A predicate transformer semantics for effects

Functional pearl

Wouter Swierstra and Tim 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

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)
```

Effects

Effectful programs

We're not only interested *pure* functions.

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 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: examples

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

Free monads: examples

- Exceptions
  - Commands `Abort : C`
  - Responses $\perp$
- State
  - Commands `Get : C` and `Put : s` $\to$ `C`
  - Responses `s` for `Get` and $\top$ for `Put`

Free monads: examples

- Exceptions
  - Commands `Abort : C`
  - Responses $\perp$
- State
  - Commands `Get : C` and `Put : s` $\to$ `C`
  - Responses `s` for `Get` and $\top$ for `Put`
- Non-determinism
  - Commands `Choice : C` and `Fail : C`
  - Responses `Bool` for `Choice` and $\perp$ for `Fail`

Free monads: examples

- Exceptions
  - Commands `Abort : C`
  - Responses $\bot$
- State
  - Commands `Get : C` and `Put : s` $\rightarrow$ `C`
  - Responses `s` for `Get` and $\top$ for `Put`
- 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

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

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?

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

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

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

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

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 $\perp$ 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 – 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 $\vee$

The paper defines similar predicate transformers for state, general recursion, etc.

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?

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?

- 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.

Refinement

Refinement calculus

Another approach is to use probably correct refinement steps to transform a specification into a design, which is ultimately transformed into an implementation that is *correct by construction*.

(Source: wikipedia page on Formal specification)

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 partial functions, $\mathtt{f} \sqsubseteq \mathtt{g}$ precisely when `f` and `g` agree on the domain of `f`;
- For non-deterministic functions, $\mathtt{f} \sqsubseteq \mathtt{g}$ is equivalent to the subset relation.
- The gambler's non-deterministic semantics flips `f` and `g`.
- For state, $\mathtt{f} \sqsubseteq \mathtt{g}$ corresponds to the usual weaker-pre's and stronger-posts.

Compositionality of refinement

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

Compositionality of refinement

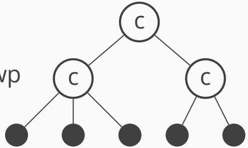
Pure functional programmers are spoiled. We're used to referential transparency, which allows us to employ equational reasoning.

For all sensible predicate transformers, refinement is inherently *compositional*.

```
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)
```

Program verification

$\text{wpSpec [pre,post]} \sqsubseteq \text{wp}$

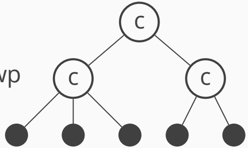


In this fashion we can show a program—given by a `Free C R a`—satisfies some specification.

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

Program verification

wpSpec [pre,post] $\sqsubseteq$ 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?

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

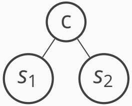
We can assign them semantics by composing the semantics for specifications and effects.

Program calculation

[pre,post]

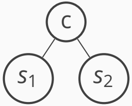
Program calculation

[pre,post] $\sqsubseteq$

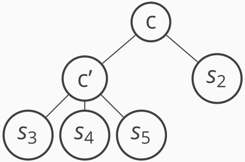


Program calculation

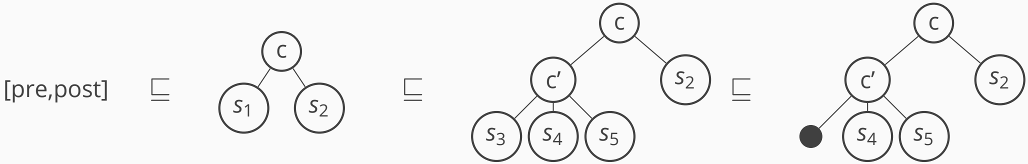
$[pre, post] \sqsubseteq$



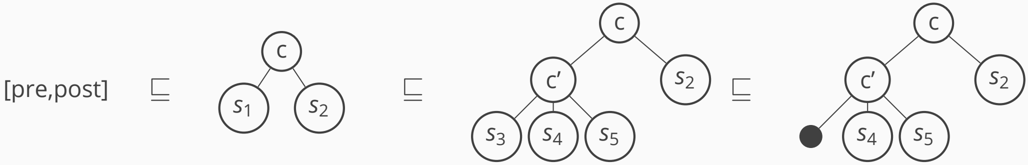
$\sqsubseteq$



Program calculation

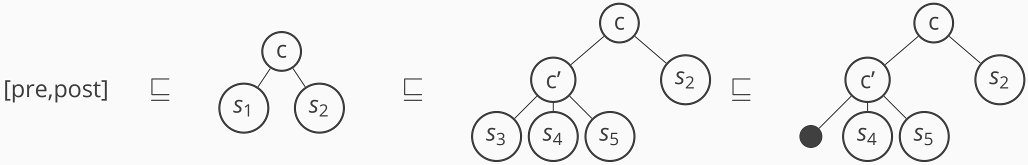


Program calculation



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*.

Conclusion

Conclusion

If we know the semantics of an effect,

Conclusion

If we know the semantics of an effect,

And have a pre- and post spec,

Conclusion

If we know the semantics of an effect,

And have a pre- and post spec,

Taking their predicate transformers combined,

Conclusion

If we know the semantics of an effect,

And have a pre- and post spec,

Taking their predicate transformers combined,

The spec can be refined,

Conclusion

If we know the semantics of an effect,

And have a pre- and post spec,

Taking their predicate transformers combined,

The spec can be refined,

To ensure that our programs ain't rekt.

A predicate transformer semantics for effects

Functional pearl

Wouter Swierstra and Tim Baanen

Utrecht University