A predicate transformer semantics for effects

A Research Agenda for Formal Methods in the Netherlands

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 this talk

- Sketch how to give a *constructive* account of Dijkstra's weakest precondition semantics in Martin-Löf type theory
- Motivate why we would ask this question in the first place.

Programming language and proof assistant

Martin-Löf type theory — the basis for modern proof assistants such as Coq, Agda, Idris, Lean and others – is a language for **proofs** and **programs**

```
plus-commutes : ∀ n m → n + m ≡ m + n
```

To prove a lemma, amounts to showing that the corresponding **type** is inhabited.

Programming, but not as we know it

```
plus-commutes : ∀ n m → n + m ≡ m + n
plus-commutes n m = plus-commutes n m
```

While this definition is type correct, it is ruled out (and rightly so).

The ‘programming language’ in most proof assistants requires all functions to be **total**.

Programming, but not as we know it

```
plus-commutes : ∀ n m → n + m ≡ m + n
plus-commutes n m = plus-commutes n m
```

While this definition is type correct, it is ruled out (and rightly so).

The ‘programming language’ in most proof assistants requires all functions to be **total**.

- No exceptions;
- No mutable state;
- No concurrency;
- No missing cases;
- No general recursion;
- ...

Termination woes

In a richly typed language - there's much more structure to exploit!

'*If your recursion isn't structural, you're using the wrong structure*' – Conor McBride

Termination woes

In a richly typed language - there's much more structure to exploit!

'*If your recursion isn't structural, you're using the wrong structure*' – Conor McBride

```
f91 : Int ⟶ Int
f91 n = if 100 < n then n - 10 else f91 (f91 (n + 11))
```

But there are plenty of programs that we'd like to study, even if their recursive definition is very strange indeed!

Algebraic effects

Algebraic effects have recently gained traction in the wider programming languages community.

- Separate **syntax** and **semantics** of effects;
- The syntax merely describes the different primitive operations;
- The semantics assigns meaning to these operations.

Algebraic effects

Algebraic effects have recently gained traction in the wider programming languages community.

- Separate **syntax** and **semantics** of effects;
- The syntax merely describes the different primitive operations;
- The semantics assigns meaning to these operations.

In this talk, I'll sketch how to assign a **predicate transformer semantics** to such effects.

And show how to model **recursion** as an algebraic effect.

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)
```

Recursion as an effect

```
data Rec (I : Set) (O : Set) (a : Set) : Set where
  Pure : a → Rec I O a
  Call : I → (O → Rec I O a) → Rec I O a
```

The `Rec I O a` data type explicitly models computations that may make calls to a (recursive) 'oracle' of type `I → O`, before returning a value of type `a`.

A function of type `I → Rec I O O` corresponds to a function where the 'recursive' calls are made explict.

It's easy to show that this type is a *monad*, for any choice of `I` and `O`.

Example

Written using the `do` notation, we can define our `f91` function as follows.

```
f91 i = do
  x ← call (i + 11)
  call x
```

This gives a *finite representation* of a recursive program.

This definition itself is **not** recursive – but we can:

- produce a coinductive trace by repeatedly unfolding the definition on a given input;
- run the computation for a fixed number of steps;
- prove it terminates using well-founded recursion;
- ...

Beyond recursion...

Many other effects can be described in this fashion:

- Exceptions
- State
- Non-determinism

Each of these effects give rise to a *free monad* describing their syntax.

We can assign each of these effects a *predicate transformer semantics*

```
(b → Set) → (Free b → Set)
```

Beyond recursion...

Many other effects can be described in this fashion:

- Exceptions
- State
- Non-determinism

Each of these effects give rise to a *free monad* describing their syntax.

We can assign each of these effects a *predicate transformer semantics*

```
(b → Set) → (Free b → Set)
```

Combined with the `wp` function we saw previously, we can compute the weakest precondition necessary for a particular program to produce a result satisfying the desired postcondition.

But what semantics should we assign to recursion?

Specifications

We define the following datatype of *specifications* on a function of type `a → b`

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

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

We can even assign predicate transformer semantics to such specifications.

Semantics for specifications

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

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

Sketching the semantics for recursion

Using this, we can even give semantics to our recursive functions:

Given the desired spec of the recursive function – we need to show that the ‘call graph’ representation respects the corresponding ‘loop invariant’.

Unsurprisingly, to assign meaning to recursive functions, we need some hint from the programmer.

We can prove that, for example, when the weakest precondition holds and we run a function for `n` steps and it does terminate, the desired post also holds.

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.

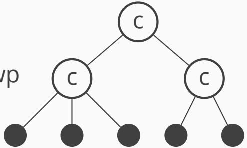
Compositionality of refinement

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 b) (g1 g2 : b → Free c) →
  wp f1 ⊑ wp f2 →
  wp g1 ⊑ wg g2 →
  wp (f1 >=> g1) ⊑ wp (f2 >=> g2)
```

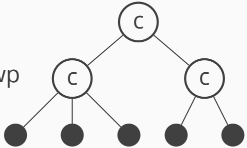
Program verification

wpSpec [pre,post] $\sqsubseteq$ wp 

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

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

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

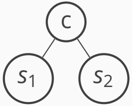
We can assign them semantics by composing the semantics for specifications and effects.

Program calculation

[pre,post]

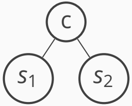
Program calculation

[pre,post] $\sqsubseteq$

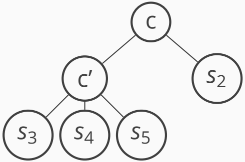


Program calculation

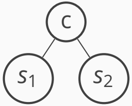
[pre,post] $\sqsubseteq$

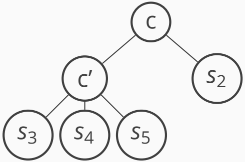


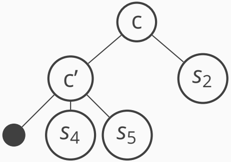
$\sqsubseteq$



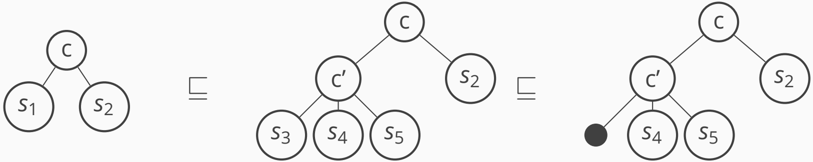
Program calculation

[pre,post] $\sqsubseteq$ 

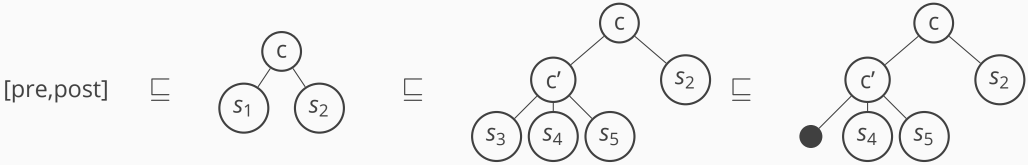
$\sqsubseteq$ 

$\sqsubseteq$ 

$[pre,post] \sqsubseteq$



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*.

Conclusions

- Programming in proof assistants is not always easy;

Conclusions

- Programming in proof assistants is not always easy;
- But we have the formal methods to make it feasible.

Utrecht University

A predicate transformer semantics for effects

A Research Agenda for Formal Methods in the Netherlands

Wouter Swierstra and Tim Baanen

Utrecht University

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

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)
```