Algebraic Effects for Calculating Compilers

Luke Geeson under Supervision of Jeremy Gibbons

Department of Computer Science

luke.geeson, jeremy.gibbons, (@cs.ox.ac.uk)

UNIVERSITY OF OXFORD

Abstract

We combine *algebraic effects* and *calculating compilers*. We implement algebraic effect handlers for languages with computational effects and calculate compiler definitions via constructive induction. We adopt a *roll-your-own* approach inspired by Swierstra's smart constructors [4].

Background

Bahr and Hutton [2] proposed a method to calculate compiler and virtual machine definitions that are *correct by construction*. Given a source language, evaluation semantics and correctness specification. For instance Hutton's razor as a source language:

```
data Expr
  = Val Int
  | Add Expr Expr
eval :: Expr → Int
eval (Val n) = n
eval (Add e1 e2) = eval e1 + eval e2
data ExprValue = Num Int
```

With correctness specification:

$$exec\ (comp'\ s\ t)\ c = exec\ t\ (eval'\ s\ c) \qquad (1)$$

we calculate compiler and virtual machine definitions:

```
exec  :: Code → Stack → Stack
comp' :: Expr → Code → Code
```

with a more general $eval'$ function for *stack-based* configurations:

```
type Stack = [Int]
eval' :: Expr → Stack → Stack
```

We model computational effects in the *source* language and configuration using algebraic effects. The stack forms a weaker form of the $state$ effect; we capture the stack with the $StackFunctor$ effect functor and $Push/Pop$ abstract operations:

```
data StackFunctor s a
  = Pop (s → a)
  | Push s a deriving Functor
```

We separate the concerns of syntax and semantics so that the stateful $Stackfunctor$ over $Expr$ forms the *free monad* abstract syntax tree, consisting of abstract operations.

```
data Free f a = Var a | Cons (f (Free f a))
```

We capture the semantics through algebraic handlers, which fold algebras (semantics) over the tree to interpret it in the semantic domain (Int) [5]. We use Swierstra's datatypes *à la carte* [4] to construct abstract syntax trees from the effect functors:

```
class (Functor f, Functor g) ⇒ f ⊂ g where
  inj :: f a → g a
  prj :: g a → Maybe (f a)
```

with injections/projections into the tree:

```
inject  :: (g ⊂ f) ⇒ g (Free f a) → Free f a
inject          = Cons ∘ inj
project :: (f ⊂ g) ⇒ Free g a → Maybe (f (Free g a))
project (Cons s) = prj s
project _        = Nothing
```

and combine handlers with co-product functors:

```
data (+) f g a where
  Inl :: f a → (f + g) a
  Inr :: g a → (f + g) a deriving Functor
```

Contributions

- Generalise Bahr and Hutton's calculation method [2] to machines with *configurations*, calculating correct compilers for Hutton's razor.
- Implement First and Higher-Order effect handlers using Swierstra's datatypes *à la carte* [4] and Wu *et al*'s *Higher-Order Syntax* [6] for languages with interacting effects and scoping constructs.
- Calculate compilers and virtual machines for languages with and without exceptions on *stack-based* machines.
- Implement typeclasses to capture correctness specifications for compilers with handlers, scoping constructs and interacting effects.
- Calculate a compiler for Levy's Call-By-Push-Value λ-Calculus [3] with exceptions as a non-trivial case study.

Calculating Compilers with Handlers

We model $eval$ using the free monad machinery:

```
eval_free :: Expr → Free (StackFunctor ExprValue) ()
eval_free (Val n)      = Cons (Push (Num n) (Var ()))
eval_free (Add e1 e2) = do
  eval_free e1
  eval_free e2
  (Num n) ← Cons (Pop Var)
  (Num m) ← Cons (Pop Var)
  Cons (Push (Num (n + m)) (Var ()))
```

Monadic evaluation semantics expressed as abstract syntax trees and compiler IR ASTs are thus the same. We capture the tree notation of $eval_{free}$ with a syntactic trick:

```
pop :: (StackFunctor ExprValue ⊂ g) ⇒
  Free g ExprValue
pop = inject (Pop Var)
push :: (StackFunctor ExprValue ⊂ g) ⇒
  ExprValue → Free g ()
push v = inject (Push v (Var ()))
```

This grafts sub-trees into the tree corresponding to the abstract operations in question. We redefine $eval$ as an abstract computation $eval'$ that uses these operations, thus separating the concerns of syntax:

```
eval' :: (StackFunctor ExprValue ⊂ g) ⇒
  Expr → Free g () → Free g ()
eval' (Val n) c      = do {c; push (Num n)}
eval' (Add e1 e2) c = do
  eval' e2 (eval' e1 c)
  (Num n) ← pop
  (Num m) ← pop
  push (Num (m + n))
```

So that we deal with the $Stackfunctor$ injected into another effect functor g. For $Val\ n$, we feed in the existing stack and then use the $push$ operation to update it. In the Add case, we evaluate each sub-expression, implicitly passing the state using do-notation before finally popping from the stack and putting the combined result. We use open handlers to fold the semantics over the tree, thus separating concerns of semantics:

```
handleStackOpen :: Functor g ⇒
  Stack →
  Free (StackFunctor ExprValue + g) a →
  Free g (Stack, a)
handleStackOpen s (Var a)
  = return (s, a)
handleStackOpen (x : xs) (Cons (Inl (Pop k)))
  = handleStackOpen xs (k x)
handleStackOpen xs (Cons (Inl (Push x k)))
  = handleStackOpen (x : xs) k
```

with handlers for pure computations [5]:

```
data Void k deriving Functor
handleVoid :: Free Void a → a
handleVoid = fold ⊥ id
```

and can run it as follows:

```
Main > (handleVoid ∘ handleStackOpen [Num 2])
  (eval' (Add (Val 1) (Val 2)) (return ()))
([Num 3, Num 2], ())
```

With this we can calculate the compiler and virtual machine definitions using the same correctness specification, but replacing $eval$ with $eval'$. We proceed by performing constructive induction on the term s in the equation $exec\ (comp'\ s\ t)\ c = exec\ t'\ c$. We start with the base case $s = Val\ n$:

Proof. Base case $s = Val\ n$

```
  exec (comp' (Val n) t) c
= {-Equation 1 -}
  exec t (eval' (Val n) c)
= {-Definition of eval' -}
  exec t (do {c; push (Num n)})
= {-Define exec (PUSH v t) c and PUSH (below) -}
  exec (PUSH (Num n) t) c
```

so we define:

```
exec (PUSH v t) c = exec t (do
  c
  push v)
data Code where {...}
  PUSH :: ExprValue → Code → Code
comp' (Val n) t = PUSH (Num n) t
```

and we have:

```
exec (comp' (Val n) t) c = exec (PUSH (Num n) t) c
```

as required. □

We can see this in action as follows:

```
Main > (handleVoid ∘ handleStackOpen [])
  (exec (comp' (Val 2) HALT) (return ()))
([Num 2], ())
```

Next, we tackle the inductive Add case, we have the inductive hypothesis for sub-expressions $e1$ and $e2$:

$$exec\ (comp'\ e\ t')\ c' = exec\ t'\ (eval'\ e\ c') \qquad (2)$$

Proof. Inductive case $s = Add\ e1\ e2$

```
  exec (comp' (Add e1 e2) t) c
= {-Equation 1 -}
  exec t (eval' (Add e1 e2) c)
= {-Definition of eval' -}
  exec t (do
    eval' e2 (eval' e1 c)
    (Num n) ← pop
    (Num m) ← pop
    push (Num (m + n)))
= {-Define exec (ADD t) c and ADD (below) -}
  exec (ADD t) (eval' e2 (eval' e1 c))
= {-Induction hypothesis for e2, equation 2 -}
  exec (comp' e2 (ADD t)) (eval' e1 c)
= {-Induction hypothesis for e1, equation 2 -}
  exec (comp' e1 (comp' e2 (ADD t))) c
```

so we define:

```
exec (ADD t) c = exec t (do
  c
  (Num n) ← pop
  (Num m) ← pop
  push (Num (m + n)))
data Code where {...} ADD :: Code → Code
comp' (Add e1 e2) t = comp' e1 (comp' e2 (ADD t))
```

and we have:

```
exec (comp' (Add e1 e2) t) c =
  exec (comp' e1 (comp' e2 (ADD t))) c
```

as required. □

So we have compiler and VM definitions:

```
data Code where
  HALT :: Code
  PUSH :: Int → Code → Code
  ADD :: Code → Code
comp'  :: Expr → Code → Code
comp'  (Val n)      t = PUSH n t
comp'  (Add e1 e2) t = comp' e1 (comp' e2 (ADD t))
exec :: (StackFunctor ExprValue ⊂ g) ⇒
  Code → Free g () → Free g ()
exec HALT        c = c
exec (PUSH v t) c = exec t (do {c; push v})
exec (ADD t)     c = exec t (do
  c
  (Num n) ← pop
  (Num m) ← pop
  push (Num (m + n)))
```

Further Work

- Extend the approach to other configurations, such as queue-based or register-based machines.
- Apply the approach to realistic compilers, such as RISC architectures or the Multicore OCaml compiler.
- Formalise calculations in a theorem prover.
- Calculate algebraic handlers using Atkey and Johann's *f-and-m-algebras* [1], which extend initial algebra semantics from pure inductive datatypes to inductive datatypes interleaved with computational effects.
- Explore compiler optimisation using Wu and Shrijver's *fold fusion* [5] for algebraic handlers.

References

[1] R. Atkey and P. Johann. Interleaving data and effects. *JFP*, 25, November 2015.

[2] P. Bahr and G. Hutton. Calculating Correct Compilers. *JFP*, 25, September 2015.

[3] P. B. Levy. Call-by-push-value: A functional/imperative synthesis, September 2012.

[4] W. Swierstra. Data types à la carte. *JFP*, 18(4), July 2008.

[5] N. Wu and T. Schrijvers. Fusion for free: Efficient algebraic effect handlers. In *MPC*, June 2015.

[6] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. *Haskell Symposium*, 49(12), September 2014.

ICFP 2017, 22nd ACM SIGPLAN International Conference on Functional Programming, Student Research Competition Undergraduate Category, 3-9 September 2017, Oxford United Kingdom