# A Predicate Transformer Semantics for Effects (Functional Pearl)

WOUTER SWIERSTRA and TIM BAANEN, Universiteit Utrecht, The Netherlands

Reasoning about programs that use effects can be much harder than reasoning about their pure counterparts. This paper presents a predicate transformer semantics for a variety of effects, including exceptions, state, non-determinism, and general recursion. The predicate transformer semantics gives rise to a refinement relation that can be used to relate a program to its specification, or even calculate effectful programs that are correct by construction.

CCS Concepts: • **Theory of computation** → *Logic and verification*; *Constructive mathematics*; *Type theory*; *Program semantics*; *Program specifications*; • **Software and its engineering** → *Functional languages*.

Additional Key Words and Phrases: predicate transformers, effects, refinement, program calculation, weakest precondition semantics, programming with dependent types, free monads, Agda

**ACM Reference Format:**
Wouter Swierstra and Tim Baanen. 2019. A Predicate Transformer Semantics for Effects (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 103 (August 2019), 26 pages. https://doi.org/10.1145/3341707

## 1 INTRODUCTION

One of the key advantages of pure functional programming is *compositionality*. Pure programs may be tested in isolation; referential transparency—the ability to freely substitute equals for equals—enables us to employ equational reasoning to prove two expressions equal [Wadler 1987]. This has resulted in a rich field of program calculation in the Bird-Meertens style [Bird 2010; Bird and De Moor 1996], transforming an inefficient executable specification to an efficient alternative implementation.

Many programs, however, are *not* pure, but instead rely on a variety of effects, such as mutable state, exceptions, general recursion, or non-determinism. Unfortunately, it is less clear how to reason about such impure programs in a compositional fashion, as we can no longer exploit referential transparency to reason about subexpressions regardless of their context.

In recent years, *algebraic effects* have emerged as a technique to incorporate effectful operations in a purely functional language [Plotkin and Power 2002; Pretnar 2010]. Algebraic effects clearly separate the syntax of effectful operations and their semantics, described by *effect handlers*. In contrast to monad transformers [Liang et al. 1995], different effects may be processed in any given order using a series of handlers.

This paper defines a predicate transformer semantics for effectful programs, culminating in a constructive framework for deriving verified effectful programs from their specifications, inspired by existing work on program calculation in the refinement calculus [Back and von Wright 2012; Morgan 1994]. We will briefly sketch the key techniques, before illustrating them with numerous examples throughout the remainder of the paper.

Authors' address: Wouter Swierstra, w.s.swierstra@uu.nl; Tim Baanen, t.baanen@uu.nl.

2475-1421/2019/8-ART103
https://doi.org/10.1145/3341707

- The syntax of effectful computations may be represented by a free monad in type theory. Assigning meaning to such free monads amounts to assigning meaning to the syntactic operations each effect provides.
- In this paper, we show how to assign *predicate transformer semantics* to computations arising from the Kleisli arrows on these free monads. This enables us to compute the weakest precondition associated with a given postcondition. By defining these semantics as a *fold* over the free monad, we can establish *compositionality* results, allowing us to decompose the verification of a large program into smaller parts. These results hold for *any* semantics defined as a fold, provided the predicate transformers are *monotone*.
- Using these weakest precondition semantics, we can define a notion of *refinement* on computations. We show how to use this refinement relation to show a program satisfies its specification, or indeed, *calculate* a program from its specification. By allowing specifications to appear in the leaves of our free monad, we can mix operations and specifications, enabling the step by step refinement of a specification to a complete program.

These principles are applicable to a range of different effects, including exceptions (Section 3), state (Section 4), non-determinism (Section 5), and general recursion (Section 6). Each section is illustrated with numerous examples, each selected for their portrayal of proof principles rather than being formidable feats of formalisation. Besides relating effectful programs to their specification, we show how programs and specifications may be mixed freely, allowing verified programs to be calculated from their specification one step at a time (Section 7).

The definitions, examples, theorems and proofs presented in this paper have all been formally verified in the dependently typed programming language Agda [Norell 2007], but the techniques translate readily to other proof assistants based on dependent types such as Idris [Brady 2013a] or Coq [Coq Development Team 2017].

## 2 BACKGROUND

### 2.1 Free Monads

We begin by defining a datatype for free monads in the style of Hancock and Setzer [2000a,b]:

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

You may want to think of C as being the type of *commands*. A computation described by the free monad Free C R either returns a result of type a or issues a command c : C. For each c : C, there is a set of responses R c. The second argument of the Step constructor corresponds to the continuation, describing how to proceed after receiving a response of type R c. It is straightforward to show that the Free C R datatype is indeed a monad:

```
map : (a → b) → Free C R a → Free C R b
map f (Pure x)   = Pure (f x)
map f (Step c k) = Step c (λ r → map f (k r))

return : a → Free C R a
return = Pure

_>>=_ : Free C R a → (a → Free C R b) → Free C R b
Pure x   >>= f = f x
Step c x >>= f = Step c (λ r → x r >>= f)
```

The examples of effects studied in this paper will be phrased in terms of such free monads; each effect, described in a separate section, chooses C and R differently, depending on its corresponding operations. This choice of operations—as is usually the case for algebraic effects—determines a syntax to which we must still assign semantics [Hyland et al. 2006].

## 2.2 Weakest Precondition Semantics

The idea of of associating weakest precondition semantics with imperative programs has a rich history, dating back to Dijkstra's Guarded Command Language [1975]. In this section, we recall the key notions that we will use throughout the remainder of the paper.

There are many different ways to specify the behaviour of a function f : a → b. One might provide a reference implementation, define a relation R : a → b → Set, or write contracts and test cases. In this paper, we will, however, focus on *predicate transformer semantics*. Where these predicate transformers traditionally relate the state space of an (imperative) program, they can be readily adapted to the functional setting.

In general, we will refer to values of type a → Set as a *predicate* on the type a; *predicate transformers* are functions between such predicates. The most famous example of a predicate transformer is the *weakest precondition*, given by the function wp below:

```
wp : (f : a → b) → (b → Set) → (a → Set)
wp f P = λ x → P (f x)
```

The wp predicate transformer maps a function f : a → b and a desired postcondition on the function's output, b → Set, to the weakest precondition a → Set on the function's input that ensures the postcondition will be satisfied. Its definition, however, is simply (reverse) function composition.

This notion of weakest precondition semantics is often too restrictive. In particular, there is no way to specify that the output is related in a particular way to the input. This can be addressed easily enough by allowing the function f to be *dependent*, yielding the following definition for weakest preconditions:

```
wp : (f : (x : a) → b x) → ((x : a) → b x → Set) → (a → Set)
wp f P = λ x → P x (f x)
```

Although this type is a bit more complicated, wp f still maps a predicate to a predicate—hence we refer to it as a predicate transformer semantics for the function f.

When working with predicates and predicate transformers, we will sometimes use the following shorthand notation:

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

Predicate transformer semantics give rise to a notion of *refinement* [Back and von Wright 2012; Morgan 1994]:

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

This refinement relation is defined between *predicate transformers*. As we will assign predicate transformer semantics to both programs and specifications, we can relate them using this refinement relation. For example, we can use this refinement relation to show a program satisfies its specification; or to show that one program is somehow 'better' than another, where the notion of 'better' arises from our choice of predicate transformer semantics.

It is straightforward to show that this refinement relation is both transitive and reflexive:

```
⊑-trans : P ⊑ Q → Q ⊑ R → P ⊑ R
⊑-refl  : P ⊑ P
```

In a pure setting, this refinement relation is not particularly interesting: the refinement relation corresponds to extensional equality between functions. The following lemma follows from the 'Leibniz rule' for equality in intensional type theory:

```
refinement : ∀ (f g : a → b) →
  (wp f ⊑ wp g) ↔ (∀ x → f x ≡ g x)
```

In the impure setting, however, we will use different notions of weakest precondition, which in turn lead to different notions of refinement.

In the remainder of this paper, we will define predicate transformer semantics for *Kleisli arrows* of the form a → Free C R b. While we could use the wp function to assign semantics to these computations directly, we are typically not interested in syntactic equality between free monads—but rather want to study the semantics of the effectful programs they represent. To define a predicate transformer semantics for effects we need to define a function of the following form:

```
pt : (a → Set) → (Free C R a → Set)
```

These functions show how to lift a predicate on the type a over an effectful computation returning values of type a. The definition of pt depends very much on the semantics we wish to assign to the effects of the free monad; the coming sections will give many examples of such semantics. Crucially, the choice of pt and our weakest precondition semantics, wp, together give us a way to assign weakest precondition semantics to Kleisli arrows representing effectful computations. Using these semantics for effectful computations, we can then specify, verify, and calculate effectful programs.

## 3 PARTIALITY

We can define the datatype for Partial computations, corresponding to the Maybe monad, by making the following choice for commands C and responses R in our Free datatype:

```
data C : Set where
  Abort : C

R : C → Set
R Abort = ⊥

Partial : Set → Set
Partial = Free C R
```

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

```
abort : Partial 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. Such smart constructors are sometimes referred to as *generic effects* in the algebraic effects literature [Plotkin and Power 2003]. With the syntax in place, we can turn our attention to verifying programs using a suitable predicate transformer semantics.

## 3.1 Example: Division

We begin by defining a small expression language, closed under division and natural numbers:

```
data Expr : Set where
  Val : Nat → Expr
  Div : Expr → Expr → Expr
```

We can specify the semantics of this language using an inductively defined *relation*:

```
data _⇓_ : Expr → Nat → Set where
  Base : Val x ⇓ x
  Step : l ⇓ v₁ → r ⇓ (Succ v₂) → Div l r ⇓ (v₁ div (Succ v₂))
```

In this definition, we rule out erroneous results by requiring that the divisor always evaluates to a non-zero value.

Alternatively we can evaluate expressions by defining a *monadic* interpreter, using the Partial monad to handle division-by-zero errors:

```
⟦_⟧ : Expr → Partial Nat
⟦ Val x ⟧      = return x
⟦ Div e₁ e₂ ⟧ = ⟦ e₁ ⟧ ≫= λ v₁ →
                 ⟦ e₂ ⟧ ≫= λ v₂ →
                 v₁ ÷ v₂
```

This interpreter uses the following division operator that may fail when the divisor is Zero:

```
_÷_ : Nat → Nat → Partial Nat
n ÷ Zero     = abort
n ÷ (Succ k) = return (n div (Succ k))
```

The division operator from the standard library (div) requires an implicit proof that the divisor is non-zero. In the case when the divisor is Zero, we fail explicitly using abort.

How can we relate these two definitions? We can assign a weakest precondition semantics to Kleisli arrows of the form a → Partial b as follows:

```
wpPartial : (f : (x : a) → Partial (b x)) → (P : (x : a) → b x → Set) → (a → Set)
wpPartial f P = wp f (mustPT P)
  where
  mustPT : (P : (x : a) → b x → Set) → (x : a) → Partial (b x) → Set
  mustPT P _ (Pure y)         = P _ y
  mustPT P _ (Step Abort _) = ⊥
```

To call the wp function we defined previously, we need to show how to transform a predicate P : b → Set to a predicate on partial results, Partial b → Set. To do so, we define the auxiliary function mustPT; the proposition mustPT P c holds when a computation c of type Partial b successfully returns a value of type b that satisfies P. The predicate transformer semantics we wish to assign to partial computations is determined by how we define mustPT. In this case, we wish to rule out failure entirely; hence the case for the Abort constructor returns the empty type. Alternatively, we could consider a different semantics for partiality, such as requiring that computations fail or return a result satisfying some desired property. As we shall see in the rest of this paper, there is often some freedom to choose different semantics for a single effect.

Now that we have a predicate transformer semantics for Kleisli arrows in general, we can study the semantics of our monadic interpreter. To do so, we pass the interpreter, ⟦_⟧, and desired postcondition, _⇓_, as arguments to wpPartial:

```
wpPartial ⟦_⟧ _⇓_ : Expr → Set
```

This results in a predicate on expressions. For all expressions satisfying this predicate, we know that the monadic interpreter and the relational specification, _⇓_, must agree on the result of evaluation.

But what does this tell us about the correctness of our interpreter? To understand the resulting predicate better, we might consider manually defining our own predicate on expressions:

```
SafeDiv : Expr → Set
SafeDiv (Val x)       = ⊤
SafeDiv (Div e₁ e₂)  =  (e₂ ⇓ Zero  →  ⊥) ∧ SafeDiv e₁ ∧ SafeDiv e₂
```

We would expect that any expression e for which SafeDiv e holds can be evaluated without encountering a division-by-zero error. Indeed, we can prove that SafeDiv is a sufficient condition for our two notions of evaluation to coincide:

```
correct : SafeDiv ⊆ wpPartial ⟦_⟧ _⇓_
```

This lemma relates the two semantics, expressed as a relation and an evaluator, for those expressions that satisfy the SafeDiv property.

We may not want to define predicates such as SafeDiv ourselves. Instead, we can define the more general predicate characterising the *domain* of a partial function:

```
dom : ((x : a) → Partial (b x)) → (a → Set)
dom f = wpPartial f (λ _ _ → ⊤)
```

Once again, we can show that the two semantics agree precisely on the domain of the interpreter.

```
sound    : dom ⟦_⟧              ⊆  wpPartial ⟦_⟧ _⇓_
complete : wpPartial ⟦_⟧ _⇓_  ⊆  dom ⟦_⟧
```

Both proofs proceed by induction on the argument expression; despite the necessity of a handful of auxiliary lemmas, they are fairly straightforward.

## 3.2 Refinement

The weakest precondition semantics on partial computations defined above give rise to a refinement relation on Kleisli arrows of the form a → Partial b. We can characterise this relation by proving the following lemma:

```
refinement : (f g : a → Partial b) →
  (wpPartial f ⊑ wpPartial g)  ↔  (∀ x → (f x ≡ g x) ∨ (f x ≡ abort))
```

Why care about this refinement relation? Not only can we use it to relate Kleisli morphisms, but it can also relate a program to a specification given by a pre- and postcondition, as we shall see shortly.

## 3.3 Example: Add

Suppose we are writing an interpreter for a simple stack machine. To interpret the ADD instruction, we replace the top two elements of the stack with their sum; this may fail if the stack has too few elements. This section shows how to prove that the obvious definition meets its specification.

We begin by defining a notion of specification in terms of a pre- and postcondition. In general, the specification of a function of type (x : a) → b x consists of a precondition on a and a postcondition relating inputs that satisfy this precondition and the corresponding outputs:

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

As is common in the refinement calculus literature, we will write [ P , Q ] for the specification consisting of the precondition P and postcondition Q. In many of our examples, the type b does not depend on x : a, motivating the following type synonym:

```
SpecK : Set → Set → Set
SpecK a b = Spec a (K b)
```

This definition uses the combinator K to discard the unused argument of type a.

Using this definition, we can define the following specification for our addition function:

```
data Add : List Nat → List Nat → Set where
  AddStep : Add (x₁ :: x₂ :: xs) ((x₁ + x₂) :: xs)
addSpec : SpecK (List Nat) (List Nat)
addSpec = [ (λ xs → length xs > 1) , Add ]
```

That is, provided we are given a list with at least two elements, we should replace the top two elements with their sum. Here we describe the desired postcondition by introducing a new datatype, Add, relating the input and output stacks.

How can we relate this specification to an implementation? We have seen how the wpPartial function assigns predicate transformer semantics to functions—but we do not yet have a corresponding predicate transformer *semantics* for our specifications. The wpSpec function does precisely this:

```
wpSpec : Spec a b → (P : (x : a) → b x → Set) → (a → Set)
wpSpec [ pre , post ] P = λ x → (pre x) ∧ (post x ⊆ P x)
```

Given a specification, Spec a b, the wpSpec function computes the weakest precondition necessary to satisfy an arbitrary postcondition P: namely, the specification's precondition should hold and its postcondition must imply P.

Using this definition we can precisely formulate the problem at hand: can we find a program add : List Nat → Partial (List Nat) that refines the specification given by addSpec:

```
correctness : wpSpec addSpec ⊑ wpPartial add
```

Defining such a program and verifying its correctness is entirely straightforward:

```
pop : List a → Partial (a × List a)
pop Nil       = abort
pop (x :: xs) = return (x , xs)
```

```
add : List Nat → Partial (List Nat)
add xs =
  pop xs ≫= λ {(x₁ , xs) →
  pop xs ≫= λ {(x₂ , xs) →
  return ((x₁ + x₂) :: xs)}}
```

We include this example here to illustrate how to use the refinement relation to relate a *specification*, given in terms of a pre- and postcondition, to its implementation. When compared to the refinement calculus, however, we have not yet described how to mix code and specifications—a point we will return to later (Section 7). Before doing so, however, we will explore several other effects, their semantics in terms of predicate transformers, and the refinement relation that arises from these semantics.

## 3.4 Alternative Semantics

The predicate transformers arising from the wpPartial function are not the only possible choice of semantics. In particular, sometimes we may use the Abort command to 'short-circuit' a computation and handle the corresponding exception. This section explores how to adapt our definitions accordingly.

Suppose we have a function that computes the product of the numbers stored in a list:

```
product : List Nat → Nat
product = foldr _*_ 1
```

If this list contains a zero, we can short circuit the computation and return zero immediately. To do so, we define the following computation:

```
fastProduct : List Nat → Partial Nat
fastProduct Nil          = return 1
fastProduct (Zero :: xs) = abort
fastProduct (k :: xs)    = map (_*_ k) (fastProduct xs)
```

To run this computation, we provide a handler that maps abort to some default value.

```
defaultHandler : a → Partial a → a
defaultHandler _ (Pure x)         = x
defaultHandler d (Step Abort _)   = d
```

Now the question arises how to assign a suitable predicate transformer semantics to the fastProduct function. We could choose to use the wpPartial function we defined previously; doing so, however, would require the input list to not contain any zeros. It is clear that we want to assign a different semantics to our aborting computations. To do so, we provide the following wpDefault function that requires the desired postcondition P holds of the default value when the computation aborts:

```
wpDefault : (d : b) → (f : a → Partial b) → (P : a → b → Set) → (a → Set)
wpDefault d f P = wp f defaultPT
  where
  defaultPT : (x : a) → Partial b → Set
  defaultPT x (Pure y)         = P x y
  defaultPT x (Step Abort _)   = P x d
```

The wpDefault function computes *some* predicate on the function's input. But how do we know that this predicate is meaningful in any way? We could compute simply return a trivial predicate that is always holds. To relate the predicate transformer semantics to the defaultHandler we need to prove the following soundness result:

```
soundness : (P : a → b → Set) → (d : b) → (c : a → Partial b) →
  ∀ x → wpDefault d c P x → P x (defaultHandler d (c x))
```

Put simply, this soundness result ensures that whenever the precondition computed by wpDefault holds, the output returned by running the defaultHandler satisfies the desired postcondition.

Now we can finally use our refinement relation to relate the fastProduct function to the original product function:

```
correctness : wp product ⊑ wpDefault 0 fastProduct
```

This example shows how to prove soundness of our predicate transformer semantics with respect to a given handler. The predicate transformers, such as wpDefault and wpPartial, return *some* predicate; by proving such soundness results, we can ensure that the semantics is meaningful. Furthermore, this example shows how different choices of handler may exist for the *same* effect—a point we shall return to when discussing non-determinism (Section 5).

## 4 MUTABLE STATE

In this section, we will explore how to develop similar predicate transformer semantics for mutable state, giving rise to a familiar Hoare logic. In what follows, we will assume a fixed type s : Set, representing the type of the state. As before, we can define the desired free monad in terms of commands C and responses R:

```
data C : Set where
  Get : C
  Put : s → C

R : C → Set
R Get     = s
R (Put _) = ⊤

State : Set → Set
State = Free C R
```

To facilitate writing stateful computations, we can define a pair of smart constructors:

```
get : State s
get = Step Get return

put : s → State ⊤
put s = Step (Put s) (\_ → return tt)
```

The usual handler for stateful computations maps our free monad, State s, to the state monad:

```
run : State a → s → a × s
run (Pure x)           s = (x , s)
run (Step Get k)       s = run (k s) s
run (Step (Put s) k) _ = run (k tt) s
```

Inspired by the previous section, we can define the following predicate transformer that for every stateful computation of type State b, maps a postcondition on b × s to the required precondition on the initial state of type s:

```
statePT : (b × s → Set) → State b → (s → Set)
statePT P (Pure x)           = λ s → P (x , s)
statePT P (Step Get k)       = λ s → statePT P (k s) s
statePT P (Step (Put s) k)   = λ _ → statePT P (k tt) s
```

We can generalise this predicate transformer slightly. As we saw before, we sometimes describe postconditions as a *relation* between inputs and outputs. In the case for stateful computations, this amounts to allowing the postcondition to also refer to the initial state:

```
statePT′ : (s → b × s → Set) → State b → (s → Set)
statePT′ P c i = statePT (P i) c i
```

In the remainder of this section, we will overload the variable name statePT to refer to both variations of the same function; the context should disambiguate the version being used.

Finally, we can define a weakest precondition semantics for Kleisli morphisms of the form a → State b:

```
wpState : (a → State b) → (P : a × s → b × s → Set) → (a × s → Set)
wpState f P (x , i) = wp f (λ c → statePT (λ j → P (x , j)) c i) x
```

Given any predicate P relating the input, initial state, final state and result of the computation, the wpState function computes the weakest precondition required of the input and initial state to ensure P holds upon completing the computation. The definition amounts to composing the wp and statePT functions we have seen previously. As we did in the previous section for wpDefault, we can prove soundness of this semantics with respect to the run function:

```
soundness : (P : a × s → b × s → Set) → (f : a → State b) →
  ∀ i x → wpState f P (x , i) → P (x , i) (run (f x) i)
```

## 4.1 Example: Tree Labelling

To show how to reason about stateful programs using our weakest precondition semantics, we revisit a classic verification problem proposed by Hutton and Fulger [2008]: 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 : SpecK (Tree a × Nat) (Tree Nat × Nat)
relabelSpec = [ K ⊤ , relabelPost ]
  where
    relabelPost : Tree a × Nat → Tree Nat × Nat → 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 + size t, where t is the initial input tree; furthermore, the output state s' should be precisely size t larger than the input state s. Note that our size function only counts the number of leaves, as these are only of interest for relabelling.

We can now define the obvious relabelling function as follows:

```
relabel : Tree a → State (Tree Nat)
relabel (Leaf x)     = map Leaf fresh
relabel (Node l r) =
  relabel l ≫= λ l' →
  relabel r ≫= λ r' →
  return (Node l' r')
```

Here 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 our wpState function to compute the weakest precondition semantics of the relabelling function and formulate the desired correctness property:

```
correctness : wpSpec relabelSpec ⊑ wpState relabel
```

The proof is interesting. Initially, it proceeds by induction on the input tree. The base case for the Leaf constructor is easy enough to discharge; the inductive case, however, poses a greater challenge. In particular, we assume that the wpSpec relabelSpec P holds for some arbitrary predicate P; the goal we wish to prove in the case for the Node constructor amounts to proving the following statement:

```
statePT (P (Node l r , i)) (relabel l ≫= (λ l' → relabel r ≫= (λ r' → Pure (Node l' r')))) i
```

At first glance, it is not at all obvious how to use our induction hypothesis! Although we can use our induction hypothesis to show P holds for l and r—it is not clear how to use this information to prove the above goal, without knowing anything further about P.

## 4.2 Compositionality

To complete the proof, we need an auxiliary lemma that enables us to prove a property of a composite computation, c ≫= f, in terms of the semantics of c and f:

```
compositionality : (c : State a) (f : a → State b) →
  ∀ i P → statePT P (c ≫= f) i ≡ statePT (wpState f P) c i
```

Most predicate transformer semantics of imperative languages have a similar rule, mapping sequential composition of programs to the composition of their associated predicate transformers:

$$\mathrm{wp}\ (c_1; c_2,\ R) = \mathrm{wp}\ (c_1,\ \mathrm{wp}\ (c_2,\ R))$$

By defining semantics for Kleisli morphisms, wpState, in terms of the predicate transformer semantics of computations, statePT, we can prove this analogous result. The proof, by induction on the stateful computation c, is trivial.

Using this compositionality property, we can massage the proof obligation of our correctness lemma to the point where we can indeed apply our induction hypotheses and complete the remaining proof obligations.

At this point, it is worth pointing out that this compositionality property does not hold exclusively for stateful computations. In fact, we can prove a more general result that holds for *any* predicate transformer semantics pt defined as a fold over the free monad:

```
compositionality : (c : Free C R a) (f : a → Free C R b) →
  ∀ P → pt (c >>= f) P ≡ pt c (wp f P)
```

Note that this proof requires that the semantics of Kleisli morphisms, wp, is defined in terms of the predicate transformer pt. If we restrict ourselves to Kleisli arrows, however, we can formulate similar properties even more succinctly. First, we can define the usual composition of Kleisli morphisms as follows:

```
_>=>_ : (a → Free C R b) → (b → Free C R c) → a → Free C R c
f >=> g = λ x → f x >>= g
```

Using this composition operator, we can show that for *any* compositional predicate transformer semantics, the following property holds:

```
compositionality-left : (f₁ f₂ : a → Free C R b) (g : b → Free C R c) →
  wp f₁ ⊑ wp f₂ →
  wp (f₁ >=> g) ⊑ wp (f₂ >=> g)
```

This is a central result of our development—it shows how the compositionality of any weakest precondition semantics is respected when considering refinement proofs. These results establish that these predicate transformers form an *ordered monad* [Katsumata and Sato 2013]. Just as referential transparency guarantees that *pure* expressions may be substituted freely during equational reasoning, this lemma guarantees that predicate transformers may be substituted freely during refinement proofs. It is worth repeating that this lemma holds for *any* predicate transformer semantics defined as a fold over a free monad.

A similar property also holds when considering refinements on the second argument of a Kleisli composition.

```
compositionality-right : (f : a → Free C R b) (g₁ g₂ : b → Free C R c) →
  wp g₁ ⊑ wp g₂ →
  wp (f >=> g₁) ⊑ wp (f >=> g₂)
```

This second property, however, only holds under the assumption that the predicate transformers computed over a free monad are *monotone*, that is to say, the function pt satisfies the following property:

```
monotonicity : P ⊆ Q → (c : Free C R a) → pt c P → pt c Q
```

This monotonicity property holds of all the predicate transformers presented in this paper and is straightforward to prove for all of them.

## 4.3 Rule of Consequence

This example illustrates how reasoning about programs written using the state monad give rise to the typical pre- and postcondition reasoning found in the verification of imperative programs. Indeed, we can also show that the familiar laws for the weakening of preconditions and strengthening of postconditions also hold:

```
weakenPre     : (P ⊆ P') → (wpSpec [ P , Q ] ⊑ wpSpec [ P' , Q ])
strengthenPost : (∀ (x : a) → Q' x ⊆ Q x) → (wpSpec [ P , Q ] ⊑ wpSpec [ P , Q' ])
```

Such laws are particularly useful when ‘bookkeeping’ large proof obligations that can sometimes arise during program verification.

## 4.4 Equations

Typically the intended semantics of algebraic effects is given by means of *equations*, identifying syntactically different terms. Indeed, the genesis of algebraic effects can be found in the work by Plotkin and Power [2002], that identified a handful of equations on relating get and put operations that completely determined the state monad. How do these equations relate to the weakest precondition semantics presented here?

Firstly, we can define the following equivalence relation between stateful computations:

```
_≃_ : State b → State b → Set
t₁ ≃ t₂ = (wpState t₁ ⊑ wpState t₂) ∧ (wpState t₂ ⊑ wpState t₁)
  where
  wpState : State b → (P : s → b × s → Set) → (s → Set)
```

Here we define a degenerate instance of the previous wpState function that works on terms of type State b rather than Kleisli arrows a → State b. To do so, we simply call the previous semantics, instantiating the type variable a to the unit type.

To establish that an equation between two terms $t_1$ and $t_2$ holds with respect to the wpState semantics, amounts to proving that $t_1 \simeq t_2$. For example, the following four laws follow immediately from our definitions for all k, x, and y:

```
law₁ : k ≃ (get >>= λ s → put s >> k)
law₂ : (get >>= λ s₁ → get >>= λ s₂ → k s₁ s₂) ≃ (get >>= λ s → k s s)
law₃ : (put y >> (put x >> k)) ≃ (put x >> k)
law₄ : (put x >> (get >>= k)) ≃ (put x >> k x)
```

More generally, we can use such an equivalence relation to verify that the predicate transformer semantics respect a set of equations that are expected to hold for a given algebraic effect.

## 5 NON-DETERMINISM

Can we repeat this construction of predicate transformer semantics for other effects? In this section, we will show how we can define a weakest precondition semantics for non-deterministic computations. Once again, we begin by defining a free monad describing the effects that can be used to describe such computations:

```
data C : Set where
  Fail : C
  Choice : C

R : C → Set
R Fail   = ⊥
R Choice = Bool
```

Here we have chosen to define two possible commands: Fail and Choice. The Fail constructor corresponds to a non-deterministic computation that will not return any results; conceptually, the Choice constructor takes two arguments and non-deterministically chooses between them. To model this, the response used in the continuation of the free monad, R Choice, is a Bool, indicating which argument to choose. We can make this more clear by defining the following shorthands for non-deterministic computations, ND, and its constructors.

```
ND : Set → Set
ND = Free C R

fail : ND a
fail = Step Fail (λ ())

choice : ND a → ND a → ND a
choice c₁ c₂ = Step Choice (λ b → if b then c₁ else c₂)
```

Next, we turn our attention to defining a suitable predicate transformer semantics on Kleisli arrows of the form (x : a) → ND (b x). There are two canonical ways to do so:

```
allPT : (P : (x : a) → b x → Set) → (x : a) → ND (b x) → Set
allPT P _ (Pure x)          = P _ x
allPT P _ (Step Fail k)     = ⊤
allPT P _ (Step Choice k)   = allPT P _ (k True) ∧ allPT P _ (k False)

wpAll : ((x : a) → ND (b x)) → (P : (x : a) → b x → Set) → (a → Set)
wpAll f P = wp f (allPT P)

anyPT : (P : (x : a) → b x → Set) → (x : a) → ND (b x) → Set
anyPT P _ (Pure x)          = P _ x
anyPT P _ (Step Fail k)     = ⊥
anyPT P _ (Step Choice k)   = anyPT P _ (k True) ∨ anyPT P _ (k False)

wpAny : ((x : a) → ND (b x)) → (P : (x : a) → b x → Set) → (a → Set)
wpAny f P = wp f (anyPT P)
```

These two predicate transformers are dual: allPT P holds for a non-deterministic computation precisely when *all* possible results satisfy P; anyPt P holds for a non-deterministic computation precisely when *some* possible result satisfies P. As we saw for other effects, we can relate both these predicates to the usual 'list handler' for non-determinism.

```
run : ND a → List a
run (Pure x)          = [ x ]
run (Step Fail _)     = Nil
run (Step Choice k)   = run (k True) ++ run (k False)
```

Finally, we can prove that our predicate transformers are sound with respect to this semantics. In the case for the wpAll function, for example, this boils down to showing:

```
wpAllSoundness : (f : (x : a) → ND (b x)) →
  ∀ P x → wpAll f P x → All (P x) (run (f x))
```

The predicate All P xs holds whenever the predicate P holds for all the elements of the list xs.

## 5.1 Refinement

These two predicate transformer semantics give rise to two different refinement relations. To characterise both these refinement relations, we begin by defining the Elem relation below, that states that a given value may be returned by a non-deterministic computation:

```
data Elem (x : a) : ND a → Set where
  Here  : Elem x (Pure x)
  Left  : Elem x (k True)  → Elem x (Step Choice k)
  Right : Elem x (k False) → Elem x (Step Choice k)
```

We can extend this relation to define a 'subset' relation on non-deterministic computations:

```
_⊆_ : ND a → ND a → Set
nd₁ ⊆ nd₂ = ∀ x → Elem x nd₁ → Elem x nd₂
```

With these relations in place, we can give the following characterisation of the refinement relation induced by both the wpAll and wpAny predicate transformers:

```
refineAll  : (f g : a → ND b) → (wpAll f  ⊑ wpAll g)  ↔ ((x : a) → g x ⊆ f x)
refineAny  : (f g : a → ND b) → (wpAny f ⊑ wpAny g) ↔ ((x : a) → f x ⊆ g x)
```

Interestingly, the case for the wpAny predicate flips the subset relation. Intuitively, if you know that a predicate P holds for *some* element returned by a non-deterministic computation, it is even 'better' to know that P holds for a non-deterministic computation that returns fewer possible results.

## 5.2 Example: Non-deterministic Deletion

To illustrate how to reason about such non-deterministic computations, we will define a function that non-deterministically removes a single element from an input list, returning both the element removed and the remaining list. Such a function can typically be used to non-deterministically inspect the elements of an input list one-by-one.

Once again, we begin by defining the specification of our function:

```
selectPost : List a → a × List a → Set
selectPost xs (y , ys) = Σ (y ∈ xs) (λ e → delete xs e ≡ ys)

removeSpec : SpecK (List a) (a × List a)
removeSpec = [ K ⊤ , selectPost ]
```

The precondition holds trivially; the postcondition consists of two parts, paired together using a Σ-type. The first component of the postcondition states that the element returned, y, is an element of the input list xs. Here we use the _∈_ relation characterising the elements of a list from the standard library. The second component of the postcondition states that removing this element from the input list produces the output list. Here we use an auxiliary function, delete, that removes an existing element from a list:

```
delete : (xs : List a) → x ∈ xs → List a
```

The definition recurses over the proof of x ∈ xs, reconstructing the output list along the way.

With the specification in place, we can define the following function that draws an element from its input list non-deterministically.

```
remove : List a → ND (a × List a)
remove Nil       = fail
remove (x :: xs) = choice (return (x , xs)) (map (retain x) (remove xs))
  where
  retain : a → a × List a → a × List a
  retain x (y , ys) = (y , (x :: ys))
```

Verifying the correctness of this function amounts to proving the following lemma:

```
removeCorrect : wpSpec removeSpec ⊑ wpAll remove
```

Note that this correctness property merely states that all the pairs returned by remove satisfy the desired postcondition. It does not require that all possible decompositions of the input list also occur as possible results of the remove function. There is a trivial proof that the fail computation also satisfies this specification:

```
trivialCorrect : wpSpec removeSpec ⊑ wpAll (const fail)
```

In other words, the lemma removeCorrect guarantees the *soundness*, but not the *completeness* of our non-deterministic computation.

We can address this by proving an additional lemma, stating that the remove function returns every possible list decomposition:

```
completeness : (y : a) (xs ys : List a) → selectPost xs (y , ys) → Elem (y , ys) (remove xs)
```

The proof proceeds by induction on the first component of the postcondition, y ∈ xs.

## 6 GENERAL RECURSION

Giving a constructive semantics for *general recursion* may seem quite difficult at first glance. There are a variety of techniques that account for general recursion in type theory [Bove et al. 2016]. Inspired by McBride [2015], however, we show how the call graph of a recursive function can be described as a free monad, to which we can in turn assign predicate transformer semantics.

Suppose we wish to define a recursive function of type (i : I) → O i, for some input type I : Set and output type O : I → Set. If the recursion is structural, we typically do so by induction on the argument of type I. If it is not, we may still want to describe the intended function and its behaviour—deferring any proof of termination for the moment. We can describe such functions as follows:

```
_ ⇀ _ : (I : Set) (O : I → Set) → Set
I ⇀ O = (i : I) → Free I O (O i)
```

Once again, we have a Kleisli arrow on the Free monad. The choice of 'commands' and 'responses', however, are somewhat puzzling at first. The intuition is that the 'effect' we are allowed to use amounts to consulting an oracle, that given an input j : I returns the corresponding output in O j. A Kleisli arrow of the form I ⇀ O takes an input i : I and may make any number of recursive calls, before returning a value in O i.

As before, we define a smart constructor to make such calls:

```
call : (i : I) → Free I O (O i)
call x = Step x Pure
```

Note that we do *not* define recursive functions—but rather define an explicit representation of the call graph of the function we wish to define. This gives a finite representation of the recursive structure of our program.

To illustrate this point, we can define McCarthy's 91-function. The recursive structure of this function is notoriously difficult to express in a total language such as Agda:

```
f91 : Nat ⇀ K Nat
f91 i with 100 lt i
f91 i | yes _ = return (i - 10)
f91 i | no _ = call (i + 11) ≫= call
```

This definition is not recursive, but merely makes the recursive structure of the function body, f91 (f91 (i + 11)), explicit. The first call corresponds to the inner application f91 (i + 11); the result of this is fed to the second call, corresponding to the outer application.

How can we reason about such functions? As is customary in the literature on predicate transformer semantics, we distinguish between *total correctness* and *partial correctness*. For the moment, we will only concern ourselves with proving *partial correctness* of our programs: provided a program terminates, it should produce the right result.

To prove partial correctness of the f91 function, we define the following specification:

```
f91Post : Nat → Nat → Set
f91Post i o with 100 lt i
f91Post i o | yes _ = o ≡ i - 10
f91Post i o | no _  = o ≡ 91

f91Spec : SpecK Nat Nat
f91Spec = [ K ⊤ , f91Post ]
```

Although we cannot directly run 'recursive' functions defined in this style, such as the f91 function, we can reason about their correctness. To do so, we would like to show that a Kleisli arrow I ⇸ O satisfies some specification of type Spec I O. To achieve this, we begin by defining an auxiliary function, invariant, that asserts that a given call-graph Free I O (O i) respects the invariant arising from a given specification:

```
invariant : (i : I) → Spec I O → Free I O (O i) → Set
invariant i [ pre , post ] (Pure x)   = pre i → post i x
invariant i [ pre , post ] (Step j k) = (pre i → pre j)
                                        ∧ ∀ o → post j o → invariant i [ pre , post ] (k o)
```

If there are no recursive calls, the postcondition must hold, provided the precondition does. If there is a recursive call on the argument j : I, the precondition must hold for j, assuming it holds initially for i. Furthermore, for any for result o : O j satisfying the postcondition, the remaining continuation k o must continue to satisfy the desired specification.

Using this definition, we can now formulate a predicate transformer semantics for Kleisli arrows of the form I ⇸ O.

```
wpRec : Spec I O → (f : I ⇸ O) → (P : (i : I) → O i → Set) → (I → Set)
wpRec spec f P i = wpSpec spec P i ∧ invariant i spec (f i)
```

In contrast to the semantics we have seen so far, the wpRec function requires a *specification* as argument to determine a semantics of a *computation*. This is analogous to how imperative programs require an explicit loop invariant: assigning semantics to recursive functions requires an explicit specification. The predicate transformer semantics wpRec states that this specification is indeed satisfied if any recursive call respects the corresponding invariant.

Using the wpRec function, we can formulate the partial correctness of the f91 function as follows.

```
f91Partial-correctness : wpSpec f91Spec ⊑ wpRec f91Spec f91
```

The proof mimics the definition of the f91 function. After comparing the input i to 100, the base case follows immediately. The recursive case, however, requires various auxiliary lemmas stating properties of subtraction.

What do we know about the soundness of wpRec? The semantics compute some predicate on the input I, but we would like to have some guarantee that this predicate is meaningful. Unfortunately, there is no way to run arbitrary recursive functions without compromising the soundness of Agda's type system. There are, however, a variety of techniques to guarantee the termination of recursive functions such as: bounding the number of iterations, generating a coinductive trace, adding a coinductive fixpoint operator [Capretta 2005], proving the recursive calls are well-founded, or performing induction on an auxiliary data structure [Bove and Capretta 2005].

We can prove a simple soundness result in terms of the 'petrol-driven semantics' that runs a computation for a fixed number of steps.

```
petrol : (f : I ⇸ O) → Free I O a → Nat → Partial a
petrol f (Pure x)    n         = return x
petrol f (Step _ _) Zero       = abort
petrol f (Step c k) (Succ n)   = petrol f (f c ≫= k) n
```

The last case is the only interesting one: it unfolds the function f once, decrementing the number of steps remaining. We would like to use this semantics, to formulate and prove the soundness of wpRec. There is one problem: the petrol function may fail to return a result and abort. Fortunately, we can define yet another predicate transformer semantics for partial computations:

```
mayPT : (a → Set) → (Partial a → Set)
mayPT P (Pure x)          = P x
mayPT P (Step Abort _)    = ⊤
```

With these definitions in place, we can finally formulate and prove a soundness result regarding our wpRec semantics:

```
soundness : (f : I ⇸ O) (spec : Spec I O) (P : (i : I) → O i → Set) →
  (∀ i → wpRec spec f P i) → ∀ n i → mayPT (P i) (petrol f (f i) n)
```

This lemma guarantees that—under the assumption that wpRec holds for all inputs—whenever the petrol-driven semantics manages to produce a result, this result is guaranteed to satisfy the predicate P. We could show similar soundness results for the other handlers that McBride [2015] proposes for general recursion; this soundness result, however, provides at least some evidence that the predicate transformer semantics for recursion, wpRec, is correct.

## 7 STEPWISE REFINEMENT

In the examples we have seen so far, we have related a *complete* program to its specification. Most work on the refinement calculus, however, allows programs and specifications to mix freely, thereby enabling the step-by-step refinement of a specification into an executable program. How can we support this style of program calculation using the predicate transformer semantics we have seen so far?

Until now we have concerned ourselves with free monads of the form Free C R a and the Kleisli arrows that produce them. Such free monads give a structured representation of a series of interactions, (potentially) ending in a value of type a. By varying this information stored in the leaves of the free monad, we can mix unfinished specifications and program fragments.

To this end, we begin by defining the following shorthand for specifications on values, rather than the specifications on (Kleisli) arrows we have considered previously:

```
SpecVal : Set → Set
SpecVal a = SpecK ⊤ a
```

These specifications consist of a precondition of type Set and a predicate a → Set. Next, we can define the datatype I a, corresponding to either a specification on a or a value of type a.

```
data I (a : Set) : Set where
  Done : a → I a
  Hole : SpecVal a → I a
```

Here we use the Hole constructor to store a specification, corresponding to some unfinished part of the program being calculated. We can assign a predicate transformer semantics to values of type I a easily enough, reusing our previous wpSpec function:

```
ptI : I a → (a → Set) → Set
ptI (Done x)    P = P x
ptI (Hole spec) P = wpSpec spec P tt
```

Furthermore, given the commands C and responses R that determine the operations of a free monad, we can define the following datatype for partially finished programs:

```
M : Set → Set
M a = Free C R (I a)
```

The type M a describes computations that *mix* code and specifications. A value of type M a consists of a number of operations, given by the Step constructor of the Free C R type constructor; in contrast to free monads we have seen so far, however, the leaves contain either values of type a or specifications, representing unfinished parts of the program's derivation.

In what follows, we will be careful to distinguish *executable code*—that is programs without specification fragments—from programs, that may still contain specifications. The following predicate characterises the executable fragment of M a:

```
isExecutable : M a → Set
isExecutable (Pure (Done _)) = ⊤
isExecutable (Pure (Hole _)) = ⊥
isExecutable (Step c k)      = ∀ r → isExecutable (k r)
```

Although we have defined the syntactic structure of our mixed computations, M a, we have not yet given their semantics. We can reuse the notion of weakest precondition on I to define a notion of weakest precondition for the computations in M. To do so, however, we need to assume that we have some weakest precondition semantics for Kleisli morphisms:

```
wp : ((x : a) → Free C R (b x)) → ((x : a) → b x → Set) → (a → Set)
```

We have seen many examples of such semantics in the previous sections for specific choices of C and R. We can now assign semantics to 'unfinished' programs as follows:

```
wpM : ((x : a) → M (b x)) → ((x : a) → b x → Set) → (a → Set)
wpM f P x = wp f (λ x ix → ptI ix (P x)) x
```

The crucial step here is to transform the argument predicate P to work on specifications or values of type I a, using the ptI function defined above.

## 7.1 Defining Derivations

The wpM function assigns a predicate transformer semantics to unfinished programs, where the leaves of a free monad may consist of values or specifications. We can use this semantics to derive a program from its specification in series of refinement steps. A *program derivation* consists of a series of refinement steps from some initial specification:

$$\text{wpSpec spec} \sqsubseteq \text{wpM } i_1 \sqsubseteq \text{wpM } i_2 \sqsubseteq \ldots \sqsubseteq \text{wp c}$$

Here the intermediate steps ($i_1$, $i_2$, and so forth) may mix specifications and effectful computations; the final program, c, must be executable.

Before calculating an example program in this style, we will develop a handful of auxiliary definitions, specialised to the stateful computations described in Section 4; it should be straightforward to adapt the definitions to work for other effects. In what should be a familiar pattern, we begin by defining a handful of smart constructors:

```
done : a → M a
get  : M Nat
put  : Nat → M ⊤
```

Note that these smart constructors now produce computations in M, mixing effects and specifications, rather than the free monad, State, we saw previously. We can define the following postconditions characterising get and put:

```
getPost : Nat → Nat × Nat → Set
getPost t (x , t') = (t ≡ x) ∧ (t ≡ t')
putPost : Nat → Nat → ⊤ × Nat → Set
putPost t _ (_ , t') = t ≡ t'
```

As you would expect, these postconditions state that get returns the current state but does not modify it; the put command overwrites the current state. We can prove that get and put commands satisfy these postconditions using our wpM semantics:

```
getCorrect : ∀ pre   → wpSpec [ pre , (λ i o → pre i ∧ getPost i o) ]   ⊑ ptM get
putCorrect : ∀ pre x → wpSpec [ pre , (λ i o → pre i ∧ putPost x i o) ] ⊑ ptM (put x)
```

While we will not use these properties in the remainder of our calculation directly, they form an important sanity check, guaranteeing that the specifications we have chosen for get and put are correct.

To derive a program from its specification, we will perform a series of refinement steps. While we could use the transitivity of the refinement relation to chain together various intermediate programs explicitly, we take a slightly different approach. Each refinement step is allowed to introduce a single new command of type C, thereby changing the remaining refinement problem. We can try to make this manifest in the following (incomplete) datatype:

```
data Derivation (spec : SpecVal b) : Set where
  Done : (x : b) → wpSpec spec ⊑ ptM (done x)    → Derivation spec
  Step : (c : C) → (∀ (r : R c) → Derivation ...) → Derivation spec
```

A value of type Derivation spec describes a series of refinement steps, yielding a value satisfying the desired specification. In the base case, the derivation is done and returns a value x : b that satisfies the desired specification; otherwise, we perform the command c and continue the remainder of the program derivation— but how does introducing the command c modify the remaining specification problem? To answer this question, we need to make explicit what the effect of each command is on the current specification goal. Fortunately, the specifications of get and put defined previously will allow us to do just that.

We would like to define a function that, given a new command c, computes the specification of the remaining continuation:

```
step : (c : C) (spec : SpecVal (b × Nat)) → SpecK (R c × Nat) (b × Nat)
```

To achieve this, we define a pair of predicate transformers that use the postcondition associated with the command c to modify the current specification. The first such transformer computes a new precondition of type b → Set, given the current specification's precondition P : a → Set.

```
_◃_ : (Q : a → b → Set) (P : a → Set) → b → Set
_◃_ Q P = λ y → Σ a (λ x → P x ∧ Q x y)
```

This limited form of relational composition requires an intermediate result, x : a, and proofs that x satisfies both P and Q. Our second transformer computes a new postcondition of the remaining specification:

```
_▹_ : (Q : a → b → Set) → (SpecK a c) → b → c → Set
_▹_ Q [ pre , post ] = λ y z → ∀ x → pre x ∧ Q x y → post x z
```

The new postcondition requires that the original postcondition post holds, whenever the post-condition Q—associated with the new command we are introducing—and initial precondition pre hold.

Using these definitions, we can complete the definition of the step function:

```
step : (c : C) (spec : SpecVal (b × Nat)) → SpecK (R c × Nat) (b × Nat)
step Get     [ pre , post ] = [ getPost ◃ pre , getPost ▹ [ pre , post ] ]
step (Put x) [ pre , post ] = [ (putPost x) ◃ pre , (putPost x) ▹ [ pre , post ] ]
```

The step function uses the above operators, together with the postconditions associated with get and put, to compute a new specification for the remaining derivation.

Before completing the definition of derivations, there is one last issue to address. Using the step function, we can compute a new specification for the remaining continuation after a put or get command. Our derivations, however, only contain specifications of *values*—represented by SpecVal—rather than the specification of a function, represented by Spec. Fortunately, we can easily convert between the two:

```
intros : SpecK (a × Nat) (b × Nat) → a → SpecVal (b × Nat)
```

The intros function (partially) applies the precondition and postcondition to the argument of type a; the name is suggestive of the corresponding Coq tactic. Finally, we can complete the definition of derivations using the step and intros functions.

```
data Derivation (spec : SpecVal (b × Nat)) : Set where
  Done : (x : b) → wpSpec spec ⊑ ptM (done x) → Derivation spec
  Step : (c : C) → (∀ (r : R c) → Derivation (intros (step c spec) r)) → Derivation spec
```

The interesting case is in the continuation of the Step constructor. For each possible response r : R c, we need to provide a derivation of the remaining specification. Note that we have specialized this type to work over stateful computations by requiring a specification on the result and output state.

It is no coincidence that the structure of our derivations mimics that of the computations described by our Free datatype. One advantage of giving a manifest representation of such derivations is that we can easily extract executable code from a given derivation:

```
extract : (spec : SpecVal (b × Nat)) → Derivation spec → State b
extract _ (Done x _) = Pure x
extract _ (Step c k)  = Step c (λ r → extract _ (k r))
```

Furthermore, we can prove that any extracted program does indeed satisfy its intended specification.

## 7.2 Case Study: Calculating the Maximum

With our definition of derivations in place, we can finally turn our attention to an example calculation. In this section, we will derive a simple program that computes the greatest element of a non-empty list. Although there is a simple, purely functional implementation, we choose to derive a stateful version that stores the greatest element encountered so far. Once again, we stress that our aim is *not* to perform complex program calculations, but rather to illustrate the definitions we have given in the previous pages.

We begin by defining the specification of our desired function:

```
maxPre : List Nat × Nat → Set
maxPre (xs , i) = (i ≡ 0) ∧ (¬ (xs ≡ Nil))
maxPost : List Nat × Nat → Nat × Nat → Set
maxPost (xs , i) (o , _) = All (o ⩾_) xs ∧ (o ∈ xs)
maxSpec = [ maxPre , maxPost ]
```

Given an initial state 0 and non-empty list xs, our max function should find a number o that is both greater than or equal to all the elements of xs and also occurs in xs.

To calculate a suitable implementation amounts to proving that Derivation maxSpec is inhabited. A direct proof quickly fails, as the statement is not general enough to reuse our induction hypothesis. We can prove, however, the following lemma that allows us to modify the target of our derivation:

```
refineDerivation : wpSpec spec ⊑ wpSpec spec' → Derivation spec' → Derivation spec
```

The proof recurses over the derivation and relies on the monotonicity of our predicate transformers. In particular, we can use this lemma, together with the weakenPre and strengthenPost lemmas from Section 4, to generalise our specification and perform the bookkeeping necessary on the intermediate specifications we encounter during derivation.

We can use the refineDerivation lemma to show that the following formulation of the desired postcondition also suffices:

```
maxPost : List Nat × Nat → Nat × Nat → Set
maxPost (xs , i) (o , _) = All (o ⩾_) (i :: xs) × (o ∈ (i :: xs))
```

Now we can finally turn our attention to completing our derivation. Here is where using an *interactive* proof assistant such as Agda is invaluable. We begin by performing induction on the input list, and splitting our goal accordingly. We can then repeatedly fill in parts of the program, one command at a time, inspecting the remaining derivation problem as we go. Along the way, we can call the refineDerivation lemma to simplify any open specifications, if they grow too complex. Indeed, if we replace the proofs in our derivation with ellipses, we can read off the program we have calculated directly:

```
max : (xs : List Nat) → Derivation (intros maxSpec xs)
max Nil       = Step Get λ i →
                Done i ...
max (x :: xs) = Step Get λ i →
                if x <? i
                  then (λ lt   → ... (max xs))
                  else (λ geq → Step (Put x) (... (max xs)))
```

The proof steps themselves are not particularly hard. In both recursive calls, we need to perform a call to the refineDerivation lemma and prove that the result of the call is strong enough to fulfil the current proof goal. In the base case, we can give a direct proof that the current state i satisfies the desired specification.

One result of formulating the step function in terms of the predicate transformers ◃ and ▹ is that the new goals after issuing a particular command are computed automatically from these definitions, but may be more verbose than what a programmer might write by hand, for example, by containing superfluous equalities between intermediate states. We could avoid this by defining custom transformations on our specifications for put and get directly, at the expense of losing some generality. Nonetheless, it is encouraging to see that we can calculate a program *together with* our proof assistant, rather than verify a program post-hoc.

## 8 DISCUSSION

Throughout this paper, we have had to choose between presenting the most general definition possible and a less general choice, that sufficed for the examples we intended to cover. When possible, we have favoured simplicity over generality. For instance, the type of our specifications can be generalised even further, making the postcondition dependent on the precondition:

```
record Spec (a : Set) (b : a → Set) : Set where
  field
    pre : a → Set
    post : (x : a) → pre x → b x → Set
```

The resulting definition is that of an *indexed container* [Altenkirch et al. 2015]. We have chosen to present a simply-typed version of a function—even if a more general dependently typed alternative exists—when the added generality was unnecessary for our examples.

Throughout this paper, we have not concerned ourselves with issues of size. Yet some of our definitions, such as those for specifications and derivations, are too large to live in Set. In the accompanying Agda development, we show how a suitable choice of universe level can be used to stratify these definitions; for the sake of presentation, however, we have omitted these annotations in the code in this paper.

### 8.1 Related Work

Traditionally, reasoning about pure functional programs is done through equational reasoning. There are several attempts to extend these techniques to the kinds of effectful programs we have presented in this paper [Gibbons 2013; Gibbons and Hinze 2011; Hutton and Fulger 2008].

There is a great deal of work studying how to reason about effects in type theory [Brady 2013b; Nanevski and Morrisett 2005; Nanevski et al. 2006, 2008; Swierstra 2009a,b; Swierstra and Altenkirch 2007]. F★ has introduced the notion of Dijkstra monads [Swamy et al. 2011, 2013] to collect the verification conditions arising from a program using a weakest precondition semantics. The wpSpec function corresponds to the predicate transformer semantics that F★ associates with specifications [Ahman et al. 2017; Swamy et al. 2016]. The compositionality results presented here correspond to subtyping of the sequential composition in F★. Where F★ typically uses an SMT solver to resolve verification conditions, the use of an interactive theorem prover and higher-order logic may facilitate the verification of properties that are difficult to encode in the SMT solver's logic. More recently, Maillard et al. [2019] have investigated the predicate transformer and specification semantics of effectful programs, similar to the effects presented here.

There is also a great deal of existing work on using interactive theorem provers to perform program calculation. Back and von Wright [1989] have given a formalisation of several notions, such as weakest precondition semantics and the refinement relation, in the interactive theorem prover HOL. This was later extended to the *Refinement Calculator* [Butler et al. 1997], that built a new GUI on top of HOL. Dongol et al. [2015] have extended these ideas even further in HOL, adding a separation logic and its associated algebraic structure. Boulmé [2007] has given a direct embedding of the refinement calculus in Coq. Alpuim and Swierstra [2018; 2016] have given an similar development to the one presented here, tailored specifically to stateful computations. Chlipala et al. [2017] have recently proposed using the Coq proof assistants to derive correct programs from their specification. Their work on the Fiat framework is geared towards describing *data refinement* and the synthesis of abstract datatypes, packaging methods and data.

## 8.2 Further Work

This paper does not yet consider *combinations* of different effects. In principle, however, we believe it should be possible to take the coproduct of our free monads in the style of Swierstra [2008] to combine the different effects syntactically. We hope that the composition of predicate transformers can be used to assign semantics to programs using a variety of different effects—much as we defined a semantics of mixed programs and specifications from their constituent parts. Similar ideas have already been explored when embedding algebraic effects in Haskell by Wu et al. [2014].

There are well-known efficiency problems when working with free monads directly, as we have done here. While efficiency was never our primary concern, we hope that we might adapt existing solutions to avoid these issues [Kiselyov and Ishii 2015; Voigtländer 2008].

## 8.3 Conclusions

We have presented several small example programs and verified their correctness. The aim of these examples is to *illustrate* our definitions and *validate* our design choices, rather than solve any realistic verification challenge. There is a great deal of further engineering work necessary to ensure these ideas scale easily beyond such simple examples: custom tactics and notation could help facilitate program calculation; further proof automation is necessary to keep the complexity of intermediate calculations in check. Nonetheless, we believe that the predicate transformer semantics defined in this paper offer a *functional* account of effects that is worth exploring further.

## ACKNOWLEDGMENTS

We would like to thank the members of Utrecht University's Software Technology Reading Group and the members of IFIP WG 2.1 for feedback on this work. Furthermore, the ICFP reviewers provided us with insightful and detailed feedback for which we are most grateful.

## REFERENCES

Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra Monads for Free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 515–529. https://doi.org/10.1145/3009837.3009878

João Alpuim and Wouter Swierstra. 2018. Embedding the refinement calculus in Coq. *Science of Computer Programming* 164 (2018), 37–48.

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* 25 (2015). https://doi.org/10.1017/S095679681500009X

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

R. J. R. Back and J. von Wright. 1989. Refinement Concepts Formalized in Higher Order Logic. *Formal Aspects of Computing* 2 (1989).

Richard Bird. 2010. *Pearls of Functional Algorithm Design*. Cambridge University Press.

Richard Bird and Oege de Moor. 1996. *The algebra of programming*. Prentice Hall.

Sylvain Boulmé. 2007. Intuitionistic refinement calculus. In *Typed Lambda Calculi and Applications*. Springer, 54–69.

Ana Bove and Venanzio Capretta. 2005. Modelling general recursion in type theory. *Mathematical Structures in Computer Science* 15, 4 (2005), 671–708.

Ana Bove, Alexander Krauss, and Matthieu Sozeau. 2016. Partiality and recursion in interactive theorem provers – an overview. *Mathematical Structures in Computer Science* 26, 1 (2016), 38–88. https://doi.org/10.1017/S0960129514000115

Edwin Brady. 2013a. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

Edwin Brady. 2013b. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, 133–144. https://doi.org/10.1145/2500365.2500581

M. J. Butler, J. Grundy, T. Långbacka, R. Ruksenas, and J. von Wright. 1997. The Refinement Calculator: Proof Support for Program Refinement. In *Proc. Conf. Formal Methods Pacific'97, Springer Series in Discrete Mathematics and Theoretical Computer Science*, L. Groves and S. Reeves (Eds.). 40–61.

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* 1, 2 (2005), 1–28.

Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. 2017. The end of history? Using a proof assistant to replace language design with library design. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

The Coq Development Team. 2017. *The Coq Proof Assistant Reference Manual, version 8.7.* ADT Coq (Action for Technological Development). http://coq.inria.fr

Edsger W. Dijkstra. 1975. Guarded commands, non-determinacy and formal. derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.

Brijesh Dongol, Victor B.F. Gomes, and Georg Struth. 2015. A Program Construction and Verification Tool for Separation Logic. In *Mathematics of Program Construction (LNCS)*, Vol. 9129. Springer, 137–158.

Jeremy Gibbons. 2013. Unifying Theories of Programming with Monads. In *Unifying Theories of Programming*, Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi (Eds.). Springer, 23–67.

Jeremy Gibbons and Ralf Hinze. 2011. Just Do It: Simple Monadic Equational Reasoning. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, 2–14. https://doi.org/10.1145/2034773.2034777

Peter Hancock and Anton Setzer. 2000a. Interactive Programs in Dependent Type Theory. In *Computer Science Logic: 14th InternationalWorkshop, CSL 2000 Annual Conference of the EACSL Fischbachau, Germany, August 21 – 26, 2000 Proceedings*, Peter G. Clote and Helmut Schwichtenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 317–331. https://doi.org/10.1007/3-540-44622-2_21

Peter Hancock and Anton Setzer. 2000b. Specifying interactions with dependent types. In *Workshop on Subtyping and Dependent Types in Programming*. INRIA, Ponte de Lima, Portugal, Article 5, 13 pages.

Graham Hutton and Diana Fulger. 2008. Reasoning about effects: Seeing the wood through the trees. (2008).

Martin Hyland, Gordon Plotkin, and John Power. 2006. Combining effects: Sum and tensor. *Theoretical Computer Science* 357, 1-3 (2006), 70–99.

Shin-ya Katsumata and Tetsuya Sato. 2013. Preorders on monads and coalgebraic simulations. In *International Conference on Foundations of Software Science and Computational Structures*. Springer, 145–160.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, 94–105. https://doi.org/10.1145/2804302.2804319

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *Conference record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 333–343.

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martinez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. (2019). arXiv:cs.PL/1903.01237

Conor McBride. 2015. Turing-completeness totally free. In *International Conference on Mathematics of Program Construction*. Springer, 257–275.

Carroll Morgan. 1994. *Programming from specifications.* Prentice Hall,.

Aleksandar Nanevski and Greg Morrisett. 2005. *Dependent Type Theory of Stateful Higher-Order Functions.* Technical Report TR-24-05. Harvard University.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and Separation in Hoare Type Theory. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM, 62–73. https://doi.org/10.1145/1159803.1159812

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent Types for Imperative Programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, 229–240. https://doi.org/10.1145/1411204.1411237

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Chalmers University of Technology.

Gordon Plotkin and John Power. 2002. Notions of computation determine monads. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 342–356.

Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied categorical structures* 11, 1 (2003), 69–94.

Matija Pretnar. 2010. *Logic and handling of algebraic effects*. Ph.D. Dissertation. The University of Edinburgh.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 266–278. https://doi.org/10.1145/2034773.2034811

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 256–270. https://doi.org/10.1145/2837614.2837655

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-order Programs with the Dijkstra Monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 387–398. https://doi.org/10.1145/2491956.2491978

Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Wouter Swierstra. 2009a. *A functional specification of effects*. Ph.D. Dissertation. University of Nottingham.

Wouter Swierstra. 2009b. A Hoare Logic for the State Monad. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 440–451.

Wouter Swierstra and João Alpuim. 2016. From Proposition to Program - Embedding the Refinement Calculus in Coq. In *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings (LNCS)*, Vol. 9613. Springer, 29–44. https://doi.org/10.1007/978-3-319-29604-3_3

Wouter Swierstra and Thorsten Altenkirch. 2007. Beauty in the Beast: a functional semantics for the awkward squad. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07)*. ACM, 25–36. https://doi.org/10.1145/1291201.1291206

Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–403.

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

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. ACM, 1–12. https://doi.org/10.1145/2633357.2633358