ppParser` simply by composing the semantics for mutable state and ambivalent choice using the `_▼_` combinator.

```
ppParser : ⅠⅠ Parser ⇒ contT String (String →_)
ppParser = ppSt' ppString ▼ ppAmb ▼ escape
```

To see `ppParser` in action, let us look at a simple parser `parseBit`, which reads a single bit and returns it as a natural number.

```
parseBit : {{_ : Parser⊆ f⁺}} → (ⅠⅠ f⁺) ⋆ ℕ
parseBit = token "0" *> return 0 <|> token "1" *> return 1
```

We provide a pretty-printer that prints the resulting number along with the final state.

```
ppℕ×String : ℕ → String → String
ppℕ×String n s = ppℕ n ++ "[" ++ ppString s ++ "]"
```

By applying this parser to the input state "1", we can see how the computation branches and then, after reading the input state, aborts the first branch and updates the state in the second branch before returning `1`.

```
> ⟦ ppParser ⟧ parseBit ppℕ×String "1"
"branch(read()[''1''];abort(),read()[''1''];write('''');1['''']))"
```

Chapter 8

General recursion

An important feature of many programming languages is that of recursion, allowing the programmer to write functions in terms of themselves. It is not always clear whether a recursive function will terminate or get stuck in an infinite loop, making recursive functions difficult to reason about. Take, for example, the function `quickSort`, a classic example of the elegance and expressivity of functional programming languages:

```
quickSort : List ℕ → List ℕ
quickSort [] = []
quickSort (x :: xs) =
  let smaller = quickSort (filter (_≤ x) xs)
      greater = quickSort (filter (_> x) xs)
  in smaller ++ [ x ] ++ greater
```

As elegant as this definition is, Agda cannot infer that `filter (_≤ x) xs` and `filter (_> x) xs` are smaller than `x :: xs`, so `quickSort` will not pass the termination checker. To ensure that only total functions can be defined, Agda allows only structural recursion. That is, at least one argument to the recursive call has to be a strict subexpression of the corresponding argument to the main function. For example, we can define `plus` recursively by stripping away the `suc` constructor in the recursive call.

```
plus : ℕ → ℕ → ℕ
plus zero    m = m
plus (suc n) m = suc (plus n m)
```

8.1 Well-founded recursion

To prove to Agda that `quickSort` terminates, we have to define it using structural recursion, while proving that the recursive calls are on strictly smaller lists. We will use a technique called well-founded recursion, where the required proof is passed along in an inductively defined data type such that the recursive step depends on a structurally smaller proof. The type `Acc` denotes whether a value `x : a` is *accessible* with respect to a relation `_<_ : a → a → Set`, meaning that every value less than `x` (according to `_<_`) is also accessible. For example, `zero : ℕ` is accessible with respect to `_<_` on natural numbers because there are no natural numbers less than `zero`. Every other natural number is accessible by induction.

```
data Acc (_<_ : a → a → Set) (x : a) : Set where
  acc : (∀ {y} → y < x → Acc _<_ y) → Acc _<_ x
```

To use it, we add the accessibility predicate as an extra argument to our function. At the problematic recursive step, we use structural recursion by stripping away one `acc` constructor and applying the proof that `y` is smaller than `x` to obtain the accessibility predicate for `y`. In the case of `quickSort`, we add an argument stating that the length of the input list is accessible and apply the proof that, for every predicate `p`, the length of `filter p xs` is less than the length of `x :: xs`.

```
quickSort : (xs : List ℕ) → Acc _<_ (length xs) → List ℕ
quickSort [] _ = []
quickSort (x :: xs) (acc f) =
  let smaller = quickSort (filter (_≤ x) xs) (f (filterProof x xs))
      greater = quickSort (filter (_> x) xs) (f (filterProof x xs))
  in smaller ++ [ x ] ++ greater
```

Unfortunately, this definition is not as elegant anymore, and it gets only worse for more complex functions that require large termination proofs. Ideally, we would like to define `quickSort` syntactically and only worry about the termination upon execution. Additionally, there are many more techniques for proving termination other than well-founded recursion. The choice of technique should not influence the syntax of our functions, but only the semantics.

8.2 General recursion

McBride [2015] shows how we can define general recursion as an algebraic effect, where a recursive call to a function of type `I → O` is represented by a call to a command of type `I` with a response of type `O`. A generally recursive function of type `I → O` can be defined as a Kleisli arrow on the free monad with signature `I ▶ O`. For convenience, we will write `I ⇀ O` to denote such a function.

```
_⇀_ : Set → Set → Set
I ⇀ O = I → (I ▶ O) ⋆ O
```

Now, rather than explicitly call a function recursively, we can call the generic effect `recurse`.

```
recurse : I ⇀ O
recurse i = call i return
```

This allows us to define `quickSort` syntactically without requiring any termination proofs.

```
quickSort : List ℕ ↬ List ℕ
quickSort [] = return []
quickSort (x :: xs) = do
  smaller ← recurse (filter (_≤ x) xs)
  greater ← recurse (filter (_> x) xs)
  return (smaller ++ [ x ] ++ greater)
```

Note that this definition is not actually recursive, but rather describes the recursive structure. In order to execute it, we require a handler function that gives semantics to these recursive calls in a way that complies with the termination checker. There exist many different semantics for general recursion, like, for example, a semantics based on invariants (Swierstra [2008], Baanen and Swierstra [2020], Baanen [2019]). We will, however, restrict ourselves to the petrol-driven semantics as defined by McBride [2015], which assigns semantics to recursive functions by choosing a maximum recursion depth.

8.2.1 Petrol-driven semantics

Note how the type `I ↬ O` is equal to `(I ▶ O) ⇒ ((I ▶ O) ⋆_)`, a morphism for general recursion. This morphism describes exactly how to unroll the function `f` once, by inlining `f` at the recursive call. This allows us to use morphism composition to unroll `quickSort` a number of times. For example:

```
quickSort₃ : List ℕ ↬ List ℕ
quickSort₃ = quickSort • quickSort • quickSort
```

The morphism `abandon` throws an exception at all remaining recursive calls, essentially giving up after the maximum recursion depth is reached.

```
abandon : (I ▶ O) ⇒ (Exc String ⋆_)
abandon _ = throw "max recursion depth reached"
```

To set the maximum recursion depth for `quickSort`, we abandon after a number of unrolls.

```
runQuickSort₃ : (List ℕ ▶ List ℕ) ⇒ (Exc String ⋆_)
runQuickSort₃ = abandon • quickSort • quickSort • quickSort
```

Using a fold function over natural numbers, we define the petrol-driven semantics `petrol`, which unrolls a computation `n` times before abandoning.

```
foldℕ : a → (a → a) → ℕ → a
foldℕ z s zero    = z
foldℕ z s (suc n) = s (foldℕ z s n)

petrol : (I ↬ O) → ℕ → (I ▶ O) ⇒ (Exc String ⋆_)
petrol unroll = foldℕ abandon (_• unroll)
```

Note that the resulting morphism has type `I → Exc String ⋆ O`. To run a generally recursive function `f`, we simply apply the input of type `I` to `petrol f n`, after which we apply a handler function for `Exc String`, like `try`.

```
runQuickSort : ℕ → List ℕ → Maybe (List ℕ)
runQuickSort n = try • petrol quickSort n
```

We can use `runQuickSort` to sort, for example, the list `4 :: 2 :: 3 :: []` with a recursive depth of 4.

```
> runQuickSort 4 (4 :: 2 :: 3 :: [])
just (2 :: 3 :: 4 :: [])
```

If we use a smaller recursive depth, it will abandon the computation before reaching a solution and return nothing.

```
> runQuickSort 3 (4 :: 2 :: 3 :: [])
nothing
```

8.2.2 Petrol-driven predicate transformers

To reason about the correctness of a generally recursive function, in addition to showing that the result of executing that function adheres to some specified postcondition, we also have to make sure that the function terminates. For petrol-driven semantics, this comes down to showing that, for every possible input, there exists a large enough maximum recursion depth. We compose our petrol-driven semantics with the modal predicate transormer `pt◇`, which states that, after unrolling `n` times, there should be at least one result adhering to the postcondition.

```
ptQuickSort : ℕ → (List ℕ ▶ List ℕ) ⇒ cont Set
ptQuickSort n = pt◇ • petrol quickSort n
```

To define a postcondition for `quickSort`, we define a relation `sorted`, where `sorted xs ys` denotes that `ys` is the result of sorting `xs`.

```
sorted : List ℕ → List ℕ → Set
sorted xs ys = permutes xs ys ∧ ascending ys
```

In the case of `quickSort`, the worst possible input is a list `xs` that is sorted in descending order, requiring a maximum recursion depth of at least `1 + length xs`. The proposition `quickSortCorrect` then states that applying `quickSort` to the list `xs` is guaranteed to give a correct result if we choose a maximum recursion depth of `1 + length xs`.

```
quickSortCorrect : (xs : List ℕ)
  → ptQuickSort (1 + length xs) xs (sorted xs)
```

8.3 Modular general recursion

The efficiency of `quickSort` depends on which pivot is chosen from the input list. Rather than always choose the first element from the list, we might choose the pivot at random. This way, we can reason about the efficiency of `quickSort` independent of the order of the input list. We will model this random choice using the effect of ambivalent choice. First, we define the function `pick`, which non-deterministically picks an element from a list.

```
pick : {{_ : Amb ∈ f⁺}} → List a → Π f⁺ ⋆ (a × List a)
pick [] = fail⁺
pick (x :: xs) = return (x , xs) <|> do
  y , ys ← pick xs
  return (y , x :: ys)
```

For convenience, we write `QuickSort` to denote the list of signatures containing both `Amb` and `(List ℕ ▶ List ℕ)` and write `QuickSort`$\subseteq f^+$ to show that these signatures are contained in f^+. Using `pick`, along with the modular generic effect `recurse⁺`, we define a non-deterministic variant of `quickSort`:

```
quickSort⁺ : {{_ : QuickSort⊆ f⁺}}
  → List ℕ → Π f⁺ ⋆ List ℕ
quickSort⁺ [] = pure []
quickSort⁺ xs = do
  y , ys ← pick xs
  smaller ← recurse⁺ (filter (_≤ y) ys)
  greater ← recurse⁺ (filter (_> y) ys)
  return (smaller ++ [ y ] ++ greater)
```

In order to define petrol-driven semantics for combinations of effects, we adapt `abandon` to work on lists of effects, effectively replacing the effect of general recursion with the effect of exceptions.

```
abandon⁺ : Π (I ▶ O :: f⁺) ⇒ (Π (Exc String :: f⁺) ⋆_)
abandon⁺ (inj₁ _) = throw⁺ "max recursion depth reached"
abandon⁺ (inj₂ c) = call (inj₂ c) return
```

The modular petrol-driven semantics `petrol⁺` are then defined by abandoning after unrolling the computation `n` times, all the while forwarding the other algebraic operations.

```
fwd : Π f⁺ ⇒ (Π (I ▶ O :: f⁺) ⋆_)
fwd c = call (inj₂ c) return

petrol⁺ : (I → Π (I ▶ O :: f⁺) ⋆ O) → ℕ
  → Π (I ▶ O :: f⁺) ⇒ (Π (Exc String :: f⁺) ⋆_)
petrol⁺ unroll = foldℕ abandon⁺ (_• unroll ▽ fwd)
```

Using morphism composition, we can first unroll the computation `n` times using `petrol⁺`, after which we simply use the modular semantics as defined in chapter 4. Note that we have to apply the petrol-driven semantics before handling any other effects, since handling an effect will change the return type of our generally recursive function, which means that it is no longer a correct morphism and cannot be used to unroll the computation. We will use `collect` semantics (rather than, for example, random choice) so that we can better inspect the different possible results.

```
runQuickSort⁺ : ℕ → ∐ QuickSort ⇒ (List ∘ Maybe)
runQuickSort⁺ n = escape
                ● partialCollect
                ● partialTry
                ● petrol⁺ quickSort⁺ n
```

If we use the non-deterministic `quickSort⁺` to sort the same list as before[1], we can see that, as expected, every possible combination of pivots gives the same result.

```
> runQuickSort⁺ 5 (inj₁ (4 ∷ 2 ∷ 3 ∷ []))
just (2 ∷ 3 ∷ 4 ∷ []) ∷ just (2 ∷ 3 ∷ 4 ∷ []) ∷
just (2 ∷ 3 ∷ 4 ∷ []) ∷ just (2 ∷ 3 ∷ 4 ∷ []) ∷
just (2 ∷ 3 ∷ 4 ∷ []) ∷ []
```

To prove that this is the case for every possible input, we will define a modular predicate transformer semantics for `quickSort⁺`. Using modal operators (section 2.4), we want this semantics to express that every possible branch (□) of `quickSort⁺` should return at least one correct answer (◇). First, we postpone this choice of interpretation and use the free monad transformers for `Exc` and `Amb` to construct a modular predicate transformer semantics.

```
ptQuickSort⁺ : ℕ → ∐ QuickSort ⇒ (excT String ∘ ambT) (cont Set)
ptQuickSort⁺ n = escape
               ● partialAmb
               ● partialExc
               ● petrol⁺ quickSort⁺ n
```

Then, we use the predicate transformer transformers `ptt□` and `ptt◇` to give our predicate transformer semantics the right interpretation.

```
ptQuickSort□◇ : ℕ → ∐ QuickSort ⇒ cont Set
ptQuickSort□◇ n c = init λ fin →
  fin (ptQuickSort⁺ n c) ∘ (ptt□ ∘ ptt◇) id
```

Finally, the proposition `quickSortCorrect⁺` states that every possible result of applying `quickSort⁺` to the list `xs` is guaranteed to be correct if we choose a maximum recursion depth of `1 + length xs`.

[1] Note that we have to use `inj₁` to apply the input, because `runQuickSort⁺` can get any command from ∐ `QuickSort` as input.

```
quickSortCorrect⁺ : (xs : List ℕ)
  → ptQuickSort□◇ (1 + length xs) (inj₁ xs) (sorted xs)
```

To conclude, petrol-driven semantics allow us to execute and reason about generally recursive effectful functions, provided that we choose a maximum recursion depth that is large enough.

Chapter 9

Soundness and Dijkstra monads

So far, we have shown how to define computational and specificational semantics for many different effects, such that we can execute and reason about effectful programs. It is not immediately obvious what it means to use specificational semantics to reason about a program. In this chapter, we will show how to reason about programs in terms of their specificational monads and how we can guarantee that this reasoning is sound, as well as show how computational and specificational semantics relate to Dijkstra monads.

9.1 Refinement

The specificational monad corresponding to an effectful computation gives some insight into the behaviour of that computation. In the case of predicate transformer semantics, the specificational monad `cont Set` describes, for every postcondition `P`, the weakest precondition that should hold in order to satisfy `P`. By showing that, for some postcondition `P`, the corresponding weakest precondition is inhabited, we prove that `P` is guaranteed to hold on the output of executing our computation.

Alternatively, we can reason about a computation by relating its weakest precondition to the weakest precondition of another computation. To do so, we define a refinement relation (Morgan [1994]):

```
_⊑_ : (wp₁ wp₂ : cont Set a) → Set
wp₁ ⊑ wp₂ = ∀ {P} → wp₁ P → wp₂ P
```

For two computations c_1 and c_2, with respective weakest preconditions wp_1 and wp_2, we say that c_2 refines c_1 if $wp_1 \sqsubseteq wp_2$, i.e. if every possible postcondition holding on the output of c_1 also holds on the output of c_2. We can use this refinement relation to show that a computation is, in some sense, at least as good as another computation. For example, we can show that `quickSort` is a refinement of the less efficient, but known to be correct `insertionSort`, thereby expressing the correctness of `quickSort` in terms of a reference implementation. For some predicate transformer `pt`, we write

$$\texttt{pt insertionSort} \sqsubseteq \texttt{pt quickSort}$$

Rather than specify a computation using a reference implementation, we can define its specification more directly, simply by constructing a value of type `cont Set a`. One way to do this is to define the specification of our computation in terms of a pre- and postcondition monad `pre/post`, and from this pre- and postcondition derive the weakest precondition.

```
pre/post : Set → Set
pre/post a = Set × (a → Set)

pre/post⇒wp : pre/post a → cont Set a
pre/post⇒wp (pre , post) P = pre ∧ ∀ x → post x → P x
```

For example, we can define a pre- and postcondition for a function that sorts lists of natural numbers in ascending order. The precondition ⊤ states that the function should work on any input list. The postcondition states that the output list is sorted ascendingly and is a permutation of the input list.

```
sort-pre/post : (xs : List ℕ) → pre/post (List ℕ)
sort-pre/post xs = ⊤ , λ ys → sorted xs ys

sort-wp : (xs : List ℕ) → cont Set (List ℕ)
sort-wp = pre/post⇒wp ∘ sort-pre/post
```

This allows us to express the correctness of `quickSort` in terms of its specification:

$$\texttt{sort-wp} \sqsubseteq \texttt{pt quickSort}$$

Usually, programs are not written in one go, but rather build up incrementally. For example, in the style of Swierstra and Baanen [2019], Baanen [2019], we might define partially implemented programs as a mix between pure code and typed holes containing specifications. Starting from only a single hole, at every step we refine the current program by filling a hole with another partially implemented program adhering to its specification until eventually no holes are left. For some predicate transformer semantics `pt`, which computes the weakest precondition of a partially implemented program, this gives us a series of refinement steps:

$$\texttt{pt hole} \sqsubseteq \texttt{pt}\ p_1 \sqsubseteq \ldots \sqsubseteq \texttt{pt}\ p_n \sqsubseteq \texttt{pt code}$$

To ensure that the final program is a refinement of the initial specification, we show that the refinement relation is a preorder on the computational monad. That is, it is both reflexive and transitive:

```
⊑-refl  : x ⊑ x
⊑-trans : x ⊑ y → y ⊑ z → x ⊑ z
```

In fact, we can define a refinement relation for any specificational monad that is ordered (Katsumata and Sato [2013]). For example, the refinement relation `_⊑ₛ_` for stateful computations is defined as follows:

```
_⊑ₛ_ : (wp₁ wp₂ : stateT s (cont Set) a) → Set
wp₁ ⊑ₛ wp₂ = ∀ {P} s → wp₁ s P → wp₂ s P
```

For simplicity, we will restrict ourselves to `_⊑_` in the rest of this chapter, but the described techniques readily extend to other refinement relations like `_⊑ₛ_`.

9.2 Soundness as a refinement

As briefly discussed in the introduction, we can use specificational semantics to reason about the behaviour of a computation syntactically. To ensure that our reasoning still holds after executing a computation, we have to show that our computational semantics are sound with respect to our specificational semantics. For example, we can show that, if a postcondition `P` holds on a computation of ambivalent choice according to the demonic predicate transformer ⟦ `pt∀` ⟧, this implies that, after applying the handler function ⟦ `collect` ⟧, the predicate `P` should hold on every value in the resulting list:

```
∀ P (x : Amb ⋆ a) → ⟦ pt∀ ⟧ x P → All P (⟦ collect ⟧ x)
```

Note how this definition is a specific instance of a refinement relation! We can specify the soundness of ⟦ `pt∀` ⟧ and ⟦ `collect` ⟧ as a refinement relation as follows:

```
(x : Amb ⋆ a) → ⟦ pt∀ ⟧ x ⊑ flip All (⟦ collect ⟧ x)
```

Intuitively, executing a program using a handler function is the last step in refining that program. Note how both ⟦ `pt∀` ⟧ and `flip All` are functions to the specificational monad `cont Set`. In the style of Dijkstra monads (Ahman et al. [2017], Maillard et al. [2019]), we will call `flip All` an effect observation.

Since we defined our semantics in terms of monad homomorphisms, let us try and prove the soundness of our semantics in terms of the soundness of the corresponding monad homomorphisms. We define `sound∀` as a refinement relation, describing that `collect` refines commands from the `Amb` signature. That is to say, within the target monad, we can represent the command `c` as `collect c`, and this representation is sound with respect to `pt∀`.

```
sound∀ : ∀ c → pt∀ c ⊑ flip All (collect c)
sound∀ branch (pf , pt) = pf ∷ pt ∷ []
sound∀ abort  tt        = []
```

To show that the soundness of ⟦ `collect` ⟧ and ⟦ `pt∀` ⟧ follows from `sound∀`, we require `All` to be a monotone predicate transformer. A predicate transformer is monotone if it preserves the relative order of predicates. We define the preorder `_⊆_` on predicates, which states that, for two predicates `P` and `Q`, the set of values on which `P` holds is a subset of the set of values on which `Q` holds.

```
_⊆_ : (P Q : a → Set) → Set
P ⊆ Q = ∀ {x} → P x → Q x
```

The monotonicity of `All` is now defined as follows:

```
mono : P ⊆ Q → All P ⊆ All Q
```

Furthermore, we require `flip All` to be a monad homomorphism from the list monad to the continuation monad on `Set`. We define the following functions `homo-return` and `homo-bind`, which are essentially specific instances of the homomorphism laws (equation 4.1 and equation 4.2 respectively).

```
homo-return : return x ⊆ flip All (return x)

homo-bind : (flip All x >>= flip All ∘ k) ⊆ flip All (x >>= k)
```

In general, we prove that ⟦ `run` ⟧ is sound with respect to ⟦ `pt` ⟧ if `run` is sound with respect to `pt` and the effect observation `ϑ : m a → cont Set a` is a monotone monad homomorphism. We model this within the function ⟪_⟫, so that the soundness of ⟦ `collect` ⟧ with respect to ⟦ `pt∀` ⟧ can be defined as ⟪ `sound∀` ⟫.

```
⟪_⟫ : (sound : ∀ c → pt c ⊑ ϑ (run c))
    → ∀ (x : f ⋆ a) → ⟦ pt ⟧ x ⊑ ϑ (⟦ run ⟧ x)
⟪ sound ⟫ (pure x)   p rewrite homo-return x = p
⟪ sound ⟫ (call c k) p rewrite homo-bind (run c) (⟦ run ⟧ ∘ k) =
  mono (⟪ sound ⟫ (k _)) (sound c p)
```

While it is not in the scope of this thesis to prove for every combination of handler functions that they are sound, this approach gives a good starting point for proving such soundness results.

9.3 Dijkstra monads

In recent work on Dijkstra monads, Ahman et al. [2017], Maillard et al. [2019] show how we can reason about a computational monad m in terms of an ordered specificational monad w with a refinement relation $\preccurlyeq$. They identify a monad-like structure that relates a computation with its specification as a dependent pair of a computation and the proof that it adheres to its specification. This proof is defined in terms of the effect observation ϑ, which is a monad morphism from m to w that *observes* the behaviour of a computation from m within the specificational monad w. This *observed* behaviour is then related to the specification using the refinement relation $\preccurlyeq$. We can model a Dijkstra monad as follows:

```
record Dijkstra
  (ϑ : ∀ {a} → m a → w a)
  (spec : w a) : Set where
  field
    comp  : m a
    proof : spec ≼ ϑ comp
```

A value of type `Dijkstra ϑ spec` represents a computation in m that is guaranteed to adhere to the specification `spec`, according to the effect observation ϑ. To be able to compute with a Dijkstra monad D, we require it to be equipped with the monadic operations $return_D$ and $bind_D$, which essentially extend the usual *return* and $\gg\!=$ operations of the computational monad by computing the correct specifications to which the resulting computations adhere.

$$return_D : \forall\ x \rightarrow D\ (return\ x)$$
$$bind_D : (mx : D\ wx)\ (mk : \forall\ x \rightarrow D\ (wk\ x)) \rightarrow D\ (wx \gg\!= wk)$$

It is easy to show that we can define both $return_D$ and $bind_D$ if the effect observation ϑ is a monotone monad homomorphism. The definition of $bind_D$ is very similar to the soundness proof as described in the previous section, which also required a monotone monad homomorphism.

As such, it is not surprising that we can define a Dijkstra monad with computational monad `List` and specificational monad `cont Set`, by choosing `flip All` as its effect observation. Similarly, the free monad with signature f forms a Dijkstra monad with specificational monad `cont Set` if the effect observation ⟦ `pt` ⟧ is a monotone predicate transformer. All predicate transformers that we defined in this thesis can easily be proven to be monotone and thus give rise to Dijkstra monads.

Knowing that predicate transformer semantics give rise to Dijkstra monads, what is the meaning of handler functions in this context? Take for example ⟦ `collect` ⟧, which is a monad morphism from the free monad of ambivalent choice to the list monad. These two monads are exactly the computational monads of the Dijkstra monads that arise from the effect observations ⟦ `pt` ⟧ and `flip All` respectively. We might try and define a morphism `collectD` between these two Dijkstra monads:

```
collectD : (D : Dijkstra ⟦ pt∀ ⟧ spec) → Dijkstra (flip All) spec
```

Let `x` denote the computational part of the original Dijkstra monad `D`. We can define the computational part of the target monad as simply ⟦ `collect` ⟧ `x`. To complete `collectD`, we have to prove that `flip All (⟦ collect ⟧ x)` refines the specification `spec`. Since `D` has a proof that ⟦ `pt∀` ⟧ `x` refines `spec`, using the transivity of refinement we only have to show that `flip All (⟦ collect ⟧ x)` refines $[\![$ `pt∀` $]\!]$ `x`. In other words, we have to prove that $[\![$ `pt∀` $]\!]$ is sound with respect to $[\![$ `collect` $]\!]$!

In general, any monotone predicate transformer semantics $[\![$ `pt` $]\!]$ gives rise to a Dijkstra monad and every sound handler function $[\![$ `run` $]\!]$ gives rise to a Dijkstra monad morphism.

Chapter 10

Conclusions & further work

In this thesis, we have described techniques for the verification of effectful computations, by assigning semantics to programs using a variety of different effects in terms of the semantics of the individual effects. We define semantics for individual effects as monad homomorphisms from the free monad to some target monad. By choosing as the target monad a monad transformer applied to a polymorphic base monad, combining semantics comes down to building up a monad transformer stack, where the choice of base monad determines the nature of the semantics. This approach gives us a lot of freedom when writing effectful computations: we can handle effects one at a time and reason about the intermediate results because the computational and specificational semantics are defined in the same way. The monad homomorphism laws guarantee that we get the same results regardless of whether we compose two programs before or after applying our semantics.

Whereas the ease at which new effects and semantics can be introduced is an important part of the techniques we described, a well-documented collection of effects along with semantics and soundness proofs relating them would be an instrumental contribution towards the usability of our techniques. So far, we have shown how to define modular semantics for the effects of mutable state, exceptional behaviour, ambivalent choice and general recursion. Apart from these effects and combinations thereof, there are many more interesting effects, such as probabilistic choice and cooperative multithreading (Bauer and Pretnar [2015]), as well as the well-known operation *callCC* (call-with-current-continuation), which, as described by Schrijvers et al. [2019], can be modelled using algebraic effects. It would be interesting to see for which of these effects it is possible to define modular semantics using the techniques we described.

In the same vein, every effect can be given a variety of different semantics. For example, Swierstra and Baanen [2019], Baanen and Swierstra [2020], Baanen [2019] describe semantics for general recursion in terms of an invariant. Compared to the petrol-driven semantics, this invariant-based semantics is a lot more sophisticated and it is not immediately obvious whether it can be integrated with the techniques described in this thesis.

Additionally, there are some effects which are not strictly algebraic that we would still like to be able to handle. For example, we might want to define an effect for handling files which only allows the programmer to use a write operation if a file is currently open. We believe it should be possible to define such effects by generalising effect signatures to indexed containers (Altenkirch et al. [2015]).

For every effect and every combination of effects we introduce, it is important to make sure that their semantics are sound. In chapter 9 we described soundness as a refinement relation and showed that we can prove the soundness of semantics in terms of the soundness of their morphisms exactly if the corresponding effect observation is a monotone monad homomorphism. We would like to be able to show that the modular semantics we described preserve the soundness of our semantics. That is, when combining semantics that are sound, the resulting semantics should also be sound. While there is still some work to be done before we can state that this is the case, we believe that the techniques we described give a good starting point for proving the soundness of modular semantics.

In practice, most programmers do not write programs in one go, but rather in a series of small steps that eventually lead to executable code. This idea of stepwise refinement, which we discussed briefly in section 9.1, requires that every refinement step correctly updates the current proof obligations. Ahman et al. [2017], Maillard et al. [2019] achieve this by equipping Dijkstra monads with the monadic operations $return_D$ and $bind_D$. Baanen [2019] proposes a slightly different approach, introducing similar operations for predicate transformer semantics, and ponders whether this approach is more expressive compared to Dijkstra monads, given that Dijkstra monads do not inherently separate syntax and semantics. It seems to us that this potential loss in expressivity is mitigated by the introduction of Dijkstra monad morphisms between syntactical Dijkstra monads (arising from specificational semantics on the free monad) and computational Dijkstra monads, as described in section 9.3.

Much of this thesis comes down to providing techniques for generating proof obligations, which the programmer is then required to discharge by hand. In terms of usability, there is still a lot of room for improvement. We might, for example, adapt the techniques by O'Connor [2019] to defer proof obligations, use macros to enhance the syntax of effectful computations, or take inspiration from the implementation of Dijkstra monads within the dependent programming language $F\star$ (Swamy et al. [2013, 2016]).

There is still a great deal of work to be done before modular predicate transformer semantics can be used in the formal verification of large-scale applications, but we believe that the findings in this thesis form a valuable contribution towards this goal.

Bibliography

Philip Wadler. A critique of abelson and sussman or why calculating is better than scheming. *ACM SIGPLAN Notices*, 22(3):83–94, 1987.

Wouter Swierstra and Anne Baanen. A predicate transformer semantics for effects (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi: 10.1145/3341707. URL `https://doi.org/10.1145/3341707`.

Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(04), July 2008. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796808006758. URL `http://www.journals.cambridge.org/abstract_S0956796808006758`.

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: what binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell - Haskell 2019*, pages 98–113, Berlin, Germany, 2019. ACM Press. ISBN 9781450368131. doi: 10.1145/3331545.3342595. URL `http://dl.acm.org/citation.cfm?doid=3331545.3342595`.

Conor McBride. Turing-completeness totally free. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction*, pages 257–275, Cham, 2015. Springer International Publishing. ISBN 978-3-319-19797-5.

Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. *ACM SIGPLAN Notices*, 52(1):515–529, January 2017. ISSN 03621340. doi: 10.1145/3093333.3009878. URL `http://dl.acm.org/citation.cfm?doid=3093333.3009878`.

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martinez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi: 10.1145/3341708. URL `https://doi.org/10.1145/3341708`.

Gordon Plotkin and John Power. Notions of computation determine monads. In *Proc. FOSSACS 2002, Lecture Notes in Computer Science 2303*, pages 342–356. Springer, 2002.

Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11:2003, 2003.

Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In Peter G. Clote and Helmut Schwichtenberg, editors, *Computer Science Logic*, pages 317–331, Berlin, Heidelberg, 2000a. Springer Berlin Heidelberg. ISBN 978-3-540-44622-4.

Peter Hancock and Anton Setzer. Specifying interactions with dependent types. In *Workshop on Subtyping and Dependent Types in Programming*, Ponte de Lima, Portugal, 2000b. INRIA.

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In *Proceedings of Foundations of Software Science and Computation Structures*, pages 23–38, 2003.

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers – constructing strictly positive types, 2004.

Graham Hutton and Erik Meijer. Monadic parser combinators, 1996.

Anne Baanen. Algebraic effects, specification and refinement. Master's thesis, Utrecht University, the Netherlands, 2019.

Anne Baanen and Wouter Swierstra. Combining predicate transformer semantics for effects: a case study in parsing regular languages. *Electronic Proceedings in Theoretical Computer Science*, 317:39–56, May 2020. ISSN 2075-2180. doi: 10.4204/EPTCS.317.3. URL `http://arxiv.org/abs/2005.00197v1`.

Mauro Jaskelioff and Russell O'Connor. A representation theorem for second-order functionals. *Journal of Functional Programming*, 25:e13, 2015. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796815000088. URL `https://www.cambridge.org/core/product/identifier/S0956796815000088/type/journal_article`.

Carroll Morgan. *Programming from specifications*. Prentice Hall international series in computer science. Prentice Hall, New York, 2nd ed edition, 1994. ISBN 9780131232747.

Shin-ya Katsumata and Tetsuya Sato. Preorders on monads and coalgebraic simulations. In *In Proc. FoSSaCS 2013, LNCS 7794, pp.145–160*. Springer, 2013.

Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, January 2015. ISSN 23522208. doi: 10.1016/j.jlamp.2014.02.001. URL `https://linkinghub.elsevier.com/retrieve/pii/S2352220814000194`.

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor Mcbride, and Peter Morris. Indexed containers, 2015.

Liam O'Connor. Deferring the details and deriving programs. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2019, page 27–39, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368155. doi: 10.1145/3331554.3342605. URL `https://doi.org/10.1145/3331554.3342605`.

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*, page 387, Seattle, Washington, USA, 2013. ACM Press. ISBN 9781450320146. doi: 10.1145/2491956.2491978. URL `http://dl.acm.org/citation.cfm?doid=2491956.2491978`.

Nikhil Swamy, Markulf Kohlweiss, Jean-Karim Zinzindohoue, Santiago Zanella-Béguelin, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, and Pierre-Yves Strub. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*, pages 256–270, St. Petersburg, FL, USA, 2016. ACM Press. ISBN 9781450335492. doi: 10.1145/2837614.2837655. URL `http://dl.acm.org/citation.cfm?doid=2837614.2837655`.