Algebraic effects – specification and refinement

Dagstuhl 18172

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Algebraic effects go mainstream

Algebraic effects go mainstream

This talk: Back into the ivory tower!

How to reason about programs written using algebraic effects?

Program verification

1. A program p
2. A specification S
3. A proof that p satisfies S

Specifications of `f : a → b`?

- A property of a function:

```
P : (a → b) → Set
```

- A relation between input and output:

```
data R : a → b → Set where
  ...
```

- A predicate transformer:

```
(b → Set) → (a → Set)
```

- a reference implementation:

```
g : a → b
```

- and many others...

What is the specification of a program written algebraic effects?

What is the specification of a program written algebraic effects?

That depends on the handler!

What is the specification of a handler?

What is the specification of a handler?

Jeremy: The equations it must satisfy!

Your mission, should you choose to accept it...

Consider the usual `Put` and `Get` operations used in mutable state...

Your mission, should you choose to accept it...

Consider the usual `Put` and `Get` operations used in mutable state...

But the memory is self-destructing. Reading from memory more than once, crashes your program.

Equations?

```
i <- Get; j <- Get ≡ Abort
```

Equations?

```
i <- Get; j <- Get ≡ Abort

i <- Get; Put x; j <- Get k ≡ Abort
```

Equations?

`i <- Get; j <- Get` $\equiv$ `Abort`

`i <- Get; Put x; j <- Get k` $\equiv$ `Abort`

I'm sure that – with some thought – we can find a suitable set of equations.

(Note: the usual `Put; Get; k` $\equiv$ `k` and `Get; Get` $\equiv$ `Get` do not hold!)

A small modification to the spec

A small modification to the spec

But reading from memory more than **63** times, crashes your program.

A small modification to the spec

But reading from memory more than **63** times, crashes your program.

Exercise: Please update the equations accordingly.

Proofs using equations

- Familiar and simple concept from universal algebra
- Equational proofs are familiar to functional programmers

Proofs using equations

- Familiar and simple concept from universal algebra
- Equational proofs are familiar to functional programmers
- … equations are typically not first-class.
- … syntactic approach of relating programs may be unsuitable for describing some program properties.

How to reason about programs using algebraic effects?

How to reason about programs using algebraic effects?

- Prehistoric approach to algebraic effects and handlers using free monads;
- A few examples in Agda to illustrate the approach.

How to reason about programs using algebraic effects?

- Prehistoric approach to algebraic effects and handlers using free monads;
- A few examples in Agda to illustrate the approach.
- The unindexed intro to Conor's talk.

What is an algebraic effect?

You can specify the operations associated with an algebraic effect by giving:

- `C : Set` – the type of operations
- `R : C → Set` – the responses passed to the continuation

What are computations?

From these ingredients, we can define the usual free monad:

```
data Free (C : Set) (R : C → Set) (A : Set) : Set where
  pure : A → Free C R A
  op : (c : C) → (R c → Free C R A) → Free C R A
```

A *handler* then corresponds to an algebra to fold over the free monad.

Example: state

```
data C : Set where
  get : C
  put : S → C

R : C → Set
R get = S
R put = Unit

State = Free C R

run : State A → S → A × S
run (pure x)         s = (x , s)
run (op get k)       s = run (k s) s
run (op (put s) k) _ = run (k tt) s
```

Reasoning about state

How can we reason about programs of type `State A`?

- We can `run` the handler to achieve a function of type $\mathtt{A} \longrightarrow \mathtt{A} \times \mathtt{S}$ and reason about that…

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Reasoning about state

How can we reason about programs of type `State A`?

- We can `run` the handler to achieve a function of type $\mathtt{A} \longrightarrow \mathtt{A} \times \mathtt{S}$ and reason about that...
- But this fixes a *specific handler* – rather than reasoning about possible handlers.

Weakest precondition

```
wp : (P : S → A → Set) → State A → (S → Set)
wp (pure x)         s = P s x
wp (op get k)       s = wp (k s) s
wp (op (put s) k) _ = wp (k tt) s
```

Claim: Here the `wp` handler computes the weakest precondition on `S` in order for the computation to return a value and state satisfying `P`.

(You can achieve the usual *relational* presentation from Hoare type theory from this by reordering the arguments slightly)

Faculty of Science
Information and Computing Sciences

Soundness

```
wp : (P : S → A → Set) → State A → S → Set
```

Given a predicate, stateful computation and initial state, `wp` computes a proposition. Who says this proposition is sensible in any way?

Soundness

```
wp : (P : S → A → Set) → State A → S → Set
```

Given a predicate, stateful computation and initial state, `wp` computes a proposition. Who says this proposition is sensible in any way?

We should show that our handlers are sound with respect to this proposition:

```
soundness : (s : S) → wp P c s → P (run c s)
```

What about other effects?

Abort

```
data C : Set where
  abort : C

R : C → Set
R abort = ⊥

pt : (P : A → Set) → Free C R A → Set
pt P (pure x) = P x
pt P (op _ _) = ⊥
```

Idea: the computation of type `Free R C A` returns a value satisfying `P`.

Weakest preconditions

```
wp : (P : B → Set) → (A → Free C R B) → (A → Set)
wp P f = pt P . f
```

This computes the weakest precondition necessary for our computation to satisfy `P`.

Weakest preconditions

```
wp : (P : B → Set) → (A → Free C R B) → (A → Set)
wp P f = pt P . f
```

This computes the weakest precondition necessary for our computation to satisfy `P`.

Other choices exist, for example, mapping to `Maybe` or asserting `P d` for some default value `d`.

Non-determinism

```
data C : Set where
  or : C
  fail : C

R : C → Set
R or    = Bool
R fail  = ⊥

pt : (P : A → Set) → Free C R A → Set
pt = ?
```

Non-determinism

```
data C : Set where
  or : C
  fail : C

R : C → Set
R or    = Bool
R fail  = ⊥

pt : (P : A → Set) → Free C R A → Set
pt = ?
```

There are different ways to transform a predicate over `A` to one over the free monad `Free C R A`…

Non-determinism: all or any?

```
all : (P : A → Set) → Free C R A → Set
all P (pure x)    = P x
all P (op or k)   = k true × k false
all P (op fail k) = unit

any : (P : A → Set) → Free C R A → Set
any P (pure x)    = P x
any P (op or k)   = k true + k false
any P (op fail k) = ⊥
```

Weakest preconditions

- Given a Kleisli arrow `c : A → Free C R B`
- a predicate transformer `(P : B → Set) → Free C R B → Set`
- we can compute the weakest precondition `A → Set` by composing the pieces.

This works independently of the particular choice of operations or handlers!

Refinement

Based on the $\mathtt{wp}$ semantics, we can define a notion of *program refinement*, $\mathtt{p}_1 \sqsubseteq \mathtt{p}_2$.

This refinement holds precisely when

$$(\mathtt{P} : \mathtt{B} \to \mathtt{Set}) \to \mathtt{wp}\ \mathtt{p}_1\ \mathtt{P} \to \mathtt{wp}\ \mathtt{p}_2\ \mathtt{P}$$

Intuitively, when $\mathtt{p}_2$ refines $\mathtt{p}_1$, we may think of $\mathtt{p}_2$ 'more specific' than $\mathtt{p}_1$.

Examples

Given two functions `f` and `g` of type $\mathtt{A} \longrightarrow \mathtt{Free\ C\ R\ B}$, what does refinement mean?

- For Abort, the domain of `f` must be included in the domain of `g` and both functions coincide on the domain of `f`.
- For stateful computations, you get the 'standard' notion of program refinement (postcondition of `f` implies that of `g`; preconditions work the other way around).
- For nondeterminism, under the `any` or `all` predicate transformers this gives rise to subset inclusions.

Examples

Given two functions `f` and `g` of type `A → Free C R B`, what does refinement mean?

- For Abort, the domain of `f` must be included in the domain of `g` and both functions coincide on the domain of `f`.
- For stateful computations, you get the 'standard' notion of program refinement (postcondition of `f` implies that of `g`; preconditions work the other way around).
- For nondeterminism, under the `any` or `all` predicate transformers this gives rise to subset inclusions.
- And if you have no effects, the functions be equal for all inputs.

Towards program calculation

We can extend our free monad with pieces of unfinished programs:

```
data Free (C : Set) (R : C → Set) (A : Set) : Set where
  pure : A → Free C R A
  op : (c : C) → (R c → Free C R A) → Free C R A
  spec : (P : B → Set) → (B → Free C R A) → Free C R A
```

Our `wp` semantics extend to these structures.

Starting from a `spec P pure`, we can derive a complete program by a series of refinement steps, replacing specifications with operations until we have computed the desired result satisfying the spec. See the SCP paper with Joao Alpuim for details of the construction for mutable state.

Limitations & further work

- Free monads, rather than full algebraic effects;
- Can Morgan et al.'s work on refinement of probablistic programs be formulated in this style?
- Invariants and recursion?
- Some ideas about interaction between different effects...

Questions?

Self-destructing memory

```
sd : (P : s → a → Set) → Nat → State a → s → Set
sd P n        (Pure x)         s = P s x
sd P n        (Step (Put s) x) _ = sd P n (x tt) s
sd P Zero     (Step Get x)     s = ⊥
sd P (Succ n) (Step Get x)     s = sd P n (x s) s

soundness : (n : Nat) → (P : s → a → Set) →
  (c : State a) → (i : s) →
  sd P n c i →
  P (snd (handle c i)) (fst (handle c i))
```

This is just...

- presheaves
- (indexed) containers
- predicate transformers semantics
- adjunctions
- Kan extensions
- Hoare type theory
- monad transformers
- ...