Algebraic effects – specification and refinement

Dagstuhl 18172

Wouter Swierstra
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 \( f : a \rightarrow b \)?

- A property of a function:
  \( P : (a \rightarrow b) \rightarrow \text{Set} \)

- A relation between input and output:
  \[
  \text{data } R : a \rightarrow b \rightarrow \text{Set} \text{ where} \\
  \ldots 
  \]

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

- A reference implementation:
  \( g : a \rightarrow b \)

- 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?

\[ i \leftarrow \text{Get}; \, j \leftarrow \text{Get} \equiv \text{Abort} \]
Equations?

\[ i \leftarrow \text{Get}; \ j \leftarrow \text{Get} \equiv \text{Abort} \]

\[ i \leftarrow \text{Get}; \ \text{Put} \ x; \ j \leftarrow \text{Get} \ k \equiv \text{Abort} \]
Equations?

\[ i \leftarrow \text{Get}; \quad j \leftarrow \text{Get} \equiv \text{Abort} \]

\[ i \leftarrow \text{Get}; \quad \text{Put} \ x; \quad j \leftarrow \text{Get} \ k \equiv \text{Abort} \]

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

(Note: the usual \text{Put}; \ \text{Get}; \ k \equiv k \text{ and Get; Get} \equiv \text{Get} \text{ do not hold!})
A small modification to the spec

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

Exercise:
Please update the equations accordingly.
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:

- \( C : \text{Set} \) – the type of operations
- \( R : C \rightarrow \text{Set} \) – the responses passed to the continuation
What are computations?

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

```haskell
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 $\text{State } A$?

- We can run the handler to achieve a function of type $A \rightarrow A \times S$ and reason about that...
Reasoning about state

How can we reason about programs of type $\text{State } A$?

- We can run the handler to achieve a function of type $A \rightarrow A \times S$ and reason about that...

- But this fixes a specific handler – rather than reasoning about possible handlers.
Weakest precondition

\[ \text{wp} : (\mathcal{P} : S \rightarrow A \rightarrow \text{Set}) \rightarrow \text{State} \quad A \rightarrow (S \rightarrow \text{Set}) \]
\[ \text{wp} (\text{pure } x) \quad s = \mathcal{P} \ s \ x \]
\[ \text{wp} (\text{op get } k) \quad s = \text{wp} (k \ s) \ s \]
\[ \text{wp} (\text{op (put } s \text{) } k) \_ = \text{wp} (k \ \text{tt}) \ s \]

\textbf{Claim:} Here the \text{wp} handler computes the weakest precondition on \( S \) in order for the computation to return a value and state satisfying \( \mathcal{P} \).

(You can achieve the usual \textit{relational} presentation from Hoare type theory from this by reordering the arguments slightly)
Soundness

\[ \text{wp} : (P : S \rightarrow A \rightarrow \text{Set}) \rightarrow \text{State A} \rightarrow S \rightarrow \text{Set} \]

Given a predicate, stateful computation and initial state, \( \text{wp} \) computes a proposition. Who says this proposition is sensible in any way?
Soundness

\[ \text{wp} : (P : S \rightarrow A \rightarrow \text{Set}) \rightarrow \text{State} \ A \rightarrow S \rightarrow \text{Set} \]

Given a predicate, stateful computation and initial state, \( \text{wp} \) 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:

\[ \text{soundness} : (s : S) \rightarrow \text{wp} \ P \ c \ s \rightarrow P (\text{run} \ c \ s) \]
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

\[ wp : (P : B \rightarrow \text{Set}) \rightarrow (A \rightarrow \text{Free C R B}) \rightarrow (A \rightarrow \text{Set}) \]

\[ wp \ P \ f = \text{pt} \ P \ . \ f \]

This computes the weakest precondition necessary for our computation to satisfy \( P \).
Weakest preconditions

\[ wp : (P : B \to \text{Set}) \to (A \to \text{Free C R B}) \to (A \to \text{Set}) \]
\[ wp P f = \text{pt} P \cdot f \]

This computes the weakest precondition necessary for our computation to satisfy \( P \).

Other choices exist, for example, mapping to \texttt{Maybe} or asserting \( P \ d \) for some default value \( d \).
Non-determinism

data C : Set where
  or : C
  fail : C

R : C \rightarrow Set
R \text{ or} = \text{Bool}
R \text{ fail} = \bot

pt : (P : A \rightarrow Set) \rightarrow \text{Free C R A} \rightarrow 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 $c : A \rightarrow \text{Free } C \ R \ B$
- a predicate transformer $(P : B \rightarrow \text{Set}) \rightarrow \text{Free } C \ R \ B \rightarrow \text{Set}$
- we can compute the weakest precondition $A \rightarrow \text{Set}$ by composing the pieces.

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

Based on the \( wp \) semantics, we can define a notion of \textit{program refinement}, \( p_1 \sqsubseteq p_2 \).

This refinement holds precisely when

\[(P : B \rightarrow \text{Set}) \rightarrow \text{wp } p_1 \ P \rightarrow \text{wp } p_2 \ P\]

Intuitively, when \( p_2 \) refines \( p_1 \), we may think of \( p_2 \) ‘more specific’ than \( p_1 \).
Examples

Given two functions \( f \) and \( g \) of type \( \text{Free} \ C \ R \ B \), 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 \text{any} or \text{all} predicate transformers this gives rise to subset inclusions.
Given two functions \( f \) and \( g \) of type \( A \to \text{Free C R B} \), 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:

```haskell
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 \(\text{wp}\) semantics extend to these structures.

Starting from a \(\text{spec } P \ 	ext{pure}\), 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 probabilistic programs be formulated in this style?
- Invariants and recursion?
- Some ideas about interaction between different effects...
Questions?
Self-destructing memory

\[ sd : (P : s \rightarrow a \rightarrow \text{Set}) \rightarrow \text{Nat} \rightarrow \text{State} \; a \rightarrow s \rightarrow \text{Set} \]

\[ sd \; P \; n \; (\text{Pure} \; x) \; s = P \; s \; x \]

\[ sd \; P \; n \; (\text{Step} \; (\text{Put} \; s) \; x) \_ = sd \; P \; n \; (x \; \text{tt}) \; s \]

\[ sd \; P \; \text{Zero} \; (\text{Step} \; \text{Get} \; x) \; s = \bot \]

\[ sd \; P \; (\text{Succ} \; n) \; (\text{Step} \; \text{Get} \; x) \; s = sd \; P \; n \; (x \; s) \; s \]

\[ \text{soundness} : (n : \text{Nat}) \rightarrow (P : s \rightarrow a \rightarrow \text{Set}) \rightarrow (c : \text{State} \; a) \rightarrow (i : s) \rightarrow \\
\quad sd \; P \; n \; c \; i \rightarrow \\
\quad P \; (\text{snd} \; (\text{handle} \; c \; i)) \; (\text{fst} \; (\text{handle} \; c \; i)) \]
This is just...

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

Faculty of Science
Information and Computing Sciences