Algebraic effects, specification and refinement

Utrecht University

Master Thesis in Computing Science and Mathematical Sciences

Tim Baanen

Supervisors: Wouter Swierstra and Jaap van Oosten

July 2019

Abstract

In the process of software engineering, we want to be sure that our code will function according to our requirements. The refinement calculus is a system that allows for mathematical correctness proofs, for imperative programs using a specific set of side effects. In the thesis, we explain how to use algebraic effects to add side effects to a purely functional program, and how to generalize concepts from the refinement calculus to develop and verify programs with algebraic effects. We work in the dependently typed programming language Agda, illustrating that this approach allows for formal verification of programs.

Acknowledgements

I dedicate the thesis to my partner Xareni, who has supported me throughout the process, from the Netherlands and from Mexico. I would also like to thank my family and friends, especially the students in the Mathematics library who were around for a quick discussion and/or sharing a cup of hot chocolate. Finally, I would like to thank my supervisors. Jaap van Oosten was my tutor and has offered his guidance during the whole Master programme. Wouter Swierstra offered me the opportunity to collaborate on his research and helped immensely with useful suggestions and insightful discussions. This thesis would have been impossible to write without the aid of all of these people, and the aid of many more people whose names I cannot fit within one page.

> "Beware of bugs in the above code; I have only proved it correct, not tried it."
> — Donald Knuth, *Notes on the van Emde Boas construction of priority deques: An instructive use of recursion*

Contents

Chapter 1

Introduction

When we are writing software, we want to demonstrate that our work is correct. The result of a successful engineering effort is not only executable code, but also an assurance that the code meets its specifications. Commonly, this is done by running the program (either manually or automatically) in a set of test circumstances and checking whether the behaviour meets our expectations. As Dijkstra said repeatedly, program testing is hopelessly inadequate for showing the absence of bugs [Dij76]. The alternative, then, is to give a mathematical proof of the desired properties of our program. If we capture the specifications in a formal statement and give a proof of this statement (again, we can do this reasoning either on paper or in a computer system), then we are assured that all possible cases are covered.

Many programming languages offer a type system that assures that operations are performed on values of the correct type. If this type system is expressive enough, it will allow us to perform mathematical proofs in the style of intuitionistic type theory. A computation can be represented as a term in type theory, the type of a computation is its specification, and verifying this specification consists of verifying that the term is well-typed. One language that offers such a type system is Agda, which can simultaneously serve as a functional programming language and proof assistant [Nor07]. Since computations in intuitionistic type theory are always mathematical functions, they are not able to directly represent the side effects that are essential in writing useful programs. Moreover, many programming languages are imperative, which means their programs do not map well onto terms of a specific type. The alternative to the intrinsic representation of correct programs as well-typed terms, is to separate the syntax of a program and its semantics. We represent the program syntactically as a mathematical object, which is well-typed independent of the program's verification, and try to find another term that expresses the proof of correctness.

In the realm of imperative programming, the *refinement calculus* allows us to mix specifications and executable code [Mor98], and uses these to express the correctness of a program mathematically. An abstract program may include parts that are only specified and not yet executable code. Whenever an (abstract) program meets the specifications of another, the *refinement* relation holds. The refinement calculus gives the rules for this relation, in terms of *weakest precondition* semantics. An idealized process of software engineering using the refinement calculus starts with writing down the specification formally, as the starting program. Then, we repeatedly apply the rules of the refinement calculus to replace parts of this specification with executable code, until the whole program has been transformed from specification to code. Each intermediate program in this process of *derivation*, although the program is not yet executable, still satisfies the original specification.

In the case of purely functional programming, correctness proofs typically make use of *equational reasoning* [Wad87]. Instead of a specification, we write a program that solves the given problem. This initial program might find its solution in a brute-force or otherwise inefficient manner. After finishing the final program, the proof of correctness then shows for all inputs that both the initial program and the final program have equal output. Equational reasoning assumes *referential transparency*: whenever a program occurs in an expression, we can replace it by its definition. This is only the case if we have a pure language and we do not worry about termination. The language Agda supports equational reasoning since the type system

enforces that all functions are pure and total. The fundamental property of side effects is that they break referential transparency, so equational reasoning cannot be carried out unmodified in the presence of effects. Moreover, equational reasoning uses the same language for specification and implementation, so it does not separate the concerns of specifying and writing code.

In this thesis, we will investigate correctness of functional programs with effects [BP15]. Effectful functional programs using monads can be verified using equational reasoning [GH11]. We propose that a generalisation of the refinement calculus is a better approach, since it allows from the separation of specification from implementation. The language used in the refinement calculus is imperative and based on the Guarded Command Language by Dijkstra [Dij75a], and only allows for one set of effects involving state and nondeterminism. Thus, we generalise in the effects that we allow in the program.

Our main results in this thesis are the following:

- We demonstrate that predicate transformer semantics allow us to verify the partial correctness and termination of generally recursive functions, in Chapter 3.
- We show how to apply predicate transformers to algebraic effects, allowing for a similar specification and verification approach to the refinement calculus, in Chapter 4.
- If a computation contains multiple sets of effects, we will show how to combine predicate transformers of the individual effects into semantics for the computation, in Chapter 5.
- We compare predicate transformer semantics with other semantics for effects, giving conditions on consistency with handlers and demonstrating that the refinement relation subsumes equational reasoning, in Chapter 6.
- We can treat a specification as an effect, allowing us to derive executable code from a specification by incorporating specifications into code, in Chapter 7.
- To show verification using the refinement relation in a more practical setting, we verify the generation of a parser from a regular expression and from context-free grammar rules, in Chapter 8.

Parts of this thesis have been adapted from published work. Chapters 3 and 4 are based on text from an article that I was a co-author on, published as [SB19]. The results of Chapter 3 are mine where not noted otherwise, and I wrote this chapter without copying the article to make clear this is my own work, under the guidance of my supervisors. For Chapter 4, I supplied proofs of *fold-bind*, *compositionality-left* and *compositionality-right*, and I showed that the compositionality property requires the monotonicity of predicate transformers. The effect, predicate transformer and refinement framework of Chapter 4 was designed by Wouter Swierstra, my contributions consist mostly of discussing the process of using and understanding the framework. By separating out these two chapters, I hope it is clear which are my contributions to the article. Chapter 7 discusses deriving programs from specifications, also discussed in the article, but this thesis includes an alternative approach to including specifications in programs that does not re-use the content of the article. Finally, the other chapters are my own work, distinct from the article.

Chapter 2

Background: Agda, monads and predicate transformers

In the thesis, we will look at verifying computations that use effects. This chapter gives background on programming with effects in functional languages, and more specifically in the functional language Agda. After explaining the basics of Agda, we will explain how to model side effects in functional languages using monads.

2.1 Agda

The dependent programming language Agda [Nor07] will play a crucial role in our work. As a programming language, it serves as the metalanguage in which we will implement the effect and handler systems we will discuss. As a predicate language, it will specify the properties in our program verification framework. As a formal system of logic, it specifies our theorems about the framework and their proofs. In fact, type theories such as Agda's already contain a complete programming language and complete facilities for program verification, as Martin-Löf already remarked [Mar84].

We will use inline Agda code to write our results formally. In Agda, we can express an effect system, programs in this system, predicates on programs and proofs of these predicates in one language. Moreover, our proofs will be automatically formally checked. For readability, we do not include all details of the code.

In many forms of intuitionistic mathematics, and Agda in particular, we do not assume that functions are extensional. That is, $\forall\, x \to f\ x == g\ x$ does not necessarily imply $f = g$ [TD91]. At certain points, especially when we deal with equational reasoning, we care about extensional equality and only extensional equality. Specifically, Agda regards two functions with equal definitions but different names as distinct for the _==_ relation, which means the equality relation is too small for full referential transparency. Thus we assume extensional equality:

```
postulate extensional : {a b : Set} {f g : a → b} → (∀ x → f x == g x) → f == g
```

For notational convenience, we will assume that the type of *Set* is *Set* itself, i.e. use the `--type-in-type` option for Agda. This assumption will save a lot of bookkeeping of universe levels, but it is possible to derive a contradiction from this [OCo15]. We expect that all code should work when an explicit universe hierarchy is constructed in the examples, as we did so for the code in the published article [SB19].

2.2 Monads

Practical programs will involve side effects such as input and output, but the λ-calculus only allows for pure computations. The notion of *monads* provide a useful system to describe computations with effects in a

categorical way. Similar to pure computations, we will associate the type of a computation with an object in the category *Set*, but we will distinguish values of type *a* from the computation that produces such a value, of type *T a*. Here, *T* is an endofunctor on *Set*, and for all *a b* : *Set*, we have two operations called *bind* : *T a* → (*a* → *T b*) → *T b* and *return* : *a* → *T a* that satisfy certain equations, the *monad laws*. These equations specify exactly that the pair (*T* , *return*, λ *mx* → *bind mx id*) is a monad in the category *Set* [Mog91]. Monads are the method by which the purely functional programming language Haskell allows side effects. In Agda, we can write the monad structure on a map *T* : *Set* → *Set* as a record type:

```
record Monad (T : Set → Set) : Set where
  constructor monad
  field
    bind : T a → (a → T b) → T b
    return : a → T a
    left-identity : (x : a) (f : a → T b) → bind (return x) f == f x
    right-identity : (mx : T a) → bind mx return == mx
    associativity : (mx : T a) (f : a → T b) (g : b → T c) →
      bind (bind mx f) g == bind mx (λ x → bind (f x) g)
```

We will often use the operator _>>=_ instead of *bind M* if the monad structure *M* is clear from context. Similarly, for *return M* we will also write *pure*.

In category theory, monads arise from adjoint functors, for example the *State* monad that is given by the adjunction between the exponential object and the Cartesian product functor [Mac71]. We will work out the details of this monad in the context of functional programming, illustrating how we can formalise mathematics in Agda.

Example 2.2.1. The *State monad* gives us the ability to read from and write to a single variable. A computation using a variable of type *s* and producing a value of type *a*, can be given as a function that takes the initial value *i* : *s*, and returns a pair consisting of the computation's outcome and the new value of the variable. We express such computations as elements of the type *State s a*:

```
State : Set → Set → Set
State s a = s → Pair a s
```

Reading from the variable and writing to it can be expressed as specific stateful computations, called *get* and *put*.

```
get : {s : Set} → State s s
get t = t , t
put : {s : Set} → s → State s ⊤
put t _ = tt , t
```

The two operations *bind* and *return* on *State* are given as follows:

```
MonadState : (s : Set) → Monad (State s)
Monad.bind (MonadState s) mx f = λ t → uncurry f (mx t)
Monad.return (MonadState s) x = λ t → (x , t)
```

Here, *uncurry* converts a function that takes two arguments into one that takes a pair of arguments. The proof of the monad laws can be given as an instance of reflexivity, since Agda is able to evaluate both sides into the same result.

```
Monad.left-identity (MonadState s) x f = refl
Monad.right-identity (MonadState s) mx = refl
Monad.associativity (MonadState s) mx f g = refl
```

◊

2.3 Program verification using equations

The previous sections explain a representation of effects in functional programming. Let us now move from syntax to semantics, so we can talk about correctness of the programs. In a pure and total functional programming language, programs have no side effects, so we can fully characterise the behaviour of a pure computation by its output value for each given input value. This means we can perform *equational reasoning* on pure computations: we define the semantics of a pure computation in terms of equations on (sub-)expressions [Wad87]. In other words, we reason using the *denotational semantics* of the computations. An example of such equations is that the *map* function satisfies $map\ (f \circ g) == map\ f \circ map\ g$ for all f and g, or stating that the *State* monad satisfies the *leftid*, *rightid* and *assoc* monad laws. Note that the requirement of pure functions also means that the programs must always terminate, and cannot crash or go into an infinite loop. This is a stricter notion than that of "pure" function in Haskell.

Equational reasoning can be extended to monadic computations by introducing equational axioms for the monad under consideration. These consist not only of the monad laws *leftid*, *rightid*, *assoc*, but also laws that specify the interaction of each monadic operation. The potential drawback to this approach is that it is no longer clear which laws we want to specify for a given situation, as an implementation might have stricter laws than the interface we are trying to construct. Additionally, the interaction between multiple operations does not follow obviously from the laws of each operation individually [GH11].

Example 2.3.1. For example, in the *State* monad where we have operations *get* and *put*, we have equations for all four possible compositions of two effects. For ease of notation, we introduce the _≫_ operator, which takes stateful computations and discards the value produced by the first computation, retaining its effect on the state [GH11].

```
_≫_ : State s a → State s b → State s b
mx ≫ my = mx >>= const my
```

Using this notation, we can formulate and prove in Agda the laws of stateful computation according to Power and Plotkin [PP02] and show that *State* satisfies them:

```
put-put : {s : Set} (t t' : s) → put t ≫ put t' == put t'
put-put t t' = refl
put-get : {s : Set} (t : s) → put t ≫ get == put t ≫ pure t
put-get t = refl
get-put : {s : Set} → get >>= put == pure {s} tt
get-put = refl
get-get : {s a : Set} (t : s) (k : s → s → State s a) →
  get >>= (λ t → get >>= k t) == get >>= (λ t → k t t)
get-get t k = refl
```

◊

We have again included all details of the code in this example. In the following text, we will decrease the level of detail, for example omitting trivial proofs.

2.4 Program verification using predicate transformers

In the thesis, we will use an axiomatic semantics in the form of predicate transformers. The refinement calculus is a specific instance of program verification using predicate transformer semantics. Predicate transformer semantics have long been used for program verification, for instance in Dijkstra's Guarded Command Language [Dij75a]. This section gives an overview of predicate transformers as used in the imperative language GCL. One of the results in this thesis is to demonstrate how to apply predicate transformer semantics in functional programs with effects. A description of predicate transformer semantics for effects can be found in Section 4.1.

In these semantics, we view programs as producing output states for given input states. We will not specify in depth when something is an input or output state, as this depends on many aspects of our verification system and program, and is not very relevant to the remainder of our discussion.

First, we will discuss how predicate transformers arise from operational semantics. If we already have the operational semantics, we can determine which output states may result from a given input state.

Definition 2.4.1. For a program S, input state i and output state o, we write $o \in S\ i$ if running S on input state i may give output state o. △

Remark 2.4.2. In this formulation, programs may be nondeterministic: the input state does not correspond uniquely to an output state. Most often we have that the output state is absent: if the program enters an infinite loop on input i, then there is no $o \in S\ i$. ◊

Validity of a program then comes down to giving the right output for all relevant input. We adapt this view on validity and the following definition from Hoare [Hoa69].

Definition 2.4.3. For imperative programs, a *specification* consists of predicates P (the *precondition*) and Q (the *postcondition*). The precondition P depends on the input state i, while Q depends on the input and output states i and o.

An imperative program S *satisfies the specification* P , Q, if for all input states i and $o \in S\ i$, we have that $P\ i \to Q\ i\ o$. This condition is also written $\{P\}\ S\ \{Q\}$, and together the precondition, program and postcondition are called a *Hoare triple*. △

The previous definitions use the operational semantics to define validity, but we can directly build an axiomatic semantics. When we view programs as transformations of predicates, the behaviour of programs is axiomatically specified in terms of the predicates they satisfy. Thus, the Hoare triples are the fundamental relations, from which we derive the relation between in- and output states. In this view, a program does not fundamentally operate on states but on predicates, so a semantics in this style is called *predicate transformer semantics*. In fact, we can make preconditions derive from the postcondition as well, according to the following definition:

Definition 2.4.4 ([Dij75a])**.** Let Q be a predicate over an input and an output state, and S be a program. The *weakest precondition* $wp\ S\ Q$ is a predicate over an input state, such that S satisfies the specification with precondition $wp\ S\ Q$ and postcondition Q. Moreover, for all preconditions P' for S and Q, we have that $wp\ S\ Q$ is weaker: for all i we have $P'\ i \to wp\ S\ Q\ i$. △

The weakest precondition transforms one predicate, the postcondition, into another, the precondition; this is why we call $wp\ S$ a *predicate transformer*.

Instead of defining the weakest precondition operator in terms of Hoare triples, we can give axioms for computing the weakest precondition for all program statements and postconditions, and recover Hoare triples by defining that $\{P\}\ S\ \{Q\}$ holds if for all i we have $P\ i \to wp\ S\ Q\ i$ [Dij75a].

2.5 Refinement calculus

In this subsection, we will discuss the refinement calculus as proposed by Morgan [Mor98] and Back and von Wright [Bv12], which extends the program verification process from predicate transformer semantics. Instead of the imprecise reasoning employed by Dijkstra to derive correct programs, the refinement calculus gives a mathematical foundation for this process.

The *refinement calculus* allows us to unify the process of writing code to match a specification, and verifying the correctness of the code we have written. In between a specification and an executable program we often resort to pseudocode, a program where formal language constructs have been extended with natural language descriptions of things yet to be implemented. At this stage, we have made choices that cut our original specification into a few smaller subspecifications that we hope to implement more easily [Dij76]. If we want to apply formal methods for program verification, we have to find a way to incorporate pseudocode in the process.

The contribution of refinement calculus to predicate transformer semantics is to allow specifications as statements in a program, dropping the requirement that a program is wholly made up of executable code. This gives us a single language both for programs satisfying a specification, and programs having equivalent behaviour to another. Both are expressed using the *refinement* relation $\_ \sqsubseteq \_$.

Definition 2.5.1. We say a program S *refines* a program S', and write $S \sqsubseteq S'$, if for all predicates Q and all states i, we have $wp\ S\ Q\ i \rightarrow wp\ S'\ Q\ i$. △

We introduce a new program construct $[\_,\_]$, which takes two predicates P and Q. Informally, this construct does nothing more than satisfy $\{P\}\ [P, Q]\ \{Q\}$ and those specifications that are consequences of this triple. Its formal semantics are given in terms of wp: $wp\ [P, Q]\ Q'\ i = P\ i \wedge (\forall\ o \rightarrow Q\ i\ o \rightarrow Q'\ i\ o)$. This gives us another way of expressing program correctness: a program S satisfies the specification $P\ ,\ Q$ if and only if $[P\ ,\ Q] \sqsubseteq S$. The central innovation of the refinement calculus is allowing the use of the $[\_, \_]$ operator as a program construct. In Chapter 7, we will show how to use this operator in functional programming with effects. Before we can reach that point, we must generalise the concepts such as weakest precondition semantics to the functional setting, which we do in the following chapters.

Chapter 3

Reasoning about general recursion

It turns out that many of our concepts of interest will reveal themselves if we try to represent and reason about non-terminating programs in a total language. Agda allows for recursion in definitions, for example in the definition of the _+_ operator:

```
_+_ : ℕ → ℕ → ℕ
Zero   + b = b
Succ a + b = Succ (a + b)
```

However, such definitions are only allowed if they terminate. Agda uses a built-in termination checker that uses a *size-change* principle: it checks that each infinite call chain gives rise to an infinite descending chain of some data values in a sub-expression order [Agda2.6; LJB01]. Since Agda's data types are well-founded, this implies that there is no such infinite descending chain, so the computation terminates if it satisfies the size-change principle. Still, there are terminating recursive computations that are not accepted by the size-change principle, and it is still useful to represent non-terminating programs in Agda, perhaps as an object to study, even if we never get to fully execute them.

To make the distinction clear, we will use *well-founded recursion* to refer to recursion principles that are guaranteed to terminate by the size-change principle (or a comparable principle). If the recursion does not necessarily terminate, we will say this is *general recursion*. In this chapter, we will show a representation of general recursion in Agda, and demonstrate that this gives rise to effects and predicate transformer semantics.

3.1 The problem of *f91*

A classical problem for termination checking is *McCarthy's 91 function*, which we can represent in Agda syntax as the following definition of *f91* [MP70].

```
f91 : ℕ → ℕ
f91 i with 100 <? i
f91 i | yes _ = i - 10
f91 i | no _  = f91 (f91 (i + 11))
```

For values $i \leq 101$, if we manually unfold the definition of *f91 i* enough times, we get a result of *91*. Still, the recursion is not based on calling on subexpressions (in this case, that would mean strictly smaller natural numbers), so Agda's termination checker will reject the definition.

Thus, the definition of *f91* in Agda cannot be directly given in the form of a total function of type $\mathbb{N} \to \mathbb{N}$. We can still represent *f91* by changing the body of the definition (and changing the type of the codomain to match). We replace the recursive calls with values expressing where this call should occur. An elegant way to do this is to use McBride's representation in the *General* data type [McB15], which is an example of a free monad. Thus, we use the name *Free* for the data type.

Definition 3.1.1 ([KI15])**.** The *Free monad* on type *a* is given by the following data type:

```
data Free (C : Set) (R : C → Set) (a : Set) : Set where
  Pure : a → Free C R a
  Step : (c : C) → (k : R c → Free C R a) → Free C R a
```

△

As we can see, the *Free* monad has three parameters: the type *C* represents the type of arguments to recursive calls, the type *R c* represents the return type for a call with argument *c*, and the type *a* represents the type returned by the whole computation. At each point, the computation either finishes and outputs a *Pure* value of type *a*, or it needs an extra step, making a call with argument *c* : *C*, and continuing according to *k* when a value of type *R c* has been returned.

Definition 3.1.2 ([McB15])**.** The type of generally recursive functions consist of Kleisli arrows for the *Free* monad:

```
_ ↬ _ : (C : Set) (R : C → Set) → Set
C ↬ R = (c : C) → Free C R (R c)
```

△

Note that this allows for dependent functions, by having *R* depend on *C*. A single recursive call can be represented as follows:

```
call : C ↬ R
call c = Step c Pure
```

Using the _ ↬ _ type, the recursive definition of *f91* would have type $\mathbb{N} \rightsquigarrow \lambda \_ \to \mathbb{N}$. For conciseness, we will not distinguish between a dependent type for *R* or a fixed type, i.e. we write *f91* : $\mathbb{N} \rightsquigarrow \mathbb{N}$.

We claimed before that *Free* is a monad, and we will show that it, or more precisely that *Free C R* for each choice of *C* and *R*, is indeed a monad with unit *Pure* and bind given as follows:

```
_>>=_ : Free C R a → (a → Free C R b) → Free C R b
Pure x  >>= f = f x
Step c k >>= f = Step c λ x → k x >>= f
MonadFree : Monad (Free C R)
Monad.bind MonadFree = _>>=_
Monad.return MonadFree = Pure
```

To show the monad laws hold for *Free*, we need to use the *extensionality* postulate on the continuations. The proofs are otherwise straightforward.

Using the *Free* monad, we make the recursive calls in the definition of *f91* explicit, giving the following representation:

```
f91 : ℕ ↬ ℕ
f91 i with 100 ≟ i
f91 i | yes _ = Pure (i - 10)
f91 i | no _  = call (11 + i) >>= call
```

Note that this definition is *not* recursive, but explicitly represents the points at which recursive calls would be made.

Another motivating example of general recursion is the *quicksort* function, which recursively sorts a list by splitting it up into sub-lists and sorting those.

```
quicksort : List ℕ ↬ List ℕ
quicksort Nil = Pure Nil
quicksort (x :: xs)
  = call (filter (λ y → y ≤? x) xs) >>= λ ys →
    call (filter (λ z → z ≥? x) xs) >>= λ zs →
    Pure (ys ++ (x :: zs))
```

The recursive calls of *f91* are nested, so the argument of the second call depends on the return value of the first. In contrast, the calls in *quicksort* do not depend on each other, only on the argument to the definition. Although the recursion used in *quicksort* is not structural, there is a simple argument for its termination: each call is made to a strict sublist of the original list, so the length of the argument *xs* is a natural number that strictly decreases for each call. To get a recursive representation that is accepted by Agda, we can pass in an upper bound on the length of *xs*, and show that this bound decreases by at least *1* for each call.

The next example of general recursion is the Ackermann function:

```
ackermann : Pair ℕ ℕ ↬ ℕ
ackermann (Zero , b)       = Pure (Succ b)
ackermann (Succ a , Zero)  = call (a , 1)
ackermann (Succ a , Succ b) = call (Succ a , b) >>= λ c → call (a , c)
```

Each call either decreases the first argument, or keeps the first argument the same and decreases the second argument. Viewing the argument of type *Pair* ℕ ℕ as an ordinal number up to $\omega \times \omega$, each recursive call is made to a strictly smaller ordinal number, so the recursion is well-founded and the evaluation of the Ackermann function will terminate.

However, we can also give recursive definitions that do not terminate, or ones of which we want to study termination. The *Collatz sequence* starts with a given natural number *n*. Given a term a_i in the sequence we determine the next term as follows: if a_i is even, the next term is a_i divided by two, and if a_i is odd, the next term is $3a_i + 1$. If $a_{i+1} = 1$, the sequence stops, otherwise we compute the next term as before and continue. It is not known whether the Collatz sequence stops for all initial terms *n*. The following generally recursive program computes the Collatz sequence for a given number *n*:

```
collatz : ℕ ↬ List ℕ
collatz 0 = Pure (0 :: Nil)
collatz 1 = Pure (1 :: Nil)
collatz n = if   even n
            then (call (⌊ n /2 ⌋)     >>= λ ns → Pure (n :: ns))
            else (call (3 * n + 1) >>= λ ns → Pure (n :: ns))
```

3.2 Functions as relations between in- and output

We claimed that *f91* always results in the value *91* if we call it on values $i \leq 91$, so we should be able to prove this. We can define a relation *R91* that expresses this claim formally, as a relation between the input of *f91* and its output.

```
R91 : ℕ → ℕ → Set
R91 i o with 100 <? i
R91 i o | yes _ = o == (i - 10)
R91 i o | no _  = o == 91
```

For verifying a recursive function of type $C \rightarrowtail R$, we can give a specification in the form of a relation of type $(c : C) \to R\ c \to Set$. If we want to verify that a given definition satisfies such a relation, we should check that the final *Pure* value is related to the input value, but at the recursive *Step*, we may assume that the relation holds between argument and result. This gives rise to the following definition:

```
invariant : ((c : C) → R c → Set) → (c : C) → Free C R (R c) → Set
invariant rel c (Pure x)    = rel c x
invariant rel c (Step c' k) = ∀ x' → rel c' x' → invariant rel c (k x')
```

We call this function *invariant* since it checks that the relation is an invariant for the call graph.

Definition 3.2.1. Suppose we are verifying a program f with respect to some specification on input and output. If all output produced by f satisfies the specification, we will say that f is *sound*, or if f is generally recursive that it is *partially correct*. Conversely, to prove that f is *complete*, or if f is generally recursive that it *terminates*, we check that all of the expected output of f is (eventually) produced. Soundness and completeness together are called *total correctness* of f. △

Example 3.2.2. For example, we can show that *f91* indeed satisfies the invariant, i.e. that *f91* is sound with respect to *R91*. The proof mirrors the definition of *f91*: we start by making a case distinction between the cases $100 < i$ and $100 \geq i$. In the first case, we are immediately finished by the definition of *R91*, while in the second case we use that the invariant holds on the call graph, to either have that one of the two calls returns *91*, or prove that i was exactly *100*, giving *91* after two calls. The last case needs a lemma with a long but uncomplicated proof, which we omit.

```
f91-sound : (i : ℕ) → invariant R91 i (f91 i)
f91-sound i with 100 <? i
f91-sound i | yes p with 100 <? i
f91-sound i | yes p | yes p' = refl
f91-sound i | yes p | no ¬p' = magic (¬p' p)
f91-sound i | no ¬p with 100 <? i
f91-sound i | no ¬p | yes p' = magic (¬p p')
f91-sound i | no ¬p | no ¬p' = lemma i ¬p
  where
  lemma :
    ∀ i → ¬ (100 < i) →
    ∀ i' → R91 (11 + i) i' →
    ∀ i'' → R91 i' i'' →
    i'' == 91
```

◊

At this point, it is worth remarking that the argument to *invariant* is not a Kleisli arrow of type $C \nrightarrow R$, but a monadic value of type *Free C R* (*R c*). The basic reason for this is that it is possible to do induction on values of type *Free C R a*, and we need this induction to check that the continuations, of the form *R c* → *Free C R a*, fit the definition. If we forget that this continuation is part of the recursive definition it accesses in a call, what we are left with is a computation with access to an oracle. In other words, it can perform an extra computation step where the program gets a response of type *R c'* for any given command *c'* : *C*. The relation passed to *invariant* specifies which responses may be received for each command. The observation that the *Free* monad allows for access to an oracle will be important in our treatment of effects in Chapter 4. In the coming sections, the main consequence will be that we define the semantics of values in the *Free* monad, not of Kleisli arrows for the *Free* monad.

Although many functions can be specified as relations between in- and output, there are oracles that are more naturally specified as a predicate transformer. In the next section, we will see how *invariant* generalizes to allow specifications in the form of predicate transformers.

3.3 Functions as predicate transformers

Note that we have defined *invariant* as a specific fold on the *Free* type, so it makes sense to generalize *invariant* by making it more resemble the general catamorphism. Each catamorphism on *Free* can be given by instantiating the arguments to the following *fold* function [Mei+91].

```
fold : ((c : C) → (R c → b) → b) → (a → b) → Free C R a → b
fold step pure (Pure x) = pure x
fold step pure (Step c k) = step c λ x → fold step pure (k x)
```

Since we want to compute properties of a computation in the *Free* monad, and propositions in intuitionistic type theory are represented by a type, we fix that the type b is equal to *Set*.

Definition 3.3.1. We define the specialisation of *fold* that returns values in *Set* to be *wp*.

```
wp : ((c : C) → (R c → Set) → Set) → (a → Set) → Free C R a → Set
wp = fold
```

Its name already hints that it corresponds to the weakest precondition operation in predicate transformer semantics. △

In *wp*, the meaning of the argument *pure* : $a \to$ *Set* is straightforward: it is the condition that each output value must satisfy. The meaning of the argument *step* : $(c : C) \to (R\ c \to Set) \to Set$ is somewhat less obvious: based on the types it takes an argument to a call and a predicate on the return type, and returns a proposition. However, if we do not have R depending on C, we can reorder the arguments to $(R \to Set) \to (C \to Set)$. In this light, *step* can be viewed as a predicate transformer for the recursive call: given a postcondition that the output should satisfy, return a precondition on the input that ensures the postcondition. Moreover, if the precondition returned by *step* is always the weakest precondition for the given postcondition, then by induction *wp step pure* gives the weakest precondition for any given postcondition *pure*. In Chapter 6, we give a formal proof that *wp* determines the weakest precondition, such that the postcondition holds on the result of running the computation.

Remark 3.3.2. In Definition 2.4.3, we defined that a precondition depends on an input state i and the postcondition depends on an input and output state i and o. Here, the *wp* function has a postcondition of type $a \to$ *Set*, and the precondition is of type *Set*. The intended interpretation for these types is that the computations we are considering are pure, so the input state i is trivial and is ignored, while the output state consists solely of the returned value. Later on, we will define the wp^S function, which allows for in- and output state.

An alternative interpretation is that *wp* specifies an interpreter for the *Free* monad: the input to this interpreter is a computation in *Free C R a*, while the output is a pure value of type a. A predicate transformer for such an interpreter would have the type $(a \to Set) \to (Free\ C\ R\ a \to Set)$, which is equivalent to the type of *wp step* for each choice of *step*. ◊

As claimed, *wp* is a generalisation of *invariant*, since we can define the latter equivalently as follows:

```
invariant' : ((c : C) → R c → Set) → (c : C) → Free C R (R c) → Set
invariant' rel c = wp (λ c' P → ∀ x' → rel c' x' → P x') (rel c)
```

Before we continue with verification, it is good to state the following lemma on the behaviour of *fold* and the bind operator _≫=_.

Lemma 3.3.3. *fold-bind* : $(S :$ *Free C R a*$)\ (f : a \to$ *Free C R b*$) \to \forall\ (pure : b \to c)\ step \to$
fold step pure $(S \gg\!= f) ==$ *fold step* $(\lambda\ x \to$ *fold step pure* $(f\ x))\ S$

Its proof goes by induction on the structure of S, where the case *Pure* is trivial and the case *Step* applies function extensionality to the induction hypothesis on $k\ x \gg\!= f$. The use of this lemma is to deal with verifying computations defined with _≫=_. If we want to prove something of the form *wp step pure* $(S \gg\!= f)$, and we have a result about the weakest precondition of S, the previous lemma allows us to use this result.

Now we have seen how to verify a recursive definition, given a relation between in- and output. Can we extend this verification to specifications in the form of a predicate transformer? We are looking for an equivalent predicate to *invariant* that has the form:

$$\mathit{verify} : (\mathit{pt} : (c : C) \to (R\ c \to \mathit{Set}) \to \mathit{Set}) \to ((c : C) \to \mathit{Free}\ C\ R\ (R\ c)) \to \mathit{Set}$$

If we can verify a program according to a predicate transformer, and program semantics are given as a predicate transformer, then we can verify that one program satisfies the specification given by another's semantics. This means we can verify that an optimised program has the same behaviour as a slower, but more obviously correct, implementation. For this, we need to figure out which postconditions and predicate transformers to pass to *wp*. From a predicate transformer, it is not obvious how to determine a postcondition, since the predicate transformer uses postconditions as input instead of giving them as output. The solution can be found in the refinement calculus.

The fundamental idea of the refinement calculus is to relate two predicate transformers using the *refinement* relation.

Definition 3.3.4 ([Mor98])**.** A predicate transformer pt_1 *is refined by* pt_2 if for each postcondition, the precondition given by pt_1 is stronger than the precondition given by pt_2. Formally:

$$\begin{aligned}&\_\sqsubseteq\_ : (\mathit{pt}_1\ \mathit{pt}_2 : (c : C) \to (R\ c \to \mathit{Set}) \to \mathit{Set}) \to \mathit{Set}\\ &\mathit{pt}_1 \sqsubseteq \mathit{pt}_2 = \forall\ c\ P \to \mathit{pt}_1\ c\ P \to \mathit{pt}_2\ c\ P\end{aligned}$$

△

Again, this definition assumes that the programs represented by the predicate transformers are pure, so there is no state apart from argument and return value. In Section 4.4, we define a refinement relation for stateful programs.

Thus, to verify a recursive definition is correct according to a predicate transformer, we calculate its own predicate transformer using *wp*, then check that this is a refinement of the original predicate transformer:

$$\mathit{verify}\ \mathit{pt}\ f = \mathit{pt} \sqsubseteq \lambda\ c\ P \to \mathit{wp}\ \mathit{pt}\ P\ (f\ c)$$

As in the imperative refinement calculus, we use mutual refinement to express that the predicate transformers denote the same computational effects.

Definition 3.3.5. Two predicate transformers pt_1 and pt_2 are *equivalent* if they refine each other.

$$\begin{aligned}&\_\equiv\_ : (\mathit{pt}_1\ \mathit{pt}_2 : (c : C) \to (R\ c \to \mathit{Set}) \to \mathit{Set}) \to \mathit{Set}\\ &\mathit{pt}_1 \equiv \mathit{pt}_2 = (\mathit{pt}_1 \sqsubseteq \mathit{pt}_2) \times (\mathit{pt}_2 \sqsubseteq \mathit{pt}_1)\end{aligned}$$

△

For example, suppose we want to verify that $\mathit{quicksort} : \mathit{List}\ \mathbb{N} \nrightarrow \mathit{List}\ \mathbb{N}$ is a correct sorting algorithm. The insertion sort algorithm $\mathit{insertionsort} : \mathit{List}\ \mathbb{N} \to \mathit{List}\ \mathbb{N}$ is structurally recursive on the input list, so we can use it as a reference implementation. For a formal verification of *quicksort*, we can give a term of type $\mathit{verify}\ (\lambda\ \mathit{xs}\ P \to P\ (\mathit{insertionsort}\ \mathit{xs}))\ \mathit{quicksort}$.

Now we can verify programs by comparing them to a reference implementation, but the same refinement relation works for verifying with respect to specifications. We can copy the definition from the refinement calculus.

Definition 3.3.6. A *specification* for a computation returning a value in a consists of a pair of pre- and postcondition, represented by the $\mathit{Spec}\ a$ data type.

```
record Spec (a : Set) : Set where
  constructor [_,_]
  field
    pre  : Set
    post : a → Set
```

△

Since a pure computation (represented as a value of some type a) does not involve any input state, the precondition is simply a proposition, while the postcondition only refers to the returned value.

However, we are usually concerned with specifying functions instead of values. While we can set a to be equal to a function type, it is more natural to let the precondition depend on the input type of the function. We will define an auxiliary type $Spec^F$ for specifications of such functions, expressed in terms of the $Spec$ type. Compare this to the construction of $\_ \nrightarrow \_$ from $Free$.

$$
\begin{aligned}
&Spec^F : (C : Set)\,(R : C \to Set) \to Set \\
&Spec^F\ C\ R = (c : C) \to Spec\ (R\ c) \\
&[\_,\_]^F : (pre : C \to Set)\,(post : (c : C) \to R\ c \to Set) \to Spec^F\ C\ R \\
&[\ pre\ ,\ post\ ]^F\ c = [\ pre\ c\ ,\ post\ c\ ]
\end{aligned}
$$

We want to define that a computation satisfies this specification if for all input values c such that $pre\ c$ holds, all potential output values x satisfy $post\ c\ x$. Thus, a predicate $P : R\ c \to Set$ holds on the output of each computation that satisfies the specification if $pre\ c$ holds and $post\ c\ x$ implies $P\ x$. This gives a predicate transformer for specifications:

$$
\begin{aligned}
&wpSpec : Spec\ a \to (a \to Set) \to Set \\
&wpSpec\ [\ pre\ ,\ post\ ]\ P = Pair\ pre\ (\forall\ x \to post\ x \to P\ x) \\
&wpSpec^F : Spec^F\ C\ R \to (c : C) \to (R\ c \to Set) \to Set \\
&wpSpec^F\ PQ\ c = wpSpec\ (PQ\ c)
\end{aligned}
$$

Checking that a recursive definition satisfies this specification is just a matter of filling in this predicate transformer in the *verify* function.

$$
\begin{aligned}
&specSatisfied : ((c : C) \to Spec\ (R\ c)) \to ((c : C) \to Free\ C\ R\ (R\ c)) \to Set \\
&specSatisfied\ PQ = verify\ (wpSpec^F\ PQ)
\end{aligned}
$$

Coming back to our remark that Kleisli arrows for the $Free$ monad represent generally recursive functions, while monadic values represent any computation with access to an oracle, we explain verification of generally recursive definitions in terms of the semantics of computations that can access an oracle. The semantics of such computations cannot be given without the semantics of the oracle, in our case the argument *step* of the *wp* function. In verification of generally recursive definitions, the semantics passed to *wp* are exactly those we want to verify against. Compare this to the situation for the *while* loop in imperative programming: an invariant is required to give the predicate transformer semantics of a *while* statement. Here too, the semantics of a recursive definition cannot be given without already assuming some part of the semantics.

Sometimes, we will check that the postcondition P holds for a program S, i.e. give an element for $wp\ pt\ P\ S$, to express that a computation is valid. We can view this as a special case of verifying the program S with respect to a specification, namely the specification $[\top\ ,\ P]$ with trivially true precondition and postcondition P. Showing that the program S is valid with respect to this specification is defined to be giving an element $\top \to wp\ pt\ P\ S$, which is equivalent to finding an element of $wp\ pt\ P\ S$.

3.4 Termination and the petrol-driven semantics

Up to now, we have only looked at partial correctness of recursive definitions, where we check that all output satisfies a postcondition, if there is output at all. Let us now talk about total correctness, where we require that the computation finishes within a finite time. We make this explicit by adding a counter to the semantics, which is decremented by each recursive call. If this counter reaches zero and a recursive call is attempted, the computation fails. This approach to semantics of recursion is called the petrol-driven semantics by McBride [McB15].

Suppose we are given a recursive definition, written as $f : C \nrightarrow R$ for some C and R, and we want to verify that some (sub-)computation $S : Free\ C\ R\ a$ produces a result that also satisfies a postcondition

$P : a \to Set$. There are two ways to implement this in the petrol-driven semantics. The first way is to compute a value in the *Maybe* monad and then lift P to the type $Maybe\ a \to Set$. In other words, we run the computation for the given number of steps and then check that there is output, and that this output satisfies the postcondition. The second way uses predicate transformers instead of directly running the computation.

Definition 3.4.1. Fix a predicate transformer pt that depends on the number of computation steps still remaining. The *petrol-driven semantics* of a computation S are given by folding the predicate transformer over this computation, making sure to decrease the number of remaining steps each time.

```
petrol : (pt : (c : C) → (R c → ℕ → Set) → ℕ → Set)
  (P : a → ℕ → Set) (S : Free C R a) → ℕ → Set
petrol pt = fold (λ { c P' 0 → ⊥; c P' (Succ n) → pt c P' n })
```

△

The definition of *petrol* still depends on a predicate transformer, but we can use *petrol* itself to transform the predicate. This is the case because the number of steps remaining is strictly decreasing. In other words, the following recursive definition is well-founded by induction on n.

```
terminates-with : (f : C ↬ R) (S : Free C R a) (P : a → ℕ → Set) → ℕ → Set
terminates-with f S P n = petrol (λ c P n' → terminates-with f (f c) P n') P S n
```

However, Agda's termination checker does not accept this definition. If we make the recursion explicit (and use the *fold-bind* lemma), the following modified definition is accepted by the termination checker:

Definition 3.4.2. Given a recursive definition $f : C \leadsto R$, the petrol-driven semantics of a computation S that calls f are equivalently defined as:

```
terminates-with : (f : C ↬ R) (S : Free C R a) (P : a → Set) → ℕ → Set
terminates-with f (Pure x)   P n       = P x
terminates-with f (Step c k) P Zero    = ⊥
terminates-with f (Step c k) P (Succ n) = terminates-with f (f c >>= k) P n
```

If for all c, *terminates-with* $f\ (f\ c)\ P\ n$ holds for some $n : \mathbb{N}$, we have that the recursive definition f is totally correct for the postcondition P. △

In contrast to partial correctness, it is possible to give semantics for general recursion without specifying some form of invariant. We can do this because *terminates-with* calculates the predicate transformer recursively, with the base case being $\bot$.

A disadvantage of defining total correctness in this way is that we are really mixing two concerns: the first concern is partial correctness, which asserts that all output conforms to the specification, and the second is termination, which asserts that the computation always results in output. We split up these concerns in the following definition.

Definition 3.4.3. Let f be a recursive definition used in a computation S. We say S *terminates in n steps* if *terminates-with* holds for a trivially true postcondition. A recursive definition $f : C \leadsto R$ *terminates in the petrol-driven semantics* if for all $c : C$ there is a n, such that $f\ c$ terminates in n steps. Formally:

```
terminates-in : (f : C ↬ R) (S : Free C R a) → ℕ → Set
terminates-in f S = terminates-with f S (λ _ → ⊤)

termination : (f : C ↬ R) → Set
termination f = ∀ c → ∃ (n : ℕ) → terminates-in f (f c) n
```

△

As with the total correctness of *terminates-with*, termination allows us to give semantics without requiring a predicate transformer for the recursive calls. The difference is that now, the semantics are a consequence of the definition, instead of interwoven with them. Because the recursive calls terminate, it is possible to evaluate the computation by unfolding the definition repeatedly.

```
evaluate : (f : C ↬ R) (S : Free C R a) (n : ℕ) → terminates-in f S n → a
evaluate f (Pure x)   n          term-S = x
evaluate f (Step c k) Zero       ()
evaluate f (Step c k) (Succ n) term-S = evaluate f (f c >>= k) n term-S
```

Applying a postcondition to the output of evaluation gives rise to a predicate transformer for f. We introduce two versions of the semantics, where *pt-for-term'* gives the predicate transformer semantics of an arbitrary computation using f, while *pt-for-term* gives the semantics of f itself.

```
pt-for-term' : (f : C ↬ R) (S : Free C R a) (n : ℕ) → terminates-in f S n → (a → Set) → Set
pt-for-term' f S n term-S P = P (evaluate f S n term-S)
pt-for-term : (f : C ↬ R) → termination f → (c : C) → (R c → Set) → Set
pt-for-term f term c P = pt-for-term' f (f c) (Sigma.fst (term c)) (Sigma.snd (term c)) P
```

We said that total correctness is a combination of termination and partial correctness, and we will make this precise in the following two theorems. The first theorem relates refining a specification in the *pt-for-term* semantics with partial correctness for that specification. The second theorem shows that satisfying a postcondition in the *terminates-with* semantics follows from termination combined with partial correctness in the *pt-for-term* semantics.

Theorem 3.4.4. *Suppose $f : C \nrightarrow R$ is a recursive computation that terminates. If it is partially correct with respect to a predicate transformer, this predicate transformer is refined by the semantics given by the termination of f.*

```
refine-pt-for-term : (f : C ↬ R) (pt : (c : C) → (R c → Set) → Set) →
  (term : termination f) → verify pt f → pt ⊑ pt-for-term f term
```

Proof. Writing out the definition of refinement and *pt-for-term*, it suffices to show for all $c : C$ and $P : R\ c \to Set$ that *wp pt P (f c)* implies *pt-for-term' f (f c) n term-n P*, where *term-n* is a proof that $f\ c$ terminates in n steps. We perform induction on n, combined with case distinction on the structure of $f\ c$. The case *Pure x* follows directly, since *wp* and *pt-for-term'* directly apply the postcondition to x.

For the case *Step c' k'*, we perform one evaluation step, so we need to show *pt-for-term' f (f c' >>= k') n term-n P*. This is exactly the conclusion of the induction hypothesis, which has the assumption *wp pt P (f c' >>= k')*. Because of the *fold-bind* lemma, this assumption is equal to *wp pt* (λ *x* → *wp pt P (k' x)*) *(f c')*. The partial correctness of f makes it sufficient to show *pt c'* (λ *x* → *wp pt P (k' x)*), which is equivalent to the assumption *wp pt P (Step c' k')* of the current case. □

Theorem 3.4.5. *If a recursive definition terminates and is partially correct for a given postcondition, it is totally correct for the postcondition.*

```
correct-from-sound-and-complete : (f : C ↬ R) (P : (c : C) → R c → Set) →
  (term : termination f) → (∀ c → wp (pt-for-term f term) (P c) (f c)) →
  ∀ c → ∃ (n : ℕ) → terminates-with f (f c) (P c) n
```

Proof. The proof goes by simultaneous induction on the number of steps remaining n and the structure of $f\ c$. The base cases are immediate, but for *Step c k* it is more difficult: we must show the weakest precondition (under *pt-for-term*) of computation of the form *Step c k* is equivalent to the weakest precondition of $f\ c >\!\!>\!\!= k$, but the two might have a different (but sufficient) number of computation steps remaining. Moreover, the

semantics of *Step c k* requires an extra application of *pt-from-term* compared to *f c* $\gg=$ *k*, and we need to prove that this difference does not matter.

Using structural induction, we can show the following lemmas, which may be applicable to other situations.

```
termination-bind : ∀ (S : Free C R a) (k : a → Free C R b) n →
  terminates-in f (S >>= k) n → terminates-in f S n
pt-for-term-independence : ∀ P (S : Free C R a) n n' term-n term-n' →
  pt-for-term' f S n term-n P == pt-for-term' f S n' term-n' P
pt-for-term-equality : ∀ P (S S' : Free C R a) n term-n (p : S == S') →
  pt-for-term' f S n term-n P ==
  pt-for-term' f S' n (subst (λ S → terminates-in f S n) p term-n) P
```

This solves the problem of the varying number of computation steps, but we also need to harmonize the different number of applications of *pt-from-term*. For that, we have the following two lemmas:

```
pt-for-term-wp : ∀ P (S : Free C R a) n term-n →
  pt-for-term' f S n term-n P == wp (pt-for-term f term) P S
pt-for-term-bind : ∀ P (S : Free C R a) (k : a → Free C R b) n term-n →
  pt-for-term' f (S >>= k) n term-n P ==
  pt-for-term' f S n (termination-bind S k n term-n) (λ x → wp (pt-for-term f term) P (k x))
```

The proofs of these lemmas are given by mutual recursion on the structure of the computations, where *pt-for-term-wp* uses *pt-for-term-bind* to collect all continuations into one call to *wp*, while *pt-for-term-bind* uses *pt-for-term-wp* to add a call to *wp* if *S* is a *Pure* value.

With these lemmas, we conclude that the weakest precondition of *Step c k* is the same as the weakest precondition of *f c* $\gg=$ *k*, allowing us to continue induction on *f c* when we encounter a *Step*. □

Example 3.4.6. Let us return to our running example: verifying *f91*. To show *f91* terminates, we can prove the following lemma:

```
f91-correct : ∀ i → Σ ℕ (terminates-with f91 (f91 i) (R91 i))
```

The proof applies *correct-from-sound-and-complete* to the partial correctness proof of *f91* in Example 3.2.2 and a proof of termination. ♢

3.5 Termination and a descending chain condition

The petrol-driven semantics are based on a syntactic argument: we know a computation terminates because expanding the call tree will eventually result in no more *call*s. Termination can also be defined based on a well-foundedness argument, such as the size-change principle of Agda's termination checker. Thus, we want to say that a recursive definition is well-founded if all recursive calls are made to a smaller argument, according to a well-founded relation.

Definition 3.5.1 ([Acz77])**.** In intuitionistic type theory, we say that a relation _≺_ on a type *a* is well-founded if all elements *x* : *a* are *accessible*, which is defined by (well-founded) recursion to be the case if all elements in the downset of *x* are accessible.

```
data Acc (_≺_ : a → a → Set) : a → Set where
  acc : (∀ y → y ≺ x → Acc _≺_ y) → Acc _≺_ x
```

△

To see that this is equivalent to the definition of well-foundedness in set theory, recall that a relation _≺_ on a set *a* is well-founded if and only if there is a monotone function from *a* to a well-founded order. Since

all inductive data types are well-founded, and the termination checker ensures that the argument to *acc* is a monotone function, there is a function from $x : a$ to $Acc\ \_\prec\_\ x$ if and only if $\_\prec\_$ is a well-founded relation in the set-theoretic sense.

The condition that all calls are made to a smaller argument is related to the notion of a loop *variant* in imperative languages. While an invariant is a predicate that is true at the start and end of each looping step, the variant is a relation that holds between successive looping steps.

Definition 3.5.2. Given a recursive definition $f : C \twoheadrightarrow R$, a relation $\_\prec\_$ on C is a recursive *variant* if for each argument c, and each recursive call made to c' in the evaluation of $f\ c$, we have $c' \prec c$. Formally:

```
variant : (f : C ↠ R) (pt : (c : C) → (R c → Set) → Set) → (C → C → Set) → Set
variant f pt _≺_ = ∀ c → variant' c (f c)
  where
  variant' : (c : C) (S : Free C R a) → Set
  variant' c (Pure x) = ⊤
  variant' c (Step c' k) = Pair (c' ≺ c) (pt c' λ x → variant' c (k x))
```

△

Note that *variant* depends on the semantics *pt* we give to f. We cannot derive the semantics in *variant* from the structure of f, since we do not yet know whether f terminates. Using *variant*, we can define another termination condition:

Definition 3.5.3. A recursive definition f is *well-founded* if it has a variant that is well-founded.

```
record Termination (f : C ↠ R) (pt : (c : C) → (R c → Set) → Set) : Set where
  field
    _≺_ : C → C → Set
    w-f : ∀ c → Acc _≺_ c
    var : variant f pt _≺_
```

△

A generally recursive function that terminates in the petrol-driven semantics is also well-founded, since a variant is given by the well-order $\_<\_$ on the amount of fuel consumed by each call. The converse also holds: if we have a descending chain of calls *cs* after calling f with argument c, we can use induction on the type $Acc\ \_\prec\_\ c$ to bound the length of *cs*. This bound gives the amount of fuel consumed by evaluating a call to f on c.

Example 3.5.4. Let us show the well-foundedness of the Ackermann function. Recall that this function is well-founded because each call either decreases the first argument, or keeps the first argument the same and decreases the second argument. This suggests that the variant we need is the lexicographic order on $Pair\ \mathbb{N}\ \mathbb{N}$.

```
data _<lex_ : Pair ℕ ℕ → Pair ℕ ℕ → Set where
  Fst : a < c → (a , b) <lex (c , d)
  Snd : b < d → (a , b) <lex (a , d)
```

The predicate transformer we use in the proof is very strong: the precondition is that the postcondition holds for all return values of the correct type. Still, showing that the lexicographic order is the variant is straightforward.

```
ackermann-terminates : Termination ackermann (λ c P → ∀ x → P x)
```

To show the lexicographic order is well-founded, we use helper lemmas to perform induction on the second element of the pair. The base case is *acc-zero* (which performs induction on the first element of the pair), while *acc-succ* is the inductive case.

```
Termination.w-f ackermann-terminates (a , b) = acc-snd a (acc (acc-zero a)) b
  where
  acc-snd : ∀ a → Acc _<lex_ (a , 0) → ∀ b → Acc _<lex_ (a , b)
  acc-zero : ∀ a → ∀ y → y <lex (a , 0) → Acc _<lex_ y
  acc-succ : ∀ a b → Acc _<lex_ (a , b) → ∀ y → y <lex (a , Succ b) → Acc _<lex_ y
```

◊

3.6 Going from recursion to other effects

In this section, we will explain why we view general recursion as a prototype for algebraic effects. Recall that we defined the weakest precondition predicate transformer *wp* for an arbitrary computation $S : \mathit{Free}\ C\ R\ a$, without restricting S to be a sub-expression in some recursive definition. If we loosen this restriction, it is no longer necessary that the semantics of the *Step* operation can be implemented in a pure function. In other words, S can access any oracle that returns values of type $R\ c$ when called on values $c : C$. As long as the semantics of the oracle are given as a predicate transformer $(c : C) \to (R\ c \to \mathit{Set}) \to \mathit{Set}$, *wp* gives us the semantics of S.

To illustrate why the *Free* monad and predicate transformer semantics allow for more than giving semantics of pure functions, let us consider the predicate transformer used by the petrol-driven semantics, which can return a precondition of $\bot$, independent of the given postcondition. Since any pure function of type $(c : C) \to R\ c$ must return some value, there is no pure function that is *unable* to satisfy the postcondition $\lambda\ \_ \to \mathit{tt}$. In the next chapter, we will discuss how we can use predicate transformers to describe the semantics of side effects.

Chapter 4

Predicate transformer semantics for effects

As suggested in Section 3.6, the style of predicate transformer semantics we constructed for general recursion is also applicable to other side effects such as partiality and non-determinism. In this chapter, we will refactor and recapitulate these definitions, giving predicate transformer semantics for algebraic effects.

Definition 4.0.1. An *effect type* consists of a type $C : Set$ and a dependent type $R : C \to Set$:

```
record Effect : Set where
  constructor eff
  field
    C : Set
    R : C → Set
```

The intended semantics are that C is the type of commands or calls, so each different value of C corresponds to a different sort of side effect. For example, in the context of mutable state, we will define a distinct element of C for each different value that can be written to the variable. For a given $c : C$, the type $R\ c$ represents the type of responses or potential "internally observable" results, such as the value read out from a variable.

We redefine the *Free* type to depend on an effect type instead of the individual C and R.

```
data Free (e : Effect) (a : Set) : Set where
  Pure : a → Free e a
  Step : (c : Effect.C e) → (Effect.R e c → Free e a) → Free e a
```

△

We do not make this change for any proof-theoretic power but for convenience, especially when we combine effects as in Chapter 5. Importantly, the syntax of terms in the *Free* monad is the same as in Chapter 3 so it is trivial to port the results of the previous chapter.

4.1 Semantics for the *Free* monad

In Chapter 3, we discussed predicate transformer semantics for general recursion. Given a predicate transformer of the form $(c : Effect.C\ c) \to (Effect.R\ e\ c \to Set) \to Set$), we can use a fold over the type $Free\ Effect\ a$ to turn a postcondition of type $a \to Set$ into a precondition of type Set.

Definition 4.1.1. The *predicate transformers* for an effect e are given by elements of the following record type $PT\ e$:

```
record PT (e : Effect) : Set where
  field
    pt : (c : Effect.C e) → (Effect.R e c → Set) → Set
    mono : ∀ c P Q → (∀ x → P x → Q x) → pt c P → pt c Q
```

△

We include a requirement *mono*, short for *monotonicity*, since we will need it in later theorems. Intuitively, the monotonicity requirement holds for all well-behaved semantics: if a very strong postcondition is satisfied after a computation, surely weaker postconditions are satisfied too.

Definition 4.1.2. As before, the *weakest precondition* predicate transformer is defined by a fold over the predicate transformer for an effect.

```
fold : ((c : Effect.C e) → (Effect.R e c → b) → b) → (a → b) → Free e a → b
fold step pure (Pure x)   = pure x
fold step pure (Step c k) = step c λ x → fold step pure (k x)

wp : PT e → Free e a → (a → Set) → Set
wp step S P = fold (PT.pt step) P S
```

△

We have changed the order of the arguments to *wp* to give it the same form as the *pt* field in the *PT* record type.

Lemma 4.1.3. *The weakest precondition of a composition* $S \gg\!= f$ *is given by composing the weakest precondition transformers.*

```
compositionality : (pt : PT e) (S : Free e a) (f : a → Free e b) →
  wp pt (S >>= f) P == wp pt S (λ x → wp pt (f x) P)
```

This *compositionality* property is no more than a specialisation of the *fold-bind* lemma of the previous chapter, and the proof is completely analogous.

Definition 4.1.4 ([Mor98])**.** Predicate transformer semantics give rise to the notion of *refinement*, which we define more generally than in Definition 3.3.4, as:

```
_⊑_ : (pt1 pt2 : (a → Set) → Set) → Set
pt1 ⊑ pt2 = ∀ P → pt1 P → pt2 P
```

△

The refinement relation is defined between predicate transformers, most notably the predicate transformers for an effect e, given by elements of $PT\ e$, and the weakest precondition of a computation, given by *wp*. We will say "a computation refines another", when we more precisely mean "the weakest precondition predicate transformer for a computation refines that of another". We can also assign predicate transformer semantics to specifications, as in Chapter 3, allowing us to relate specifications and (purported) implementations. If $pt_1 \sqsubseteq pt_2$ holds, the predicate transformer pt_2 is "better" in some way than pt_1, where the notion of "better" arises from our choice of semantics.

It is straightforward to show that the refinement relation is both transitive and reflexive, giving the following lemma.

Lemma 4.1.5. *The refinement relation is a pre-order.*

```
⊑-refl : (pt : (a → Set) → Set) → pt ⊑ pt
⊑-refl pt P H = H
⊑-trans : (pt1 pt2 pt3 : (a → Set) → Set) → pt1 ⊑ pt2 → pt2 ⊑ pt3 → pt1 ⊑ pt3
⊑-trans pt1 pt2 pt3 r12 r23 P H = r23 P (r12 P H)
```

Moreover, the refinement relation between computations works well with composition (since *wp* does), allowing us to refine the left- or right hand side of a _≫=_.

Lemma 4.1.6. *A composition $S \gg\!= f$ is refined if one of its parts is refined:*

```
compositionality-left : (pt : PT e) (S S' : Free e a) (f : a → Free e b) →
  wp pt S ⊑ wp pt S' → wp pt (S ≫= f) ⊑ wp pt (S' ≫= f)
compositionality-right : (pt : PT e) (S : Free e a) (f f' : a → Free e b) →
  (∀ x → wp pt (f x) ⊑ wp pt (f' x)) → wp pt (S ≫= f) ⊑ wp pt (S ≫= f')
```

Proof. The proof of *compositionality-left* consists simply of applying the *compositionality* equality twice, once to $S \gg\!= f$ and once to $S' \gg\!= f$. To prove *compositionality-right*, we need that *pt* is monotone. If *wp pt S* were anti-monotone, for example if *pt* returns the negation of the postcondition, then $S \gg\!= f'$ is refined by $S \gg\!= f$ instead of vice versa. With monotonicity of *pt*, the proof follows from induction on the structure of *S*, since the *Pure* case is immediate and the *Step* case follows from applying monotonicity to the induction hypothesis. □

4.2 Example: partial computation

The first example of a computational effect we will consider is that of a partial computation, one which calls to an oracle which never returns.

Definition 4.2.1. A *partial computation* is represented by the effect *EPartial*, and has predicate transformer semantics given by *wpPartial*, as follows:

```
data CPartial : Set where
  Abort : CPartial
RPartial : CPartial → Set
RPartial Abort = ⊥
EPartial : Effect
EPartial = eff CPartial RPartial
ptPartial : PT EPartial
PT.pt ptPartial Abort P = ⊥
PT.mono ptPartial Abort P Q imp ()
wpPartial : Free EPartial a → (a → Set) → Set
wpPartial = wp ptPartial
```

△

There is a single command, *Abort*; there is no continuation after issuing this command, hence the type of valid responses is empty. It is sometimes convenient to define a smart constructor for failure:

```
abort : Free EPartial a
abort = Step Abort λ ()
```

A computation of type *Partial a* will either return a value of type *a* or fail, issuing the *abort* command. Note that the responses to the *Abort* command are empty; the smart constructor *abort* uses this to discharge the continuation in the second argument of the *Step* constructor. With the syntax in place, we can turn our attention to verifying programs using suitable predicate transformer semantics.

Example 4.2.2. We begin by defining a small stack-based language, where the stack only contains natural numbers.

```
Stack = List ℕ
```

There is an operator for pushing a natural number onto the stack and an operator for popping the top two numbers and pushing their sum. These operators are represented by the terms of type *Op*. A full program, of type *Ops*, consists of a list of elements of *Op*.

```
data Op : Set where
  Push : ℕ → Op
  Plus : Op
Ops = List Op
```

After a program is run, we will require that there is at least one value left on the stack. The top of the stack is the result of the program. There are various ways to define the semantics of a program of type *Ops*. We can define operational semantics as an inductively defined relation between operations, the state of the stack, and the displayed value.

```
data _,_⇒_ : Ops → Stack → ℕ → Set where
  Base : Nil , (n :: sp) ⇒ n
  Plus : ops , ((a + b) :: sp) ⇒ c → (Plus :: ops) , (a :: b :: sp) ⇒ c
  Push : ops , (a :: sp) ⇒ b       → (Push a :: ops) , sp ⇒ b
```

In this definition, we rule out the erroneous operation of popping from an empty stack by only including non-empty stacks in the relation.

Alternatively, we can give semantics by defining a *monadic* interpreter that evaluates the programs, using the *Free EPartial* monad to handle invalid operations. The *pop* function pops a value off the stack, or *abort*s if the stack is empty.

```
pop : Stack → Free EPartial (Pair ℕ Stack)
pop Nil       = abort
pop (x :: xs) = Pure (x , xs)
```

Using *pop*, it is straightforward to write the interpreter for *Ops*.

```
run : Ops →       Stack →  Free EPartial ℕ
run Nil           Nil      = abort
run Nil           (s :: sp) = Pure s
run (Push a :: ops) sp     = run ops (a :: sp)
run (Plus   :: ops) sp = do
  (a , sp') ← pop sp
  (b , sp'') ← pop sp'
  run ops (a + b :: sp'')
```

Now we can ask ourselves whether the relation and interpreter describe the same semantics. If *run* was a pure function, it would give rise to a relation between input and output values. In that case, we could check whether the two relations coincide. Since *run* uses the *EPartial* effect, we can use the *wpPartial* predicate transformer to compute a relation between in- and output.

```
run-semantics : (ℕ → Set) → Ops → Stack → Set
run-semantics P ops sp = wpPartial (run ops sp) P
```

Now that we have two ways of expressing the semantics of *Ops*, we should verify that they come down to the same thing.

```
run-soundness :    ∀ ops sp n → (ops , sp ⇒ n) → run-semantics (_== n) ops sp
run-completeness : ∀ ops sp n → run-semantics (_== n) ops sp → (ops , sp ⇒ n)
```

The proof consists of induction on the list of operations.

```
run-soundness Nil             (n :: sp)       n Base      = refl
run-soundness (Plus :: ops)   (a :: b :: sp) n (Plus H)  = run-soundness ops (_ :: sp) n H
run-soundness (Push a :: ops) sp              n (Push H)  = run-soundness ops (a :: sp) n H

run-completeness Nil             Nil          n ()
run-completeness Nil             (a :: sp)    .a refl = Base
run-completeness (Plus :: ops)   Nil          n ()
run-completeness (Plus :: ops)   (a :: Nil)   n ()
run-completeness (Plus :: ops)   (a :: b :: sp) n H = Plus (run-completeness ops (_ :: sp) n H)
run-completeness (Push a :: ops) sp           n H = Push (run-completeness ops (a :: sp) n H)
```

This example also illustrates that *wp* generalises the predicate transformer semantics used for imperative languages. The language *Ops* is a simple model of imperative programs, where the state consists of a *Stack*. By combining *wpPartial* and *run*, we have constructed predicate transformer semantics for each program in *Ops*. ◊

4.3 Example: non-determinism

A pure computation always has exactly one output value. An effectful computation might have a different number of potential output values, zero or multiple. We have already seen how to represent partial computations, which result in no output if they fail. In this section, we represent multiple potential outputs with the effect of binary choice, nondeterministically selecting between two options.

Definition 4.3.1. A *nondeterministic computation* is represented by the effect *ENondet*. The effect *ENondet* allows for two calls: *Fail* is equivalent to *Abort* of *EPartial*, and *Choice* allows for a nondeterministic choice between two alternatives.

```
data CNondet : Set where
  Fail : CNondet
  Choice : CNondet
RNondet : CNondet → Set
RNondet Fail = ⊥
RNondet Choice = Bool
ENondet : Effect
ENondet = eff CNondet RNondet
```

Again, we define smart constructors to make the definitions more concrete:

```
fail : Free ENondet a
fail = Step Fail λ ()
choice : Free ENondet a → Free ENondet a → Free ENondet a
choice alt1 alt2 = Step Choice λ b → if b then alt1 else alt2
```

There are multiple choices for the semantics of nondeterministic computations, of which we will discuss two: the semantics of *wpAny* correspond to the *angelic* oracle, where a computation is successful if any of the choices leads to success. We omit the proofs of monotonicity since they are trivial.

```
ptAny : PT ENondet
PT.pt ptAny Fail     P = ⊥
PT.pt ptAny Choice P = Either (P True) (P False)


wpAny : Free ENondet a → (a → Set) → Set
wpAny = wp ptAny
```

The semantics of *wpAll* are *demonic*, where a computation is successful if all of the choices lead to success.

```
ptAll : PT ENondet
PT.pt ptAll Fail      P = ⊤
PT.pt ptAll Choice P = Pair (P True) (P False)


wpAll : Free ENondet a → (a → Set) → Set
wpAll = wp ptAll
```

△

Example 4.3.2. For a given set of strings, a *generalized portmanteau* contains each string as a substring with overlap. For example, "refinelement" is a portmanteau of "element" and "refine", while "elementrefine" is not since the substrings do not overlap. A *portmantout* is a generalized portmanteau where each word in a given word list appears at least once. Since repetitions are allowed, long portmantouts are easy to find, so a question arises: what is the shortest portmantout for a wordlist [Mur15; RM16; Ren17]?

Here, we will consider a converse problem: what is the longest generalized portmanteau, not allowing for repetitions? We give a nondeterministic program for generating a repetition-free generalized portmanteau, which we will call *portplusieurs*. The implementation mirrors the portmantout algorithm by Murphy [Mur15]. We keep track of the unused words and an accumulator that holds a portmanteau under construction. As long as there are unused words, we iterate by nondeterministically selecting an arbitrary unused word and try to add it to the accumulator to produce a longer portmanteau. If we end up in a dead end where there is no unused word that overlaps, we return the accumulated result.

To implement the algorithm, we start by introducing a function *pruffix* that nondeterministically generates a portmanteau of two strings *xs* and *ys*, such that *xs* is the prefix and *ys* is the suffix. Note that *pruffix* may return multiple results. For example, "headed" and "deduction" result in "headeduction" and "headededuction".

```
hasPrefix : String → String → Bool
hasPrefix xs       Nil      = True
hasPrefix Nil      (y :: ys) = False
hasPrefix (x :: xs) (y :: ys) = x ≟ y ∧ hasPrefix xs ys
pruffix : String → String → Free ENondet String
pruffix Nil      ys = fail
pruffix (x :: xs) ys = choice
  (guard (hasPrefix ys (x :: xs)) (Pure ys))
  (pruffix xs ys >>= λ xys → Pure (x :: xys))
```

To ensure there is this overlap, *pruffix* makes use of the *guard* function; *guard p S* performs the computation *S* if the Boolean *p* is true, otherwise it *abort*s.

Generating a portplusieurs is then a case of repeatedly applying *pruffix* to an unused word. Since deleting a word from the list of unused words does not always result in a sub-expression of the original list, we use the type *Vec String n* of lists of a fixed length *n* to satisfy the termination checker, making the induction on the length *n* explicit.

```
portplusieurs : Vec String n → String → Free ENondet String
portplusieurs Nil          acc = Pure acc
portplusieurs unused@ (_ :: _) acc = choice
  (sampleVec unused >>= λ {(word , unused') → pruffix word acc >>= portplusieurs unused'})
  (Pure acc)
```

Here, *sampleVec* takes a vector and returns an arbitrary element, together with the vector with that element deleted from it.

In our example, we will use the words of the opening paragraph of *Moby-Dick* as our word list. Taking the first successful branch of *portplusieurs* results in "coffinteresthishmaelongrimindamphilosophicall", a string of 45 letters. With predicate transformer semantics, we can prove that we can do better:

```
longPortplusieurs : wpAny (portplusieurs moby-dick "call") (λ xs → length xs > 50)
```

The proof of *longPortplusieurs* consists of a sequence of *Inl* and *Inr* specifying the choices made, followed by a proof that the corresponding portmanteau is indeed longer than 50 characters. ◊

The angelic semantics of *wpAll*, as discussed in the previous example, are mostly seen in complexity theory. The semantics used in predicate transformer semantics for imperative programs, such as in the work of Dijkstra [Dij75a] and Morgan [Mor98], correspond to the semantics of *wpAll*. In the following example, we will use the refinement relation to verify a computation in the *wpAll* semantics.

Example 4.3.3. To illustrate equational reasoning and nondeterminism, Gibbons and Hinze [GH11] give a program that nondeterministically generates a permutation of a given list. We will develop a similar program to illustrate our approach. First of all, we need to define when two given lists are permutations of another. Our approach will be that of the Agda standard library, giving an inductive definition of keeping the first element in place or swapping it with the second, and closing this under reflexivity and transitivity.

```
data Permutation : List a → List a → Set where
  Refl  : Permutation xs xs
  Prep  : Permutation xs ys → Permutation (x :: xs) (x :: ys)
  Swap  : Permutation xs ys → Permutation (x :: y :: xs) (y :: x :: ys)
  Trans : Permutation xs ys → Permutation ys zs → Permutation xs zs
```

This allows us to define the specification of the permutation generating function.

```
permsSpec : Spec^F (List a) (List a)
permsSpec xs = [ ⊤ , Permutation xs ]
```

While Gibbons and Hinze [GH11] generate a permutation by nondeterministically choosing an element of the input list, and putting it on the head of a permutation of the remainder of the list, this definition is not structurally recursive on the input list. To satisfy Agda's termination checker, we will use an opposite approach: we insert the head of the input list in a nondeterministic location in a permutation of the tail of the input. The insertion is done by the function *insert*.

```
insert : a → List a → Free ENondet (List a)
insert x Nil         = Pure (x :: Nil)
insert x (x' :: xs) = choice (Pure (x :: x' :: xs)) (insert x xs >>= λ xs' → Pure (x' :: xs'))
```

This leads to the following definition of *perms*:

```
perms : List a → Free ENondet (List a)
perms Nil         = Pure Nil
perms (x :: xs) = perms xs >>= insert x
```

To verify that all possible outputs of *perms* are permutations, we prove that it refines the specification *permsSpec*. Since the definition of *perms* includes a call to itself and to a recursive function *insert*, we should have some way of writing the proof recursively. At first glance, we might be able to use one of the *compositionality* lemmas, since those deal with *wp* applied to the composition of two computations. However, those lemmas only relate computations to each other, not computations with specifications. When we introduce specifications as effects in Chapter 7 we solve that issue, but for now we need to introduce a new lemma. It states that a composition refines a specification, if we have a postcondition for the first part that is a precondition for the second part.

```
compositionalitySpec : ∀ pt pre mid post (S : Free e a) (f : a → Free e b) →
  wpSpec [ pre , mid ] ⊑ wp pt S → (∀ x → wpSpec [ mid x , post ] ⊑ wp pt (f x)) →
  wpSpec [ pre , post ] ⊑ wp pt (S >>= f)
compositionalitySpec pt pre mid post S f rS rF P (fst , snd) =
  coerce (sym (compositionality pt S f))
  (rS _ (fst , λ o H → rF o _ (H , snd)))
```

Since *perms* calls *insert*, the first step is to verify the behaviour of *insert*. For our purposes, it is sufficient to prove that inserting *x* into *xs* produces a permutation of *x* :: *xs*; we do not need to show that the elements of *xs* are kept in their original order. The proof follows the definition of *insert*, and uses induction on the input list to show that all alternatives satisfy the specification. As for *perms*, verification proceeds similarly up to the point we have to deal with the call to *insert*. The precondition to *insert* does not match the postcondition of the recursive call to *perms*. We use the transitivity of $\_ \sqsubseteq \_$ to change the pre- and postcondition to a fitting form.

```
permsSound : (xs : List a) → wpSpec (permsSpec xs) ⊑ wpAll (perms xs)
permsSound Nil P (fst , snd) = snd Nil Refl
permsSound (x :: xs) = compositionalitySpec ptAll _ (Permutation xs) _ (perms xs) (insert x)
  (permsSound xs)
  λ xs' → ⊑-trans _ _ _
    (λ { P (fst , snd) → tt , (λ xs'' H → snd _ (Trans (Prep fst) H)) })
    (insertSound x xs')
```

However, we have not finished the verification of *perms* at this point. Although we have shown that all the output of *perms* is a permutation of the input, i.e. the soundness of *perms*, we also need to show completeness: that each permutation occurs in the output. We can accomplish this by showing the converse refinement, that *perms* is refined by *permsSpec*. In the refinement calculus, programs are equivalent if the refinement relation holds in both directions. Again, we need a lemma for refinement between compositions and specifications.

```
compositionalitySpec' : ∀ pt pre mid post (S : Free e a) (f : a → Free e b) →
  wp pt S ⊑ wpSpec [ pre , mid ] →
  (∀ (P : b → Set) → (∀ x → mid x → wp pt (f x) P) → ∀ y → post y → P y) →
  wp pt (S >>= f) ⊑ wpSpec [ pre , post ]
compositionalitySpec' pt pre mid post S f rS rF P H =
  let postS = rS _ (coerce (compositionality pt S f) H)
  in Pair.fst postS , rF P (Pair.snd postS)
```

As with soundness, the completeness proof of *perms* goes by induction on the input list, using compositionality and a separate completeness proof of *insert* to perform the inductive step. The completeness proof of *insert* makes use of a lemma that states that *x* :: *xs* is a permutation of *ys* if and only if *x* ∈ *ys* and *xs* is a permutation of *ys* with this occurrence of *x* deleted. Apart from the details of this lemma, the proof follows quickly from case distinction.

```
permsComplete : (xs : List a) → wpAll (perms xs) ⊑ wpSpec (permsSpec xs)
permsComplete Nil P H = tt , permNil P H
permsComplete (x :: xs) = compositionalitySpec' ptAll _ (Permutation xs) _ (perms xs) (insert x)
  (permsComplete xs)
  (insertComplete x xs)
```

◊

4.4 Example: stateful computation

Up to now, we have only considered effects where the input state is trivial, and the output state consists only of the return value. Predicate transformer semantics can also be applied to non-trivial state, which is required for imperative programming. To incorporate a state of type s, we need to change the type of the pre- and postcondition, replacing Set with $s \to Set$. We already did a similar substitution in Section 3.4, to accommodate the fuel level used by the petrol-driven semantics.

Definition 4.4.1. The effect of *mutable state* for a single variable of type s is represented by the effect $EState\ s$. The variable can be read out with the command Get, and written to with the command Put.

```
data CState (s : Set) : Set where
  Get : CState s
  Put : s → CState s
RState : CState s → Set
RState Get = s
RState (Put _) = ⊤
EState : Set → Effect
EState s = eff (CState s) RState
```

The intended semantics of $EState$ are that it behaves exactly as the state monad. In the more precise words that we will introduce in Definition 6.1.1, the handlers of $EState$ map the smart constructors get and put to the respective operations of the state monad.

```
get : Free (EState s) s
get = Step Get Pure
put : s → Free (EState s) ⊤
put x = Step (Put x) Pure
```

We cannot directly define the semantics of stateful computations by giving an element of the $PT\ EState$ type, since the type of PT does not allow for non-trivial states in the pre- or postcondition. The solution we take is to define a new analogous type PT^S, and define the fold over such a predicate transformer to be wp^S.

```
record PTS (s : Set) (e : Effect) : Set where
  field
    pt : (c : Effect.C e) → (Effect.R e c → s → Set) → s → Set
    mono : ∀ c P Q → (∀ x t → P x t → Q x t) → ∀ t → pt c P t → pt c Q t
wpS : PTS s e → Free e a → (a → s → Set) → (s → Set)
wpS step S P = fold (PTS.pt step) P S
```

The PT^S type allows us to give the predicate transformer semantics of stateful computation by passing through the state for a Get operation, and updating it for a Put operation.

```
ptState : PTS s (EState s)
PTS.pt ptState Get      P t = P t t
PTS.pt ptState (Put t') P t = P tt t'
PTS.mono ptState Get      P Q imp t asm = imp t t asm
PTS.mono ptState (Put t') P Q imp t asm = imp tt t' asm
wpState : Free (EState s) a → (a → s → Set) → s → Set
wpState = wpS ptState
```

△

While *Free EState* allows for a single state variable of type s, we can support multiple variables by setting s to be the product of all the variables' types. Similarly, if we want to interpret an imperative language, we can set the state to contain the whole stack and heap, and update this state for each operation.

If we want to verify stateful computations, we should apply a similar change to the specification type, making the pre- and postcondition in that type non-trivial. The postcondition can refer to the input and output state, so we can express predicates like "the variable has been incremented by 1".

```
record Spec^S (s a : Set) : Set where
  constructor [_,_]^S
  field
    pre : s → Set
    post : s → a → s → Set
wpSpec^S : Spec^S s a → (a → s → Set) → s → Set
wpSpec^S [ pre , post ]^S P s = pre s × (∀ x s' → post s x s' → P x s')
```

Finally, we need to change the refinement relation to also take into account the state.

```
_⊑^S_ : (pt1 pt2 : (a → s → Set) → s → Set) → Set
pt1 ⊑^S pt2 = ∀ P s → pt1 P s → pt2 P s
```

As we can see, a somewhat unfortunate consequence of including the current state in predicates is that many definitions now come in two incompatible forms, one without access to the state and one that has access to the state. Moreover, to neatly handle multiple stateful variables, for example to deal with combinations of effects as in Chapter 5, we would have to re-introduce every definition using this new number of variables. In the remainder of this thesis, if we only give one form of a definition, we will tend to choose predicates with access to one mutable variable, allowing for more generality. The drawback to this choice is that sometimes the unused variables clutter up the definition, and that multiple mutable variables have to be handled by careful dealing with tuples. Since the programs in this thesis only require a single variable, we will mention a potential solution without developing it in detail.

Instead of a single mutable variable, we can have a list (or map) of variables accessed through references in the style of the language ML [Mil+97]. Moreover, we can compute the type of predicates based on the elements of the list: when the list contains two natural numbers, a precondition will have type $\mathbb{N} \to \mathbb{N} \to Set$, and the empty list corresponds to preconditions of type Set. Thus, the singleton list with one element of type s corresponds to the single mutable variable, and the empty list to the absence of mutable state, subsuming both forms at the expense of more complicated machinery.

Example 4.4.2. To show how to reason about stateful programs using weakest precondition semantics, we revisit a classic verification problem proposed by Hutton and Fulger [HF08]: given a binary tree as input, relabel this tree so that each leaf has a unique number associated with it. A typical solution uses the state monad to keep track of the next unused label. The challenge that Hutton and Fulger pose is to reason about the program, without expanding the definition of the monadic operations.

We begin by defining the type of binary trees:

```
data Tree (a : Set) : Set where
  Leaf : a → Tree a
  Node : Tree a → Tree a → Tree a
```

One obvious choice of specification might be the following:

```
relabelSpec : Tree a → Spec^S ℕ (Tree ℕ)
relabelSpec t = [ K ⊤ , relabelPost t ]^S
  where
    relabelPost : Tree a → ℕ → Tree ℕ → ℕ → Set
    relabelPost t s t' s' = (flatten t' == [s ... s + size t]) × (s + size t == s')
```

The precondition of this specification is trivially true regardless of the input tree and initial state; the postcondition consists of a conjunction of two auxiliary statements: first, flattening the resulting tree t' produces the sequence of numbers from s to $s + \mathit{size}\ t$, where t is the initial input tree; furthermore, the output state s' should be precisely $\mathit{size}\ t$ larger than the input state s.

We can now define the obvious relabelling function as follows:

```
relabel : Tree a → Free (EState ℕ) (Tree ℕ)
relabel (Leaf x)   = fresh >>= (Pure ∘ Leaf)
relabel (Node l r) =
  relabel l >>= λ l' →
  relabel r >>= λ r' →
  Pure (Node l' r')
```

The auxiliary function *fresh* increments the current state and returns its value.

Next, we would like to show that this definition satisfies the intended specification. To do so, we can use the wp^S function to compute the weakest precondition semantics of the relabelling function and formulate the desired correctness property:

```
correctness : (t : Tree a) → wpSpec^S (relabelSpec t) ⊑^S wpState (relabel t)
```

The proof mirrors the definition of *relabel*, as previous correctness proofs did: we do a case distinction on the tree, then use the compositionality properties to deal with the _>>=_ operators. Finally, we prove the equality of the postcondition as a lemma on natural numbers. ◊

In Example 4.2.2 we wrote an interpreter for a small imperative language, making use of the partiality effect. Looking at the definition of *run*, we can see that we could also express it as a stateful computation, where the state consists of the stack of values. Rewriting *run* to additionally use *EState Stack* as an effect would allow for more readable code, but we cannot use *Free* (*EState Stack*) to implement *run*: we also need the partiality of *EPartial*. Algebraic effects solve this conflict by allowing multiple effects to be combined in the same syntax. In the next chapter, we will describe how to assign semantics to combinations of effects.

Chapter 5

Combinations of effects using coproducts

In the previous chapters, we have investigated the predicate transformer semantics arising from the catamorphism of the *Free* monad. The *Free* monad allows us to write effectful programs in the style of algebraic effects.

5.1 Combining effects in programs

Combinations of effects in the *Free* monad are given by taking the free monad over the coproduct of the respective functors [WSH14]. In our framework, given two effects $e_1\ e_2 : \mathit{Effect}$, the coproduct consists of the coproduct of the type of calls, together with the unique map from the combined type of calls to *Set*, given by the response type for each alternative.

```
_:+:_ : (e1 e2 : Effect) → Effect
Effect.C ((eff C1 R1) :+: (eff C2 R2)) = Either C1 C2
Effect.R ((eff C1 R1) :+: (eff C2 R2)) (Inl c1) = R1 c1
Effect.R ((eff C1 R1) :+: (eff C2 R2)) (Inr c2) = R2 c2
```

Combining effects in this way allows us to reuse the results of the previous chapter, but also has some disadvantages. Most notably, using a coproduct to combine effects is semantically associative and commutative, but syntactically it is neither. A program in the monad *Free* $(e_1 :+: (e_2 :+: e3))$ cannot be directly composed with one in the monad *Free* $((e_2 :+: e_1) :+: e3)$. We solve the disadvantages in two steps.

The first step is to redefine *Free* to take a list of effect types instead of a single one.

Definition 5.1.1. For a list of effect types *es*, the *Free* monad on a type *a* is given by the following data type:

```
data Free (es : List Effect) (a : Set) : Set where
  Pure : a → Free es a
  Step : (i : eff C R ∈ es) (c : C) (k : R c → Free es a) → Free es a
```

△

The *Step* takes an index into the list of effects, expressed as a proof that the specific effect is in the list. When writing code, we pass this index as an instance argument, allowing for automatic inference of its value, analogously to using a type class. Thus, a smart constructor for e.g. the *fail* of nondeterminism will have a type of the form $\mathit{fail} : \{\!\{ \mathit{iND} : \mathit{ENondet} \in \mathit{es} \}\!\} \to \mathit{Free}\ \mathit{es}\ \bot$. This is, up to notational differences, the

same as the type of the smart constructor *fail* defined by Wu, Schrijvers, and Hinze. By using a list of effects instead of a tree of coproducts, there is no more syntactic problem of associativity.

The second step allowing for composing programs with different effects is a change in programming style. Instead of the list of effects being fixed by the type of an effectful computation, we parametrise the computation over the index of each effect in the list. Thus, we can add effects without needing to change the syntax. This means that each effectful program is in the *Free es* monad for large enough *es*, so we can always compose two of these programs, as long as we pass the correct indices. For example, the *fail* operation of nondeterminism will have type $fail : \{\!\{ iND : ENondet \in es \}\!\} \to Free\ es\ \bot$ instead of $fail : Free\ (ENondet :: Nil)\ \bot$, to allow for generality in the list of effects.

Example 5.1.2. Let us rewrite the *run* function for the simple stack-based language to make use of two effects; partiality and statefulness. We can reuse most of the definitions in Example 4.2.2, and only rewrite the values in the *Free* monad.

```
pop : {{ iP : EPartial ∈ es }} {{ iS : EState Stack ∈ es }} → Free es ℕ
pop = do
  (a :: sp) ← get
    where Nil → abort
  put sp
  Pure a
push : {{ iS : EState Stack ∈ es }} → ℕ → Free es ⊤
push a = do
  sp ← get
  put (a :: sp)
run : {{ iP : EPartial ∈ es }} {{ iS : EState Stack ∈ es }} → Ops → Free es ℕ
run Nil = pop
run (Push a :: ops) = do
  push a
  run ops
run (Plus :: ops) = do
  a ← pop
  b ← pop
  push (a + b)
  run ops
```

◊

5.2 Combining effect semantics

Recall from Definitions 3.3.1 and 4.1.2 that the weakest precondition predicate transformer is given as a fold of the *Free* monad over the predicate transformer. With the new definition of *Free*, folds are also indexed by the effects in the list.

```
fold : ((eff C R ∈ es) → (c : C) → (R c → b) → b) → (a → b) → Free es a → b
fold step pure (Pure x) = pure x
fold step pure (Step i c k) = step i c λ x → fold step pure (k x)
```

The argument *step* to *fold* describes the semantics of all effects simultaneously, but to fit with the combinatorial nature of the effects, we want to specify the semantics as a combination of the individual semantics of each effect. Since we put the effect types in a list, it makes sense to create a list of predicate transformers, and have the *step* argument look up the corresponding predicate transformer.

Definition 5.2.1. For a list *es* of effect types, the type of predicate transformer semantics for *es* is defined inductively:

```
data PTs : List Effect → Set where
  Nil  : PTs Nil
  _::_ : PT e → PTs es → PTs (e :: es)
```

The weakest precondition predicate transformer of a computation in the monad *Free es* is given by folding over the corresponding predicate transformer at each step.

```
wp : PTs es → Free es a → (a → Set) → Set
wp pts S P = fold (lookupPT pts) P S
```

Here, *lookupPT* takes an index into the list of effect types, and returns the corresponding predicate transformer. △

We can give different semantics to calls to the same effect type, by including two different copies of the effect type in *es*, and two different predicate transformers in *pts*. This also implies that to apply *wp* to a program, we need to fix the order in which the effect types occur in the list *es*. We will see this pattern in the rest of our work: first introduce a program that takes any list of effect types *es*, as long as it allows for the needed effects, then choose a value for *es* and verify the instantiation of the program for the given list.

The semantics are applied in the order they occur in the program, not in the order they occur in the list of predicate transformers. For instance, the predicate transformers *ptAll* :: *ptAny* :: *Nil* for *ENondet* :: *ENondet* :: *Nil* will not necessarily result in a precondition that is a conjunction of disjunctions.

Example 5.2.2. To make the way in which predicate transformers combine more explicit, we will look at the combination of the *ptAll* and *ptAny* predicate transformers in more detail. For a more practical example, verifying the *run* program of Example 5.1.2, we first need to also introduce combinations of stateful effect semantics. We start the current, simpler example by introducing smart constructors for angelic and demonic choice, both renamings of the semantics-independent nondeterministic *choice*. We fix the order of effects at this point to make clear the operations have distinct semantics.

```
angelic = choice {{ ∈Tail ∈Head }}
demonic = choice {{ ∈Head }}
```

The semantics we assign to these effects are a combination of *ptAll* and *ptAny*:

```
pts = ptAll :: ptAny :: Nil
```

Let us apply the *wp* function to a few programs to show how the combinations interact. First of all, if we do not use *angelic* and *demonic* in the same program, the weakest precondition is the same as for the situation without combinations:

```
ex1 : wp pts (angelic (Pure 1) (angelic (Pure 2) (Pure 3))) P == Either (P 1) (Either (P 2) (P 3))
ex2 : wp pts (demonic (Pure 1) (demonic (Pure 2) (Pure 3))) P == Pair (P 1) (Pair (P 2) (P 3))
```

Combining the *angelic* choice and *demonic* choice will result in a weakest precondition of the same form as the program.

```
ex3 : wp pts (angelic (Pure 1) (demonic (Pure 2) (Pure 3))) P == Either (P 1) (Pair (P 2) (P 3))
ex4 : wp pts (demonic (Pure 1) (angelic (Pure 2) (Pure 3))) P == Pair (P 1) (Either (P 2) (P 3))
```

This result is independent of reordering *ptAll* and *ptAny* in the list of effects, as long as we also reassign the indices for the *angelic* and *demonic* operations. ◊

5.3 Re-incorporating mutable state

In Section 4.4, we discussed the predicate transformer semantics for programs using a single mutable variable. The weakest precondition can still be written as a fold over a predicate transformer, but the predicates

now take into account the value of the variable. When we have a combination of effects and want to give semantics that include mutable state, the question arises whether we keep track of a list of variables, one for each effect, or the mutable state is the same for each predicate transformer. The first case allows for easy definition of multiple mutable variables, since to introduce a new variable can be done by adding a new state effect (and predicate transformer). The second case allows us more expressivity: we can carry through the state from one branch of nondeterministic computation to the next.

We choose to allow for a single mutable variable of an arbitrary type s, just as in the uncombined case. If we need multiple variables, we can set s to be a product type. This choice is better for expressivity but carries some notational burden. For a practical framework that allows for verifying arbitrary programs, different choices might be better. For a single mutable variable, we can modify 5.2.1 to use stateful predicate transformers in the PT^S type instead of the stateless predicate transformers in the PT type.

Definition 5.3.1. The type of stateful predicate transformers for state of type s and a list of effect types es are given by $PT^S\ s\ es$, as follows:

```
data PTs^S (s : Set) : List Effect → Set where
  Nil  : PTs^S s Nil
  _::_ : PT^S s e → PTs^S s es → PTs^S s (e :: es)
```

The weakest precondition of a stateful computation is given by folding over the corresponding predicate transformer at each step.

```
wp^S : PTs^S s es → Free es a → (a → s → Set) → s → Set
wp^S pts S P = fold (lookupPTS pts) P S
```

Here, $lookupPTS$ takes an index into the list of effect types, and returns the corresponding stateful predicate transformer. ∆

In the previous sections, we have defined stateless semantics in the PT types. To combine these with stateful semantics in the PT^S type, we define a helper function that adds the state by threading it through the computation unmodified.

```
addState : PT e → PT^S s e
PTS.pt (addState record { pt = pt; mono = mono }) c P t = pt c (λ x → P x t)
```

Example 5.3.2. In Example 5.1.2, the run function is an interpreter for a simple stack-based language. This function combines the effects of state and partiality, and adapts the function of Example 4.2.2, where we also verified the interpreter with respect to an inductively defined relation. Now that we have defined wp^S, we are ready for verification of the version that uses a combination of effects.

The weakest precondition semantics of run are given by applying wp^S with suitable predicate transformers. Although the postcondition P has access to the stack in the wp^S semantics, we will ignore this, to mirror the semantics given in Example 4.2.2. As discussed, it is at the point of verification that we fix the exact list of effects used in run, by specifying the predicate transformers.

```
run-semantics : (ℕ → Set) → Ops → Stack → Set
run-semantics P ops = wp^S
  (addState ptPartial :: ptState :: Nil)
  (run ⦃ ∈Head ⦄ ⦃ ∈Tail ∈Head ⦄ ops)
  (λ n _ → P n)
```

We defined the intended semantics of the stack-based language as an inductive relation $\_,\_ \Rightarrow \_$ between operations, the stack and the output value. As before, the correctness of run is specified by requiring that run outputs a given value if and only if this value is the result according to the relation.

$$\begin{aligned}&\mathit{run\text{-}soundness} &&: \forall\ \mathit{ops}\ \mathit{sp}\ n \rightarrow (\mathit{ops}\ ,\ \mathit{sp}\ \Rightarrow\ n) \rightarrow \mathit{run\text{-}semantics}\ (\_{==}\ n)\ \mathit{ops}\ \mathit{sp}\\ &\mathit{run\text{-}completeness} &&: \forall\ \mathit{ops}\ \mathit{sp}\ n \rightarrow \mathit{run\text{-}semantics}\ (\_{==}\ n)\ \mathit{ops}\ \mathit{sp} \rightarrow (\mathit{ops}\ ,\ \mathit{sp}\ \Rightarrow\ n)\end{aligned}$$

The proof these two statements is again a induction on the list of operations, and in fact results in proofs syntactically identical to that of the corresponding proofs in Example 4.2.2. ◊

5.4 Termination and combinations of effects

In the previous section, we have looked at combinations for multiple effects, but we have not re-introduced the general recursion of Chapter 3. This section will look at combining general recursion with other effects, and more specifically at termination in the presence of multiple effects.

First of all, we re-introduce notation for generally recursive functions as Kleisli arrows for *Free*. In Chapter 3, the notation $C \nrightarrow R$ abbreviates $(c\ :\ C) \rightarrow \mathit{Free}\ C\ R\ (R\ c)$. We cannot reuse this notation, since there are multiple effects in addition to general recursion. We incorporate the list of effect types as follows:

$$\begin{aligned}&\_\ \overset{-}{\underset{es}{\nrightarrow}}\ \_\ :\ (C\ :\ \mathit{Set})\ (\mathit{es}\ :\ \mathit{List\ Effect})\ (R\ :\ C \rightarrow \mathit{Set}) \rightarrow \mathit{Set}\\ &C \overset{es}{\nrightarrow} R\ =\ (c\ :\ C) \rightarrow \mathit{Free}\ (\mathit{eff}\ C\ R\ ::\ \mathit{es})\ (R\ c)\end{aligned}$$

In the previous sections, we let the order of the effects free, while now we explicitly choose that general recursion is the first effect. This was done for the practical consideration of simplifying the notation in this section, so that we do not have to repeatedly write the index of general recursion in the list of effects.

In Section 3.4, we were careful to state that a computation terminates in the petrol-driven semantics if unfolding the definition enough satisfies a trivially true postcondition. This is a semantic property, and should be distinguished from the syntactic property of containing no more recursive calls. When we deal with effects in combination with general recursion, these two properties are distinct. For instance, consider a recursive function that decrements a counter kept in the state, and calls itself if the counter is not yet equal to *0*. At each unfolding step, we have code of the form **if** $\mathit{counter}\ \not\equiv\ 0$ **then** $(\mathit{decrement}\ \gg\ \mathit{call})$ **else** *done*. The code will always syntactically incorporate a recursive call, even if this call is no longer made at a certain point of execution. The practical implication is that the equivalent to *terminates-in* takes a list of predicate transformers as arguments, to determine whether a call is actually made.

Definition 5.4.1. Let *es* be a list of effect types with (stateful) semantics given by $\mathit{pts}\ :\ PT^S\ s\ \mathit{es}$. Let S be a computation that calls a generally recursive function f, in addition to the effects of *es*. We say that S *terminates in the petrol-driven semantics with effects* es if unfolding f enough times satisfies a trivially true postcondition:

$$\begin{aligned}&\mathit{terminates\text{-}in}\ :\ (\mathit{pts}\ :\ \mathit{PTs}^S\ s\ \mathit{es})\ (f\ :\ C \overset{es}{\nrightarrow} R)\ (S\ :\ \mathit{Free}\ (\mathit{eff}\ C\ R\ ::\ \mathit{es})\ a) \rightarrow \mathbb{N} \rightarrow s \rightarrow \mathit{Set}\\ &\mathit{terminates\text{-}in}\ \mathit{pts}\ f\ (\mathit{Pure}\ x)\ n\ t\ =\ \top\\ &\mathit{terminates\text{-}in}\ \mathit{pts}\ f\ (\mathit{Step}\ \in\!\mathsf{Head}\ c\ k)\ \mathit{Zero}\ t\ =\ \bot\\ &\mathit{terminates\text{-}in}\ \mathit{pts}\ f\ (\mathit{Step}\ \in\!\mathsf{Head}\ c\ k)\ (\mathit{Succ}\ n)\ t\ =\ \mathit{terminates\text{-}in}\ \mathit{pts}\ f\ (f\ c\ \ggg\ k)\ n\ t\\ &\mathit{terminates\text{-}in}\ \mathit{pts}\ f\ (\mathit{Step}\ (\in\!\mathsf{Tail}\ i)\ c\ k)\ n\ t\ =\ \mathit{lookupPTS}\ \mathit{pts}\ i\ c\ (\lambda\ x \rightarrow \mathit{terminates\text{-}in}\ \mathit{pts}\ f\ (k\ x)\ n)\ t\end{aligned}$$

Note that this definition takes multiple state variables: one for the state shared by all effects, and one for the amount of computation steps left. Similarly to the situation in Chapter 3, we could write this as an application of the *fold* function, but this would not be accepted by Agda's termination checker. △

In contrast, if we do not use the predicate transformers, the case for $\mathit{Step}\ (\in\!\mathsf{Tail}\ i)\ c\ k$ has no natural way to produce a $x\ :\ R\ c$, so it must be the result of some quantifier. But this quantifier is not uniquely determined. Consider the *ENondet* effect, which has different semantics *ptAll* and *ptAny*. The *Choice* effect returns a value $x\ :\ \mathit{Bool}$ nondeterministically. The semantics of *ptAll* require that all potential output leads to success (i.e. termination), and the semantics of *ptAny* only require that one of them leads to termination. We cannot make the choice between the universal and existential quantifier without knowing the semantics of the operation. Hence we need the semantics to reason about termination.

Termination with respect to *ptAny* matches the definition of non-deterministic Turing machines in complexity theory, just as partial correctness also corresponds to the semantics of *ptAny*. Indeed, the original definition of a Turing machine does not even allow for the machine to reject input (corresponding to the *Fail* effect of *ENondet*). Instead, a failing branch avoids an accepting state (corresponding to returning a *Pure* value) by never terminating [Tur37].

We give an example that uses *terminates-in* to verify a generally recursive function with access to other effects in Subsection 8.1.4.

5.4.1 Well-founded recursion for combinations of effects

Instead of petrol-driven semantics, we can also use define termination based on a well-foundedness argument. The construction is analogous to Section 3.5. We find a relation that is a variant for the recursion, then prove this relation is well-founded. Based on this well-foundedness, all call trees have finite depth, so the computation must eventually terminate. The relation has access to the state, so the computation can also make progress by modifying the current state.

Definition 5.4.2. Let *es* be a list of effect types with stateful semantics given by $pts : PT^S\ s\ es$. Let f be a generally recursive function that has access to the effects of *es*. A relation $\_\prec\_$ on $C \times s$ is a recursive *variant* if for each argument c and state t, and each recursive call made to c' with state t' in the evaluation of $f\ c$, we have $(c' , t') \prec (c , t)$.

```
variant' : (pts : PTs^S s (eff C R :: es)) (f : C ↬^es R) (_≺_ : (C × s) → (C × s) → Set)
  (c : C) (t : s) (S : Free (eff C R :: es) a) → s → Set
variant' pts f _≺_ c t (Pure x) t' = ⊤
variant' pts f _≺_ c t (Step ∈Head c' k) t'
  = ((c' , t') ≺ (c , t)) × lookupPTS pts ∈Head c' (λ x → variant' pts f _≺_ c t (k x)) t'
variant' pts f _≺_ c t (Step (∈Tail i) c' k) t'
  = lookupPTS pts (∈Tail i) c' (λ x → variant' pts f _≺_ c t (k x)) t'

variant : (pts : PTs^S s (eff C R :: es)) (f : C ↬^es R) → ((C × s) → (C × s) → Set) → Set
variant pts f _≺_ = ∀ c t → variant' pts f _≺_ c t (f c) t
```

The recursive function f is well-founded if it has a variant that is well-founded. We package these requirements in the *Termination* data type.

```
record Termination (pts : PTs^S s (eff C R :: es)) (f : C ↬^es R) : Set where
  field
    _≺_ : (C × s) → (C × s) → Set
    w-f : ∀ c t → Acc _≺_ (c , t)
    var : variant pts f _≺_
```

△

We give an example that uses the *Termination* record to verify a generally recursive function with access to other effects in Subsection 8.2.5.

5.5 Handlers for combinations of effects

Apart from allowing combinations of effects, algebraic effects also feature the use of effect handlers as a program construct [WSH14; BP15]. These handlers catch the calls that are made to a set of effects and handle them by running some effect-specific code, with access to the command and continuation at that point. In the *Free* monad, effect handlers can be defined as a fold over the *Free* data type, applied to the scope in which they handle the effect. For example, we can define *try_catch_* as an effect handler for the *EPartial* effect, similar to the construct of the same name that is available in many programming languages.

```
try_catch_ : (S1 S2 : Free (EPartial :: es) a) → Free (EPartial :: es) a
try (Pure x)                catch S2 = Pure x
try (Step ∈Head Abort k) catch S2 = S2
try (Step (∈Tail i) c k)    catch S2 = Step (∈Tail i) c λ x → try (k x) catch S2
```

Incorporating effect handlers in predicate transformer semantics is simple in one way: we simply apply the effect handlers to their scope, then compute the weakest precondition for the result of handling the effects. In effect, we use the denotational semantics of the effect handlers to compute the predicate transformer semantics of a program that uses them.

We have not yet found a way to assign predicate transformer semantics to handlers, which could be used to verify handlers using refinement from a specification. Additionally, by handling effects in different orders, we should be able to give different semantics to the same syntax. For example, first handling nondeterminism and then mutable state will result in semantics that feed the output state of the first alternative into the input state of the second alternative, while handling state first and then nondeterminism results in semantics where each alternative sees the same input state. With the current description of combined effects, we have to explicitly specify (in)dependence between the state of alternatives in the semantics of *ENondet*.

Chapter 6

Consistency of predicate transformer semantics

In the previous chapters, we have shown that predicate transformer semantics are useful in reasoning about functional programming with effects. We have done this by showing that the weakest precondition predicate transformer arises naturally as a fold, or catamorphism, over the *Free* monad. Moreover, the weakest precondition allows for the refinement relation between two programs, or between program and specification, allowing for verification of programs in the style of the refinement calculus.

We should not only show that we can productively perform verification work with predicate transformer semantics, but also that its results are consistent. To use effects in a computation, we give a handler that returns a value in a suitable monad. Thus, for consistency of semantics we want to show that computing the weakest precondition is equivalent to applying a running the computation in a monad and applying a postcondition to it. First we show how a handler can be used to run algebraic effects in a specific monad, then we give conditions on the handler and on the predicate transformer for an effect, that ensure that the semantics given by the predicate transformer are consistent with the semantics given by the handler. Additionally, we compare verification using predicate transformer semantics to equational reasoning, a style of verification used in the context of functional programming.

6.1 Effect handlers

We will begin our discussion of handlers by considering a specific case of consistency. We have introduced the *EState* effect by stating it should have the semantics of the *State* monad. Let us state precisely what we mean by that, and then give the proof that this property holds.

To compare the *EState* effect and the *State* monad, we need to determine a common description of stateful computation. The *State* monad is "executable" in the sense that elements of *State s a* are functions. Thus, we can compute the outcome of running a stateful computation by applying a value in *State s a* to an initial state *t : s*. If we can translate a computation described by the *EState* effect to the *State* monad, we can determine the result of running the computation. This gives another semantics for *EState*, in terms of the *State* monad, which we want to show is consistent with predicate transformer semantics of the wp^S function.

```
runState : Free (EState s :: Nil) a → State s a
runState (Pure x) = Monad.return _ x
runState (Step ∈Head Get     k) = Monad.bind _ get     λ x → runState (k x)
runState (Step ∈Head (Put t) k) = Monad.bind _ (put t) λ x → runState (k x)
```

The format of *runState* is a fold over the *Free* monad, with the *Pure* case mapped to *return* and the *Step* ∈Head cases mapped to a function *get* or *put* composed with running the continuation. We say that *get* and *put* are handlers for the effect *EState* into the *State* monad, a notion we will formalise in Definition 6.1.1.

We want to show the predicate transformer semantics are consistent with the semantics of running a stateful computation. Since predicate transformer semantics are based on applying postcondition to the output, for the comparison we need to be able to *lift* a predicate, so we can apply it to the output of a computation in the *State* monad.

```
liftState : (a → s → Set) → (State s a → s → Set)
liftState P S t = uncurry P (S t)
```

With these definitions, consistency of the semantics of *State* comes down to a postcondition holding according to wp^S if and only if it holds according to *liftState* on the output of running the computation with *runState*. The notion is formally represented by a term of the following type:

```
consistencyState : (P : a → s → Set) (S : Free (EState s :: Nil) a) (t : s) →
  wp^S (ptState :: Nil) S P t ⟺ liftState P (runState S) t
```

The proof of consistency is a simple induction on the program structure.

```
consistencyState P (Pure x) t = ⟺ −refl
consistencyState P (Step ∈Head Get k) t = consistencyState P (k t) t
consistencyState P (Step ∈Head (Put t') k) t = consistencyState P (k tt) t'
```

6.1.1 Running effects with handlers

Let us now move from a single effect *EState* to a list of effects *es*. To run a program in the *Free* monad, i.e. turn the syntax into a form that we can execute on the computer, we must show how each effect *e* in *es* can be executed. We will assume that there is a monad *m* that represents the full range of effects available in the system, for instance Haskell's *IO* monad. Turning each operation of the form *Step i c k* into an executable form is performed by a *handler* for *e* in the surrounding scope, which gives an operation in the monad *m* for the command *c* and continuation *k*. In essence, a handler gives an interpreter the effects in the part of the program it handles. The semantics of a computation *S* : *Free es a* are *algebraic* if for all continuations *k* : *a* → *Free es b*, we have *run* (*bind S k*) == *bind* (*run S*) (λ *x* → *run* (*k x*)). If all operations are algebraic, we can run the computation in a single monad [PP03]. Note that there are many useful effects that are not algebraic. For instance, an exception handler (as used in a *try T catch C* block) is not algebraic, since exceptions thrown in the continuation *k* are not caught in the program (*try T catch C*) >>= *k* and they are caught in the program *try* (*T* >>= *k*) *catch C*.

Assuming that all operations are algebraic allows us to only specify the semantics of each individual effect, and let the interaction between effects arise from the algebraicity. Since *bind* (*return x*) *k* is equal to *k x*, we have that running a *Pure* value is done by returning this value. Similarly, since *bind* (*Step i c Pure*) *k* is equal to *Step i c k*, we have that it suffices to give the semantics of *Step i c Pure* to give the semantics of all computation steps.

Definition 6.1.1. For a list of algebraic effects *es*, a list of *handlers* into a monad *m* is given by the following data type:

```
data Handlers (m : Set → Set) : List Effect → Set where
  Nil : Handlers m Nil
  _::_ : ((c : C) → m (R c)) → Handlers m es → Handlers m (eff C R :: es)
```

To *run* a program using these handlers, we apply the handler at the given index for each *Step i c k*, then run the continuation, and when we arrive at a pure value, we return it.

```
run : Monad m → Handlers m es → Free es a → m a
run M hs (Pure x) = Monad.return M x
run M hs (Step i c k) = Monad.bind M (lookupHandler hs i c) λ x → run M hs (k x)
```

Here, *lookupHandler* takes a list of handlers and an index and returns the corresponding handler, similar to the function *lookupPTS* used for wp^S. △

While we assume that there is one monad *m* that provides all effects we want to run, it is also possible to have each handler come with its own monad. The issue with introducing multiple monads is that in general the composition of monads is not itself a monad. Thus, if we combine a handler *h1* into a monad *m1* (say *State s*) with another handler *h2* into a monad *m2* (say *List*), we need to ensure *m1* ∘ *m2* is a monad (and *State s* ∘ *List* is not). A remedy for the issue is to associate a monad *transformer* to each handler. For simplicity, we assume that the combined monad has already been determined, for example through composing monad transformers, and focus only on the semantics for this combined monad.

6.1.2 Consistency with respect to handlers

Now that we have defined the semantics for running algebraic effects, we are ready for the main theorem of this chapter, giving sufficient conditions for consistency of predicate transformer and handlers.

Theorem 6.1.2. *Fix a list of effects* es*, a monad* m *(with structure given by* M : Monad m*) and a predicate lifter* lift : (a → Set) → (m a → Set)*. Let* pts : PTs es *and* hs : Handlers m es *be predicate transformers and handlers for these effects. For all programs* S *and postconditions* P*,* wp pts S P *is equivalent to* lift P (run M hs S) *if the following conditions hold:*

pure: For all pure values x : a *and postconditions* Q*,* Q x *is equivalent to* lift Q (return x)*.*

step: For all i*,* c *and* Q*,* lookupPT pts i c Q *is equivalent to* lift Q (lookupHandler hs i c)*.*

bind: For all S*,* k *and* Q*,* lift (λ x → lift Q (k x)) S *is equivalent to* lift Q (bind S k)*.*

In other words, predicate transformer semantics are consistent, if they are equivalent with the handler for each step of the computation, and lifting of the postcondition agrees with the return *and* bind *operator.*

In the proof, we need to make use of the monotonicity of each predicate transformer. This is already included in the definition of the type *PT*, so we do not list it above. The condition *bind* is analogous to the *compositionality* property of *wp*.

Proof. The body of the proof consists of induction on the structure of the program *S*. The case where *S* is a *Pure* value is exactly the equivalence of *pure*. For the case *Step i c k*, we want to show that *lookupPT pts i c* λ *x* → *wp pts* (*k x*) *P* is equivalent to *lift P* (*run hs* (*Step i c k*)), under the induction hypothesis that for all *x*, *wp pts* (*k x*) *P* is equivalent to *lift P* (*run hs* (*k x*)). By monotonicity of predicate transformers, if *Q* is equivalent to *Q′*, then *lookupPT pts i c Q* is equivalent to *lookupPT pts i c Q′*. Applying this to the equivalence of the induction hypothesis, it remains to show that *lookupPT pts i c* λ *x* → *lift P* (*run hs* (*k x*)) is equivalent to *lift P* (*run hs* (*Step i c k*)). We can use the assumption *step* to get that the left hand side is equivalent to *lift* (λ *x* → *lift P* (*run hs* (*k x*))) (*lookupHandler hs i c*). The assumption *bind* gives that this is itself equivalent to *lift P* (*lookupHandler hs i c* >=> λ *x* → *run hs* (*k x*)). Finally, *lookupHandler hs i c* >=> λ *x* → *run hs* (*k x*) is exactly the definition of *run hs* (*Step i c k*), so we have the desired conclusion. □

By inspecting the proof, we also find a small strengthening of the result. Since we only apply the *pure* and *bind* assumptions to one postcondition *P*, it suffices that they hold for a certain class of postconditions, to show that the equivalence holds for the same class.

6.1.3 Mapping handlers to predicate transformers

In the previous section, we have defined conditions between handlers and predicate transformers, that ensure that the operational and predicate transformer semantics of a program are equivalent. An interesting aspect of these conditions is that the combination of handler and lifting function determine the predicate transformer up to equivalence of propositions, by the definition of *step*. Thus, for each handler and suitably monotone lift, we find a canonical predicate transformer.

Definition 6.1.3. For an effect type $\mathit{eff}\ C\ R$, a handler $h : (c : C) \to m\,(R\ c)$ into a monad m, together with a predicate lifter $\mathit{lift} : (a \to \mathit{Set}) \to (m\ a \to \mathit{Set})$ determine the *evaluation predicate transformer*, given by the following equation:

$$
\begin{aligned}
&\mathit{evaluationPT} : (h : (c : C) \to m\,(R\ c))\,(\mathit{lift} : (a \to \mathit{Set}) \to (m\ a \to \mathit{Set})) \to \\
&\quad (\forall\ c\ P\ Q \to (\forall\ x \to P\ x \to Q\ x) \to \mathit{lift}\ P\ (h\ c) \to \mathit{lift}\ Q\ (h\ c)) \to \mathit{PT}\ (\mathit{eff}\ C\ R) \\
&\mathit{PT.pt}\ (\mathit{evaluationPT}\ h\ \mathit{lift}\ \mathit{mono})\ c\ P = \mathit{lift}\ P\ (h\ c) \\
&\mathit{PT.mono}\ (\mathit{evaluationPT}\ h\ \mathit{lift}\ \mathit{mono}) = \mathit{mono}
\end{aligned}
$$

△

Example 6.1.4. As an example, we will determine the predicate transformer associated with handling *ENondet* into the *List* monad. This handler runs a nondeterministic program by giving the list of its potential outcomes, and can be defined as follows:

$$
\begin{aligned}
&\mathit{handleNondet}\ \mathit{Fail} = \mathit{Nil} \\
&\mathit{handleNondet}\ \mathit{Choice} = \mathit{True} :: \mathit{False} :: \mathit{Nil}
\end{aligned}
$$

One way to lift predicates to the *List* monad is by requiring the predicate holds on all elements of the list:

$$
\begin{aligned}
&\mathit{liftNondet}\ P\ \mathit{Nil} = \top \\
&\mathit{liftNondet}\ P\ (x :: xs) = P\ x \times \mathit{liftNondet}\ P\ xs
\end{aligned}
$$

Applying *evaluationPT* to *handleNondet* and *liftNondet* gives a predicate transformer, which gives equivalent, but not identical, preconditions to those given by *ptAll*.

$$
\begin{aligned}
&\mathit{ptAllEval} = \mathit{evaluationPT}\ \mathit{handleNondet}\ \mathit{liftNondet}\ \_ \\
&\mathit{equivalenceFail} : \forall\ P \to \mathit{PT.pt}\ \mathit{ptAllEval}\ \mathit{Fail}\ P == \top \\
&\mathit{equivalenceChoice} : \forall\ P \to \mathit{PT.pt}\ \mathit{ptAllEval}\ \mathit{Choice}\ P == \mathit{Pair}\ (P\ \mathit{True})\ (\mathit{Pair}\ (P\ \mathit{False})\ \top)
\end{aligned}
$$

◊

As we can see, many effects such as nondeterminism and statefulness allow for the predicate transformer semantics to be defined in terms of running the program in the corresponding monad, then applying the predicate to the output. This may result in somewhat more complicated predicates, such as *ptAllEval* including an extra $\top$ in the semantics of the *Choice* operation, so directly defining predicate transformers is still useful. Moreover, not all effects can be run in a monad that is notably simpler than the original *Free* monad. Specifically, generally recursive functions have predicate transformer semantics for partial correctness, even if they do not terminate. By virtue of non-termination, we cannot run such computations and inspect their output. Therefore, predicate transformer semantics in general are not all evaluation-based.

6.2 Equational reasoning

The preceding sections of this chapter dealt with the relation between predicate transformer semantics and effect handlers, comparing the axiomatic semantics of predicate transformers with the operational semantics of handlers. As mentioned in Section 2.3, we can use equations to reason about and verify pure functional programs.

Many systems of effects can be fully described by giving equations for the behaviour of their operations [PP02], such as the laws of the *State* monad that we showed in Example 2.3.1. Thus, the system of program verification using equational reasoning can be extended to effectful programs, by adding the monad laws and extra equations for the effects [GH11]. An immediate drawback to monadic equational reasoning is that it is not immediately obvious which equations to postulate. In this section, we will argue that predicate transformer semantics can express (and prove) the equations postulated by equational reasoning, and that it can verify programs that equational reasoning cannot, both for theoretical and for practical reasons.

First of all, for pure computations we have that the refinement relation implies equality: if S is refined by S', and neither make use of effects, then the two programs are syntactically equal.

```
pureEquiv : (S S' : Free Nil a) → wp Nil S ⊑ wp Nil S' → S == S'
pureEquiv (Pure x) (Pure x') H = H (λ x' → Pure x == Pure x') refl
```

Conversely, the refinement relation is reflexive, so for pure programs there is no distinction between refinement and syntactic equality.

For effectful programs, syntactic equalities used as semantics for effects do not hold for the *Free* monad. Rather, we express equations using the equivalence of Definition 3.3.5 in the style of the refinement calculus, where two programs are equivalent if their associated predicate transformers refine each other. Since the equivalence relation is a preorder, and by the *compositionality* lemma substituting equivalent components preserves equivalence, the equivalence relation has the properties we require to perform equational reasoning. Moreover, equivalence is strong enough for verification since different pure values are not equivalent, so we can distinguish programs with distinct output.

To give a concrete demonstration that we can do equational reasoning based on the refinement relation, we will show that equations proposed by Gibbons and Hinze [GH11] can be proved in the appropriate setting of predicate transformer semantics. They propose that effectful programs can be verified by writing them in an appropriate type class, on which we postulate several equations as axioms, and combine these with the monad laws and reduction rules of λ-calculus, using transitivity to prove the desired equation. Since Agda already includes the reduction rules of λ-calculus and we have proved the monad laws for *Free* in 3.1, we will show that the equations for the effects, as proposed by Gibbons and Hinze, hold in predicate transformer semantics for the *Free* monad. As mentioned, we will use the equivalence relation $\_ \equiv \_$ to express the equations.

The first set of equations proposed by Gibbons and Hinze are for nondeterministic computations, and correspond to the *ENondet* effect. Proving these equations in the predicate transformer semantics is straightforward, coming down to evaluating the preconditions sufficiently that they are immediately equivalent.

```
fail-left-zero : (m : Free (ENondet :: es) a) → fail ≫ m ≡ fail
fail-left-zero S = (λ P H → H) , (λ P H → H)

choice-assoc : (S1 S2 S3 : Free (ENondet :: es) a) →
  choice S1 (choice S2 S3) ≡ choice (choice S1 S2) S3
choice-assoc S1 S2 S3 =
  (λ { P (H1 , (H2 , H3)) → (H1 , H2) , H3 }) ,
  (λ { P ((H1 , H2) , H3) → H1 , (H2 , H3) })
choice-dist : (S1 S2 : Free (ENondet :: es) a) (f : a → Free (ENondet :: es) b) →
  (choice S1 S2) >>= f ≡ choice (S1 >>= f) (S2 >>= f)
choice-dist S1 S2 f = (λ P H → H) , (λ P H → H)
```

The second set of equations are about exceptions and handlers, using the *abort* and *catch* operations. We represent *catch* as an effect handler for *EPartial*:

```
catch : (S1 S2 : Free (EPartial :: es) a) → Free (EPartial :: es) a
catch (Pure x) S2 = Pure x
catch (Step ∈Head Abort k) S2 = S2
catch (Step (∈Tail i) c k) S2 = Step (∈Tail i) c λ x → catch (k x) S2
```

The proofs of these equations are mostly straightforward. The only complication is that we need induction on the structure of the programs, to mirror the definition of *catch*. In the recursive step, we make use of the compositionality lemma ≡-step, which states that performing the same step followed by equivalent continuations is equivalent. This lemma is a direct consequence of monotonicity.

```
catch-unit-left : (S : Free (EPartial :: es) a) → catch abort S ≡ S
catch-unit-left S = (λ P H → H) , (λ P H → H)
```

```
catch-unit-right : (S : Free (EPartial :: es) a) → catch S abort ≡ S
catch-unit-right (Pure x) = (λ P H → H) , (λ P H → H)
catch-unit-right (Step ∈Head Abort k) = (λ P ()) , (λ P ())
catch-unit-right (Step (∈Tail i) c k) = ≡-step (λ x → catch-unit-right (k x))
catch-assoc : (S1 S2 S3 : Free (EPartial :: es) a) →
  catch S1 (catch S2 S3) ≡ catch (catch S1 S2) S3
catch-assoc (Pure x) S2 S3 = (λ P H → H) , (λ P H → H)
catch-assoc (Step ∈Head Abort k) S2 S3 = (λ P H → H) , (λ P H → H)
catch-assoc (Step (∈Tail i) c k) S2 S3 = ≡-step (λ x → catch-assoc (k x) S2 S3)
```

The final set of equations concerns *EState*. Here, all proofs follow from sufficient evaluation of the precondition.

```
get-get : get >>= λ s → get >>= k s s ≡ get >>= λ s → k s s
get-get = (λ P t H → H) , (λ P t H → H)
get-put : get >>= put ≡ Pure tt
get-put = (λ P t H → H) , (λ P t H → H)
put-get : ∀ t → put t >> get ≡ put t >> Pure t
put-get t = (λ P t' H → H) , (λ P t' H → H)
put-put : ∀ t t' → put t >> put t' ≡ put t'
put-put t t' = (λ P t'' H → H) , (λ P t'' H → H)
```

6.2.1 Comparing expressive power

We claimed before that predicate transformers have more power than equations in verifying programs. The more precise version of this claim is that we can express properties through predicate transformers that cannot be expressed through equations on the program. Informally, an equation can only compare between implementations of the specifications, not between specification and implementation. Thus, when verifying programs equationally, we cannot formalize the initial step of going from a specification to an initial (potentially very inefficient) implementation. There is an objection to this informal argument: to verify a postcondition P holds on the output of a program, we can check that the program is equivalent to the same program that applies P to the output and *abort*s if it does not hold.

A more formal argument shows the objection is not always applicable. Let $X \subseteq \mathbb{N}$ be a set that is not recursively enumerable, and suppose we want to verify the postcondition P on a natural number, which states "the output is an element of X". With predicate transformer semantics, this is done by finding a proof for *wp S P*. Since the set X is not recursively enumerable, there is no program that outputs exactly the elements of X, so there is no reference implementation that equational reasoning can compare against. Thus, we can verify P using predicate transformer semantics, but expressing the specification with equational reasoning is impossible. Therefore, predicate transformer semantics is more powerful than equational reasoning for verifying postconditions.

Moreover, there exist semantics that are impractical to describe using equations. Consider assigning the *EState* effect the semantics of self-destructing memory, where applying the *get* operation twice results in failure. Not only do we have to change the *get-get* equation to *get* ≫ *get* ≡ *abort*, but the equation *get* >>= *put* ≡ *Pure tt* does not hold any more. It is possible (with some difficulty) to find a new set of equations for self-destructing memory. How do we change these equations to deal with the *get* operation failing only when it has occurred 64 times? To give equations for dealing with self-destructing memory is a difficult task; for predicate transformer semantics it is simple to describe: add an extra number to the state, counting the number of *get* operations left before self-destruction, and in the semantics for the *get* operation, either decrement this counter or *abort* if it has reached zero.

Not only does predicate transformer semantics allow us to use any predicate (of the correct type) as a postcondition, following the refinement calculus we can use specifications as a program element. This means we can use the _⊑_ (and _≡_) relations to relate specification with programs, in the way that plain equational reasoning does not allow. The following chapter will focus on the idea of using specifications as effects.

Chapter 7

Deriving programs from specification

An idealized process of software engineering using the refinement calculus starts with writing down the specification formally, as the starting program. Then, we repeatedly apply the rules of the refinement calculus to replace parts of this specification with executable code, until the whole program has been transformed from specification to code. Each intermediate program in this process of *derivation*, although the program is not yet executable, still satisfies the original specification. The crucial ingredient in this process is the ability to treat specification and executable code as interchangeable parts of a program. This is achieved by giving the semantics for both as predicate transformers.

In this chapter, we show how the *Free* monad allows for combining specifications and code. We use this to demonstrate a derivation process that is formally computer-verified and has the assistance of Agda's interactive features. The end result of derivation consists of executable code together with a certificate that the code satisfies the specification. We also highlight the application of lemmas such as transitivity of $\_\sqsubseteq\_$ and compositionality of *wp* to define combinators that allow us for custom notation of this derivation process.

7.1 Mixing specifications and code

In Section 3.3, we give a basic description of program verification from a specification. A specification, as represented by the *Spec* data type, consists of a pre- and postcondition. The intended semantics are that the postcondition will be satisfied after running the computation, as long as the precondition is satisfied before running the computation. In this chapter, we will work with specifications of type $Spec^S$ that allow access to the state, for more generality. The formal definition of the semantics is given by the predicate transformer $wpSpec^S$. Since our semantics for effects only need a predicate transformer, and we can give an effect type for specifications, we will treat a specification as an effect. Where the type *Spec* is parametrised over the return type, for *CSpec* this parametrisation occurs in the values, allowing specifications with different return types to inhabit the same type.

Definition 7.1.1. The effect of *calling a specification* is given by the following effect type:

```
record CSpec : Set where
  constructor [_,_]
  field
    {R} : Set
    pre : s → Set
    post : s → R → s → Set
ESpec : Effect
ESpec = eff CSpec CSpec.R
```

Again, we introduce a smart constructor for performing the effect, which we will write the same as the constructors for the *Spec* and *CSpec* types.

```
[_,_] : (s → Set) → (s → a → s → Set) → Free (ESpec :: es) a
[ pre , post ] = Step ∈Head [ pre , post ] Pure
```

The semantics of this effect are given by the predicate transformer $wpSpec^S$. Writing out these semantics:

```
ptSpec : PT^S s ESpec
PTS.pt ptSpec [ pre , post ] P t = pre t × ∀ x t' → post t x t' → P x t'
```

△

One use of specification as effect is to model calling to an interface. If we have an interface for a procedure, expressed as a specification, then a program that calls this interface should have the semantics of "executing" this specification. Notably, we can call the specification before having written any implementation for the interface, allowing us to write the rest of the program before returning to implement the specification.

7.2 Derivation through incremental refinement

Now we are ready to say what it means to implement a program. Given a program S, usually in the form of a specification, we implement it by giving another program that is executable, in the sense that it does not contain *ESpec* as an effect. Then we show this code refines S (with semantics given by the weakest precondition under some given predicate transformers *pts*). This is expressed by the data type *Impl S*:

```
record Impl (S : Free (ESpec :: es) a) : Set where
  constructor impl
  field
    prog : Free es a
    refines : wp^S (ptSpec :: pts) S ⊑^S wp^S pts prog
```

We will leave the effects *es* and the predicate transformers *pts* out from *Impl* when they are clear from the context, to avoid repeating these each time they are needed. However, they are important in determining whether a given value is indeed a well-typed element of *Impl*.

The *Impl* type has a similar form to the Dijkstra monad [Swa+11; Swa+13]. Both *Impl* and the Dijkstra monad contain executable code and its specification. Additionally, the constructions of *Impl* and the Dijkstra monad ensure correctness of the program with respect to the specification. To be precise, in this chapter we typically instantiate the argument S to be a specification, giving types of the form *Impl* [*P* , *Q*], more directly comparable to the less general Hoare monad. By introducing a new specification construct that contains a predicate transformer instead of pre- and postcondition, we could also make a form of *Impl* equivalent to the full Dijkstra monad. The main distinction we observe is that our *Impl* type represents the proof as a separate field in the record type, while programs in the Dijkstra monad are correct by consistency of its type constructors. Operations such as *bind* and *return* are primitive in the Dijkstra and Hoare monads; the *Impl* type can represent them as combinators. We will give examples of combinators for *Impl* after we manually construct a few example *Impl* terms.

Example 7.2.1. We can give a specification for the ± operator on integers, which may perform addition or subtraction, but it is unspecified which. The specification can be given independent of the effect system:

```
±-spec : (a b : Int) → Free (ESpec :: es) Int
±-spec a b = [⊤ , λ c → Either (c == a + b) (c == a - b) ]
```

The operator + always adds its arguments, so it always satisfies the first alternative of the specification. Since addition is a pure function defined in the Agda standard library, we can use it as executable code that implements ±. This means we can formally write it out as a term in the *Impl* type:

```
+-impl-± : (a b : Int) → Impl (±-spec a b)
+-impl-± a b = impl (Pure (a + b)) (λ { P _ (fst , snd) → snd (a + b) _ (Inl refl) })
```

The $\pm$ operator is specified to return either of two values, so it has a clear correspondence with the effect of nondeterminism. Although we specify that $\pm$ may return either value, the correct semantics of nondeterministic choice to use in this context is given by *ptAll*: the interface is specified to return any value, so the caller using the interface must accept all values. Directly constructing a term in *Impl* is straightforward:

```
nondet-impl-± : (a b : Int) → Impl (±-spec a b)
nondet-impl-± a b = impl
  (choice (Pure (a + b)) (Pure (a - b)))
  λ { P _ (fst , snd) → (snd (a + b) _ (Inl refl)) , (snd (a - b) _ (Inr refl)) }
```

◊

7.3 Combinators for programming

While we can directly produce values of type *Impl*, this does not give us much more than a more convoluted way to express correctness, something that Agda itself can already do for us. We improve this situation by introducing combinators. These construct values of *Impl* without needing the programmer to supply (as many) proof terms. The first combinator corresponds to the *Pure* constructor, and constructs implementations where pre- and postcondition coincide:

```
doReturn : (x : a) → Impl [ λ t → P t x t , P ]
doReturn x = impl
  (Pure x)
  λ { P _ (fst , snd) → snd x _ fst }
```

Note that this is exactly the form of the *return* operation of the Hoare monad.

In designing our combinators, we want to make them compute the remaining goal based on the current goal, so we want the return type of a combinator to have the form *Impl* [*P* , *Q*] with *P* and *Q* free variables. Currently, if our specification does not have the form [λ *t* → *P t x t* , *P*], the previous combinator does not apply. Instead, we can try to sharpen the pre- and postcondition, refining the specification to one we can work with. In general, if we want to implement S, and S is refined by S', then it suffices to implement S' instead.

```
doRefine : (wp^S (ptSpec :: pts) S ⊑^S wp^S (ptSpec :: pts) S') → Impl S' → Impl S
doRefine r1 (impl prog r2) = impl prog (⊑-trans r1 r2)
```

Using *doRefine*, we can replace an original specification [*P* , *Q*] with a sharper specification [*P'* , *Q'*] that refines the original specification.

```
doSharpen : (wp^S (ptSpec :: pts) [ P , Q] ⊑^S wp^S (ptSpec :: pts) [ P' , Q']) →
  Impl [ P' , Q'] → Impl [ P , Q ]
doSharpen = doRefine
```

We can use *doSharpen* to give a more general form of *doReturn*, which takes a proof that the postcondition holds on the returned value.

```
doReturn' : (x : a) → (∀ t → P t → Q t x t) → Impl [ P , Q ]
doReturn' x pf = doSharpen (λ { P t (fst , snd) → pf t fst , λ x' t' H → snd x' t' H }) (doReturn x)
```

The combinators *doReturn* and *doSharpen* correspond to returning a *Pure* value and calling a specification, so we will also define a combinator corresponding to other *Step*s. Since the *Step* constructor depends on the call *c* we make, the combinator depends on the semantics of this call. Our approach is to use the compositionality properties of *wp* to give combinators for the composition of two programs, then specialize these to doing a *Step*. Apart from allowing the use of a *Step*, the combinators we develop can be applied generally, for

example to give a recursive definition of a program. We will demonstrate the general applicability in Example 7.4.1.

The first of these is *doCompose*, which takes two implementations and uses the _≫=_ operator to compose them. The correctness proof is given by combining *compositionality-left* for implementing S with *compositionality-right* for implementing f.

```
doCompose : Impl S → ((x : a) → Impl (f x)) → Impl (S ≫= f)
```

To use *doCompose*, we also need a way to introduce the _≫=_ operator in the program to be implemented. We introduce the combinator *doBindSpec* that splits apart a specification into a composition of two specifications. The second specification is computed by *preBind* and *postBind* based on the original specification and the first specification. Making the postcondition of the first specification a free variable allows us to do an easy composition with existing implementations.

```
doBindSpec : Impl ([pre , mid ] ≫= λ x → [preBind pre mid x , postBind pre mid post x]) →
  Impl [ pre , post ]
```

With the correct definitions of *preBind* and *postBind*, the combinator is simply an application of the *doSharpen* combinator to a simple refinement proof. Unfortunately, it is not immediately obvious how to define *preBind* and *postBind*. Our semantics are based on the weakest *pre*condition and we need to calculate the *post*condition of the first program. The correct definitions are given by quantifying over the potential initial states for the composed program: the new precondition requires that there exists some valid initial state t which leads to the current state t', as specified by *mid*, while the new postcondition requires that the original postcondition holds for all such initial states t.

```
preBind  pre mid      x = λ t'       → ∃ (t : s) → pre t × mid t x t'
postBind pre mid post x = λ t' x' t'' → ∀ (t : s) → pre t × mid t x t' → post t x' t''
```

The combinator for implementing a specification by composing two programs is then simply the composition of *doBindSpec* and *doCompose*.

```
doBind : Impl [ pre , mid] → ((x : a) → Impl [ preBind pre mid x , postBind pre mid post x]) →
  Impl [ pre , post ]
doBind mx f = doBindSpec (doCompose mx f)
```

Again, note the similarity with the *bind* operator for the Hoare monad.

Using the *doBind* combinator, we can write the *doStep* combinator as a composition of *Step i c Pure* with the continuation. We need an extra argument, containing a proof that the intermediate predicate *mid* is indeed a valid postcondition for the given precondition. This argument is required since we are working with weakest precondition semantics, and we cannot directly calculate the strongest postcondition for the program *Step i c Pure*.

```
doStep : (i : eff C R ∈ es) (c : C) →
  (∀ t → pre t → lookupPTS pts i c (mid t) t) →
  ((x : R c) → Impl [ preBind pre mid x , postBind pre mid post x]) →
  Impl [ pre , post ]
doStep i c pf = doBind
  (impl (Step i c Pure) (λ { P t (fst , snd) → lookupMono pts i c _ P snd t (pf t fst) }))
```

In practice, such as in the following example, we will instantiate a combinator for each constructor of the type C, and this instantiation will fill in all parameters except for the continuation. Thus, the combinator will be used as a function that takes one implementation and produces another.

Example 7.3.1. We will define a combinator corresponding to the effect *Choice* of nondeterministic computation, and use the combinator to implement the ± operator nondeterministically. The combinator is implemented by instantiating *doStep* with the correct proof, together with *doSharpen* to simplify the computed pre- and postcondition.

```
doChoice : (p p' : Impl [ P , Q ]) → Impl [ P , Q ]
doChoice p p' = doStep ∈Head Choice (λ t H → refl , refl)
  λ x → doSharpen _ (if x then p else p')
```

Now can give a derivation of the nondeterministic ± operator, expressed only in terms of the combinators.

```
derive-± : (a b : Int) → Impl (±-spec a b)
derive-± a b = doChoice
  (doReturn' (a + b) λ t x → Inl refl)
  (doReturn' (a − b) λ t x → Inr refl)
```

◊

The three combinators *doReturn*, *doSharpen* and *doStep* correspond neatly to the operations given by *Free* (*ESpec* :: *es*), but they alone are not sufficient for our purposes. For instance, the three combinators only support tail recursion. If we want to write non-tail recursive code, we need to perform a monadic bind on two implementations. Each of the three combinators only has a single implementation as a parameter, so they are not sufficient to express all code. This is why we need the *doBind* operator, and we can see such a situation in the next section.

7.4 Example: deriving the *index* function

Example 7.4.1. As a more involved example, we derive a partial function *index* that searches for the position of a given element in a list. If the element exists in the list, the *index* function returns its index as a natural number; otherwise the computation *abort*s. The type of *index* as a program, without a bundled correctness proof, would be:

```
index : (x : ℕ) (xs : List ℕ) → Free (EPartial :: Nil) ℕ
```

We will illustrate how the interactive features of Agda aid in the derivation process by showing the interactive goal as we apply the combinators.

Since we are going to write a partial function, we will first introduce a combinator for the command *Abort*. The continuation for an aborting computation is irrelevant, so we do not take it as an argument. The combinator *doAbort* allows any precondition as long as we can prove that it is never satisfied.

```
doAbort : (∀ t → ¬ pre t) → Impl [ pre , post ]
doAbort ¬pre = doStep ∈Head Abort ¬pre λ ()
```

The combinators *doAbort*, *doBind* and *doReturn* are sufficient to write the *index* function. Its precondition is that the element exists in the list, while the postcondition expresses that the index is valid for the list, and that the list indeed contains the element at the given position. The returned index does not necessarily match the one given in the proof of the precondition $x \in xs$.

```
indexPre x xs = x ∈ xs
indexPost x xs = λ i → ∃ (it : i < length xs) → elemAt xs i it == x
```

The first step in the derivation is to declare the type of *index*. The argument types are written explicitly; from the type of *indexPost* we can deduce that the return type is ℕ.

```
index : (x : ℕ) (xs : List ℕ) → Impl [ indexPre x xs , indexPost x xs ]
index = ?
```

The *?* represents a *hole*, a place where a definition is missing. Agda's interactive mode allows us to manipulate definitions based on these holes, for example refining them by filling in (part of) the definition. This

interactivity can be used to incrementally derive the *index* function while maintaining correctness at each step.

Introducing the arguments x and xs leads to a new goal, which we will write in the style of the Agda interactive mode: above a horizontal line is the *goal*, the inferred type of the hole, while below the line are the assumptions, or variables, that are in scope. Often, we omit the assumptions for readability, if they are not different than the previous assumptions.

$$Impl\ [\ indexPre\ x\ xs\ ,\ indexPost\ ]$$

$$xs\ :\ List\ \mathbb{N}$$
$$x\ :\ \mathbb{N}$$

We proceed by case distinction on the list xs.

7.4.1 *index x Nil*

In the *Nil* case, the goal is:

$$Impl\ [\ indexPre\ x\ Nil\ ,\ indexPost\ x\ Nil\ ]$$

Here, the precondition is not satisfied (the list *Nil* does not contain any element), so any postcondition is trivially true for a given program. This does not mean that we are immediately finished: although the postcondition is false, the goal cannot be discharged with a case distinction on the empty type *indexPre x Nil*. We must still specify the behaviour of the program, which does not have access to the precondition. Thus, we use the combinator *doAbort* to refine the goal, leaving the following goal:

$$(t\ :\ s)\ \rightarrow\ \neg\ indexPre\ x\ Nil\ t$$

This goal is easily solved, since $\neg$ *indexPre x Nil t* normalises to $x\ \in\ Nil\ \rightarrow\ \bot$ and we can give an element of the last type as an empty case distinction $\lambda\ ()$. At this point in the derivation, *index* looks like:

$$index\ :\ (x\ :\ \mathbb{N})\ (xs\ :\ List\ \mathbb{N})\ \rightarrow\ Impl\ [\ indexPre\ x\ xs\ ,\ indexPost\ x\ xs\ ]$$
$$index\ x\ Nil\ =\ doAbort\ \lambda\ t\ ()$$
$$index\ x\ (x'\ ::\ xs)\ =\ ?$$

7.4.2 *index x* (*x :: xs*)

For the case where the list is of the form $(x'\ ::\ xs)$, our goal is similar to the *Nil* case:

$$Impl\ [\ indexPre\ x\ (x'\ ::\ xs)\ ,\ indexPost\ x\ (x'\ ::\ xs)\ ]$$

At this point it is possible that x and x' are the same, in which case we are done: we return that x is in the head of the list. Since equality of natural numbers is decidable, let us do a case distinction on the equality of x and x'.

If x is indeed the head of the list, our goal is:

$$Impl\ [\ indexPre\ x\ (x\ ::\ xs)\ ,\ indexPost\ x\ (x\ ::\ xs)\ ]$$

Since we have found the value we are looking for in the head of the list, we use the *doReturn'* applied to the return value *0*. This leaves the following goal, arising from evaluating *indexPost x* $(x\ ::\ xs)$ applied to the return value *0*:

$$(0\ <\ Succ\ (length\ xs))\ \times\ (x\ ==\ x)$$

The goal can be solved automatically by the Agda synthesiser. At this point, our derivation looks like:

```
index : (x : ℕ) (xs : List ℕ) → Impl [ indexPre x xs , indexPost x xs ]
index x Nil = doAbort λ t ()
index x (x' :: xs) with x ≟ x'
index x (.x :: xs) | yes refl = doReturn' 0 (s≤s z≤n , refl)
index x (x' :: xs) | no ¬p = ?
```

7.4.3 *index x (x' :: xs)*

The final case is where the list has the form $x' :: xs$, and the element x to be found is not equal to the head of the list x'. The goal is:

```
Impl [ indexPre x (x' :: xs) , indexPost x (x' :: xs) ]
```

```
¬p : ¬ (x == x')
xs : List ℕ
x' : ℕ
x  : ℕ
```

In this case, the precondition tells us that we can find x in the tail of the list, *xs*. We can do this by recursively calling *index*, making use of the *doBind* combinator. However, the precondition for *index* does not match with the precondition we have currently, so we refine the goal with *doSharpen ?* (*doBind* (*index x xs*) *?*). This leads to two new goals, both depending on a new postcondition Q'. The first is the correctness proof of the *doSharpen* step:

```
wp^S (ptSpec :: _) [ indexPre x (x' :: xs) , indexPost x (x' :: xs)] ⊑^S
wp^S (ptSpec :: _) [ indexPre x xs , Q' ]
```

```
¬p : ¬ (x == x')
xs : List ℕ
x' : ℕ
x  : ℕ
```

The second is the implementation for the right hand side of the bind operator:

```
Impl [ preBind (indexPre x xs) (indexPost x xs) i , postBind (indexPre x xs) (indexPost x xs) Q' i ]
```

```
i  : ℕ
¬p : ¬ (x == x')
xs : List ℕ
x' : ℕ
x  : ℕ
```

The easiest choice for the new postcondition Q' is to copy the postcondition *indexPost*, which simplifies the correctness proof.

We will first give the implementation for the right hand side, since its code is not too difficult. After all, given the index $i : \mathbb{N}$ for the tail, we can simply return *Succ i* as index for the full list. Thus, we refine the goal with $\lambda\, i \to$ *doReturn'* ($\in$Tail *i*) *?*, leaving the following goal:

$$\mathit{preBind}\ (\mathit{indexPre}\ x\ xs)\ (\mathit{indexPost}\ x\ xs)\ i\ \to$$
$$\mathit{postBind}\ (\mathit{indexPre}\ x\ xs)\ (\mathit{indexPost}\ x\ xs)\ (\mathit{indexPost}\ x\ (x'\ ::\ xs))\ i\ (\mathit{Succ}\ i)$$

If we inspect the *postBind* of the return type, it normalises to:

$$(x\ \in\ xs) \times \exists\ (\mathit{lt}\ :\ i\ <\ \mathit{length}\ xs)\ \to\ \mathit{elemAt}\ xs\ i\ \mathit{lt}\ ==\ x\ \to$$
$$\exists\ (\mathit{lt}\ :\ \mathit{Succ}\ i\ <\ \mathit{Succ}\ (\mathit{length}\ xs))\ \to\ \mathit{elemAt}\ (x'\ ::\ xs)\ (\mathit{Succ}\ i)\ \mathit{lt}\ ==\ x$$

The inequality $\mathit{Succ}\ i\ <\ \mathit{Succ}\ (\mathit{length}\ xs)$ follows from applying $s{\leq}s\ :\ a\ <\ b\ \to\ \mathit{Succ}\ a\ <\ \mathit{Succ}\ b$ to the assumption; the equality $\mathit{elemAt}\ (x'\ ::\ xs)\ (\mathit{Succ}\ i)\ \mathit{lt}\ ==\ x$ normalises to the same as $\mathit{elemAt}\ xs\ i\ \mathit{lt}\ ==\ x$ so we can re-use the relevant part of the assumption.

The remaining goal is the correctness of the *doSharpen* step, which normalises to the following goal:

$$(P\ :\ x\ \in\ (x'\ ::\ xs)\ \to\ \mathit{Set})\ \to$$
$$x\ \in\ (x'\ ::\ xs) \times ((o\ :\ x\ \in\ (x'\ ::\ xs))\ \to\ \top\ \to\ P\ o)\ \to$$
$$x\ \in\ xs \times ((o\ :\ x\ \in\ (x'\ ::\ xs))\ \to\ \top\ \to\ P\ o)$$

$$\neg p\ :\ \neg\ x\ ==\ x'$$
$$xs\ :\ \mathit{List}\ \mathbb{N}$$
$$x'\ :\ \mathbb{N}$$
$$x\ :\ \mathbb{N}$$

Since we chose the new postcondition Q' to be equal to the specified postcondition $\mathit{indexPost}\ x\ xs$, the parts $(o\ :\ x\ \in\ (x'\ ::\ xs))\ \to\ \top\ \to\ P\ o$ of the assumption and $(o\ :\ x\ \in\ (x'\ ::\ xs))\ \to\ \top\ \to\ P\ o$ of the conclusion are identical, and the only interesting part concerns the preconditions:

$$x\ \in\ xs$$

$$\mathit{pre}\ :\ x\ \in\ (x'\ ::\ xs)$$
$$\neg p\ :\ \neg\ x\ ==\ x'$$
$$xs\ :\ \mathit{List}\ \mathbb{N}$$
$$x'\ :\ \mathbb{N}$$
$$x\ :\ \mathbb{N}$$

We prove this in a separate lemma by case distinction on the proof of the precondition *pre*, since the case $\in\mathsf{Head}$ leads to a contradiction and $\in\mathsf{Tail}\ i$ gives the required result.

The result is the following derivation of the *index* function.

$$\begin{aligned}
&\mathit{index}\ x\ \mathit{Nil}\ =\ \mathit{doAbort}\ \lambda\ t\ ()\\
&\mathit{index}\ x\ (x'\ ::\ xs)\ \mathbf{with}\ x\ \stackrel{?}{=}\ x'\\
&\mathit{index}\ x\ (.x\ ::\ xs)\ \mid\ \mathit{yes}\ \mathit{refl}\ =\ \mathit{doReturn'}\ 0\ (s{\leq}s\ z{\leq}n\ ,\ \mathit{refl})\\
&\mathit{index}\ x\ (x'\ ::\ xs)\ \mid\ \mathit{no}\ \neg p\ =\ \mathit{doSharpen}\ \{\ Q'\ =\ \mathit{indexPost}\ x\ (x'\ ::\ xs)\ \}\\
&\quad(\lambda\ \{\ P\ _\ (\mathit{fst}\ ,\ \mathit{snd})\ \to\ \mathit{lemma}\ \neg p\ \mathit{fst}\ ,\ \mathit{snd}\ \})\\
&\quad(\mathit{doBind}\ (\mathit{index}\ x\ xs)\ \lambda\ i\ \to\ \mathit{doReturn'}\ (\mathit{Succ}\ i)\ \lambda\ \{(_\ ,\ (\mathit{lt}\ ,\ p))\ \to\ s{\leq}s\ \mathit{lt}\ ,\ p\ \})\\
&\quad\mathbf{where}\\
&\quad\mathit{lemma}\ :\ \neg\ (x\ ==\ x')\ \to\ x\ \in\ (x'\ ::\ xs)\ \to\ x\ \in\ xs\\
&\quad\mathit{lemma}\ \neg p\ \in\mathsf{Head}\quad\ =\ \mathit{magic}\ (\neg p\ \mathit{refl})\\
&\quad\mathit{lemma}\ \neg p\ (\in\mathsf{Tail}\ i)\ =\ i
\end{aligned}$$

$\Diamond$

Chapter 8

Application: verifying parsers

Up to this point, we have illustrated the various aspects of predicate transformer semantics only with small example programs. In this chapter, we demonstrate how to apply predicate transformer semantics to the verification of programs that are not as trivial. We will verify programs that parse formal languages, matching a string with a given grammar. In Section 8.1, we write and verify regular expression parsers. In Section 8.2, we write and verify a parser for context-free grammars, and compare predicate transformer semantics for the *Free* monad with other representation of languages in intuitionistic type theory.

8.1 Regular expression parsing

The first class of languages we will write parsers for are the regular languages. Our approach is first to define the specification of a parser, then inspect this specification to write the first implementation and prove (partial) correctness of this implementation. We will later improve this implementation by refining it.

Definition 8.1.1 ([AU77])**.** The class of *regular languages* is the smallest class such that:

- the empty language is regular,
- the language containing only the empty string is regular,
- for each character `x`, the language containing only the string `"x"` is regular,
- the union and concatenation of regular languages are regular, and
- the repetition of a regular language is regular.

△

The type *Regex* of *regular expressions* is defined inductively in the same way as regular languages. An element of this type represents the syntax of a regular language, and we will generally identify a regular expression with the language it denotes.

```
data Regex : Set where
  Empty     : Regex
  Epsilon   : Regex
  Singleton : Char → Regex
  _|_       : Regex → Regex → Regex
  _ · _     : Regex → Regex → Regex
  _⋆        : Regex → Regex
```

Here, *Empty* is an expression for empty language (which matches no strings at all), while *Epsilon* is an expression for the language of the empty string (which matches exactly one string: `""`).

What should a parser for regular expressions output? Perhaps it could return a *Bool* indicating whether a given string matches the regular expression, or we could annotate the regular expression with capture groups, and say that the output of the parser maps each capture group to the substring that the capture group matches. In this case, we define the return type to be a parse tree mirroring the structure of the expression.

```
Tree : Regex → Set
Tree Empty        = ⊥
Tree Epsilon      = ⊤
Tree (Singleton _) = Char
Tree (l | r)      = Either (Tree l) (Tree r)
Tree (l · r)      = Pair (Tree l) (Tree r)
Tree (r ⋆)        = List (Tree r)
```

In Agda, we can represent the semantics of the *Regex* type by giving a relation between a *Regex* and a *String* on the one hand (the input of the parser), and a parse tree on the other hand (the output of the parser). Note that the *Tree* type itself is not sufficient to represent the semantics, since it does not say which strings result in any given parse tree. If the *Regex* and *String* do not match, there should be no output, otherwise the output consists of all relevant parse trees. We give the relation using the following inductive definition:

```
data Match : (r : Regex) → String → Tree r → Set where
  Epsilon    : Match Epsilon Nil tt
  Singleton  : Match (Singleton x) (x :: Nil) x
  OrLeft     : Match l xs x → Match (l | r) xs (Inl x)
  OrRight    : Match r xs x → Match (l | r) xs (Inr x)
  Concat     : Match l ys y → Match r zs z → Match (l · r) (ys ++ zs) (y , z)
  StarNil    : Match (r ⋆) Nil Nil
  StarConcat : Match (r · (r ⋆)) xs (y , ys) → Match (r ⋆) xs (y :: ys)
```

Note that there is no constructor for *Match Empty xs x* for any *xs* or *x*, which we interpret as that there is no way to match the *Empty* language with a string *xs*. Similarly, the only constructor for *Match Epsilon xs x* is where *xs* is the empty string *Nil*.

8.1.1 Parsing regular languages recursively

Since the definition of *Match* allows for multiple ways that a given *Regex* and *String* may match, such as in the trivial case where the *Regex* is of the form $r \mid r$, and it also has cases where there is no way to match a *Regex* and a *String*, such as where the *Regex* is *Empty*, we can predict that the implementation of the parser should be nondeterministic. Whenever we encounter an expression of the form $l \mid r$, we make a nondeterministic *choice* between either *l* or *s*. Similarly, whenever we encounter the *Empty* expression, we immediately *fail*. In this analysis, we have already assumed that we implement the parser by structural recursion on the *Regex*, so let us consider other cases.

The implementation for concatenation is not as immediately obvious. One way that we can deal with it is to change the type of the parser. Instead write a parser that returns the unmatched portion of the string, and when we have to match a regular expression of the form $l \cdot r$ with a string *xs*, we match *l* with *xs* giving a left over string *ys*, then match *r* with *ys*. We can also do without changing the return values of the parser, by nondeterministically splitting the string *xs* into $ys \mathbin{+\!\!+} zs$. That is what we do in a helper function *allSplits*, which nondeterministically chooses such *ys* and *zs* and returns them as a pair.

```
allSplits : {{ iND : ENondet ∈ es }} (xs : List a) → Free es (List a × List a)
allSplits Nil = Pure (Nil , Nil)
allSplits (x :: xs) = choice
  (Pure (Nil , (x :: xs)))
  (allSplits xs >>= λ {(ys , zs) → Pure ((x :: ys) , zs)})
```

Finally, we can handle the Kleene star $\_\star$ by treating an expression of the form $r\,\star$ as a nondeterministic choice between *Epsilon* and the concatenation $r \cdot r\,\star$. Simply performing this substitution on the input expression is not acceptable, since it would lead to an infinitely long expression, and thus non-termination of the parser. The solution is to combine the nondeterminism effect with general recursion. Instead of substituting $r\,\star$ with $\mathit{Epsilon} \mid (r \cdot r\,\star)$, we perform a generally recursive *call* to the parser with the expanded expression as an argument.

Now that we have an idea of its structure, let us define the parser in code. As explained, the main structure of the parser consists of performing a case distinction on the expression and then checking that the string has the correct format.

```
match : {{ iND : ENondet ∈ es }} → Regex × String -es-> λ{(r , xs) → Tree r}
match (Empty , xs) = fail
match (Epsilon , Nil) = Pure tt
match (Epsilon , xs@ (_ :: _)) = fail
match ((Singleton c) , Nil) = fail
match ((Singleton c) , (x :: Nil)) with c ≟ x
match ((Singleton c) , (.c :: Nil)) | yes refl = Pure c
match ((Singleton c) , (x :: Nil)) | no ¬p = fail
match ((Singleton c) , (_ :: _ :: _)) = fail
match ((l · r) , xs) = do
  (ys , zs) ← allSplits xs
  y ← call (l , ys)
  z ← call (r , zs)
  Pure (y , z)
match ((l | r) , xs) = choice
  (call (l , xs) >>= (Pure ∘ Inl))
  (call (r , xs) >>= (Pure ∘ Inr))
match ((r ★) , Nil) = Pure Nil
match ((r ★) , xs@ (_ :: _)) = do
  (y , ys) ← call ((r · (r ★)) , xs)
  Pure (y :: ys)
```

8.1.2 Partial correctness of *match*

Although the implementation of *match* was derived from the specification *Match*, we have not yet reached the goal of a verified regular expression parser. To show partial correctness, we still need to give a formal proof that *match* implements its specification, i.e. that the specification is refined by *match*. The effects we need to use for running *match* include general recursion. Thus, as discussed in Section 3.3, we first need to give the specification for *match* before we can verify a program that makes a recursive call to *match*. Since the *Match* data type already represents the semantics of a regular expression, the specification for the parser can use *Match* as the relation between in- and output.

```
matchSpec : (r,xs : Pair Regex String) → Tree (Pair.fst r,xs) → Set
matchSpec (r , xs) ms = Match r xs ms
```

For the effect of nondeterminism, we want that all matches reported by the parser are correct. Thus, we use the demonic choice as in the *ptAll* predicate transformer. We get the following predicate transformer semantics that we run *match* in:

```
wpMatch : Free (eff (Pair Regex String) (λ{(r , xs) → Tree r}) :: ENondet :: Nil) a →
  (a → Set) → Set
wpMatch = wp (ptRec matchSpec :: ptAll :: Nil)
```

Here, *ptRec* gives the semantics of a recursive call to a function of type $C \overset{es}{\nrightarrow} R$, based on a relation of type $(c : C) \to R\ c \to Set$.

The correctness proof for *match* in the *wpMatch* semantics closely matches the structure of its definition (and by extension the structure of *allSplits*). It uses the same recursion on the structure of *Regex* as in the definition of *match*. Since we make use of *allSplits* in the definition, we first verify this function, i.e. that concatenating its output gives its input:

```
allSplitsSound : ∀ (xs : String) →
  wpSpec [ ⊤ , (λ {(ys , zs) → xs == ys ++ zs}) ] ⊑ wpMatch (allSplits xs)
```

The proof is given by induction on the input string *xs*.

Then, using the compositionality of the weakest precondition, we incorporate this correctness proof in the correctness proof of *match*. Apart from the extra work we need to do to use *allSplitsSound*, the proof essentially follows automatically from the definitions.

```
matchSound : ∀ r,xs → wpSpec [ ⊤ , matchSpec r,xs ] ⊑ wpMatch (match r,xs)
matchSound (Empty , xs) P (preH , postH) = tt
matchSound (Epsilon , Nil)        P (preH , postH) = postH _ Epsilon
matchSound (Epsilon , (_ :: _)) P (preH , postH) = tt
matchSound (Singleton c , Nil)              P (preH , postH) = tt
matchSound (Singleton c , (x :: Nil))       P (preH , postH) with c ≟ x
matchSound (Singleton c , (.c :: Nil))      P (preH , postH) | yes refl = postH _ Singleton
matchSound (Singleton c , (x :: Nil))       P (preH , postH) | no ¬p = tt
matchSound (Singleton c , (_ :: _ :: _)) P (preH , postH) = tt
matchSound ((l · r) , xs) P (preH , postH) = coerce (sym (fold-bind (allSplits xs) _ P _))
  (allSplitsSound xs _ (_ , (λ {(ys , zs) splitH y lH z rH → postH (y , z)
    (coerce (cong (λ xs → Match _ xs _) (sym splitH)) (Concat lH rH))})))
matchSound ((l | r) , xs) P (preH , postH) =
  (λ o H → postH _ (OrLeft H)) ,
  (λ o H → postH _ (OrRight H))
matchSound ((r ⋆) , Nil)        P (preH , postH) = postH _ StarNil
matchSound ((r ⋆) , (x :: xs)) P (preH , postH) = λ o H → postH _ (StarConcat H)
```

At this point, we have defined a parser for regular languages and formally proved that its output is always correct. However, *match* does not necessarily terminate: if *r* is a regular expression that accepts the empty string, then calling *match* on $r \star$ and a string *xs* results in the first nondeterministic alternative being an infinitely deep recursion.

The next step is then to write a parser that always terminates and show that *match* is refined by it. Our approach is to do recursion on the input string instead of on the regular expression.

8.1.3 Parsing regular languages with derivatives

Since recursion on the structure of a regular expression does not guarantee termination of the parser, we can instead perform recursion on the string to be parsed. To do this, we make use of the Brzozowski derivative.

Definition 8.1.2 ([Brz64])**.** The *Brzozowski derivative* of a formal language *L* with respect to a character *x* consists of all strings *xs* such that $x :: xs \in L$. △

Importantly, if *L* is regular, so are all its derivatives. Thus, let *r* be a regular expression, and *d r /d x* an expression for the derivative with respect to *x*, then *r* matches a string $x :: xs$ if and only if *d r /d x* matches *xs*. This suggests the following implementation of matching an expression *r* with a string *xs*: if *xs* is empty, check whether *r* matches the empty string; otherwise let *x* be the head of the string and *xs′* the tail and go in recursion on matching *d r /d x* with *xs′*.

The first step in implementing a parser using the Brzozowski derivative is to compute the derivative for a given regular expression. Following Brzozowski [Brz64], we use a helper function $\varepsilon?$ that decides whether an expression matches the empty string.

```
ε? : (r : Regex) → Dec (Σ (Tree r) (Match r Nil))
```

The definitions of $\varepsilon?$ is given by structural recursion on the regular expression, just as the derivative operator is:

```
d_/d_ : Regex → Char → Regex
d Empty /d c = Empty
d Epsilon /d c = Empty
d Singleton x /d c with c ≟ x
... | yes p = Epsilon
... | no ¬p = Empty
d l · r /d c with ε? l
... | yes p = ((d l /d c) · r) | (d r /d c)
... | no ¬p = (d l /d c) · r
d l | r /d c = (d l /d c) | (d r /d c)
d r ⋆ /d c = (d r /d c) · (r ⋆)
```

In order to use the derivative of r to compute a parse tree for r, we need to be able to convert a tree for $d\ r\ /d\ x$ to a tree for r. We do this with the function *integralTree*:

```
integralTree : (r : Regex) → Tree (d r /d x) → Tree r
```

We can also define it with exactly the same case distinction as we used to define *d_/d_*.

The code for the parser, *dmatch*, itself is very short. As we sketched, for an empty string we check that the expression matches the empty string, while for a non-empty string we use the derivative to perform a recursive call.

```
dmatch : ⦃ iND : ENondet ∈ es ⦄ → Regex × String ↬[es] λ { (r , xs) → Tree r }
dmatch (r , Nil) with ε? r
... | yes (ms , _) = Pure ms
... | no ¬p        = fail
dmatch (r , (x :: xs)) = call ((d r /d x) , xs) >>= (Pure ∘ integralTree r)
```

8.1.4 Total correctness of *dmatch*

Now that we have written *dmatch*, we can verify that it is a correct implementation of parsing a regular language. Not only will we show that it is partially correct, as we did for *match*, we can also easily prove that it terminates. Since *dmatch* always consumes a character before going in recursion, we can bound the number of recursive calls with the length of the input string. The proof goes by induction on this string. Unfolding the recursive *call* gives $(dmatch\ (d\ r\ /d\ x\ ,\ xs) \gg\!= (Pure \circ integralTree)$, which we can rewrite in the lemma *terminates-fmap* using the associativity monad law.

```
dmatchTerminates : (r : Regex) (xs : String) →
  terminates-in (addState ptAll :: Nil) (dmatch) (dmatch (r , xs)) (length xs) t
dmatchTerminates r Nil with ε? r
dmatchTerminates r Nil | yes p = tt
dmatchTerminates r Nil | no ¬p = tt
dmatchTerminates r (x :: xs) = terminates-fmap (length xs)
  (dmatch ((d r /d x) , xs))
  (dmatchTerminates (d r /d x) xs)
```

To show partial correctness of *dmatch*, we can use the transitivity of the refinement relation. If we apply transitivity, it suffices to show that *dmatch* is a refinement of *match*. Our first step is to show that the derivative operator is correct, i.e. $d\ r\ /d\ x$ matches those strings xs such that r matches $x :: xs$.

$$derivativeCorrect\ :\ \forall\ r\ \rightarrow\ Match\ (d\ r\ /d\ x)\ xs\ y\ \rightarrow\ Match\ r\ (x\ ::\ xs)\ (integralTree\ r\ y)$$

Since the definition of *d_/d_* uses the *integralTree* function, we also prove the correctness of *integralTree*.

$$integralTreeCorrect\ :\ \forall\ r\ x\ xs\ y\ \rightarrow\ Match\ (d\ r\ /d\ x)\ xs\ y\ \rightarrow\ Match\ r\ (x\ ::\ xs)\ (integralTree\ r\ y)$$

All three proofs mirror the definitions of these functions, being structured as a case distinction on the regular expression.

Before we can prove the correctness of *dmatch* in terms of *match*, it turns out that we also need to describe *match* itself better. To show *match* is refined by *dmatch*, we need to prove that the output of *dmatch* is a subset of that of *match*. Since *match* makes use of *allSplits*, we first prove that *allSplits* returns all possible splittings of a string.

$$\begin{aligned}&allSplitsComplete\ :\ (xs\ ys\ zs\ :\ String)\ (P\ :\ String \times String\ \rightarrow\ Set)\ \rightarrow\\&\quad wpMatch\ (allSplits\ xs)\ P\ \rightarrow\ (xs\ ==\ ys\ +\!\!+\ zs)\ \rightarrow\ P\ (ys\ ,\ zs)\end{aligned}$$

The proof mirrors *allSplits*, performing induction on xs. Note that *allSplitsSound* and *allSplitsComplete* together show that *allSplits xs* is equivalent to its specification $[\ \top\ ,\ \lambda\ \{(ys\ ,\ zs)\ \rightarrow\ xs\ ==\ ys\ +\ zs\}\ ]$, in the sense of the $\_\equiv\_$ relation.

Using the preceding lemmas, we can prove the partial correctness of *dmatch* by showing it refines *match*:

$$dmatchSound\ :\ \forall\ r\ xs\ \rightarrow\ wpMatch\ (match\ (r\ ,\ xs))\ \sqsubseteq\ wpMatch\ (dmatch\ (r\ ,\ xs))$$

Since we need to perform the case distinctions of *match* and of *dmatch*, the proof is longer than that of *matchSoundness*. Despite the length, most of it consists of performing the case distinction, then giving a simple argument for each case. Therefore, we omit the proof.

With the proof of *dmatchSound* finished, we can conclude that *dmatch* always returns a correct parse tree, i.e. that *dmatch* is sound. However, *dmatch* is *not* complete with respect to the *Match* relation: since *dmatch* never makes a nondeterministic choice, it will not return all possible parse trees as specified by *Match*, only the first tree that it encounters. Still, we can express the property that *dmatch* finds a parse tree if it exists. In other words, we will show that if there is a valid parse tree, *dmatch* returns any parse tree (and this is a valid tree by *dmatchSound*). To express that *dmatch* returns something, we use a trivially true postcondition, and replace the *ptAll* semantics for nondeterminism with *ptAny*:

$$\begin{aligned}&dmatchComplete\ :\ \forall\ r\ xs\ y\ \rightarrow\\&\quad Match\ r\ xs\ y\ \rightarrow\ wp\ (ptRec\ matchSpec\ ::\ ptAny\ ::\ Nil)\ (dmatch\ (r\ ,\ xs))\ (\lambda\ \_\ \rightarrow\ \top)\end{aligned}$$

The proof is short, since *dmatch* can only *fail* when it encounters an empty string and a regex that does not match the empty string, contradicting the assumption immediately:

```
dmatchComplete r Nil y H with ε? r
... | yes p = tt
... | no ¬p = ¬p (_ , H)
dmatchComplete r (x :: xs) y H y' H' = tt
```

Here we have demonstrated the power of predicate transformer semantics for effects: by separating syntax and semantics, we can easily describe different aspects (soundness and completeness) of the one definition of *dmatch*. Since the soundness and completeness result we have proved imply partial correctness, and partial correctness and termination imply total correctness, we can conclude that *dmatch* is a totally correct parser for regular languages.

Note the correspondences of this section with a Functional Pearl by Harper [Har99], which also uses the parsing of regular languages as an example of principles of functional software development. Starting out with defining regular expressions as a data type and the language associated with each expression as an inductive relation, both use the relation to implement essentially the same *match* function, which does not terminate. In both, the partial correctness proof of *match* uses a specification expressed as a postcondition, based on the inductive relation representing the language of a given regular expression. Where we use nondeterminism to handle the concatenation operator, Harper uses a continuation-passing parser for control flow. Since the continuations take the unparsed remainder of the string, they correspond almost directly to the *EParser* effect of the following section. Another main difference between our implementation and Harper's is in the way the non-termination of *match* is resolved. Harper uses the derivative operator to rewrite the expression in a standard form which ensures that the *match* function terminates. We use the derivative operator to implement a different matcher *dmatch* which is easily proved to be terminating, then show that *match*, which we have already proven partially correct, is refined by *dmatch*. The final major difference is that Harper uses manual verification of the program and our work is formally computer-verified. Although our development takes more work, the correctness proofs give more certainty than the informal arguments made by Harper. In general, choosing between informal reasoning and formal verification will always be a trade-off between speed and accuracy.

8.2 Effects as unifying theory of parsers

In the previous section, we have developed a formally verified parser for regular languages. The class of regular languages is small, and does not include most programming languages. If we want to write a parser for a larger class of languages, we fist need a practical representation. In classical logic, the most general concept of a formal language is no more than a set of strings, or a predicate over strings, represented by the type *String* → *Set*. Constructively, such predicates (even when decidable) are not very useful: the language {*xs* | *xs* is a valid proof of the Riemann Hypothesis} has no defined cardinality. To make them more amenable to reasoning, we impose structure on languages, for example by giving their definition as a regular expression. When we have a more structured grammar, we can write a parser for these grammars, and prove its partial correctness and termination, just as we did for regular expressions and *dmatch*.

One structure we can impose on languages is that we can always perform local operations, in the style of the Brzozowski derivative. This means we can decide whether a language *l* matches the empty string (as *ε*? does for regular languages), and for each character *x*, we can compute the derivative *d l /d x*, which contains exactly those *xs* such that *x* :: *xs* is in *l*. Packaging up these two operations into a record type gives the *coinductive trie* representation of a formal language, as described by Abel [Abe16]. We augment the definition by including a list of the parser's output values for the empty string, instead of a Boolean stating whether the language contains the empty string. An empty list corresponds to the original *False*, while a non-empty list corresponds to *True*.

```
record Trie (i : Size) (a : Set) : Set where
  coinductive
  field
    ε? : List a
    d_/d_ : Char → Trie j a
```

The definition of the *Trie* type is complicated by making it coinductive and using sized types. We need *Trie* to be coinductive since it appears in a negative position in the *d_/d_* operator, or viewed in another way, since the *Trie* type needs to be nested arbitrarily deeply to describe arbitrarily long strings. The sized types help Agda to check that certain definitions terminate. Despite being needed to ensure the *Trie* type is useful, the two complications do not play an important role in the remainder of the development.

Example 8.2.1. Let us look at two simple examples of definitions using the *Trie* type. The first definition, *emptyTrie*, represents the empty language. It does not contain the empty string, so *ε*? *emptyTrie* is the empty

list. It also does not contain any string of the form $x :: xs$, so the derivatives of the empty trie are all the empty trie again.

```
emptyTrie : Trie i a
ε? emptyTrie = Nil
d emptyTrie /d x = emptyTrie
```

This is also an example of why we need the coinductive structure of the *Trie* type, since the definition `d emptyTrie /d x = emptyTrie` is not productive for an inductive type.

The second example of a construction in the *Trie* type is the union operator, which is straightforward to write out.

```
_∪_ : Trie i a → Trie i a → Trie i a
ε? (t ∪ t') = ε? t ⫴ ε? t'
d (t ∪ t') /d x = (d t /d x) ∪ (d t' /d x)
```

◊

We can also take a very computational approach to languages, representing them by a parser. This parser takes a string and returns a list of successful matches, similar to the $\varepsilon?$ operator of the coinductive *Trie*.

```
Parser : Set → Set
Parser a = String → List a
```

8.2.1 Context-free grammars with *Productions*

Using a *Trie* or a *Parser* to define a language requires a lot of low-level work, since we first need to implement operations such as the union of a language or concatenation. The *Regex* representation of regular languages has such operations built-in, allowing us to have intuition on the level of grammar rather than operations. A class of languages that is more expressive than the regular languages, while remaining tractable in parsing is that of the *context-free language*. The expressiveness of context-free languages is enough to cover most programming languages used in practice [AU77]. We will represent context-free languages in Agda by giving a grammar in the style of Brink, Holdermans, and Löh [BHL10], in a similar way as we represent a regular language using an element of the *Regex* type. Following their development, we parametrize our definitions over a collection of nonterminal symbols.

```
record GrammarSymbols : Set where
  field
    Nonterminal : Set
    ⟦_⟧ : Nonterminal → Set
    _≟_ : Decidable {A = Nonterminal} _==_
```

The elements of the type *Char* are the *terminal* symbols, for example characters or tokens. The elements of the type *Nonterminal* are the *nonterminal* symbols, representing the language constructs. As for *Char*, we also need to be able to decide the equality of nonterminals. The (disjoint) union of *Char* and *Nonterminal* gives all the symbols that we can use in defining the grammar.

```
Symbol = Either Char Nonterminal
Symbols = List Symbol
```

For each nonterminal A, our goal is to parse a string into a value of type $[\![A]\!]$, based on a set of production rules. A production rule $A \to xs$ gives a way to expand the nonterminal A into a list of symbols xs, such that successfully matching each symbol of xs with parts of a string gives a match of the string with A. Since matching a nonterminal symbol B with a (part of a) string results in a value of type $[\![B]\!]$, a production

rule for A is associated with a *semantic function* that takes all values arising from submatches and returns a value of type $[\![A]\!]$, as expressed by the following type:

```
⟦_‖_⟧ : Symbols → Nonterminal → Set
⟦ Nil          ‖ A ⟧ = ⟦ A ⟧
⟦ Inl x :: xs ‖ A ⟧ = ⟦ xs ‖ A ⟧
⟦ Inr B :: xs ‖ A ⟧ = ⟦ B ⟧ → ⟦ xs ‖ A ⟧
```

Now we can define the type of production rules. A rule of the form $A \to BcD$ is represented as *prod* A (*Inr* B :: *Inl* c :: *Inr* D :: *Nil*) f for some f.

```
record Production : Set where
  constructor prod
  field
    lhs : Nonterminal
    rhs : Symbols
    sem : ⟦ rhs ‖ lhs ⟧
```

We use the abbreviation *Productions* to represent a list of productions, and a grammar will consist of the list of all relevant productions.

Our goal in this section will be to use algebraic effects to act as a unifying theory of parsing. By introducing an effect corresponding to the basic operation of parsing a string, we can implement a parser for any given context-free grammar represented by the *Productions* type. We also show how to handle this effect to give a coinductive *Trie* or a runnable *Parser* from an algebraic parser. Using predicate transformers, we verify that the parser for a context-free grammar is correct, and that it terminates as long as the grammar is left-factored, demonstrating how we can write a fully formally verified parser using algebraic effects. In the process we also demonstrate verification of a non-trivial program in predicate transformer semantics.

8.2.2 Parsing as effect

For an description of parsing based on algebraic effects, we introduce a new effect *EParser*, and use a state consisting of a *String*. The *EParser* effect has one command *Parse*, which either returns the current character in the state (advancing it to the next character) or fails if all characters have been consumed. In our current development, we do not need more commands such as an *EOF* command which succeeds only if all characters have been consumed, so we do not incorporate them. However, in the semantics we will define that parsing was successful if the input string has been completely consumed.

```
data CParser : Set where
  Parse : CParser
RParser : CParser → Set
RParser Parse = Char
EParser = eff CParser RParser

parse : {{ iP : EParser ∈ es }} → Free es Char
parse {{ iP }} = Step iP Parse Pure
```

Note that *EParse* is not sufficient alone to implement even simple parsers such as *dmatch*: we need to be able to choose between parsing the next character or returning a value for the empty string. This is why we usually combine *EParser* with the nondeterminism effect *ENondet*. However, nondeterminism and parsing is not always enough: we also need general recursion to deal with productions of the form $E \to E$.

The denotational semantics of a parser in the *Free* monad are given by handling the effects. We give two functions, one returning a *Parser* and one returning a *Trie*.

```
toParser : Free (ENondet :: EParser :: Nil) a → Parser a
toTrie : Free (ENondet :: EParser :: Nil) a → Trie ∞ a
```

```
toParser (Pure x) Nil = x :: Nil
toParser (Pure x) (_ :: _) = Nil
toParser (Step ∈Head Fail k) xs = Nil
toParser (Step ∈Head Choice k) xs = toParser (k True) xs ++ toParser (k False) xs
toParser (Step (∈Tail ∈Head) Parse k) Nil = Nil
toParser (Step (∈Tail ∈Head) Parse k) (x :: xs) = toParser (k x) xs

toTrie (Pure x) = record {ε? = x :: Nil; d_/d_ = λ _ → emptyTrie}
toTrie (Step ∈Head Fail k) = emptyTrie
toTrie (Step ∈Head Choice k) = toTrie (k True) ∪ toTrie (k False)
toTrie (Step (∈Tail ∈Head) Parse k) = record {ε? = Nil; d_/d_ = λ x → toTrie (k x)}
```

To give the predicate transformer semantics of the *EParser* effect, we need to choose the meaning of failure, for the case where the next character is needed and all characters have already been consumed. Since we want all results returned by the parser to be correct, we use demonic choice and the *ptAll* predicate transformer as the semantics for *ENondet*. Using *ptAll*'s semantics for the *Fail* command gives the following semantics for the *EParser* effect.

```
ptParse : PT^S String EParser
PTS.pt ptParse Parse P Nil = ⊤
PTS.pt ptParse Parse P (x :: xs) = P x xs
```

Example 8.2.2. With the predicate transformer semantics of *EParse*, we can define the language accepted by a parser in the *Free* monad as a predicate over strings: a string *xs* is in the language of a parser *S* if the postcondition "all characters have been consumed" is satisfied.

```
empty? : List a → Set
empty? Nil = ⊤
empty? (_ :: _) = ⊥

_∈[_] : String → Free (ENondet :: EParser :: Nil) a → Set
xs ∈[ S ] = wp^S (addState ptAll :: ptParse :: Nil) S (λ _ → empty?) xs
```

8.2.3 A parser for context-free grammars ◊

We want to show that the effects *EParser* and *ENondet* are sufficient to parse any context-free grammar, using a generally recursive function. To show this claim, we implement a function *fromProductions* that constructs a parser for any context-free grammar given as a list of *Production*s, then formally verify the correctness of *fromProductions*. Our implementation mirrors the definition of the *generateParser* function by Brink, Holdermans, and Löh, differing in the naming and in the system that the parser is written in: our implementation uses the *Free* monad and algebraic effects, while Brink, Holdermans, and Löh use a monad *Parser* that is based on parser combinators.

We start by defining two auxiliary types, used as abbreviations in our code.

```
FreeParser = Free (eff Nonterminal ⟦_⟧ :: ENondet :: EParser :: Nil)
record ProductionRHS (A : Nonterminal) : Set where
  constructor prodrhs
  field
    rhs : Symbols
    sem : ⟦ rhs ‖ A ⟧
```

The core algorithm for parsing a context-free grammar consists of the following functions, calling each other in mutual recursion:

```
fromProductions : (A : Nonterminal) → FreeParser ⟦ A ⟧
filterLHS       : (A : Nonterminal) → Productions → List (ProductionRHS A)
```

```
fromProduction : ProductionRHS A → FreeParser ⟦ A ⟧
buildParser    : (xs : Symbols) → FreeParser (⟦ xs ‖ A ⟧ → ⟦ A ⟧)
exact          : a → Char → FreeParser a
```

The main function is *fromProductions*: given a nonterminal, it selects the productions with this nonterminal on the left hand side using *filterLHS*, and makes a nondeterministic choice between the productions.

```
filterLHS A Nil = Nil
filterLHS A (prod lhs rhs sem :: ps) with A ≟ lhs
... | yes refl = prodrhs rhs sem :: filterLHS A ps
... | no _     = filterLHS A ps
fromProductions A = foldr (choice) (fail) (map fromProduction (filterLHS A prods))
```

The function *fromProduction* takes a single production and tries to parse the input string using this production. It then uses the semantic function of the production to give the resulting value.

```
fromProduction (prodrhs rhs sem) = buildParser rhs >>= λ f → Pure (f sem)
```

The function *buildParser* iterates over the *Symbols*, calling *exact* for each literal character symbol, and making a recursive *call* to *fromProductions* for each nonterminal symbol.

```
buildParser Nil = Pure id
buildParser (Inl x :: xs) = exact tt x >>= λ _ → buildParser xs
buildParser (Inr B :: xs) = call B >>= (λ x → buildParser xs >>= λ o → Pure λ f → o (f x))
```

Finally, *exact* uses the *parse* command to check that the next character in the string is as expected, and *fail*s if this is not the case.

```
exact x t =
  parse >>= λ t' →
  if t ≟ t' then Pure x else fail
```

8.2.4 Partial correctness of the parser

Partial correctness of the parser is relatively simple to show, as soon as we have a specification. Since we want to prove that *fromProductions* correctly parses any given context free grammar given as an element of *Productions*, the specification consists of a relation between many sets: the production rules, an input string, a nonterminal, the output of the parser, and the remaining unparsed string. Due to the many arguments, the notation is unfortunately somewhat unwieldy. To make it a bit easier to read, we define two relations in mutual recursion, one for all productions of a nonterminal, and for matching a string with a single production rule.

```
data _⊢_∈⟦_⟧⇒_,_ prods where
  Produce : prod lhs rhs sem ∈ prods →
            prods ⊢ xs ∼ rhs ⇒ f , ys →
            prods ⊢ xs ∈⟦ lhs ⟧⇒ f sem , ys
data _⊢_∼_⇒_,_ prods where
  Done :    prods ⊢ xs ∼ Nil ⇒ id , xs
  Next :    prods ⊢ xs ∼ ps ⇒ o , ys →
            prods ⊢ (x :: xs) ∼ (Inl x :: ps) ⇒ o , ys
  Call :    prods ⊢ xs ∈⟦ A ⟧⇒ o , ys →
            prods ⊢ ys ∼ ps ⇒ f , zs →
            prods ⊢ xs ∼ (Inr A :: ps) ⇒ (λ g → f (g o)) , zs
```

With these relations, we can define the specification *parserSpec* to be equal to $\_ \vdash \_ \in [\![\_]\!] \Rightarrow \_,\_$ (up to reordering some arguments), and show that *fromProductions* refines this specification. For the semantics of general recursion, we also make use of the specification, while for the semantics of nondeterminism, we use the *ptAll* semantics to ensure all output is correct. This gives the partial correctness term as defined below

```
pts prods = ptRec^S (parserSpec prods) :: addState ptAll :: ptParse :: Nil
wpFromProd prods = wp^S (pts prods)
partialCorrectness : (prods : Productions) (A : Nonterminal) →
  wp^S (ptSpec :: Nil) [ ⊤ , parserSpec prods A] ⊑^S wpFromProd prods (fromProductions prods A)
```

Let us fix the production rules *prods*. How do we prove the partial correctness? Since the structure of *fromProductions* is of a nondeterministic choice between productions to be parsed, and we want to show that all alternatives for a choice result in success, we will first give a lemma expressing the correctness of each alternative. Correctness in this case is expressed by the semantics of a single production rule, i.e. the $\_ \vdash \_ \sim \_ \Rightarrow \_,\_$ relation. Thus, we want to prove a lemma with a type as follows:

```
parseStep : ∀ A xs P str →
  ((o : [[ xs ‖ A ]] → [[ A ]]) (str' : String) → prods ⊢ str ∼ xs ⇒ o , str' → P o str') →
  wpFromProd prods (buildParser prods xs) P str
```

The lemma can be proved by reproducing the case distinctions used to define *buildParser*; there is no complication apart from having to use the *fold-bind* lemma to deal with the $\_ \gg\!= \_$ operator in a few places.

```
parseStep A Nil P t H = H id t Done
parseStep A (Inl x :: xs) P Nil H = tt
parseStep A (Inl x :: xs) P (x' :: t) H with x ≟ x'
... | yes refl = parseStep A xs P t λ o t' H' → H o t' (Next H')
... | no ¬p = tt
parseStep A (Inr B :: xs) P t H o t' Ho = subst _ (sym (fold-bind (buildParser prods xs) _ P _))
  (parseStep A xs _ t' λ o' t'' Ho' → H _ _ (Call Ho Ho'))
```

To combine the *parseStep* for each of the productions in the nondeterministic choice, it is tempting to define another lemma *filterStep* by induction on the list of productions. But we must be careful that the productions that are used in the *parseStep* are the full list *prods*, not the sublist *prods'* used in the induction step. Additionally, we must also make sure that *prods'* is indeed a sublist, since using an incorrect production rule in the *parseStep* will result in an invalid result. Thus, we parametrise *filterStep* by a list *prods'* and a proof that it is a sublist of *prods*. Again, the proof uses the same distinction as *fromProductions* does, and uses the *fold-bind* lemma to deal with the $\_ \gg\!= \_$ operator.

```
filterStep : ∀ prods' → (p ∈ prods' → p ∈ prods) →
  ∀ A → wp^S (ptSpec :: Nil) [ ⊤ , parserSpec prods A] ⊑^S wpFromProd prods
    (foldr (choice) (fail) (map (fromProduction prods) (filterLHS prods A prods')))
filterStep Nil subset A P xs H = tt
filterStep (prod lhs rhs sem :: prods') subset A P xs H with A ≟ lhs
filterStep (prod .A rhs sem :: prods') subset A P xs (_ , H) | yes refl
  = subst (λ f → f xs) (sym (fold-bind (buildParser prods rhs) _ P _))
  (parseStep A rhs _ xs λ o t' H' → H _ _ (Produce (subset ∈Head) H'))
  , filterStep prods' (subset ∘ ∈Tail) A P xs (_ , H)
... | no ¬p = filterStep prods' (subset ∘ ∈Tail) A P xs H
```

With these lemmas, *partialCorrectness* just consists of applying *filterStep* to the subset of *prods* consisting of *prods* itself. As for *dmatch*, we are not done at this point. To complete the verification, not only do we need to show the partial correctness of the parser, we also need to show it terminates on all input.

8.2.5 Termination of the parser

To show termination we need a somewhat more subtle argument: since we are able to call the same nonterminal repeatedly, termination cannot be shown simply by inspecting the definitions. Consider the grammar given by $E \to aE; E \to b$, where we see that the string that matches E in the recursive case is shorter than the original string, but the definition itself is of unbounded length. Fortunately for us, predicate transformer semantics allow us to give this more subtle definition of termination, in the form of the *Termination* type in Definition 5.4.2. By taking into account the current state, i.e. the string to be parsed, in the variant, we can show that a decreasing string length leads to termination.

But not all grammars feature this decreasing string length in the recursive case, with the most pathological case being those of the form $E \to E$. The issues do not only occur in edge cases: the grammar $E \to E+E; E \to 1$ representing very simple expressions will already result in non-termination for *fromProductions* as it will go in recursion on the first non-terminal without advancing the input string. Since the position in the string and current nonterminal together fully determine the state of *fromParsers*, it will not terminate. We need to ensure that the grammars passed to the parser do not allow for such loops.

Intuitively, the condition on the grammars should be that they are not *left-recursive*, since in that case, the parser should always advance its position in the string before it encounters the same nonterminal. This means that the number of recursive calls to *fromProductions* is bounded by the length of the string times the number of different nonterminals occurring in the production rules. The type we will use to describe the predicate "there is no left recursion" is constructively somewhat stronger: we define a left-recursion chain from A to B to be a sequence of nonterminals $A, \ldots, A_i, A_{i+1}, \ldots, B$, such that for each adjacent pair A_i, A_{i+1} in the chain, there is a production of the form $A_{i+1} \to B_1 B_2 \ldots B_n A_i \ldots$, where $B_1 \ldots B_n$ are all nonterminals. In other words, we can advance the parser to A starting in B without consuming a character. Disallowing (unbounded) left recursion is not a limitation for our parsers: Brink, Holdermans, and Löh [BHL10] have shown that the *left-corner transform* can transform left-recursive grammars into an equivalent grammar without left recursion. Moreover, they have implemented this transform, including formal verification, in Agda. In this work, we assume that the left-corner transform has already been applied if needed, so that there is an upper bound on the length of left-recursive chains in the grammar.

We formalize one link of this left-recursive chain in the type *LeftRec*, while a list of such links forms the *LeftRecChain* data type.

```
record LeftRec (prods : Productions) (A B : Nonterminal) : Set where
  field
    rec : prod A (map Inr xs ++ (Inr B :: ys)) sem ∈ prods
```

(We leave *xs*, *ys* and *sem* as implicit fields of *LeftRec*, since they are fixed by the type of *rec*.)

```
data LeftRecChain (prods : Productions) : Nonterminal → Nonterminal → Set where
  Nil : LeftRecChain prods A A
  _::_ : LeftRec prods B A → LeftRecChain prods A C → LeftRecChain prods B C
```

Now we say that a set of productions has no left recursion if all such chains have an upper bound on their length.

```
chainLength : LeftRecChain prods A B → ℕ
chainLength Nil = 0
chainLength (c :: cs) = Succ (chainLength cs)

leftRecBound : Productions → ℕ → Set
leftRecBound prods n = (cs : LeftRecChain prods A B) → chainLength cs < n
```

If we have this bound on left recursion, we are able to prove termination, since each call to *fromProductions* will be made either after we have consumed an extra character, or it is a left-recursive step, of which there is an upper bound on the sequence. Thus, the relation *RecOrder* will work as a recursive variant for *fromProductions*:

```
data RecOrder (prods : Productions) : (x y : Nonterminal × String) → Set where
  Adv : length str < length str' → RecOrder prods (A , str) (B , str')
  Rec : length str ≤ length str' → LeftRec prods A B → RecOrder prods (A , str) (B , str')
```

With the definition of *RecOrder*, we can complete the correctness proof of *fromProductions*, by giving an element of the corresponding *Termination* type. We assume that the length of recursion is bounded by *bound* : ℕ.

```
fromProductionsTerminates : (prods : Productions) (bound : ℕ) → leftRecBound prods bound →
  Termination (pts prods) (fromProductions prods)
Termination._≺_ (fromProductionsTerminates prods bound H) = RecOrder prods
```

To show that the relation *RecOrder* is well-founded, we need to show that there is no infinite descending chain starting from some nonterminal *A* and string *str*. The proof is based on iteration on two natural numbers *n* and *k*, which form an upper bound on the number of allowed left-recursive calls in sequence and unconsumed characters in the string respectively. Note that the number *bound* is an upper bound for *n* and the length of the input string is an upper bound for *k*. Since each nonterminal in the production will decrease *n* and each terminal will decrease *k*, we eventually reach the base case *0* for either. If *n* is zero, we have made more than *bound* left-recursive calls, contradicting the assumption that we have bounded left recursion. If *k* is zero, we have consumed more than *length str* characters of *str*, also a contradiction.

```
Termination.w-f (fromProductionsTerminates prods bound H) A str
  = acc (go A str (length str) ≤-refl bound Nil ≤-refl)
  where
  go : ∀ A str →
    (k : ℕ) → length str ≤ k →
    (n : ℕ) (cs : LeftRecChain prods A B) → bound ≤ chainLength cs + n →
    ∀ y → RecOrder prods y (A , str) → Acc (RecOrder prods) y
  go A Nil Zero ltK n cs H' (A' , str') (Adv ())
  go A (_ :: _) Zero () n cs H' (A' , str') (Adv lt)
  go A (_ :: _) (Succ k) (s≤s ltK) n cs H' (A' , str') (Adv (s≤s lt))
    = acc (go A' str' k (≤-trans lt ltK) bound Nil ≤-refl)
  go A str k ltK Zero cs H' (A' , str') (Rec lt cs')
    = magic (<→≱ (H cs) (≤-trans H' (≤-reflexive (+-zero _))))
  go A str k ltK (Succ n) cs H' (A' , str') (Rec lt c)
    = acc (go A' str' k (≤-trans lt ltK) n (c :: cs) (≤-trans H' (≤-reflexive (+-suc _ _))))
```

To show that *RecOrder* is a variant for *fromProductions*, we cannot follow the definitions of *fromProductions* as closely as we did for the partial correctness proof. We need a complicated case distinction to keep track of the left-recursive chain we have followed in the proof. For this reason, we split the *parseStep* apart into two lemmas *parseStepAdv* and *parseStepRec*, both showing that *buildParser* maintains the variant. We also use a *filterStep* that calls the correct *parseStep* for each production in the nondeterministic choice.

```
parseStepAdv : ∀ A xs str str' → length str' < length str →
  variant' (pts prods) (fromProductions prods) (RecOrder prods) A str (buildParser xs) str'
parseStepRec : ∀ A xs str str' → length str' ≤ length str →
  ∀ ys → prod A (map Inr ys ++ xs) sem ∈ prods →
  variant' (pts prods) (fromProductions prods) (RecOrder prods) A str (buildParser xs) str'
filterStep : ∀ prods' → (x ∈ prods' → x ∈ prods) →
  ∀ A str str' → length str' ≤ length str →
  variant' (pts prods) (fromProductions prods) (RecOrder prods) A str
    (foldr (choice) (fail) (map fromProduction (filterLHS A prods')))
  str'
```

In the *parseStepAdv*, we deal with the situation that the parser has already consumed at least one character since it was called. This means we can repeatedly use the *Adv* constructor of *RecOrder* to show the variant holds.

```
parseStepAdv A Nil str str' lt = tt
parseStepAdv A (Inl x :: xs) str Nil lt = tt
parseStepAdv A (Inl x :: xs) str (c :: str') lt with x ≟ c
parseStepAdv A (Inl x :: xs) (_ :: _ :: str) (.x :: str') (s≤s (s≤s lt)) | yes refl
  = parseStepAdv A xs _ _ (s≤s (≤-step lt))
... | no ¬p = tt
parseStepAdv A (Inr B :: xs) str str' lt
  = Adv lt
  , λ o str'' H → variant-fmap (pts prods) (fromProductions prods) (buildParser xs)
    (parseStepAdv A xs str str'' (≤-trans (s≤s (consumeString str' str'' B o H)) lt))
```

Here, the lemma *variant-fmap* states that the variant holds for a program of the form $S \gg\!= (Pure \circ f)$ if it does for S, since the *Pure* part does not make any recursive calls; the lemma *consumeString str' str'' B* states that the string *str''* is shorter than *str'* if *str''* is the left-over string after matching *str''* with nonterminal B.

In the *parseStepRec*, we deal with the situation that the parser has only encountered nonterminals in the current production. This means that we can use the *Rec* constructor of *RecOrder* to show the variant holds until we consume a character, after which we call *parseStepAdv* to finish the proof.

```
parseStepRec A Nil str str' lt ys i = tt
parseStepRec A (Inl x :: xs) str Nil lt ys i = tt
parseStepRec A (Inl x :: xs) str (c :: str') lt ys i with x ≟ c
parseStepRec A (Inl x :: xs) (_ :: str) (.x :: str') (s≤s lt) ys i | yes refl
  = parseStepAdv A xs _ _ (s≤s lt)
... | no ¬p = tt
parseStepRec A (Inr B :: xs) str str' lt ys i
  = Rec lt (record { rec = i })
  , λ o str'' H → variant-fmap (pts prods) (fromProductions prods) (buildParser xs)
    (parseStepRec A xs str str'' (≤-trans (consumeString str' str'' B o H) lt)
    (ys ++ (B :: Nil)) (nextNonterminal i))
```

Apart from the previous lemmas, we make use of *nextNonterminal i*, which states that the current production starts with the nonterminals $ys \mathbin{+\!\!+} (B :: Nil)$.

The lemma *filterStep* shows that the variant holds on all subsets of the production rules, analogously to the *filterStep* of the partial correctness proof. It calls *parseStepRec* since the parser only starts consuming characters after it selects a production rule.

```
filterStep Nil A str str' lt subset = tt
filterStep (prod lhs rhs sem :: prods') subset A str str' lt with A ≟ lhs
... | yes refl
  = variant-fmap (pts prods) (fromProductions prods) (buildParser rhs)
    (parseStepRec A rhs str str' lt Nil (subset ∈Head))
    , filterStep prods' (subset ∘ ∈Tail) A str str' lt
... | no ¬p = filterStep prods' (subset ∘ ∈Tail) A str str' lt
```

As for partial correctness, the main proposition consists of applying *filterStep* to the subset of *prods* consisting of *prods* itself.

Having divided the proof into the three lemmas, the remainder is straightforward. The proofs of the lemmas use induction on the production rule for *parseStepAdv* and *parseStepRec*, and induction on the list of rules for *filterStep*, and call each other as indicated.

Now that we have finished a partial correctness and termination proof of *fromProductions*, we can conclude that we have a completely formally verified parser for context-free grammars. At this point, we should note the similarities between the development of *dmatch* and of *fromProductions*: in both cases, we start out from a data type representing the grammar of a certain class of languages, with an inductively defined relation between strings and the grammar. We use the *Free* monad to write a parser. With the refinement relation, we verified partial correctness of the parser, and finally we showed that the parser terminates (for *fromProductions*, assuming the grammar is not left recursive). We can view this as a simple development methodology for formally verified programs, and this chapter as an illustration that predicate transformer semantics for algebraic effects allows this methodology.

Chapter 9

Conclusions and further work

In the thesis, we have described verification of effectful functional programs using predicate transformer semantics. We argue that effects and predicate transformers are together useful in formally verified programming, since they allow separation of the concerns of syntax, semantics and specification. The effect system, in the form of the *Free* monad, allows us to write a program, without committing to specific semantics for evaluating or verification. By expressing the semantics of a program as a predicate transformer, we can verify programs with respect to a specification which is not in the form of executable code. This contrasts with equational reasoning as applied by Gibbons and Hinze [GH11], in which specifications must be in the form of a reference implementation. If we apply Agda's interactivity to the concepts of the refinement calculus, we gain the ability to interactively derive a formally correct program from its specifications, giving a way to make the manual development method described by Morgan [Mor98] and Dijkstra [Dij76] into a computer-formalized process. In the final chapter, we demonstrate that refinement-based verification can be used in practice, by writing non-trivial parsers of formal languages and formally verifying them.

Not only do predicate transformer semantics and the refinement calculus have useful applications, we have also illustrated that they arise naturally by taking the correct generalisations of existing concepts. Starting out from verifying partial correctness of recursive functions by applying the *invariant* function, we generalize by writing *invariant* as a fold over the *Free* data type. Specialising the catamorphism of the *Free* monad to produce propositions gives us the *wp* function, which takes a predicate transformer for a single effect and computes the weakest precondition semantics for an effectful program. The natural way in which the weakest precondition arises also suggests why it is used instead of the strongest postcondition predicate transformer. A priori, the definitions of weakest precondition and strongest postcondition are symmetrical, but the weakest precondition is used throughout the refinement calculus while the strongest postcondition is not. Notably, the main concepts of weakest precondition, algebraic effects and folds are widely known among computer scientists. The new results of the thesis are not in re-introducing them individually, but showing that together they arise naturally and work well as a verification methodology.

We have still left a number of open issues. In Chapter 5, we describe combinations of effects, including how to compute the weakest precondition for a program using multiple effects. The semantics assume that all effects are handled simultaneously. One advantage of algebraic effects is the ability to use effect handlers, which deal with some effects but leave others to higher-level handlers. We can include effect handlers, such as the *catch* handler of Section 6.2, as a function from the *Free* monad to the *Free* monad. In other words, we can describe the handling of effects one by one in denotational semantics, and apply predicate transformer semantics to the denoted program, but currently we have no description of handlers that is completely founded within predicate transformer semantics.

Related to the open work on semantics of handlers is to introduce more effects and their handlers to the predicate transformer framework. Apart from general recursion, partial computation, non-determinism, mutable state and parsing, we have already written example code for a stack with *push* and *pop* operations (in the style of Example 5.1.2) and cooperative multitasking (operations are *fork* and *yield*; *yield* nondeterministically chooses the next thread to execute); interesting additional effects include self-destructing memory

(consider mutable state where the *put* operation can only be executed up to 64 times) and probabilistic nondeterminism as described by McIver and Morgan [MM04]. If we introduce all these extra effects, is the single global state variable of wp^S sufficient or do we need more sophisticated types?

Another avenue for further work is that Chapter 7 describes deriving a program from its specification, but we have not gone through a larger derivation process than the ones included in the chapter. It could be interesting to attempt a more complicated derivation than we have, perhaps porting some of the longer informal derivations described by Morgan [Mor98] and Dijkstra [Dij76] to our formal setting.

One could also investigate the relation to other descriptions of effects in functional programming. Notably, the Dijkstra monad of the language F$\star$ includes the predicate transformer semantics of a functional program in its type [Swa+11; Swa+13], in a manner similar to the *Impl* type we have defined in Section 7.2. Since the Dijkstra monad does not separate syntax and semantics as we do, it is interesting to see whether any expressiveness is lost compared to our approach.

Finally, the true test of verification frameworks is to apply them to practical software development. Now we have evidence that refinement works in the artificial environment of the thesis, so we should look for its results when applied in practice before we can be confident of the applicability in general. Still, the smaller scale examples of Chapter 8 look promising, so we have good hope that formal verification using predicate transformer semantics for algebraic effects will can be used to improve the quality of software.

Bibliography

[Abe16] Andreas Abel. "Equational Reasoning about Formal Languages in Coalgebraic Style". preprint available at http://www.cse.chalmers.se/~abela/jlamp17.pdf. Dec. 2016.

[Acz77] Peter Aczel. "An Introduction to Inductive Definitions". In: *Handbook of Mathematical Logic*. Ed. by Jon Barwise. Vol. 90. Studies in Logic and the Foundations of Mathematics. Elsevier, 1977, pp. 739–782. DOI: https://doi.org/10.1016/S0049-237X(08)71120-0.

[Agda2.6] The Agda Team. *agda/Termination.hs*. Agda version 2.6.0.1. URL: https://github.com/agda/agda/blob/2a087473e76c96c5767e7b03b0ca403fd99c8173/src/full/Agda/Termination/Termination.hs#L3.

[AU77] Alfred Aho and Jeffrey D. Ullman. *Principles of compiler design*. Reading, Mass: Addison-Wesley Pub. Co, 1977. ISBN: 0201000229.

[BHL10] Kasper Brink, Stefan Holdermans, and Andres Löh. "Dependently Typed Grammars". In: June 2010, pp. 58–79. DOI: 10.1007/978-3-642-13321-3_6.

[BP15] Andrej Bauer and Matija Pretnar. "Programming with algebraic effects and handlers". In: *Journal of Logical and Algebraic Methods in Programming* 84.1 (2015). Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011, pp. 108–123. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2014.02.001.

[Brz64] Janusz A. Brzozowski. "Derivatives of Regular Expressions". In: *J. ACM* 11.4 (Oct. 1964), pp. 481–494. ISSN: 0004-5411. DOI: 10.1145/321239.321249. URL: http://doi.acm.org/10.1145/321239.321249.

[Bv12] Ralph-Johan Back and Joakim von Wright. *Refinement calculus: a systematic introduction*. Springer Graduate Texts in Computer Science, 2012.

[Dij75a] Edsger W. Dijkstra. "Guarded commands, non-determinacy and formal derivation of programs". published as [Dij75b]. Jan. 1975. URL: http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD472.PDF.

[Dij75b] Edsger W. Dijkstra. "Guarded commands, non-determinacy and formal derivation of programs". In: *Comm. ACM* 18.8 (1975), pp. 453–457.

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1976. ISBN: 013215871X.

[GH11] Jeremy Gibbons and Ralf Hinze. "Just Do It: Simple Monadic Equational Reasoning". In: *SIGPLAN Not.* 46.9 (Sept. 2011), pp. 2–14. ISSN: 0362-1340. DOI: 10.1145/2034574.2034777.

[Har99] Robert Harper. "Proof-directed debugging". In: *Journal of Functional Programming* 9.4 (1999), pp. 463–469. DOI: 10.1017/S0956796899003378.

[HF08] Graham Hutton and Diana Fulger. "Reasoning about effects: Seeing the wood through the trees". 2008.

[Hoa69] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[KI15] Oleg Kiselyov and Hiromi Ishii. “Freer Monads, More Extensible Effects”. In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell ’15. Vancouver, BC, Canada: ACM, 2015, pp. 94–105. ISBN: 978-1-4503-3808-0. DOI: 10.1145/2804302.2804319. URL: http://doi.acm.org/10.1145/2804302.2804319.

[LJB01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. “The Size-change Principle for Program Termination”. In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’01. London, United Kingdom: ACM, 2001, pp. 81–92. ISBN: 1-58113-336-7. DOI: 10.1145/360204.360210. URL: http://doi.acm.org/10.1145/360204.360210.

[Mac71] Saunders MacLane. *Categories for the working mathematician*. Graduate Texts in Mathematics, Vol. 5. Springer-Verlag, New York-Berlin, 1971, pp. ix+262.

[Mar84] Per Martin-Löf. *Intuitionistic Type Theory*. Lecture notes by Giovanni Sambin. 1984.

[McB15] Conor McBride. “Turing-Completeness Totally Free”. In: *Mathematics of Program Construction*. Ed. by Ralf Hinze and Janis Voigtländer. Cham: Springer International Publishing, 2015, pp. 257–275. ISBN: 978-3-319-19797-5.

[Mei+91] Erik Meijer et al. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire”. In: Aug. 1991, pp. 124–144. DOI: 10.1007/3540543961_7.

[Mil+97] Robin Milner et al. *The Definition of Standard ML*. The MIT Press, 1997.

[MM04] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. SpringerVerlag, 2004. ISBN: 0387401156.

[Mog91] Eugenio Moggi. “Notions of computation and monads”. In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 55–92. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(91)90052-4.

[Mor98] Caroll Morgan. *Programming from Specifications*. 2nd ed. Prentice Hall, 1998.

[MP70] Zohar Manna and Amir Pnueli. “Formalization of Properties of Functional Programs”. In: *J. ACM* 17.3 (July 1970), pp. 555–569. ISSN: 0004-5411. DOI: 10.1145/321592.321606. URL: http://doi.acm.org/10.1145/321592.321606.

[Mur15] Dr. Tom Murphy VII Ph.D. “The Portmantout”. In: *A Record of the Proceedings of SIGBOVIK 2015*. SIGBOVIK. Apr. 2015, pp. 85–98.

[Nor07] Ulf Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. Chalmers University of Technology, 2007.

[OCo15] Liam O’Connor. *The Trouble with Typing Type as Type*. accessed 3rd Nov 2018. 2015. URL: http://liamoc.net/posts/2015-09-10-girards-paradox.html.

[PP02] Gordon Plotkin and John Power. “Notions of computation determine monads”. In: *International Conference on Foundations of Software Science and Computation Structures*. Springer. 2002, pp. 342–356.

[PP03] Gordon Plotkin and John Power. “Algebraic Operations and Generic Effects”. In: *Applied Categorical Structures* 11.1 (Feb. 2003), pp. 69–94. ISSN: 1572-9095. DOI: 10.1023/A:1023064908962. URL: https://doi.org/10.1023/A:1023064908962.

[Ren17] David Renshaw. “Efficient Computation of an Optimal Portmantout”. In: *A Record of the Proceedings of SIGBOVIK 2017*. SIGBOVIK. Apr. 2017, pp. 176–189.

[RM16] David Renshaw and Jim McCann. “A Shortmantout”. In: *A Record of the Proceedings of SIGBOVIK 2016*. SIGBOVIK. Apr. 2016, 0x4ccd69669eb3ec09434da6ad0e127cfc7b86169bf24a3fb135042d60e3ec1fdf–0x88d34007416e70009614ed5ee1bc590881f346feebcbc122d93004be50449be1.

[SB19] Wouter Swierstra and Tim Baanen. "A predicate transformer semantics for effects (Functional Pearl)". In: *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming*. ICFP '19. 2019. DOI: 10.1145/3341707.

[Swa+11] Nikhil Swamy et al. "Secure distributed programming with value-dependent types". In: *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 266–278. ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034811.

[Swa+13] Nikhil Swamy et al. "Verifying Higher-order Programs with the Dijkstra Monad". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 387–398. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2491978.

[TD91] A. S. Troelstra and D. van Dalen. "Constructivism in Mathematics, Volume 2". In: *Studia Logica* 50.2 (1991), pp. 355–356.

[Tur37] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230. eprint: http://oup.prod.sis.lan/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf. URL: https://doi.org/10.1112/plms/s2-42.1.230.

[Wad87] P Wadler. "A Critique of Abelson and Sussman or Why Calculating is Better Than Scheming". In: *SIGPLAN Not.* 22.3 (Mar. 1987), pp. 83–94. ISSN: 0362-1340. DOI: 10.1145/24697.24706. URL: http://doi.acm.org/10.1145/24697.24706.

[WSH14] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. "Effect Handlers in Scope". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. Gothenburg, Sweden: ACM, 2014, pp. 1–12. ISBN: 978-1-4503-3041-1. DOI: 10.1145/2633357.2633358.

Index