# Algebraic Effects for Calculating Compilers

Luke Geeson

Department of Computer Science, University of Oxford
lgeeson@acm.org

S-REPLS7, University of Warwick

# Outline

# Outline

# Source languages and Semantics

Given *Hutton's Razor*[7] as the source language:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Expr}\\
&\quad = \mathit{Val}\ \mathit{Int}\\
&\quad \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}
\end{aligned}
$$

# Source languages and Semantics

Given *Hutton's Razor*[7] as the source language:

$$\begin{aligned}&\textbf{data}\ \mathit{Expr}\\&\quad = \mathit{Val}\ \mathit{Int}\\&\quad \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}\end{aligned}$$

And evaluation semantics *eval*:

$$\begin{aligned}&\mathit{eval} :: \mathit{Expr} \to \mathit{Int}\\&\mathit{eval}\ (\mathit{Val}\ n) = n\\&\mathit{eval}\ (\mathit{Add}\ e1\ e2) = \mathit{eval}\ e1 + \mathit{eval}\ e2\end{aligned}$$

We have a target language *Code*:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Code}\ \textbf{where}\\
&\quad \mathit{HALT} :: \mathit{Code}\\
&\quad \mathit{PUSH} :: \mathit{Int} \to \mathit{Code} \to \mathit{Code}\\
&\quad \mathit{ADD} :: \mathit{Code} \to \mathit{Code}
\end{aligned}
$$

We have a target language *Code*:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Code}\ \textbf{where}\\
&\quad \mathit{HALT} :: \mathit{Code}\\
&\quad \mathit{PUSH} :: \mathit{Int} \to \mathit{Code} \to \mathit{Code}\\
&\quad \mathit{ADD} :: \mathit{Code} \to \mathit{Code}
\end{aligned}
$$

with a compiler $comp'$ and top-level $comp$:

$$
\begin{aligned}
&comp' :: Expr \to Code \to Code\\
&comp'\ (Val\ n)\qquad\quad t = PUSH\ n\ t\\
&comp'\ (Add\ e1\ e2)\ t = comp'\ e1\ (comp'\ e2\ (ADD\ t))
\end{aligned}
$$

$$
\begin{aligned}
&comp :: Expr \to Code\\
&comp\ x = comp'\ x\ HALT
\end{aligned}
$$

# VMs

and a *Stack-based* virtual machine to run the compiled code:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{type}\ \mathit{Stack} = [\mathit{Int}]} \\
\multicolumn{3}{l}{\mathit{exec} :: \mathit{Code} \rightarrow \mathit{Stack} \rightarrow \mathit{Stack}} \\
\mathit{exec}\ \mathit{HALT} & c & = c \\
\mathit{exec}\ (\mathit{PUSH}\ n\ t) & c & = \mathit{exec}\ t\ (n : c) \\
\mathit{exec}\ (\mathit{ADD}\ t) & (n : m : c) & = \mathit{exec}\ t\ ((m + n) : c)
\end{array}
$$

# VMs

and a *Stack-based* virtual machine to run the compiled code:

$$
\begin{array}{l}
\textbf{type}\ Stack = [\,Int\,] \\
exec :: Code \rightarrow Stack \rightarrow Stack \\
\begin{array}{lll}
exec\ HALT & c & = c \\
exec\ (PUSH\ n\ t) & c & = exec\ t\ (n : c) \\
exec\ (ADD\ t) & (n : m : c) & = exec\ t\ ((m + n) : c)
\end{array}
\end{array}
$$

example:

$$
\begin{array}{l}
Main > exec\ (comp'\ (Val\ 1)\ HALT)\ [\,] \\
[1]
\end{array}
$$

# VMs

and a *Stack-based* virtual machine to run the compiled code:

$$\begin{array}{lll}
\textbf{type}\ Stack = [Int] & & \\
exec :: Code \to Stack \to Stack & & \\
exec\ HALT & c & = c \\
exec\ (PUSH\ n\ t) & c & = exec\ t\ (n : c) \\
exec\ (ADD\ t) & (n : m : c) & = exec\ t\ ((m + n) : c)
\end{array}$$

example:

$$Main > exec\ (comp'\ (Val\ 1)\ HALT)\ []$$
$$[1]$$

We can derive this using equational reasoning!

# Equational Reasoning

Functional programmers enjoy the benefits of *referential transparency*, that is through algebraic manipulation and a substitution of 'equals of equals'[5].

# Equational Reasoning

Functional programmers enjoy the benefits of *referential transparency*, that is through algebraic manipulation and a substitution of 'equals of equals'[5].

$$double :: Int \rightarrow Int$$
$$double\ x = x + x$$

# Equational Reasoning

Functional programmers enjoy the benefits of *referential transparency*, that is through algebraic manipulation and a substitution of 'equals of equals'[5].

$$\begin{aligned} &\mathit{double} :: \mathit{Int} \rightarrow \mathit{Int} \\ &\mathit{double}\ x = x + x \end{aligned}$$

From which we can define a *quadruple* function:

$$\begin{aligned} &\mathit{quadruple} :: \mathit{Int} \rightarrow \mathit{Int} \\ &\mathit{quadruple}\ x = \mathit{double}\ (\mathit{double}\ x) \end{aligned}$$

is the same as

$$\mathit{quadruple}\ x = (x + x) + (x + x)$$

# Equational Reasoning

Functional programmers enjoy the benefits of *referential transparency*, that is through algebraic manipulation and a substitution of 'equals of equals'[5].

$$\begin{aligned} &\mathit{double} :: \mathit{Int} \to \mathit{Int} \\ &\mathit{double}\ x = x + x \end{aligned}$$

From which we can define a *quadruple* function:

$$\begin{aligned} &\mathit{quadruple} :: \mathit{Int} \to \mathit{Int} \\ &\mathit{quadruple}\ x = \mathit{double}\ (\mathit{double}\ x) \end{aligned}$$

is the same as

$$\mathit{quadruple}\ x = (x + x) + (x + x)$$

Which we can derive:

**Proof.**

$$double\ (double\ x) = (x + x) + (x + x)$$

$$\begin{aligned} & double\ (double\ x) \\ = & \ \{\text{-Definition of inner } double \text{ -}\} \\ & double\ (x + x) \\ = & \ \{\text{-Definition of } double \text{ -}\} \\ & (x + x) + (x + x) \end{aligned}$$

as required. □

Proof.

$double\ (double\ x) = (x + x) + (x + x)$

$$
\begin{aligned}
& double\ (double\ x) \\
= & \quad \{\text{-Definition of inner } double \text{ -}\} \\
& double\ (x + x) \\
= & \quad \{\text{-Definition of } double \text{ -}\} \\
& (x + x) + (x + x)
\end{aligned}
$$

as required. □

Adopting *constructive equational reasoning* we can derive function definitions as we go.

# Calculating Compilers: Background

- Wand, deriving compilers using continuation semantics[21].

# Calculating Compilers: Background

- Wand, deriving compilers using continuation semantics[21].
- Ager *et al.* in deriving virtual machines and compilers from interpreters[1].

# Calculating Compilers: Background

- Wand, deriving compilers using continuation semantics[21].
- Ager *et al.* in deriving virtual machines and compilers from interpreters[1].
- Meijer in calculating compilers[13].

# Calculating Compilers: Background

- Wand, deriving compilers using continuation semantics[21].
- Ager *et al.* in deriving virtual machines and compilers from interpreters[1].
- Meijer in calculating compilers[13].
- Bahr and Hutton on *Calculating Correct Compilers*[3].

Notably, work by Ager *et al.*[1] derived implementations from the relationship between interpreters (evaluation semantics), compilers and virtual machines:

$$\begin{tikzcd} \mathit{Expr} \arrow[r, "\mathit{comp}"] \arrow[dr, "\mathit{eval}"'] & \mathit{Code} \arrow[d, "\mathit{exec} \circ \mathit{flip}\ {[\,]}"] \\ & \mathit{Int} \end{tikzcd}$$

This relationship was taken further by Hutton[8] and generalised by Bahr and Hutton[3] to define correctness conditions of *comp*, *comp*′ and *exec* and *eval*′:

$$
\begin{tikzcd}
Expr \arrow[r, "eval' \circ flip\ c"] \arrow[d, "comp' \circ flip\ t"'] & {[Int]} \arrow[d, "exec\ t"] \\
Code \arrow[r, "exec \circ flip\ c"] & {[Int]}
\end{tikzcd}
\qquad
\begin{tikzcd}
Expr \arrow[r, "comp"] \arrow[dr, "eval' \circ flip\ c"'] & Code \arrow[d, "exec \circ flip\ {[\,]}"] \\
 & {[Int]}
\end{tikzcd}
$$

Or put another way, we obtain the following *correctness specifications*:

$$exec\ (comp'\ s\ t)\ c = exec\ t\ (eval'\ s\ c) \tag{1}$$

$$exec\ (comp\ s)\ c = eval'\ s\ c \tag{2}$$

for source expression $s :: Expr$, target code $t :: Code$, empty configuration $c :: [Int]$ and functions

# Calculating Correct Compilers

- Bahr and Hutton[3] describe a method which when given $s$, $eval'$ and the correctness specifications, can be applied to derive an implementation of $exec$, $comp$ and $comp'$ that implement the semantics $eval'$ and satisfy the correctness specifications via *constructive induction*.

# Calculating Correct Compilers

- Bahr and Hutton[3] describe a method which when given $s$, $eval'$ and the correctness specifications, can be applied to derive an implementation of $exec$, $comp$ and $comp'$ that implement the semantics $eval'$ and satisfy the correctness specifications via *constructive induction*.
- Constructive induction is an extension of constructive equational reasoning to encompass inductively defined languages such as $Expr$.

# Calculating Correct Compilers

- Bahr and Hutton[3] describe a method which when given $s$, $eval'$ and the correctness specifications, can be applied to derive an implementation of $exec$, $comp$ and $comp'$ that implement the semantics $eval'$ and satisfy the correctness specifications via *constructive induction*.
- Constructive induction is an extension of constructive equational reasoning to encompass inductively defined languages such as $Expr$.
- We can calculate correct definitions of the compiler and virtual machines from before.

We proceed by calculating $exec\ (comp\ s)\ c = exec\ t'\ c$ as follows:

We proceed by calculating $exec\ (comp\ s)\ c = exec\ t'\ c$ as follows:

Proof.

Calculation of *comp* definition

$$
\begin{aligned}
& exec\ (comp\ s)\ c \\
= & \ \{\text{-Equation 2 -}\} \\
& eval\ s\ c \\
= & \ \{\text{-Define } exec\ HALT\ c = c \text{ and } Code\ HALT \text{ constructor -}\} \\
& exec\ HALT\ (eval\ s\ c) \\
= & \ \{\text{-Equation 1 -}\} \\
& exec\ (comp'\ s\ HALT)\ c
\end{aligned}
$$

so we define:

$$
\begin{aligned}
& exec\ HALT\ c = c \\
& comp\ s = comp'\ s\ HALT \\
& \textbf{data}\ Code\ \textbf{where}\ \{\ldots\}\ HALT :: Code
\end{aligned}
$$

# Outline

# The Specification Problem

- Bahr and Hutton[3] change the correctness specification as languages become more complex.

# The Specification Problem

- Bahr and Hutton[3] change the correctness specification as languages become more complex.
- Specification Complexity increases with the inclusion of *Computational Effects*.

# The Specification Problem

- Bahr and Hutton[3] change the correctness specification as languages become more complex.
- Specification Complexity increases with the inclusion of *Computational Effects*.
- We want to tackle the *specification problem* by simply fixing the correctness specification outright. We do this by adopting *Algebraic Effects*

# Outline

# Monads

- Monads are the canonical means to model computational effects in functional languages[14, 20].

# Monads

- Monads are the canonical means to model computational effects in functional languages[14, 20].
- Informally, a computational effect is some notion of computation that influences how a function proceeds; an effect is a pattern in execution we wish to capture.

# Monads

- Monads are the canonical means to model computational effects in functional languages[14, 20].
- Informally, a computational effect is some notion of computation that influences how a function proceeds; an effect is a pattern in execution we wish to capture.
- We can redefine the VM to be *total* by adopting the *Maybe* type:

$$
\begin{array}{l}
exec'' \; :: Code \rightarrow Stack \rightarrow Maybe\ Stack \\
exec''\ HALT \qquad\quad c = return\ c \\
exec''\ (PUSH\ n\ t)\ c = exec''\ t\ (n : c) \\
exec''\ (ADD\ t) \qquad c = \mathbf{do} \\
\quad \mathbf{case}\ c\ \mathbf{of} \\
\qquad (x : y : xs) \rightarrow exec''\ t\ (x + y : xs) \\
\qquad \_ \qquad\qquad\;\; \rightarrow Nothing
\end{array}
$$

## What if we want multiple effects?

A solution: *Monad Transformers*, problems:

- Once a transformer stack is instantiated, the stack and the order of effects becomes concrete[9].

# What if we want multiple effects?

A solution: *Monad Transformers*, problems:

- Once a transformer stack is instantiated, the stack and the order of effects becomes concrete[9].
- We may have interleaving effects and statically defined stacks where 'no complete static layering of one effect over the other provides the desired semantics'[11].

# What if we want multiple effects?

A solution: *Monad Transformers*, problems:

- Once a transformer stack is instantiated, the stack and the order of effects becomes concrete[9].
- We may have interleaving effects and statically defined stacks where 'no complete static layering of one effect over the other provides the desired semantics'[11].
- Whilst lifting algebraic operations (*Just*, *Nothing* etc...) is typically easy, lifting scoped operations is not[22].

# What if we want multiple effects?

A solution: *Monad Transformers*, problems:

- Once a transformer stack is instantiated, the stack and the order of effects becomes concrete[9].
- We may have interleaving effects and statically defined stacks where 'no complete static layering of one effect over the other provides the desired semantics'[11].
- Whilst lifting algebraic operations (*Just*, *Nothing* etc...) is typically easy, lifting scoped operations is not[22].
- Programming with monads forces a *phase distinction*: in modelling impure computation in a pure way, we break the abstraction boundary[5, 9, 6, 11]

# The Solution?

What can we use to model computational effects?

# The Solution?

What can we use to model computational effects?
Algebraic Effects!

# Algebraic Effects

- Proposed by Plotkin and Pretnar[16, 17], is an alternative approach to modelling computational effects.

# Algebraic Effects

- Proposed by Plotkin and Pretnar[16, 17], is an alternative approach to modelling computational effects.
- An algebraic effect, assigns an to an effect to a set of *abstract operations* and an equational theory to constrain the behaviour of the operations.

# Algebraic Effects

- Proposed by Plotkin and Pretnar[16, 17], is an alternative approach to modelling computational effects.
- An algebraic effect, assigns an to an effect to a set of *abstract operations* and an equational theory to constrain the behaviour of the operations.
- Computations using algebraic effects become *abstract computations*[9].

# Algebraic Effects

- Proposed by Plotkin and Pretnar[16, 17], is an alternative approach to modelling computational effects.
- An algebraic effect, assigns an to an effect to a set of *abstract operations* and an equational theory to constrain the behaviour of the operations.
- Computations using algebraic effects become *abstract computations*[9].
- We solve the abstraction problem of monads through *modular abstraction*[9].

# Algebraic Effects

- Proposed by Plotkin and Pretnar[16, 17], is an alternative approach to modelling computational effects.
- An algebraic effect, assigns an to an effect to a set of *abstract operations* and an equational theory to constrain the behaviour of the operations.
- Computations using algebraic effects become *abstract computations*[9].
- We solve the abstraction problem of monads through *modular abstraction*[9].
- We do not adopt free theories however.

# Algebraic Effect Handlers

- Abstract ops require an implementation for which we adopt algebraic handlers[15].

# Algebraic Effect Handlers

- Abstract ops require an implementation for which we adopt algebraic handlers[15].
- An algebraic handler is a concrete interface for the abstract operations, thus enabling *modular instantiation of effects*.

# Algebraic Effect Handlers

- Abstract ops require an implementation for which we adopt algebraic handlers[15].
- An algebraic handler is a concrete interface for the abstract operations, thus enabling *modular instantiation of effects*.
- Between modular abstraction and modular instantiation, Kammar *et al.* note that all the issues of monad transformers, outlined before, are solved.

# Contributions

- Generalise Bahr and Hutton's calculation method [3] to machines with *configurations*, calculating correct compilers for Hutton's razor.

# Contributions

- Generalise Bahr and Hutton's calculation method [3] 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* [19] and Wu *et al*'s *Higher-Order Syntax* [23] for languages with interacting effects and scoping constructs.

# Contributions

- Generalise Bahr and Hutton's calculation method [3] 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* [19] and Wu *et al*'s *Higher-Order Syntax* [23] for languages with interacting effects and scoping constructs.
- Calculate compilers and virtual machines for languages with and without exceptions on *stack-based* machines.

# Contributions

- Generalise Bahr and Hutton's calculation method [3] 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* [19] and Wu *et al*'s *Higher-Order Syntax* [23] 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.

# Contributions

- Generalise Bahr and Hutton's calculation method [3] 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* [19] and Wu *et al*'s *Higher-Order Syntax* [23] 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 $\lambda$-Calculus [12] with exceptions as a non-trivial case study.

# Outline

# Hutton's Razor

We return to the toy language we have been using:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Expr} \\
&\quad = \mathit{Val}\ \mathit{Int} \\
&\quad \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}
\end{aligned}
$$

# Hutton's Razor

We return to the toy language we have been using:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Expr}\\
&\quad = \mathit{Val}\ \mathit{Int}\\
&\quad \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}
\end{aligned}
$$

And a more general evaluation semantics *eval*:

$$
\begin{aligned}
&\mathit{eval}\ :: \mathit{Expr} \rightarrow [\mathit{ExprValue}] \rightarrow [\mathit{ExprValue}]\\
&\mathit{eval}\ (\mathit{Val}\ n)\qquad c = (\mathit{Num}\ n) : c\\
&\mathit{eval}\ (\mathit{Add}\ e1\ e2)\ c =\\
&\quad \textbf{case}\ \mathit{eval}\ e1\ c\ \textbf{of}\\
&\qquad c' \rightarrow \textbf{case}\ \mathit{eval}\ e2\ c'\ \textbf{of}\\
&\qquad ((\mathit{Num}\ m) : (\mathit{Num}\ n) : c'') \rightarrow ((\mathit{Num}\ (m+n)) : c'')\\
&\textbf{data}\ \mathit{ExprValue} = \mathit{Num}\ \mathit{Int}
\end{aligned}
$$

# Implicit Effects in the Semantics

- There are implicit *push* and *pop* operations in the semantics: $c' \rightarrow$ **case** $eval\ e2\ c'$ **of** $((Num\ m) : (Num\ n) : c'') \rightarrow ((Num\ (m + n)) : c'')$

# Implicit Effects in the Semantics

- There are implicit *push* and *pop* operations in the semantics: $c' \rightarrow \textbf{case}\ eval\ e2\ c'\ \textbf{of}\ ((Num\ m) : (Num\ n) : c'') \rightarrow ((Num\ (m + n)) : c'')$
- This pattern matching is semantically equivalent to popping $Num\ n$ and $Num\ m$, then subsequently pushing $Num\ (m + n)$.

# Implicit Effects in the Semantics

- There are implicit *push* and *pop* operations in the semantics: $c' \rightarrow$ **case** $eval\ e2\ c'$ **of** $((Num\ m) : (Num\ n) : c'') \rightarrow ((Num\ (m+n)) : c'')$
- This pattern matching is semantically equivalent to popping $Num\ n$ and $Num\ m$, then subsequently pushing $Num\ (m+n)$.
- The *ExprValue* stack combined with *push* and *pop* operations form a stateful computation, and a stateful computation is a computational effect.

# Implicit Effects in the Semantics

- There are implicit *push* and *pop* operations in the semantics: $c' \rightarrow \textbf{case}\ eval\ e2\ c'\ \textbf{of}\ ((Num\ m) : (Num\ n) : c'') \rightarrow ((Num\ (m+n)) : c'')$
- This pattern matching is semantically equivalent to popping $Num\ n$ and $Num\ m$, then subsequently pushing $Num\ (m+n)$.
- The *ExprValue* stack combined with *push* and *pop* operations form a stateful computation, and a stateful computation is a computational effect.
- Differentiate between effects in the *Source/Semantics* and Effects in the *Configuration*, algebraic effects handles them uniformly!

# Modelling State

We declare the stateful stack effect with a *Stackfunctor* declaration, each constructor is an abstract operation, and part of the computational effect in question:

$$\begin{aligned}
&\textbf{data}\ \mathit{StackFunctor}\ s\ a \\
&\quad = \mathit{Pop}\ (s \rightarrow a) \\
&\quad \mid\ \mathit{Push}\ s\ a \\
&\qquad\quad \textbf{deriving}\ \mathit{Functor}
\end{aligned}$$

# Modelling State

We declare the stateful stack effect with a *Stackfunctor* declaration, each constructor is an abstract operation, and part of the computational effect in question:

$$\begin{aligned}
&\mathbf{data}\ \mathit{StackFunctor}\ s\ a \\
&\quad = \mathit{Pop}\ (s \to a) \\
&\quad \mid\ \mathit{Push}\ s\ a \\
&\qquad\quad \mathbf{deriving}\ \mathit{Functor}
\end{aligned}$$

Comes with canonical state laws.

# Outline

# Free Trees and Folds

- 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.

$$\textbf{data}\ \mathit{Free}\ f\ a = \mathit{Var}\ a \mid \mathit{Cons}\ (f\ (\mathit{Free}\ f\ a))$$

# Free Trees and Folds

- 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.

$$\textbf{data}\ Free\ f\ a = Var\ a \mid Cons\ (f\ (Free\ f\ a))$$

- Capture semantics with algebraic handlers, which *fold* algebras (semantics) over the tree to interpret it in the semantic domain (*Int*) [22].

$$\begin{aligned}&\textbf{instance}\ Functor\ f \Rightarrow Monad\ (Free\ f)\ \textbf{where}\\&\quad return\ = Var\\&\quad m \gg\!= f = fold\ Cons\ f\ m\end{aligned}$$

# IR ASTs and Free ASTs

With this free monad machinery, we create an abstract monadic evaluator that produces an abstract syntax tree from an expression:

$$
\begin{aligned}
&\mathit{eval}_{\mathit{free}} :: \mathit{Expr} \to \mathit{Free}\ (\mathit{StackFunctor}\ \mathit{ExprValue})\ ()\\
&\mathit{eval}_{\mathit{free}}\ (\mathit{Val}\ n) \qquad\ = \mathit{Cons}\ (\mathit{Push}\ (\mathit{Num}\ n)\ (\mathit{Var}\ ()))\\
&\mathit{eval}_{\mathit{free}}\ (\mathit{Add}\ e1\ e2) = \mathbf{do}\\
&\quad \mathit{eval}_{\mathit{free}}\ e1\\
&\quad \mathit{eval}_{\mathit{free}}\ e2\\
&\quad (\mathit{Num}\ n) \leftarrow \mathit{Cons}\ (\mathit{Pop}\ \mathit{Var})\\
&\quad (\mathit{Num}\ m) \leftarrow \mathit{Cons}\ (\mathit{Pop}\ \mathit{Var})\\
&\quad \mathit{Cons}\ (\mathit{Push}\ (\mathit{Num}\ (n+m))\ (\mathit{Var}\ ()))
\end{aligned}
$$

# IR ASTs and Free ASTs

With this free monad machinery, we create an abstract monadic evaluator that produces an abstract syntax tree from an expression:

$$
\begin{array}{l}
eval_{free} :: Expr \rightarrow Free\ (StackFunctor\ ExprValue)\ ()\\
eval_{free}\ (Val\ n) \qquad = Cons\ (Push\ (Num\ n)\ (Var\ ()))\\
eval_{free}\ (Add\ e1\ e2) = \mathbf{do}\\
\quad eval_{free}\ e1\\
\quad eval_{free}\ e2\\
\quad (Num\ n) \leftarrow Cons\ (Pop\ Var)\\
\quad (Num\ m) \leftarrow Cons\ (Pop\ Var)\\
\quad Cons\ (Push\ (Num\ (n+m))\ (Var\ ()))
\end{array}
$$

In this sense compiler IR ASTs and ASTs produced from $eval_{free}$ are the same!

## Closed Handlers

We can additionally define a *closed* handler for such abstract computations:

$$
\begin{array}{l}
\mathit{handleStackClosed} :: \\
\quad \mathit{Stack} \rightarrow \\
\quad \mathit{Free}\ (\mathit{StackFunctor}\ \mathit{ExprValue})\ a \rightarrow \\
\quad (\mathit{Stack}, a) \\
\mathit{handleStackClosed}\ s\ (\mathit{Var}\ x) \\
\quad = (s, x) \\
\mathit{handleStackClosed}\ (x : xs)\ (\mathit{Cons}\ (\mathit{Pop}\ k)) \\
\quad = \mathit{handleStackClosed}\ xs\ (k\ x) \\
\mathit{handleStackClosed}\ xs\ (\mathit{Cons}\ (\mathit{Push}\ x\ k)) \\
\quad = \mathit{handleStackClosed}\ (x : xs)\ k
\end{array}
$$

# example

$$\mathit{Main} > \mathit{handleStackClosed}\ [\mathit{Num}\ 11]$$
$$(\mathit{eval}_{\mathit{free}}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 1)))$$
$$([\mathit{Num}\ 3, \mathit{Num}\ 11], ())$$

## example

$$\begin{aligned}&\mathit{Main} > \mathit{handleStackClosed}\ [\mathit{Num}\ 11]\\&\quad(\mathit{eval}_{\mathit{free}}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 1)))\\&([\mathit{Num}\ 3, \mathit{Num}\ 11], ())\end{aligned}$$

problems:

- Cannot handle more than one effect
- Introduced syntax tree boilerplate *Cons*(-tructions) in $\mathit{eval}_{\mathit{free}}$ and the handler.

# example

$$\mathit{Main} > \mathit{handleStackClosed}\ [\mathit{Num}\ 11]$$
$$(\mathit{eval}_{\mathit{free}}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 1)))$$
$$([\mathit{Num}\ 3, \mathit{Num}\ 11], ())$$

problems:

- Cannot handle more than one effect
- Introduced syntax tree boilerplate *Cons*(-tructions) in $\mathit{eval}_{\mathit{free}}$ and the handler.

solve these problems by adopting Swierstra's datatypes *à la carte*[19].

# Outline

# Combining Effect-Functors

Want to handle more than one effect in the language and configuration, use co-product functors for this purpose:

$$
\begin{aligned}
&\textbf{data}\ (+)\ f\ g\ a\ \textbf{where}\\
&\mathit{Inl} :: f\ a \rightarrow (f + g)\ a\\
&\mathit{Inr} :: g\ a \rightarrow (f + g)\ a\\
&\textbf{deriving}\ \mathit{Functor}
\end{aligned}
$$

# Open Handlers

Can redefine the open handlers using co-product functors:

$$
\begin{array}{l}
\mathit{handleStackOpen} :: \mathit{Functor}\ g \Rightarrow \\
\quad \mathit{Stack} \\
\quad \rightarrow \mathit{Free}\ (\mathit{StackFunctor}\ \mathit{ExprValue} + g)\ a \\
\quad \rightarrow \mathit{Free}\ g\ (\mathit{Stack}, a) \\
\mathit{handleStackOpen}\ s\ (\mathit{Var}\ a) \\
\quad = \mathit{return}\ (s, a) \\
\mathit{handleStackOpen}\ (x : \mathit{xs})\ (\mathit{Cons}\ (\mathit{Inl}\ (\mathit{Pop}\ k))) \\
\quad = \mathit{handleStackOpen}\ \mathit{xs}\ (k\ x) \\
\mathit{handleStackOpen}\ \mathit{xs}\ (\mathit{Cons}\ (\mathit{Inl}\ (\mathit{Push}\ x\ k))) \\
\quad = \mathit{handleStackOpen}\ (x : \mathit{xs})\ k
\end{array}
$$

# Syntactic sugar

Introduce Swierstra's smart constructors[19], $g$ supports $f$:

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow f \subset g\ \textbf{where} \\
\quad \mathit{inj} :: f\ a \to g\ a \\
\quad \mathit{prj} :: g\ a \to \mathit{Maybe}\ (f\ a)
\end{array}
$$

with class additional instances, (see [19])

# Syntactic sugar

Introduce Swierstra's smart constructors[19], $g$ supports $f$:

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow f \subset g\ \textbf{where} \\
\quad \mathit{inj} :: f\ a \to g\ a \\
\quad \mathit{prj} :: g\ a \to \mathit{Maybe}\ (f\ a)
\end{array}
$$

with class additional instances, (see [19]) We have *injection* and *projection* functions to capture the tree boilerplate *Cons*(-tructions):

$$
\begin{array}{lll}
\mathit{inject} & :: (g \subset f) \Rightarrow g\ (\mathit{Free}\ f\ a) \to \mathit{Free}\ f\ a & \\
\mathit{inject} & & = \mathit{Cons} \circ \mathit{inj} \\
\mathit{project} & :: (f \subset g) \Rightarrow \mathit{Free}\ g\ a \to \mathit{Maybe}\ (f\ (\mathit{Free}\ g\ a)) & \\
\mathit{project}\ (\mathit{Cons}\ s) & & = \mathit{prj}\ s \\
\mathit{project}\ \_ & & = \mathit{Nothing}
\end{array}
$$

# Syntactic Sugar II

We redefine the abstract operations used in $eval_{free}$:

$$
\begin{aligned}
&pop' :: (StackFunctor\ ExprValue \subset g) \Rightarrow Free\ g\ ExprValue \\
&pop' = inject\ (Pop\ Var) \\
&push' :: (StackFunctor\ ExprValue \subset g) \Rightarrow ExprValue \rightarrow Free\ g\ () \\
&push'\ v = inject\ (Push\ v\ (Var\ ()))
\end{aligned}
$$

# Syntactic Sugar II

We redefine the abstract operations used in $eval_{free}$:

$$
\begin{aligned}
&pop' :: (StackFunctor\ ExprValue \subset g) \Rightarrow Free\ g\ ExprValue\\
&pop' = inject\ (Pop\ Var)\\
&push' :: (StackFunctor\ ExprValue \subset g) \Rightarrow ExprValue \rightarrow Free\ g\ ()\\
&push'\ v = inject\ (Push\ v\ (Var\ ()))
\end{aligned}
$$

This captures that tricky tree notation

# *eval* redefined

We get a simpler $eval_{free}$ function:

$$
\begin{array}{l}
eval' :: (StackFunctor\ ExprValue \subset g) \Rightarrow \\
\quad Expr \rightarrow Free\ g\ () \rightarrow Free\ g\ () \\
eval'\ (Val\ n)\ c = \mathbf{do} \\
\quad c \\
\quad push'\ (Num\ n) \\
eval'\ (Add\ e1\ e2)\ c = \mathbf{do} \\
\quad \mathbf{let}\ c' = eval'\ e1\ c \\
\quad eval'\ e2\ c' \\
\quad (Num\ n) \leftarrow pop' \\
\quad (Num\ m) \leftarrow pop' \\
\quad push'\ (Num\ (m + n))
\end{array}
$$

# Expr, handled

With a handler for pure computations [22]:

$$\textbf{data}\ \mathit{Void}\ k\ \textbf{deriving}\ \mathit{Functor}$$

$$\begin{aligned}&\mathit{handleVoid} :: \mathit{Free}\ \mathit{Void}\ a \to a\\&\mathit{handleVoid} = \mathit{fold}\ \bot\ \mathit{id}\end{aligned}$$

# Expr, handled

With a handler for pure computations [22]:

$$\mathbf{data}\ \mathit{Void}\ k\ \mathbf{deriving}\ \mathit{Functor}$$

$$\begin{array}{l} \mathit{handleVoid} :: \mathit{Free}\ \mathit{Void}\ a \rightarrow a \\ \mathit{handleVoid} = \mathit{fold}\ \bot\ \mathit{id} \end{array}$$

can run as follows:

$$\begin{array}{l} \mathit{Main} > (\mathit{handleVoid} \circ \mathit{handleStackOpen}\ [\mathit{Num}\ 2]) \\ \quad (\mathit{eval}'\ (\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2))\ (\mathit{return}\ ())) \\ ([\mathit{Num}\ 3, \mathit{Num}\ 2], ()) \end{array}$$

# Overview so far

- *Roll-your-own* effect-handler approach inspired by Swierstra's *smart constructors*[19]

# Overview so far

- *Roll-your-own* effect-handler approach inspired by Swierstra's *smart constructors*[19]
- modified/simplified semantics of *eval* with effects highlighted.

# Overview so far

- *Roll-your-own* effect-handler approach inspired by Swierstra's *smart constructors*[19]
- modified/simplified semantics of *eval* with effects highlighted.
- separated concerns of syntax from semantics using effect handlers and algebraic effects.

benefits:

# Overview so far

- *Roll-your-own* effect-handler approach inspired by Swierstra's *smart constructors*[19]
- modified/simplified semantics of *eval* with effects highlighted.
- separated concerns of syntax from semantics using effect handlers and algebraic effects.

benefits:

- Scalability

# Overview so far

- *Roll-your-own* effect-handler approach inspired by Swierstra's *smart constructors*[19]
- modified/simplified semantics of *eval* with effects highlighted.
- separated concerns of syntax from semantics using effect handlers and algebraic effects.

benefits:

- Scalability
- Abstraction

# Overview so far

- *Roll-your-own* effect-handler approach inspired by Swierstra's *smart constructors*[19]
- modified/simplified semantics of *eval* with effects highlighted.
- separated concerns of syntax from semantics using effect handlers and algebraic effects.

benefits:

- Scalability
- Abstraction
- Flexibility

# Overview so far

- *Roll-your-own* effect-handler approach inspired by Swierstra's *smart constructors*[19]
- modified/simplified semantics of *eval* with effects highlighted.
- separated concerns of syntax from semantics using effect handlers and algebraic effects.

benefits:

- Scalability
- Abstraction
- Flexibility

Now to calculate a correct compiler with algebraic effects!

# Outline

# Calculating Compilers, Statefully

Using the same correctness specifications, 1 and 2 but replacing *eval* with *eval'*:

$$exec\ (comp'\ s\ t)\ c = exec\ t\ (eval'\ s\ c) \tag{3}$$

$$exec\ (comp\ s)\ c = eval'\ s\ c \tag{4}$$

We proceed by performing constructive induction on the term $s$ in the equation $exec\ (comp'\ s\ t)\ c = exec\ t'\ c$.

# Calculating Compilers, Statefully

Using the same correctness specifications, 1 and 2 but replacing *eval* with *eval′*:

$$exec\ (comp'\ s\ t)\ c = exec\ t\ (eval'\ s\ c) \tag{3}$$

$$exec\ (comp\ s)\ c = eval'\ s\ c \tag{4}$$

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 $s = Val\ n$:

$$eval'\ (Val\ n)\ c = \mathbf{do}\ \{c; push'\ (Num\ n)\}$$

## Proof.

Base case $s = \mathit{Val}\ n$

$$
\begin{aligned}
& \mathit{exec}\ (\mathit{comp}'\ (\mathit{Val}\ n)\ t)\ c \\
= & \quad \{\text{-Equation 3 -}\} \\
& \mathit{exec}\ t\ (\mathit{eval}'\ (\mathit{Val}\ n)\ c) \\
= & \quad \{\text{-Definition of } \mathit{eval}' \text{ -}\} \\
& \mathit{exec}\ t\ (\mathbf{do}\ \{ c; \mathit{push}'\ (\mathit{Num}\ n) \}) \\
= & \quad \{\text{-Define } \mathit{exec}\ (\mathit{PUSH}\ v\ t)\ c \text{ (below) and } \mathit{Code}\ \mathit{PUSH} \text{ -}\} \\
& \mathit{exec}\ (\mathit{PUSH}\ (\mathit{Num}\ n)\ t)\ c
\end{aligned}
$$

so we define:

$$
\begin{aligned}
& \mathit{exec}\ (\mathit{PUSH}\ v\ t)\ c = \mathit{exec}\ t\ (\mathbf{do}\ \{ c; \mathit{push}'\ v \}) \\
& \mathbf{data}\ \mathit{Code}\ \mathbf{where}\ \{ \ldots \}\ \mathit{PUSH} :: \mathit{ExprValue} \to \mathit{Code} \to \mathit{Code} \\
& \mathit{comp}'\ (\mathit{Val}\ n)\ t = \mathit{PUSH}\ (\mathit{Num}\ n)\ t
\end{aligned}
$$

# Inductive case

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

$$exec\ (comp'\ e\ t')\ c' = exec\ t'\ (eval'\ e\ c') \tag{5}$$

$$
\begin{array}{l}
eval'\ (Add\ e1\ e2)\ c = \mathbf{do} \\
\quad \mathbf{let}\ c' = eval'\ e1\ c \\
\quad eval'\ e2\ c' \\
\quad (Num\ n) \leftarrow pop' \\
\quad (Num\ m) \leftarrow pop' \\
\quad push'\ (Num\ (m + n))
\end{array}
$$

## Proof.

Inductive case $s = Add\ e1\ e2$

$$
\begin{aligned}
& exec\ (comp'\ (Add\ e1\ e2)\ t)\ c \\
= \ & \{\text{-Equation 3 -}\} \\
& exec\ t\ (eval'\ (Add\ e1\ e2)\ c) \\
= \ & \{\text{-Definition of } eval' \text{ -}\} \\
& exec\ t\ (\mathbf{do} \\
& \quad \mathbf{let}\ c' = eval'\ e1\ c \\
& \quad eval'\ e2\ c' \\
& \quad (Num\ n) \leftarrow pop' \\
& \quad (Num\ m) \leftarrow pop' \\
& \quad push'\ (Num\ (m + n)))
\end{aligned}
$$

□

## Proof.

$$
\begin{aligned}
= &\quad \{\text{-let substitution -}\} \\
& exec\ t\ (\mathbf{do} \\
&\quad eval'\ e2\ (eval'\ e1\ c) \\
&\quad (Num\ n) \leftarrow pop' \\
&\quad (Num\ m) \leftarrow pop' \\
&\quad push'\ (Num\ (m+n))) \\
= &\quad \{\text{-Define } exec\ (ADD\ t)\ c \text{ and } Code\ ADD \text{ -}\} \\
& exec\ (ADD\ t)\ (eval'\ e2\ (eval'\ e1\ c)) \\
= &\quad \{\text{-Induction hypothesis for e2, eq 5 -}\} \\
& exec\ (comp'\ e2\ (ADD\ t))\ (eval'\ e1\ c) \\
= &\quad \{\text{-Induction hypothesis for e1, eq 5 -}\} \\
& exec\ (comp'\ e1\ (comp'\ e2\ (ADD\ t)))\ c
\end{aligned}
$$

□

# Calculation end

so we define:

$$
\begin{array}{l}
exec\ (ADD\ t)\ c = exec\ t\ (\mathbf{do} \\
\quad c \\
\quad (Num\ n) \leftarrow pop' \\
\quad (Num\ m) \leftarrow pop' \\
\quad push'\ (Num\ (m + n))) \\
\mathbf{data}\ Code\ \mathbf{where}\ \{...\}\ ADD :: Code \rightarrow Code \\
comp'\ (Add\ e1\ e2)\ t = comp'\ e1\ (comp'\ e2\ (ADD\ t))
\end{array}
$$

and we have:

$$
\begin{array}{l}
exec\ (comp'\ (Add\ e1\ e2)\ t)\ c = \\
\quad exec\ (comp'\ e1\ (comp'\ e2\ (ADD\ t)))\ c
\end{array}
$$

as required.

# Calculation Summary

- Calculation similar to calculation of Bahr and Hutton[3], except now with monadic equational reasoning to boot.

# Calculation Summary

- Calculation similar to calculation of Bahr and Hutton[3], except now with monadic equational reasoning to boot.
- Compiler and VM definitions are correct by construction[18].

# Calculation Summary

- Calculation similar to calculation of Bahr and Hutton[3], except now with monadic equational reasoning to boot.
- Compiler and VM definitions are correct by construction[18].
- algebraic effects preserve abstractions and come with algebraic laws (compiler optimisation?)
- Calculation of *comp* comes from straightforward equational reasoning, see *Background*.

# Contributions Revisited

- Generalise Bahr and Hutton's calculation method [3] to machines with *configurations*, calculating correct compilers for Hutton's razor. ✓

# Contributions Revisited

- Generalise Bahr and Hutton's calculation method [3] 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* [19] and Wu *et al*'s *Higher-Order Syntax* [23] for languages with interacting effects and scoping constructs.

# Contributions Revisited

- Generalise Bahr and Hutton's calculation method [3] 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* [19] and Wu *et al*'s *Higher-Order Syntax* [23] for languages with interacting effects and scoping constructs.
- Calculate compilers and virtual machines for languages with and without exceptions on *stack-based* machines.

# Contributions Revisited

- Generalise Bahr and Hutton's calculation method [3] 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* [19] and Wu *et al*'s *Higher-Order Syntax* [23] 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.

# Contributions Revisited

- Generalise Bahr and Hutton's calculation method [3] 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* [19] and Wu *et al*'s *Higher-Order Syntax* [23] 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 $\lambda$-Calculus [12] with exceptions as a non-trivial case study.

# Outline

# Related Work

- Gibbons and Hinze[5] on monadic equational reasoning and Bahr and Hutton on calculating correct compilers[3]

# Related Work

- Gibbons and Hinze[5] on monadic equational reasoning and Bahr and Hutton on calculating correct compilers[3]
- Kiselyov *et al.*'s *extensible effects* Haskell library[11] uses open unions and more recently *freer monads*[10].

# Related Work

- Gibbons and Hinze[5] on monadic equational reasoning and Bahr and Hutton on calculating correct compilers[3]
- Kiselyov *et al.*'s *extensible effects* Haskell library[11] uses open unions and more recently *freer monads*[10].
- Day and Hutton come close to effect-handlers for compilers[4], developing datatypes *à la carte* however the link with free monads and algebraic effect handlers was not made.

# Related Work

- Gibbons and Hinze[5] on monadic equational reasoning and Bahr and Hutton on calculating correct compilers[3]
- Kiselyov *et al.*'s *extensible effects* Haskell library[11] uses open unions and more recently *freer monads*[10].
- Day and Hutton come close to effect-handlers for compilers[4], developing datatypes *à la carte* however the link with free monads and algebraic effect handlers was not made.
- Wu *et al.* make use of *pattern synonyms* and *view patterns*[23] to capture the abstract operations.

# Outline

# Further Work

- Extend the approach to other configurations, such as queue-based or register-based machines.

# 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 e.g. Multicore OCaml compiler.

# 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 e.g. Multicore OCaml compiler.
- Formalise calculations in a theorem prover.

# 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 e.g. Multicore OCaml compiler.
- Formalise calculations in a theorem prover.
- Calculate algebraic handlers using Atkey and Johann's *f-and-m-algebras* [2], which extend initial algebra semantics from pure inductive datatypes to inductive datatypes interleaved with computational effects.

# 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 e.g. Multicore OCaml compiler.
- Formalise calculations in a theorem prover.
- Calculate algebraic handlers using Atkey and Johann's *f-and-m-algebras* [2], 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* [22] for algebraic handlers.

Thank you for listening!

Any Questions?

Code available at:

# References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. *BRICS Report Series*, 10(14), 2003.

[2] R. Atkey and P. Johann. Interleaving data and effects. *Journal of Functional Programming*, 25, 2015.

[3] P. Bahr and G. Hutton. Calculating Correct Compilers. *Journal of Functional Programming*, 25, Sept. 2015.

[4] L. E. Day and G. Hutton. Compilation À la carte. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, IFL '13, pages 13:13–13:24, New York, NY, USA, 2014. ACM.

[5] J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 2–14, September 2011.

[6] D. Hillerström, S. Lindley, R. Atkey, and K. Sivaramakrishnan. Continuation passing style for effect handlers. Accepted for FSCD, 2017.

# References (cont.)

[7] G. Hutton. Fold and Unfold for Program Semantics. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming(ICFP)*, Baltimore, Maryland, Sept. 1998.

[8] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, September 2016.

[9] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ICFP '13, pages 145–158, New York, NY, USA, 2013. ACM.

[10] O. Kiselyov and H. Ishii. Freer monads, more extensible effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 94–105, New York, NY, USA, 2015. ACM.

[11] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 59–70. ACM, 2013.

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

[13] E. Meijer. *calculating compilers*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.

# References (cont.)

[14] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science (LICS).

[15] G. Plotkin and M. Pretnar. *Handlers of Algebraic Effects*, pages 80–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[16] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science (LICS)*, Volume 9, Issue 4, December 2013.

[17] M. Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19 – 35, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[18] N. Shah. Program construction: Calculating implementations from specifications by r.c. backhouse, john wiley & sons, 2004. *J. Funct. Program.*, 14(5):598–600, Sept. 2004.

[19] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008.

# References (cont.)

[20] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.

[21] M. Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):496–517, July 1982.

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

[23] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 49(12):1–12, September 2014.