A predicate transformer semantics for effects

Wouter Swierstra and Anne Baanen

Utrecht University

Phil. Trans. R. Soc. Lond. A **312**, 501–518 (1984) [501]
Printed in Great Britain

Constructive mathematics and computer programming†

By P. Martin-Löf

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.

EWD 879

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.

Aim of our paper

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 → Set
```

A predicate transformer semantics for effects

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

```
a → Set
```

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

```
(a → Set) → (b → Set)
```

A predicate transformer semantics for effects

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

```
a → Set
```

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

```
(a → Set) → (b → Set)
```

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

```
wp : (a → b) → (b → Set) → (a → Set)
wp = ...
```

A predicate transformer semantics for effects

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

```
a → Set
```

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

```
(a → Set) → (b → Set)
```

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

```
wp : (a → b) → (b → Set) → (a → Set)
wp = ·
```

A predicate transformer semantics for effects

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

```
a → Set
```

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

```
(a → Set) → (b → Set)
```

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

```
wp : (a → b) → (b → Set) → (a → Set)
wp = ·
```

Or more generally, using *dependent types*

```
wp : ((x : a) → b x) → (∀ x → b x → Set) → (a → Set)
```

Example: predicates

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

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

If we have a function `double : ℕ → ℕ` and the predicate:

```
gt17 : ℕ → Set
gt17 x = x > 17
```

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

Example: predicates

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

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

If we have a function `double : ℕ → ℕ` and the predicate:

```
gt17 : ℕ → Set
gt17 x = x > 17
```

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

```
Q : ℕ → Set
Q x = gt17 (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) → b x) → (∀ x → b x → Set) → (a → Set)
```

Consider the following (slightly contrived) example:

- take for `p : (xs : List a) → Permutation xs` as the first argument;
- and `isSorted : (xs : List a) → Permutation xs → Set` as the second.

Than `wp` computes the precondition necessary for `p` to be a sorting function.

Computing with effects

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.

Computing with effects

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.

Computing with effects

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:

```
silly : ∀ x → x < x
silly x = silly x
```

Effects

Effectful programs

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

```
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 → Set` of responses associated with every command.

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

Free monads

```
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 → 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 `Get : C` and `Put : s → C`
  - Responses `s` for `Get` and ⊤ for `Put`

State explicitly

Instantiating `C` and `R` accordingly yields the following data type (for some type of states `s : 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:

```
incr : FS a
incr = get >>= λ x → put (x + 1) >> return x
```

Free monads: other examples

- Exceptions
  - Commands `Abort : C`
  - Responses $\bot$

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 $\bot$
- Non-determinism
  - Commands `Choice : C` and `Fail : C`
  - Responses `Bool` for `Choice` and $\bot$ for `Fail`
- General recursion on a function `I` $\rightarrow$ `O`
  - Commands `call : I` $\rightarrow$ `C`
  - Responses `O`

Semantics for effects

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

```
a → 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`.

Semantics for effects

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

```
a → 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 → Free C R b) → (Free C R b → Set) → (a → Set)
```

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

What happened to keeping syntax and semantics separate?

Semantics for effects

We'd like to define semantics with the following type:

```
(a → Free C R b) → (b → Set) → (a → Set)
```

But our `wp` semantics has the following form:

```
(a → Free C R b) → (Free C R b → Set) → (a → Set)
```

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

```
(b → Set) → (Free C R b → 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

```
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.

But other choices exist!

- Replace $\bot$ with $\top$
- Require that `P` holds for some default value `d : a`
- ...

Semantics for effects – non-determinism

```
allPT : (P : b → Set) → (ND b → 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)
```

Here we require `P` to hold for every possible result.

Semantics for effects – non-determinism

```
allPT : (P : b → Set) → (ND b → 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)
```

Here we require `P` to hold for every possible result.

But again, alternatives exist.

The *gambler's nondeterminism* replaces $\top$ with $\bot$ and $\wedge$ with $\vee$

Semantics for effects – state

```
statePT : (P : (b × s) → Set) → FS b → (s → Set)
statePT P (Return x) s = P (x , s)
statePT P (Get k)    s = statePT P (k s) s
statePT P (Put s' k) s = statePT P k s'
```

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

```
s → (b × s) → Set
```

So that we can observer 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

```
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 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.

Relabelling

One way to specify the desired behaviour of our relabelling is:

```
P : Tree a × Nat → Tree Nat × Nat → Set
P (t , s) (t' , s') = flatten t' ≡ seq s (size t)
```

Where `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.

Relabelling

One way to specify the desired behaviour of our relabelling is:

```
P : Tree a × Nat → Tree Nat × Nat → Set
P (t , s) (t' , s') = flatten t' ≡ seq s (size t)
```

Where `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.

Unfortunately a direct proof showing that `relabel` satisfies this specification gets stuck quite quickly.

Compositionality

We do not yet know how to reason about composite programs written using binds.

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

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

Compositionality

We do not yet know how to reason about composite programs written using binds.

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

```
compositionality : (c : FS a) (f : a → FS b) →
  ∀ i P → statePT P (c >>= f) i ≡ statePT (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:

```
wp(c1 ; c2, R) = wp(c1, wp(c2, 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.

Semantics for effects

This shows how to assign a weakest precondition semantics to Kleisli arrows:

```
(a → Free C R b) → (b → Set) → (a → Set)
```

But why bother with such semantics in the first place?

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

Semantics for effects

This shows how to assign a weakest precondition semantics to Kleisli arrows:

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

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

- A *precondition* consisting of a predicate on `a`
- A *postcondition* consisting of a relation between `(x : a)` and `b x`.

I'll often write such specifications as `[ pre , post ]`.

But how can we assign semantics to such specifications?

Semantics for specifications

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

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

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

Refinement

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

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

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

Proving a program `p` satisfies it specification `s` amounts to showing:

```
wpSpec s ⊑ 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)
```

Compositionality of refinement

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

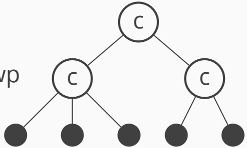
For all sensible predicate transformers, we can show the following result:

```
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**.

Program verification

wpSpec [pre,post] $\sqsubseteq$ wp

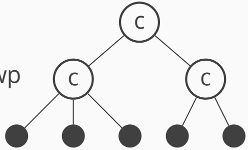


In this fashion we can show a program—given by a `Free C R a`—satisfies some specification.

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

Program verification

wpSpec [pre,post] ⊑ wp



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)`

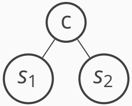
We can assign them semantics by composing the semantics for specifications and effects.

Program calculation

[pre,post]

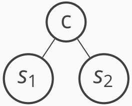
Program calculation

[pre,post] $\sqsubseteq$

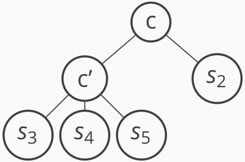


Program calculation

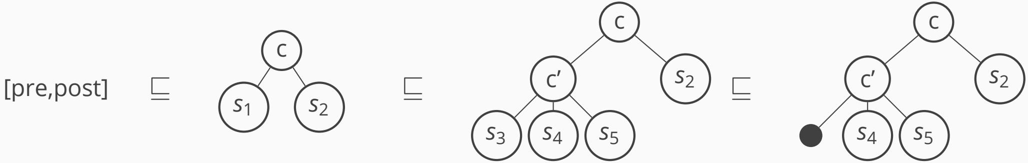
[pre,post] $\sqsubseteq$



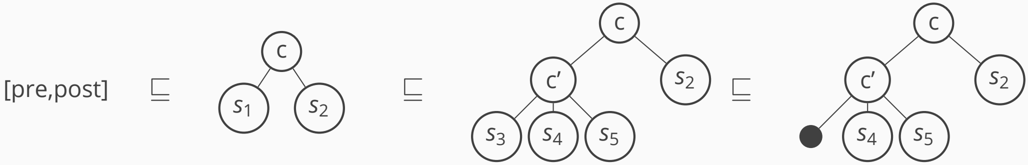
$\sqsubseteq$



Program calculation

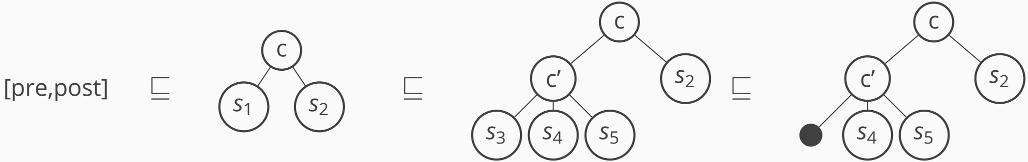


Program calculation



Typically, we prove little lemmas showing how each individual choice of command gives rise to new specifications, for which we must subsequently derive programs.

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*.

Soundness

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

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

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

The degenerate case:

```
pt P c = ⊤
```

is type correct, but why is it still wrong?

Soundness

Typically, we write these predicate transformers with an ‘intended’ semantics in mind.

```
runState : FS a → s → a × s
```

We should show that the predicates we compute are *sound* with respect to these ‘handlers’.

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

Soundness

Typically, we write these predicate transformers with an 'intended' semantics in mind.

```
runState : FS a → s → a × s
```

We should show that the predicates we compute are *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?

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 alse compose nicely;
- We can put these together to study the predicate transformer semeanticsc of compositions of effects.

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

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.

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.