A predicate transformer semantics for effects

Wouter Swierstra and Anne Baanen

Utrecht University
Constructive mathematics and computer programming†

BY P. MARTIN-LOF

Department of Mathematics, University of Stockholm, Box 6701, S-113 85 Stockholm, Sweden

If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods of computation that it is the computer’s duty to execute (a difference that Dijkstra has referred to as the difference between computer science and computing science), then it no longer seems possible to distinguish the discipline of programming from constructive mathematics. This explains why the intuitionistic theory of types (Martin-Löf 1975 In Logic Colloquium 1973 (ed. H. E. Rose & J. C. Shepherdson), pp. 73–118. Amsterdam: North-Holland), which was originally developed as a symbolism for the precise codification of constructive mathematics, may equally well be viewed as a programming language. As such it provides a precise notation not only, like other programming languages, for the programs themselves but also for the tasks that the programs are supposed to perform. Moreover, the inference rules of the theory of types, which are again completely formal, appear as rules of correct program synthesis. Thus the correctness of a program written in the theory of types is proved formally at the same time as it is being synthesized.
The day was closed by P. Martin-Löf... But the 50 minutes were not enough to introduce an ignorant audience to intuitionistic type theory to the extent that it could follow a comparison with Scottery. He was a very sympathetic speaker and convinced at least me that something (possibly even of great conceptual elegance) was going on.
Can we give a *constructive* account of Dijkstra’s weakest precondition semantics in Martin-Löf type theory?
A predicate transformer semantics for effects

- A **predicate** on type \(a\) is some value of type \(a \rightarrow \text{Set}\)

\[ a \rightarrow \text{Set} \]
A predicate transformer semantics for effects

• A **predicate** on type $a$ is some value of type $a \rightarrow \text{Set}$

\[
a \rightarrow \text{Set}
\]

• A **predicate transformer** maps predicates to predicates:

\[
(a \rightarrow \text{Set}) \rightarrow (b \rightarrow \text{Set})
\]
A predicate transformer semantics for effects

- A **predicate** on type `a` is some value of type

  \[ a \rightarrow \text{Set} \]

- A **predicate transformer** maps predicates to predicates:

  \[(a \rightarrow \text{Set}) \rightarrow (b \rightarrow \text{Set})\]

- A **predicate transformer semantics** assigns a predicate transformer to

  \[ wp : (a \rightarrow b) \rightarrow (b \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set}) \]

  \[ wp = \ldots \]
A predicate transformer semantics for effects

- A **predicate** on type \( a \) is some value of type \( a \rightarrow \text{Set} \)

- A **predicate transformer** maps predicates to predicates:
  \[
  (a \rightarrow \text{Set}) \rightarrow (b \rightarrow \text{Set})
  \]

- A **predicate transformer semantics** assigns a predicate transformer to
  \[
  \text{wp} : (a \rightarrow b) \rightarrow (b \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set})
  \]
  \[
  \text{wp} = \cdot
  \]
A predicate transformer semantics for effects

- A **predicate** on type \( a \) is some value of type \( a \rightarrow \text{Set} \).

- A **predicate transformer** maps predicates to predicates:

  \[(a \rightarrow \text{Set}) \rightarrow (b \rightarrow \text{Set})\]

- A **predicate transformer semantics** assigns a predicate transformer to

  \[\text{wp} : (a \rightarrow b) \rightarrow (b \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set})\]

  \[\text{wp} = \cdot\]

  Or more generally, using *dependent types*

  \[\text{wp} : ((x : a) \rightarrow b \, x) \rightarrow (\forall \, x \rightarrow b \, x \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set})\]
We can illustrate the general principle using a (trivial) example:

\[
wp : (a \rightarrow b) \rightarrow (b \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set})
\]

\[
wp f P x = P (f x)
\]

If we have a function \text{double} : \mathbb{N} \rightarrow \mathbb{N} and the predicate:

\[
gt17 : \mathbb{N} \rightarrow \text{Set}
\]

\[
gt17 x = x > 17
\]

What is the (weakest) precondition that needs to hold in order for the result of \text{double} to satisfy \text{gt17} – that is \text{double} produces a number greater than 17?
Example: predicates

We can illustrate the general principle using a (trivial) example:

\[ wp : (a \rightarrow b) \rightarrow (b \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set}) \]
\[ wp f P x = P (f x) \]

If we have a function \( \text{double} : \mathbb{N} \rightarrow \mathbb{N} \) and the predicate:

\[ \text{gt17} : \mathbb{N} \rightarrow \text{Set} \]
\[ \text{gt17} x = x > 17 \]

What is the (weakest) precondition that needs to hold in order for the result of \( \text{double} \) to satisfy \( \text{gt17} \) – that is \( \text{double} \) produces a number greater than 17?

\[ Q : \mathbb{N} \rightarrow \text{Set} \]
\[ Q x = \text{gt17} (\text{double} x) \]
Example: relations

But many specifications *relate* inputs and outputs – instead of just requiring a number greater than 17, we may want a sorted permutation of our input list.

This follows naturally if you use dependent types.
Example: relations

But many specifications relate inputs and outputs – instead of just requiring a number greater than 17, we may want a sorted permutation of our input list.

This follows naturally if you use dependent types.

\[
wp : ((x : a) \rightarrow b x) \rightarrow (\forall x \rightarrow b x \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set})
\]

Consider the following (slightly contrived) example:

- take for \( p : (xs : \text{List } a) \rightarrow \text{Permutation } xs \) as the first argument;
- and \( \text{isSorted} : (xs : \text{List } a) \rightarrow \text{Permutation } xs \rightarrow \text{Set} \) as the second.

Then \( wp \) computes the precondition necessary for \( p \) to be a sorting function.
So far this is not particularly exciting – it is no surprise that we can compute with predicates in Agda to reason about total functions.

But we can use the same techniques to reason about *effectful* functions.
So far this is not particularly exciting – it is no surprise that we can compute with predicates in Agda to reason about total functions.

But we can use the same techniques to reason about *effectful* functions.

This is problematic at first glance – many proof assistants based on type theory are careful to *avoid* effects, but restrict themselves to a total language to ensure the soundness of the underlying logic.
So far this is not particularly exciting – it is no surprise that we can compute with predicates in Agda to reason about total functions.

But we can use the same techniques to reason about effectful functions.

This is problematic at first glance – many proof assistants based on type theory are careful to avoid effects, but restrict themselves to a total language to ensure the soundness of the underlying logic.

For example, if we are not careful about handling unbounded recursion, we can define ‘bogus’ proofs such as:

\[ \text{silly} : \forall x \rightarrow x < x \]
\[ \text{silly} \ x = \text{silly} \ x \]
Effects
Inspired by work on algebraic effects, we are careful separate **syntax** and **semantics**.

- A free monad fixes the syntax;
- the semantics is defined by a predicate transformer.

Our ICFP paper describes the syntax and semantics for a variety of different effects in this style:

- exceptions
- mutable state
- non determinism
- general recursion
Free monads

```haskell
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

• A set C of commands;
• A function \( R : C \rightarrow \text{Set} \) of responses associated with every command.

Different choices of C and R give arise to different effects.
Free monads

\[
data \text{ Free } (C : \text{ Set}) (R : C \to \text{ Set}) (a : \text{ Set}) : \text{ Set} \quad \text{where}
\]

\[
\begin{align*}
\text{Pure} & : a \to \text{ Free } C R a \\
\text{Step} & : (c : C) \to (R c \to \text{ Free } C R a) \to \text{ Free } C R a
\end{align*}
\]

- A set \( C \) of \textit{commands};
- A function \( R : C \to \text{ Set} \) of responses associated with every command.

Different choices of \( C \) and \( R \) give arise to different effects.

- For example, to represent the familiar operations from the state monad, we can choose:
  - Commands \( \text{Get} : C \) and \( \text{Put} : s \to C \)
  - Responses \( s \) for \( \text{Get} \) and \( \top \) for \( \text{Put} \)
Instantiating C and R accordingly yields the following data type (for some type of states $s : \text{Set}$):

```
data FS (a : Set) : Set where
    Get : (s → FS a) → FS a
    Put : s → FS a → FS a
    Return : a → FS a
```

If we choose $s$ to be the natural numbers, we can write simple programs in this style:

```
inr : FS a

incr = get >>= λ x → put (x + 1) >> return x
```
Free monads: other examples

• Exceptions
  • Commands Abort : C
  • Responses ⊥
Free monads: other examples

• Exceptions
  • Commands Abort : C
  • Responses \bot

• Non-determinism
  • Commands Choice : C and Fail : C
  • Responses Bool for Choice and \bot for Fail
Free monads: other examples

• Exceptions
  • Commands Abort : C
  • Responses ⊥

• Non-determinism
  • Commands Choice : C and Fail : C
  • Responses Bool for Choice and ⊥ for Fail

• General recursion on a function \( I \rightarrow 0 \)
  • Commands call : I \rightarrow C
  • Responses 0
In general, we want to study the meaning of Kleisli arrows – that is, programs of the form:

\[ a \rightarrow \text{Free } C \rightarrow R \rightarrow b \]

These correspond to ‘effectful programs’, taking an input of type \( a \), performing effects from \( C \) and computing a value of type \( b \).
Semantics for effects

In general, we want to study the meaning of Kleisli arrows – that is, programs of the form:

\[ a \rightarrow \text{Free C R b} \]

These correspond to ‘effectful programs’, taking an input of type \( a \), performing effects from \( C \) and computing a value of type \( b \).

Given our \( \wp \) function, we compute the weakest precondition associated with a Kleisli arrow:

\[ \wp : (a \rightarrow \text{Free C R b}) \rightarrow (\text{Free C R b} \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set}) \]

But the postcondition here is expressed as a predicate on a free monad.

What happened to keeping syntax and semantics separate?
We’d like to define semantics with the following type:

\[ (a \to \text{Free C R b}) \to (b \to \text{Set}) \to (a \to \text{Set}) \]

But our \( \wp \) semantics has the following form:

\[ (a \to \text{Free C R b}) \to (\text{Free C R b} \to \text{Set}) \to (a \to \text{Set}) \]

To do so, requires a predicate transformer semantics for effects:

\[ (b \to \text{Set}) \to (\text{Free C R b} \to \text{Set}) \]

Defining predicate transformer semantics for effects boils down to defining such a function.
Semantics for effects – exceptions

wpPartial : (a → Partial b) → (b → Set) → (a → Set)

wpPartial f P = wp f (mustPT P)

where

mustPT : (b → Set) → (Partial b → Set)

mustPT P (Pure y) = P y
mustPT P (Step Abort) = ⊥

Here Partial refers to the free monad with a single command, Abort.

This semantics produces preconditions that guarantee Abort never happens.
Semantics for effects – exceptions

\[
\text{wpPartial} : (a \rightarrow \text{Partial } b) \rightarrow (b \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set})
\]

\[
\text{wpPartial } f \ P = \text{wp } f \ (\text{mustPT } P)
\]

\text{where}

\[
\text{mustPT} : (b \rightarrow \text{Set}) \rightarrow (\text{Partial } b \rightarrow \text{Set})
\]

\[
\text{mustPT } P \ (\text{Pure } y) = P \ y
\]

\[
\text{mustPT } P \ (\text{Step } \text{Abort}) = \bot
\]

Here \text{Partial} refers to the free monad with a single command, \text{Abort}.

This semantics produces preconditions that guarantee \text{Abort} never happens.

But other choices exist!

- Replace \bot with \top
- Require that \text{P} holds for some default value \text{d} : a
- ...
Semantics for effects – non-determinism

\[
\text{allPT} : (P : b \rightarrow \text{Set}) \rightarrow (\text{ND} b \rightarrow \text{Set})
\]

\[
\text{allPT} P \text{ (Pure } x) = P \ x
\]

\[
\text{allPT} P \text{ (Step Fail } k) = \top
\]

\[
\text{allPT} P \text{ (Step Choice } k) = \text{allPT} P \text{ (k True)} \land \text{allPT} P \text{ (k False)}
\]

Here we require \( P \) to hold for every possible result.
Semantics for effects – non-determinism

\[
\text{allPT} : (P : b \to \text{Set}) \to (\text{ND} b \to \text{Set}) \\
\text{allPT} P \text{ (Pure } x) = P x \\
\text{allPT} P \text{ (Step Fail } k) = \top \\
\text{allPT} P \text{ (Step Choice } k) = \text{allPT} P (k \text{ True}) \land \text{allPT} P (k \text{ False})
\]

Here we require \( P \) to hold for every possible result.

But again, alternatives exist.

The \textit{gambler’s nondeterminism} replaces \( \top \) with \( \bot \) and \( \land \) with \( \lor \)
Semantics for effects – state

\[ \text{statePT} : (P : (b \times s) \rightarrow \text{Set}) \rightarrow \text{FS} \ b \rightarrow (s \rightarrow \text{Set}) \]

\[ \text{statePT} \ P \ (\text{Return } x) \ s = P (x, s) \]

\[ \text{statePT} \ P \ (\text{Get } k) \ s = \text{statePT} \ P \ (k \ s) \ s \]

\[ \text{statePT} \ P \ (\text{Put } s' \ k) \ s = \text{statePT} \ P \ k \ s' \]

If necessary, we can also define a variant that takes an argument predicate:

\[ s \rightarrow (b \times s) \rightarrow \text{Set} \]

So that we can observe the relation between input and output states.
data Tree (a : Set) : Set where
  Leaf : a → Tree a
  Node : Tree a → Tree a → Tree a

Exercise
Relabel such a binary tree with unique numbers assigned to each leaf.
State – larger example

```haskell
(data Tree (a : Set) : Set where

  Leaf : a → Tree a

  Node : Tree a → Tree a → Tree a

Exercise
Relabel such a binary tree with unique numbers assigned to each leaf.

relabel : Tree a → FS (Tree \(\mathbb{N}\))
relabel (Leaf _) = incr >>= Leaf
relabel (Node l r) = relabel l >>= λ l' →
  relabel r >>= λ r' →
  return (Node l' r')

How do we show this is correct? Well to start with, we need a specification.
```
One way to specify the desired behaviour of our relabelling is:

\[ P : \text{Tree } a \times \text{Nat} \rightarrow \text{Tree } \text{Nat} \times \text{Nat} \rightarrow \text{Set} \]

\[ P (t, s) (t', s') = \text{flatten } t' \equiv \text{seq } s (\text{size } t) \]

Where \text{seq } s x is the sequence of natural numbers starting from \( s \) of length \( x \) – it’s easy to show that this does not contain duplicates.
One way to specify the desired behaviour of our relabelling is:

\[ P : \text{Tree } a \times \text{Nat} \rightarrow \text{Tree } \text{Nat} \times \text{Nat} \rightarrow \text{Set} \]

\[ P (t , s) (t' , s') = \text{flatten } t' \equiv \text{seq } s \text{ (size } t) \]

Where seq \( s \times x \) is the sequence of natural numbers starting from \( s \) of length \( x \) – it’s easy to show that this does not contain duplicates.

Unfortunately a direct proof showing that relabel satisfies this specification gets stuck quite quickly.
We do not yet know how to reason about composite programs written using binds.

But fortunately, we can prove a lemma along these lines:

\[
\text{compositionality} : (c : \text{FS} \ a) (f : a \rightarrow \text{FS} \ b) \rightarrow \\
\forall \ i \ P \rightarrow \text{statePT} \ P \ (c >>= f) \ i \equiv \text{statePT} \ (\text{wpState} \ f \ P) \ c \ i
\]
We do not yet know how to reason about composite programs written using binds.

But fortunately, we can prove a lemma along these lines:

\[
\text{compositionality : } (c : \text{FS } a) \, (f : a \rightarrow \text{FS } b) \rightarrow \\
\forall i \, P \rightarrow \text{statePT } P \, (c >>= f) \, i \equiv \text{statePT } (\text{wpState } f \, P) \, c \, i
\]

If you squint a bit, this is very similar to the usual relational composition used to reason about predicate transformers:

\[
\text{wp}(c_1 \, ; \, c_2, R) = \text{wp}(c_1, \text{wp}(c_2, R))
\]

Only here we have a monadic bind, passing an argument to \( f \), rather the (more implicit) dependency between imperative programs.

Using this result, we can check that our relabelling function is indeed correct.
This shows how to assign a weakest precondition semantics to Kleisli arrows:

\[(a \to \text{Free C R b}) \to (b \to \text{Set}) \to (a \to \text{Set})\]

But why bother with such semantics in the first place?

It seems like a rather indirect way to reason about programs!
This shows how to assign a weakest precondition semantics to Kleisli arrows:

\[(a \rightarrow \text{Free C R b}) \rightarrow (b \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set})\]

But why bother with such semantics in the first place?

It seems like a rather indirect way to reason about programs!

- We can also assign predicate transformer semantics to *specifications*;
- And use this semantics prove that a program satisfies its specification;
- Or even derive a program from its specification.
Specifications
Specifications

We define the following datatype of specifications on a function of type \((x : a) \rightarrow b \times\)

\[
\text{record \ Spec (a : Set) (b : a \rightarrow Set) : Set where}
\]

\[\begin{align*}
\text{field} \\
\quad \text{pre} : a \rightarrow Set \\
\quad \text{post} : (x : a) \rightarrow b \times \rightarrow Set
\end{align*}
\]

• A \textit{precondition} consisting of a predicate on \(a\)

• A \textit{postcondition} consisting of a relation between \((x : a)\) and \(b \times\).

I'll often write such specifications as \([\text{pre}, \text{post}]\).

But how can we assign semantics to such specifications?
Semantics for specifications

\[
\text{wpSpec} : \text{Spec} \ a \ b \rightarrow (P : (x : a) \rightarrow b \times \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set})
\]

\[
\text{wpSpec} \ [\ \text{pre} \ , \ \text{post} \ ] \ P = \lambda x \rightarrow (\text{pre} x) \land (\forall y \rightarrow \text{post} x \ y \rightarrow P \ x \ y)
\]

We can relate programs and specifications by relating the corresponding predicate transformers.

This idea – assigning predicate transformer semantics to \textit{specifications} – is one of the key insights of the \textit{refinement} calculus studied by Morgan, Back and von Wright.
Refinement

Given two predicate transformers, we can use the refinement relation to compare them:

\[ \sqsubseteq : (pt1 \; pt2 : (b \rightarrow \text{Set}) \rightarrow (a \rightarrow \text{Set})) \rightarrow \text{Set} \]

\[ \text{pt1} \sqsubseteq \text{pt2} = \forall P \; x : \text{pt1} P \; x \rightarrow \text{pt2} P \; x \]

This relation is reflexive, transitive and (morally) asymmetric.

Proving a program \( p \) satisfies it specification \( s \) amounts to showing:

\[ \text{wpSpec} \; s \sqsubseteq \text{wpEffect} \; p \]
Refinements between programs

Not only can relate a program with its specification, but we can also compare two different programs using the refinement relation.

- For pure functions, \( f \sqsubseteq g \) holds precisely when \( f \) and \( g \) are extensionally equal;
- For partial functions, \( f \sqsubseteq g \) precisely when \( f \) and \( g \) agree on the domain of \( f \);
- For non-deterministic functions, \( f \sqsubseteq g \) is equivalent to the subset relation.
- The gambler’s non-deterministic semantics flips \( f \) and \( g \).
- For state, \( f \sqsubseteq g \) corresponds to the usual weaker-pres and stronger-posts.
Refinements between programs

Not only can relate a program with its specification, but we can also compare two different programs using the refinement relation.

- For pure functions, \( f \sqsubseteq g \) holds precisely when \( f \) and \( g \) are extensionally equal;
- For partial functions, \( f \sqsubseteq g \) precisely when \( f \) and \( g \) agree on the domain of \( f \);
- For non-deterministic functions, \( f \sqsubseteq g \) is equivalent to the subset relation.
- The gambler’s non-deterministic semantics flips \( f \) and \( g \).
- For state, \( f \sqsubseteq g \) corresponds to the usual weaker-pres and stronger-posts.

This is rather a nice result – the refinement relation captures the expected relation between effectful programs in a general way.
Compositionality of refinement

Pure functional programmers are spoiled. We’re used to referential transparency, which allows us to employ equational reasoning.
Compositionality of refinement

Pure functional programmers are spoiled. We’re used to referential transparency, which allows us to employ equational reasoning.

For all sensible predicate transformers, we can show the following result:

compositionality : (f1 f2 : a → Free C R b) (g1 g2 : b → Free C R c) →
    wp f1 ▼ wp f2 →
    wp g1 ▼ wg g2 →
    wp (f1 >= g1) ▼ wp (f2 >= g2)

Here we can reap the rewards of indirection: the verification of effectful programs is compositional.
Pure functional programmers are spoiled. We’re used to referential transparency, which allows us to employ equational reasoning.

For all sensible predicate transformers, we can show the following result:

\[
\text{compositionality} : (f_1 : a \rightarrow \text{Free C R b}) (g_1 : b \rightarrow \text{Free C R c}) \rightarrow \\
\text{wp } f_1 \sqsubseteq \text{wp } f_2 \rightarrow \\
\text{wp } g_1 \sqsubseteq \text{wp } g_2 \rightarrow \\
\text{wp } (f_1 \Rightarrow g_1) \sqsubseteq \text{wp } (f_2 \Rightarrow g_2)
\]

Here we can reap the rewards of indirection: the verification of effectful programs is \textbf{compositional}. 
In this fashion we can show a program—given by a \texttt{Free C R a}—satisfies some specification.

But can we \textit{calculate} a program from its specification?
In this fashion we can show a program—given by a Free C R a—satisfies some specification.

But can we **calculate** a program from its specification?

Let’s consider values of the type Free C R (a + Spec a)

We can assign them semantics by composing the semantics for specifications and effects.
Typically, we prove little lemmas showing how each individual choice of command gives rise to new specifications, for which we must subsequently derive programs. This style of calculation relies heavily on the compositionality of our semantics. Even if you're not interested in program calculation, this gives you a 'small-step debugger' that you can use during verification.
Program calculation

Typically, we prove little lemmas showing how each individual choice of command gives rise to new specifications, for which we must subsequently derive programs. This style of calculation relies heavily on the compositionality of our semantics. Even if you're not interested in program calculation, this gives you a 'small-step debugger' that you can use during verification.
Typically, we prove little lemmas showing how each individual choice of command gives rise to new specifications, for which we must subsequently derive programs. This style of calculation relies heavily on the compositionality of our semantics. Even if you're not interested in program calculation, this gives you a 'small-step debugger' that you can use during verification.
Typically, we prove little lemmas showing how each individual choice of command gives rise to new specifications, for which we must subsequently derive programs. This style of calculation relies heavily on the compositionality of our semantics. Even if you're not interested in program calculation, this gives you a ‘small-step debugger’ that you can use during verification.
Typically, we prove little lemmas showing how each individual choice of command gives rise to new specifications, for which we must subsequently derive programs.
Typically, we prove little lemmas showing how each individual choice of command gives rise to new specifications, for which we must subsequently derive programs.

This style of calculation relies heavily on the **compositionality** of our semantics.

Even if you’re not interested in program calculation, this gives you a ‘small-step debugger’ that you can use during *verification*.
Soundness

All of our predicate transform semantics for effects have the following form:

\[ \text{pt} : (b \rightarrow \text{Set}) \rightarrow (\text{Free } C R b \rightarrow \text{Set}) \]

But this is quite strange in a way – why is the predicate you return is meaningful in any way?

The degenerate case:

\[ \text{pt} \ P \ c = \top \]

is type correct, but why is it still wrong?
Typically, we write these predicate transformers with an ‘intended’ semantics in mind.

\[
\text{runState} : \text{FS a } \rightarrow \text{s } \rightarrow \text{a } \times \text{s}
\]

We should show that the predicates we compute are \textit{sound} with respect to these ‘handlers’.

In words, every result returned by this handler satisfies the desired postcondition when the computed precondition holds.
Typically, we write these predicate transformers with an ‘intended’ semantics in mind.

\[
\text{runState} : \text{FS } a \rightarrow s \rightarrow a \times s
\]

We should show that the predicates we compute are \textit{sound} with respect to these ‘handlers’.

In words, every result returned by this handler satisfies the desired postcondition when the computed precondition holds.

If you’re familiar with Dijkstra monads:

- the computational monad corresponds to this run function;
- the specification monad corresponds to the predicate transformer semantics.
More than a single effect?

So far, we’ve only talked about the semantics of different effects *in isolation*. What about combining state and non-determinism? Or general recursion?

Anne Baanen and I have a recent paper at MSFP where we use this to write parsers for regular languages.
More than a single effect?

So far, we’ve only talked about the semantics of different effects *in isolation*.

What about combining state and non-determinism? Or general recursion?

- Free monads are closed under coproducts and compose nicely;
- Our predicate transformer semantics are defined as folds over free monads – these also compose nicely;
- We can put these together to study the predicate transformer semantics of compositions of effects.

Anne Baanen and I have a recent paper at MSFP where we use this to write parsers for regular languages.
• This gives a constructive & functional account of predicate transformer semantics.
• This approach works for a variety of different effects.
• We can relation effectful functions to their specifications in a compositional fashion.
• And even calculate programs from their spec.
Conclusion

• This gives a constructive & functional account of predicate transformer semantics.
• This approach works for a variety of different effects.
• We can relation effectful functions to their specifications in a compositional fashion.
• And even calculate programs from their spec.

Something (possibly of great conceptual elegance) is going on.