Calculating Compilers Effectively

Functional Pearl

Zac Garby
University of Nottingham
Nottingham, United Kingdom
zac.garby@nottingham.ac.uk

Graham Hutton
University of Nottingham
Nottingham, United Kingdom
graham.hutton@nottingham.ac.uk

Patrick Bahr
IT University of Copenhagen
Copenhagen, Denmark
paba@itu.dk

Abstract

Much work in the area of compiler calculation has focused on pure languages. While this simplifies the reasoning, it reduces the applicability. In this article, we show how an existing compiler calculation methodology can be naturally extended to languages with side-effects. We achieve this by exploiting an algebraic approach to effects, which keeps the reasoning simple and provides flexibility in how effects are interpreted. To make the ideas accessible we only use elementary functional programming techniques.

CCS Concepts: • **Software and its engineering → Compilers**; **Formal software verification**; • **Theory of computation → Logic and verification**; **Program verification**.

Keywords: program calculation, algebraic effects, monads

ACM Reference Format:
Zac Garby, Graham Hutton, and Patrick Bahr. 2024. Calculating Compilers Effectively: Functional Pearl. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium (Haskell '24), September 6–7, 2024, Milan, Italy*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3677999.3678283

1 Introduction

Compiler calculation aims to develop compilers whose correctness is guaranteed by construction. The process typically begins with a semantics for the language being compiled, and seeks to derive a compiler that preserves the semantics using some form of stepwise reasoning. A range of approaches for achieving this aim have been developed over the years, such as [Ager et al. 2003; Bahr and Hutton 2015, 2020; Elliott 2017; Gibbons 2022; Meijer 1992; Wand 1982].

To date, however, work in this area has primarily focused on pure languages. Side effects, such as mutable state, tend to complicate the reasoning process: semantic domains becomes more complex, evaluation order becomes important, and it is often necessary to reason explicitly with monads. Those that have considered effects usually simulate them in a pure manner, rather than using real side effects. For example, [Bahr and Hutton 2015; Meijer 1992; Wand 1982] simulate state using pure functions of type $State \to (Result, State)$, rather than using real mutable references.

Despite the above complications, side-effects are important for any practical language. In this article, we show how an algebraic approach to effects inspired by [Bahr and Hutton 2023; Pretnar 2015; Swierstra and Altenkirch 2007] can be combined with the compiler calculation methodology of [Bahr and Hutton 2015] to derive compilers for languages with effects using simple equational reasoning.

We introduce our approach via three examples of increasing complexity, starting with printing, then state, and finally non-determinism. A key aspect of our algebraic approach is flexibility in how effects are interpreted. We demonstrate this by providing a range of semantics for each effect, including semantics that perform real effects, and versions with extra features such as single stepping and logging.

To make the ideas accessible we don't assume prior knowledge of compiler calculation or algebraic effects, and we take an elementary approach using simple, explicit definitions where possible. We discuss how our techniques can be generalised and abstracted at the end of the article.

All the programs and calculations are written in Haskell, exploiting its built-in support for monads, but the underlying ideas are applicable in any functional language. All the calculations have also been verified in Agda, and all the code is available online as supplementary material.

2 A language with printing

To introduce our approach to compiler calculation for languages with effects, we begin by considering a simple language of expressions built up from integer values using an addition operator [Hutton 2023], extended with a primitive for printing the value of an expression:

data *Expr* = *Val Int* | *Add Expr Expr* | *Print Expr*

For example, the expression *Print* (*Add* (*Val* 1) (*Val* 2)) adds together two integers and prints the result. This behaviour could be realised in practice by defining a semantic function $eval :: Expr \to IO\ Int$ that evaluates an expression

Haskell '24, September 6–7, 2024, Milan, Italy

ACM ISBN 979-8-4007-1102-2/24/09
https://doi.org/10.1145/3677999.3678283

to an action that returns an integer value, where the use of the *IO* monad allows the side effect of printing.

For our purposes, however, we prefer to take a more abstract, algebraic approach to effects, which for this simple expression language can be achieved by replacing the use of the built-in *IO* monad by a custom monad of *print sequences* that captures the notion of printing integer values.

2.1 Print sequences

The type of print sequences, *PrintSeq a*, represents sequences of computations that return a final value of type *a*, and may use an operation that prints integer values:

```
data PrintSeq a where
  Ret      :: a → PrintSeq a
  PrintInt :: Int → PrintSeq a → PrintSeq a
```

This declaration could also be parameterised by the type of values being printed, but for simplicity we don't do this here. Informally, *Ret v* returns the value *v*, while *PrintInt n ps* prints the integer *n* and then behaves as the print sequence *ps*. For example, a print sequence that prints three integers and returns the void result () can be defined as follows:

```
three :: PrintSeq ()
three = PrintInt 1 (PrintInt 2 (PrintInt 3 (Ret ())))
```

Note that *three* does not print anything, but rather builds a data structure that represents the action of printing.

Print sequences naturally form a monad, with *return* being simply the *Ret* constructor, and the bind operator ≫= defined by replacing the *Ret* at the end of a print sequence by a function that returns another such sequence:

```
instance Monad PrintSeq where
  return :: a → PrintSeq a
  return v = Ret v

  (≫=) :: PrintSeq a → (a → PrintSeq b) → PrintSeq b
  Ret v          ≫= f = f v
  PrintInt n ps ≫= f = PrintInt n (ps ≫= f)
```

It is straightforward to show that these definitions satisfy the monad laws, which express, modulo the fact that ≫= involves a binding operation, that *return* is the identity for the ≫= operator, and that ≫= is associative:

$$\begin{aligned} \mathit{return}\ v \gg\!= f &= f\ v \\ ps \gg\!= \mathit{return} &= ps \\ (ps \gg\!= f) \gg\!= g &= ps \gg\!= (\lambda v \to f\ v \gg\!= g) \end{aligned}$$

We will use these properties, together with the **do** notation, when writing and reasoning about print sequences.

We conclude by noting that print sequences are essentially just lists of integers, except that lists end with the empty list [ ] whereas print sequences end with a value of type *a*. Indeed, this observation provides an alternative formulation of print sequences as a pair of type ([*Int*], *a*), which is a writer monad that accumulates a list of integers. However, we prefer the original recursive definition for print sequences, as it emphasises their algebraic nature, and naturally generalises to other forms of effect later on.

2.2 Semantics of expressions

To define a semantics for expressions, we first define a helper function that constructs a singleton print sequence:

```
printInt :: Int → PrintSeq ()
printInt n = PrintInt n (Ret ())
```

We could have defined this function to return *n* itself, rather than the void result (). However, we prefer the above definition because it mirrors the behaviour of the basic output primitives in Haskell, and also results in simpler properties when we consider the semantics of print sequences.

Using the fact the print sequences form a monad, it is then easy to define an evaluation semantics for expressions:

```
eval :: Expr → PrintSeq Int
eval (Val n)     = return n
eval (Add x y) = do n ← eval x; m ← eval y; return (n + m)
eval (Print x)   = do n ← eval x; printInt n; return n
```

Note that *eval* itself doesn't actually print anything, but builds a data structure that represents the sequence of print operations that result from evaluation.

2.3 Compiler specification

We have now defined a type *Expr* that represents the syntax of a simple expression language that supports printing, and a function *eval* that gives a monadic evaluation semantics for the language in terms of print sequences. In this section we show how to specify the desired behaviour of a compiler for this simple expression language.

Our goal is to define a function *comp* :: *Expr* → *Code* that compiles an expression into code for a suitable machine. We assume the compiler targets a stack-based machine, whose semantics is given a function *exec* :: *Code* → *Stack* → *Stack* that executes code using an initial stack to give a final stack, where a stack is simply a list of integers:

```
type Stack = [Int]
```

However, because the function *eval* :: *Expr* → *PrintSeq Int* defines the semantics of expressions in terms of print sequences, we also generalise the type of the execution function to operate in the same monadic setting:

```
exec :: Code → Stack → PrintSeq Stack
```

The definitions for the *Code* datatype and the *exec* function are not given up front, but will rather fall out naturally as part of the process of calculating the compiler itself.

Prior to specifying the desired behaviour of the compiler, we generalise the function *comp* to take additional code to be executed after the compiled code. The addition of such a

code continuation is a key aspect of the methodology and simplifies the resulting calculations [Bahr and Hutton 2015]. Using this idea, our goal now is to establish the following compiler correctness property for the generalised compilation function $comp :: Expr \to Code \to Code$:

$$exec\ (comp\ e\ c)\ s\ =\ \mathbf{do}\ v \leftarrow eval\ e; exec\ c\ (v : s) \quad (1)$$

That is, compiling an expression and executing the resulting code together with the supplied additional code should give the same print sequence as executing the additional code with the value of the expression on top of the stack.

2.4 Compiler calculation

We now show how equation (1) can be used to calculate an implementation of the compiler, by induction on the expression e. For each case of e, we start with the right-hand side of the equation, $\mathbf{do}\ v \leftarrow eval\ e; exec\ c\ (v : s)$, and seek to transform it by equational reasoning into the form $exec\ c'\ s$ for some code c'. We then define $comp\ e\ c = c'$, which gives us a clause for the compiler that is guaranteed by construction to satisfy the correctness equation.

The base case, $e = Val\ n$, begins by applying *eval*, and simplifying the resulting term using the monad laws:

$$\begin{array}{ll} & \mathbf{do}\ v \leftarrow eval\ (Val\ n); exec\ c\ (v : s) \\ = & \quad \{\ \text{definition of } eval\ \} \\ & \mathbf{do}\ v \leftarrow return\ n; exec\ c\ (v : s) \\ = & \quad \{\ \text{monad laws}\ \} \\ & exec\ c\ (n : s) \end{array}$$

To complete the calculation for this case, we need to arrive at a term of the form $exec\ c'\ s$. That is, we have to find some code c' that solves the following equation:

$$exec\ c'\ s\ =\ exec\ c\ (n : s)$$

Note that we cannot simply use this equation as a defining clause for *exec*, as the variables n and c would be unbound in the body of the definition. The solution is to package these two variables up in the code argument c', which can freely be instantiated as it is existentially quantified. This can be achieved by adding a new constructor to the *Code* datatype that takes an integer and code as arguments,

$$PUSH :: Int \to Code \to Code$$

and defining a new clause for the function *exec*:

$$exec\ (PUSH\ n\ c)\ s = exec\ c\ (n : s)$$

That is, executing code of the form $PUSH\ n\ c$ proceeds by pushing the value n onto the stack and then executing the code c, which motivates the choice of name for the new constructor. This definition solves the above equation, and allows us to conclude the calculation:

$$\begin{array}{ll} & exec\ c\ (n : s) \\ = & \quad \{\ \text{definition of } exec\ \} \\ & exec\ (PUSH\ n\ c)\ s \end{array}$$

The resulting term is now of the form $exec\ c'\ s$, with $c' = PUSH\ n\ c$, which gives the first clause for the compiler:

$$comp\ (Val\ n)\ c = PUSH\ n\ c$$

The inductive case for addition, $e = Add\ x\ y$, proceeds in a similar manner, in which we introduce a new code constructor *ADD* that adds the top two values on the stack, this time motivated by the desire to transform the term into a form to which the induction hypotheses can be applied:

$$\begin{array}{ll} & \mathbf{do}\ v \leftarrow eval\ (Add\ x\ y); exec\ c\ (v : s) \\ = & \quad \{\ \text{definition of } eval\ \} \\ & \mathbf{do}\ v \leftarrow \mathbf{do}\ \{ n \leftarrow eval\ x; m \leftarrow eval\ y; \\ & \qquad return\ (n + m)\ \}; exec\ c\ (v : s) \\ = & \quad \{\ \text{monad laws}\ \} \\ & \mathbf{do}\ n \leftarrow eval\ x; m \leftarrow eval\ y; exec\ c\ ((n + m) : s) \\ = & \quad \{\ \text{define: } exec\ (ADD\ c)\ (m : n : s) = \\ & \qquad exec\ c\ ((n + m) : s)\ \} \\ & \mathbf{do}\ n \leftarrow eval\ x; m \leftarrow eval\ y; exec\ (ADD\ c)\ (m : n : s) \\ = & \quad \{\ \text{induction hypothesis for } y\ \} \\ & \mathbf{do}\ n \leftarrow eval\ x; exec\ (comp\ y\ (ADD\ c))\ (n : s) \\ = & \quad \{\ \text{induction hypothesis for } x\ \} \\ & exec\ (comp\ x\ (comp\ y\ (ADD\ c)))\ s \end{array}$$

The resulting term is now of the form $exec\ c'\ s$, which gives the second clause for the compilation function:

$$comp\ (Add\ x\ y)\ c = comp\ x\ (comp\ y\ (ADD\ c))$$

Finally, the inductive case for printing, $e = Print\ x$, introduces a new code constructor *PRINT* that prints the top value on the stack, which again allows induction to be applied:

$$\begin{array}{ll} & \mathbf{do}\ v \leftarrow eval\ (Print\ x); exec\ c\ (v : s) \\ = & \quad \{\ \text{definition of } eval\ \} \\ & \mathbf{do}\ v \leftarrow \mathbf{do}\ \{ n \leftarrow eval\ x; printInt\ n; \\ & \qquad return\ n\ \}; exec\ c\ (v : s) \\ = & \quad \{\ \text{monad laws}\ \} \\ & \mathbf{do}\ n \leftarrow eval\ x; printInt\ n; exec\ c\ (n : s) \\ = & \quad \{\ \text{define: } exec\ (PRINT\ c)\ (n : s) = \\ & \qquad \mathbf{do}\ printInt\ n; exec\ c\ (n : s)\ \} \\ & \mathbf{do}\ n \leftarrow eval\ x; exec\ (PRINT\ c)\ (n : s) \\ = & \quad \{\ \text{induction hypothesis for } x\ \} \\ & exec\ (comp\ x\ (PRINT\ c))\ s \end{array}$$

To complete the compiler calculation, we consider a top-level function $compile :: Expr \to Code$ that compiles expressions into code, specified by the following property:

$$exec\ (compile\ e)\ s\ =\ \mathbf{do}\ v \leftarrow eval\ e; return\ (v : s) \quad (2)$$

To calculate an implementation, we aim to transform the right-hand side of this equation into the form $exec\ c\ s$ for some code c, and then define $compile\ e = c$. In this case simple calculation suffices, by first introducing a new code constructor that halts execution, and then using equation (1):

```
  do v ← eval e; return (v : s)
=   { define: exec HALT s = return s }
  do v ← eval e; exec HALT (v : s)
=   { specification (1) }
  exec (comp e HALT) s
```

In summary, we have calculated the following definitions for the code datatype, compiler and stack machine. In each case, the definition arose in a natural manner from the desire to satisfy the compiler correctness equations.

```
data Code where
  PUSH  :: Int → Code → Code
  ADD   :: Code → Code
  PRINT :: Code → Code
  HALT  :: Code

compile :: Expr → Code
compile e = comp e HALT

comp :: Expr → Code → Code
comp (Val n)     c = PUSH n c
comp (Add x y) c = comp x (comp y (ADD c))
comp (Print x)   c = comp x (PRINT c)

exec :: Code → Stack → PrintSeq Stack
exec (PUSH n c) s       = exec c (n : s)
exec (ADD c) (m : n : s) = exec c ((n + m) : s)
exec (PRINT c) (n : s)   = do printInt n; exec c (n : s)
exec HALT s             = return s
```

2.5 Semantics of print sequences

Print sequences are a data structure that represents the action of printing a sequence of integers. To realise this behaviour in practice, we define a concrete semantics using the *IO* monad, by means of a function that runs a print sequence:

```
run :: PrintSeq a → IO a
run (Ret v)          = return v
run (PrintInt n ps) = do print n; run ps
```

That is, running a return value simply returns the value, and running a print operation prints the supplied integer and runs the remaining print sequence. For instance, applying *run* to our example print sequence from earlier,

```
three :: PrintSeq ()
three = PrintInt 1 (PrintInt 2 (PrintInt 3 (Ret ())))
```

gives the following output, as expected:

```
1
2
3
```

The function $run :: PrintSeq\ a \to IO\ a$ is an example of a *monad morphism* [Wadler 1995] — a polymorphic function between monads that preserves the monad operations in the sense that the following two equations hold:

$$run\ (return\ v) \;=\; return\ v$$
$$run\ (ps \gg\!= f) \;=\; run\ ps \gg\!= (run \circ f)$$

The *return* and $\gg\!=$ on the left-hand side of these equations are for the *PrintSeq* monad, while on the right-hand side they are for the *IO* monad. In addition, *run* satisfies the following simple property, which expresses that running a singleton print sequence just prints the integer value:

$$run\ (printInt\ n) \;=\; print\ n \tag{3}$$

In fact, *run* is the *unique* monad morphism that satisfies this property, and hence the behaviour of *run* is fully determined by its behaviour on the basic operation *printInt*. It is straightforward to verify all these properties.

2.6 Fusing execution and running

If desired, the *exec* and *run* functions can be fused together to eliminate the intermediate use of print sequences, giving a function $exec' :: Code \to Stack \to IO\ Stack$. The behaviour of $exec'$ can be specified by the equation

$$exec'\ c\ s \;=\; run\ (exec\ c\ s) \tag{4}$$

from which a definition can be calculated by induction on the code *c*. The base case, for $c = HALT$, makes use of the fact that *run* is a monad morphism:

```
  exec' HALT s
=   { specification (4) }
  run (exec HALT s)
=   { definition of exec }
  run (return s)
=   { run is a monad morphism }
  return s
```

In turn, the inductive case for printing also uses the fact that *run* preserves printing, as captured by property (3):

```
  exec' (PRINT c) (n : s)
=   { specification (4) }
  run (exec (PRINT c) (n : s))
=   { definition of exec }
  run (do printInt n; exec c (n : s))
=   { run is a monad morphism }
  do run (printInt n); run (exec c (n : s))
=   { property (3) }
  do print n; run (exec c (n : s))
=   { induction hypothesis }
  do print n; exec' c (n : s)
```

The remaining cases, for pushing and adding, follow by straightforward induction, and do not require properties of *run*. In conclusion, we obtain the following definition:

```
exec′ :: Code → Stack → IO Stack
exec′ (PUSH n c) s       = exec′ c (n : s)
exec′ (ADD c) (m : n : s) = exec′ c ((n + m) : s)
exec′ (PRINT c) (n : s)   = do print n; exec′ c (n : s)
exec′ HALT s             = return s
```

It is also possible to calculate the *exec′* directly, rather than first calculating *exec* and then fusing with *run*.

2.7 Alternative semantics

Since print sequences can be fused away, and printing ultimately requires a side effect, why are they useful?

The key benefit is that using an algebraic approach that separates syntax and semantics gives *extra flexibility*. In particular, we have a syntactic approach to printing via the *PrintSeq* monad, which we give a semantics in terms of the *IO* monad via the *run* function. This separation of concerns means that we have the flexibility to consider other semantics for print sequences without the need to redo the compiler calculation, by simply modifying *run*. This is possible because the calculation does not depend on this function.

For example, suppose that we wished to allow the user to single step through the output, thereby providing a simple form of 'debug mode'. This could be achieved by defining a new semantics for print sequences that waits for the user to hit enter after each print operation:

```
runStep :: PrintSeq a → IO a
runStep (Ret v)          = return v
runStep (PrintInt n ps) = do print n; getLine; runStep ps
```

Alternatively, we may wish to filter out undesired outputs, such as 'launching missiles' [Harris et al. 2005]. For example, for our simple language we could define a new semantics that filters out negative numbers as follows:

```
runPos :: PrintSeq a → IO a
runPos (Ret v)                          = return v
runPos (PrintInt n ps) | n ⩾ 0          = do print n; runPos ps
                       | otherwise      = runPos ps
```

We also have the flexibility to interpret print sequences in other settings than the *IO* monad. For example, we could collect the outputs in a list rather than printing them,

```
runList :: PrintSeq a → [Int]
runList (Ret v)          = [ ]
runList (PrintInt n ps) = n : runList ps
```

and it is possible to reverse the order of the elements in a print sequence, using the following definition:

```
runRev :: PrintSeq a → PrintSeq a
runRev (Ret v)          = return v
runRev (PrintInt n ps) = do v ← runRev ps
                            PrintInt n (Ret v)
```

Finally, we could also interpret print sequences using more than one monad, such as using *Maybe* to define a semantics that only prints if all the outputs are non-negative:

```
runMaybe :: PrintSeq a → Maybe (IO a)
runMaybe (Ret v) = Just (return v)
runMaybe (PrintInt n p)
  | n ⩾ 0      = do a ← runMaybe p
                    return (do print n; a)
  | otherwise = Nothing
```

In general, we have the freedom to interpret print sequences in any way that we choose, which gives great flexibility.

3 A language with state

For our next example, we consider an expression language with primitives for getting and setting an integer state:

```
data Expr = Val Int | Add Expr Expr | Get | Set Expr
```

For example, *Set* (*Add Get* (*Val* 1)) increments the state by getting the current value, adding one to it, and setting the state to the resulting value. This behaviour could be realised by defining a semantics *eval* :: *Expr* → *ST Int* that uses a suitable state monad *ST* to manage the effect of manipulating an integer state. As with the previous example, however, we prefer to take a more abstract, algebraic approach and define the semantics using a monad of *state sequences*.

3.1 State sequences

The type of state sequences, *StateSeq a*, represents sequences of computations that return a final value of type *a*, and may use operations that get and set an integer state:

```
data StateSeq a where
  Ret    :: a → StateSeq a
  GetInt :: (Int → StateSeq a) → StateSeq a
  SetInt :: Int → StateSeq a → StateSeq a
```

Informally, *Ret v* returns the value *v*, while *GetInt c* feeds the current state into the continuation *c*, which takes an integer and returns a state sequence that captures what happens next, while *SetInt n ss* sets the current state to *n* and then behaves as the state sequence *ss*. For instance, our increment example above can be realised by the following state sequence:

```
inc :: StateSeq ()
inc = GetInt (λn → SetInt (n + 1) (Ret ()))
```

That is, we first get the current state and call it *n*, then set the state to *n* + 1, and finally return the void result (). Note that *inc* does not actually change a state, but rather builds a data structure that represents the action of doing so.

State sequences form a monad, and it is straightforward to show that the monad laws are satisfied:

```
instance Monad StateSeq where
  return :: a → StateSeq a
```

```
return = Ret

(>>=) :: StateSeq a → (a → StateSeq b) → StateSeq b
Ret v      >>= f = f v
GetInt c   >>= f = GetInt (λn → c n >>= f)
SetInt n ss >>= f = SetInt n (ss >>= f)
```

3.2 Compiler calculation

The starting point for calculating a compiler is an evaluation semantics for expressions in terms of state sequences:

```
eval :: Expr → StateSeq Int
eval (Val n)   = return n
eval (Add x y) = do n ← eval x; m ← eval y; return (n + m)
eval Get       = getInt
eval (Set x)   = do n ← eval x; setInt n; return n
```

The two helper functions used above construct singleton state sequences that get and set the integer state:

```
getInt :: StateSeq Int
getInt = GetInt Ret

setInt :: Int → StateSeq ()
setInt n = SetInt n (Ret ())
```

In a similar manner to Section 2, our goal now is to calculate a compilation function *comp*::*Expr* → *Code* → *Code* and an execution function *exec*::*Code* → *Stack* → *StateSeq Stack* that satisfy the following correctness equation:

$$exec\ (comp\ e\ c)\ s \quad = \quad \textbf{do}\ v \leftarrow eval\ e; exec\ c\ (v : s) \qquad (5)$$

As previously, the *Stack* type is simply a list of integers, while the *Code* type will be derived during the calculation process. The calculation itself proceeds by induction on the expression *e*, with the aim of transforming the right-hand side of the equation into the form *exec c′ s* for some code *c′*, which then allows us to define *comp e c* = *c′*.

The cases for *Val n* and *Add x y* proceed in the same way as before, and result in the same definitions. The case for *Get* follows by simply applying the definition of *eval*, and then introducing a new code constructor *GET* that pushes the current state onto the stack, in order to transform the term being manipulated into the required form:

```
  do v ← eval Get; exec c (v : s)
=   { definition of eval }
  do v ← getInt; exec c (v : s)
=   { define: exec (GET c) s = do n ← getInt; exec c (n : s) }
  exec (GET c) s
```

And finally, the case for *Set x* is similar to the case for *Print x*, except that this time around we are storing an integer in a state rather than printing an integer:

```
  do v ← eval (Set x); exec c (v : s)
=   { definition of eval }
  do v ← do { n ← eval x; setInt n; return n }; exec c (v : s)
=   { monad laws }
  do n ← eval x; setInt n; exec c (n : s)
=   { define: exec (SET c) (n : s) = do setInt n; exec c (n : s) }
  do n ← eval x; exec (SET c) (n : s)
=   { induction hypothesis for x }
  exec (comp x (SET c)) s
```

The top-level function *compile* :: *Expr* → *Code* is calculated in the same way as previously. In conclusion, we obtain the following definitions for the code datatype, compiler and stack machine, which are guaranteed to be correct by the manner in which they are constructed:

```
data Code where
  PUSH :: Int → Code → Code
  ADD  :: Code → Code
  GET  :: Code → Code
  SET  :: Code → Code
  HALT :: Code

compile :: Expr → Code
compile e = comp e HALT

comp :: Expr → Code → Code
comp (Val n)   c = PUSH n c
comp (Add x y) c = comp x (comp y (ADD c))
comp Get       c = GET c
comp (Set x)   c = comp x (SET c)

exec :: Code → Stack → StateSeq Stack
exec (PUSH n c) s       = exec c (n : s)
exec (ADD c) (n : m : s) = exec c ((m + n) : s)
exec (GET c) s          = do n ← getInt; exec c (n : s)
exec (SET c) (n : s)    = do setInt n; exec c (n : s)
exec HALT s             = return s
```

3.3 Semantics of state sequences

To realise the behaviour of state sequences in practice, we can define a concrete semantics using Haskell's state monad [Wadler 1992], in which the type *State s a* represents a state transformer of type *s* → (*a*, *s*), where *s* is the type of states and *a* is the type of result values. In our case, the state is simply an integer, and we define a semantics as follows:

```
run :: StateSeq a → State Int a
run (Ret v)      = return v
run (GetInt c)   = do n ← get; run (c n)
run (SetInt n ss) = do put n; run ss
```

In this definition, the library primitive *get* :: *State s s* returns the current value of the state, while *put* :: *s* → *State s* () sets the state to the given value. Note that in the case of *GetInt*, the continuation *c* is applied to the current state to give the remaining state sequence to be interpreted.

A state transformer can be applied to an initial state using $runState :: State\ s\ a \to s \to (a, s)$. For instance, running the increment example with zero as the initial state,

$runState\ (run\ inc)\ 0$

gives the pair $((), 1)$ comprising the void result value $()$ and the incremented state one, as expected. The semantic function *run* is a monad morphism, and also preserves the basic state primitives in the sense that we have:

$$run\ getInt \;=\; get \tag{6}$$

$$run\ (setInt\ n) \;=\; put\ n \tag{7}$$

If desired, these properties can be used to fuse together *exec* and *run*, resulting in the following definition:

```
exec' :: Code → Stack → State Int Stack
exec' (PUSH n c) s       = exec' c (n : s)
exec' (ADD c) (n : m : s) = exec' c ((m + n) : s)
exec' (GET c) s          = do n ← get; exec' c (n : s)
exec' (SET c) (n : s)    = do put n; exec' c (n : s)
exec' HALT s             = return s
```

3.4 Alternative semantics

Using an algebraic approach to stateful computation that separates syntax from semantics means that we have the flexibility to consider other semantics for state sequences, without the need to redo the compiler calculation.

For example, rather than simulating state using the *State* monad, we could use a real mutable reference, by means of Haskell's *IORef* mechanism [Peyton Jones 2001]. In particular, we can define a new semantics for state sequences that takes a reference to an integer as an extra parameter:

```
runRef :: StateSeq a → IORef Int → IO a
runRef (Ret v)      r = return v
runRef (GetInt c)   r = do n ← readIORef r; runRef (c n) r
runRef (SetInt n ss) r = do writeIORef r n; runRef ss r
```

Using this function, the following sequence of actions creates a new reference *r* that is initialised to zero, increments the reference, and then reads and prints the resulting value:

```
do r ← newIORef 0
   runRef inc r
   n ← readIORef r
   print n
```

In turn, if we wished to also track how the state changes, we could further refine the semantics to log the operation that is being applied at each step in a state sequence:

```
runLog :: Show a ⇒ StateSeq a → IORef Int → IO a
runLog (Ret v)      r = do log "Ret" v; return v
runLog (GetInt c)   r = do n ← readIORef r
                           log "Get" n; runLog (c n) r
runLog (SetInt n ss) r = do writeIORef r n
                           log "Set" n; runLog ss r
```

The helper function used above that logs an operation and its argument value is defined as follows:

```
log :: Show a ⇒ String → a → IO ()
log op v = putStrLn (op ++ "  " ++ show v)
```

For example, executing the action sequence

```
do r ← newIORef 0
   runLog inc r
```

gives the following log:

```
Get 0
Set 1
Ret ()
```

More generally, we can interpret state sequences in any way that we choose, without the need to modify the compiler for the language, as the calculation of the compiler does not depend on the semantics of state sequences.

4 A language with non-determinism

For our final example, we consider a language with simple primitives for expressing non-determinism, i.e. the ability to return multiple possible result values, including none:

```
data Expr = Val Int | Add Expr Expr | Fail | Or Expr Expr
```

Intuitively, *Fail* represents failure to return a result value, while $Or\ x\ y$ makes a non-deterministic choice between evaluating the expressions x and y. For example, $Add\ (Val\ 1)\ Fail$ fails to evaluate, because there is no second argument for the addition, while $Add\ (Val\ 1)\ (Or\ (Val\ 2)\ (Val\ 3))$ evaluates to either three or four, depending on which choice is made for the second argument. These behaviours could be realised by defining a function $eval :: Expr \to [Int]$ that uses the list monad to manage the possibility of returning multiple results. Here we take a more flexible, algebraic approach and define the semantics using a monad of *choice trees*.

4.1 Choice trees

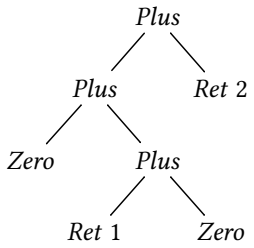
The type of choice trees, $ChoiceTree\ a$, represents trees of computations that may return a successful value of type a, and can also use operations that represent failure and choice:

```
data ChoiceTree a where
  Ret  :: a → ChoiceTree a
  Zero :: ChoiceTree a
  Plus :: ChoiceTree a → ChoiceTree a → ChoiceTree a
```

Informally, $Ret\ v$ succeeds with the value v, while *Zero* fails without returning a value, and $Plus\ x\ y$ makes a non-deterministic choice between the two computations x and y. For instance, the latter addition example above can be realised by the following choice tree:

```
ctree :: ChoiceTree Int
ctree = Plus (Ret 3) (Ret 4)
```

That is, we have the choice of either returning three or four. It can also be useful to view choice trees in a pictorial manner, as in the following larger example:



Note that such trees do not perform non-deterministic computation, but are rather data structures that represents the action of doing so, which can be interpreted in different ways depending on the desired semantics.

Choice trees naturally form a monad under the following definitions, and satisfy the monad laws:

```
instance Monad ChoiceTree where
  return :: a → ChoiceTree a
  return = Ret
  (≫=) :: ChoiceTree a → (a → ChoiceTree b) → ChoiceTree b
  Ret v    ≫= f = f v
  Zero     ≫= f = Zero
  Plus x y ≫= f = Plus (x ≫= f) (y ≫= f)
```

The final two equations above state that, by definition, *Zero* is the left zero for the ≫= operator on choice trees, and that ≫= left distributes over *Plus*. In combination with the fact that choice trees form a monoid under the *run* semantics defined later in this section, these additional laws express that choice trees form a monad with a zero and a plus.

4.2 Compiler calculation

Starting from an evaluation function that defines the semantics of expressions in terms of choice trees,

```
eval :: Expr → ChoiceTree Int
eval (Val n)   = return n
eval (Add x y) = do n ← eval x; m ← eval y; return (n + m)
eval Fail      = Zero
eval (Or x y)  = Plus (eval x) (eval y)
```

we aim to calculate functions *comp* :: *Expr* → *Code* → *Code* and *exec* :: *Code* → *Stack* → *ChoiceTree Stack* that satisfy the following correctness equation:

$$exec\ (comp\ e\ c)\ s \;=\; \mathbf{do}\ v \leftarrow eval\ e; exec\ c\ (v : s) \qquad (8)$$

As before, a *Stack* is simply a list of integers, and we also seek to derive the *Code* type. We focus here on the new cases concerning non-determinism, with the other cases proceeding in the same manner to the previous examples.

The case for *Fail* begins by applying the definition of *eval*, then uses the fact that *Zero* is the left zero for the ≫= operator, and concludes by introducing a new code constructor *FAIL* that simply discards the stack and returns *Zero*, in order to transform the term into the required form:

```
  do v ← eval Fail; exec c (v : s)
=   { definition of eval }
  do v ← Zero; exec c (v : s)
=   { left zero law }
  Zero
=   { define: exec FAIL s = Zero }
  exec FAIL s
```

In turn, the case for *Or x y* exploits the fact that ≫= left distributes over *Or*, which allows the induction hypotheses to be applied, after which we introduce a new code constructor *OR* to obtain a term of the required form:

```
  do v ← eval (Or x y); exec c (v : s)
=   { definition of eval }
  do v ← Plus (eval x) (eval y); exec c (v : s)
=   { left distributivity law }
  Plus (do v ← eval x; exec c (v : s))
       (do v ← eval y; exec c (v : s))
=   { induction hypotheses for x and y }
  Plus (exec (comp x c) s) (exec (comp y c) s)
=   { define: exec (OR c d) s = Plus (exec c s) (exec d s) }
  exec (OR (comp x c) (comp y c)) s
```

The top-level function *compile* is the same as previously. In conclusion, we obtain the following definitions:

```
data Code where
  PUSH :: Int → Code → Code
  ADD  :: Code → Code
  FAIL :: Code
  OR   :: Code → Code → Code
  HALT :: Code

compile :: Expr → Code
compile e = comp e HALT

comp :: Expr → Code → Code
comp (Val n)   c = PUSH n c
comp (Add x y) c = comp x (comp y (ADD c))
comp Fail      c = FAIL
comp (Or x y)  c = OR (comp x c) (comp y c)

exec :: Code → Stack → ChoiceTree Stack
exec (PUSH n c) s       = exec c (n : s)
exec (ADD c) (m : n : s) = exec c ((n + m) : s)
exec FAIL s             = Zero
exec (OR c d) s         = Plus (exec c s) (exec d s)
exec HALT s             = return s
```

4.3 Semantics of choice trees

Using an algebraic approach to non-determinism by means of choice trees gives us the flexibility to interpret such trees in any way that we please. In this section, we consider a number of possible semantics for choice trees.

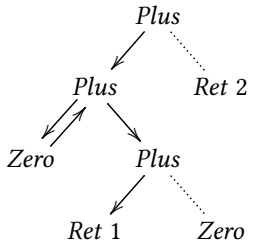
First of all, the standard semantics that collapses a choice tree down to a list of possible results is defined as follows:

```
run :: ChoiceTree a → [a]
run (Ret v)    = [v]
run Zero       = []
run (Plus x y) = run x ++ run y
```

For instance, applying *run* to the example tree pictured earlier gives the list $[1, 2]$ comprising two possible results. If desired, we could also define a semantics that returns just the first result, if there is one, using the *Maybe* monad:

```
runMaybe :: ChoiceTree a → Maybe a
runMaybe (Ret v)    = Just v
runMaybe Zero       = Nothing
runMaybe (Plus x y) = case runMaybe x of
                        Just v  → Just v
                        Nothing → runMaybe y
```

For example, *runMaybe* traverses the example tree in the manner illustrated below, with failure in the leftmost branch eventually leading to the successful result one:



However, we may prefer a more refined semantics that uses a stack to allow choice trees to be traversed in a more efficient manner. In particular, taking a stack of trees of type $[ChoiceTree\ a]$ as an extra argument allows the above semantics to be defined tail recursively:

```
runTail :: ChoiceTree a → [ChoiceTree a] → Maybe a
runTail (Ret v) _    = Just v
runTail Zero []      = Nothing
runTail Zero (y : ys) = runTail y ys
runTail (Plus x y) ys = runTail x (y : ys)
```

That is, for *Zero* we simply remove and run the top tree on the stack, if there is one. Conversely, for *Plus x y* we push the tree *y* onto the stack, in case running the tree *x* fails. The function *runTail* forms an abstract machine for evaluating choice trees, and can also be calculated from *runMaybe* using the techniques in [Hutton and Bahr 2016]. The original *Maybe* semantics can be recovered by passing in the empty stack, i.e. defining $runMaybe\ x = runTail\ x\ []$.

Alternatively, rather than simulating failure using the *Maybe* monad, we could use a real exception, by means of Haskell's *Exception* mechanism [Peyton Jones 2001]. For example, we can define a semantics using a new exception *ZERO* that can be thrown and caught in the *IO* monad:

```
data MyException = ZERO deriving (Show, Exception)

runExc :: ChoiceTree a → IO a
runExc (Ret v)    = return v
runExc Zero       = throwIO ZERO
runExc (Plus x y) = catchException (runExc x)
                                   (λZERO → runExc y)
```

This approach can also further improve efficiency, as it is using built-in language features to immediately jump to the second argument of *Plus* if the first argument fails.

Finally, we could also implement non-determinism itself for real, by randomly choosing which argument of each *Plus* operator in a choice tree to run, using Haskell's random number feature. This also takes place within the *IO* monad as the result is truly non-deterministic.

```
runRand :: ChoiceTree a → IO (Maybe a)
runRand (Ret v)    = return (Just v)
runRand Zero       = return Nothing
runRand (Plus x y) = do n :: Int ← randomRIO (1, 2)
                        case n of
                          1 → runRand x
                          2 → runRand y
```

For example, applying *runRand* to the example tree pictured above has three possible outcomes, *Nothing*, *Just* 1 or *Just* 2, depending on which path through the tree is chosen. Two further semantics, using parallelism and asynchronous concurrency, are provided in the supplementary material.

5 Generalisation

As noted in the introduction, by design we have used simple, explicit definitions where possible. In this section we take a step back and consider how the ideas could be presented in a more general or abstract manner, by modelling effect types using type classes and free monads.

5.1 Type classes

For each of the effects we have considered, the compiler calculations for the *Val* and *Add* cases were the same. The reason is that the calculations only use the monad *laws* for the underlying effect type, rather than the actual definition of the type. In a similar manner, the calculations for the effectful operations only use general monadic laws. For example, in the *Fail* and *Or* cases for non-determinism, we used the fact that choice trees form a monad with a zero and a plus.

This presents an opportunity for abstraction. Take printing, for example. Because the calculations do not depend on the specifics of the *PrintSeq* monad, we could use Haskell's type class system to capture the class of monads which support the operation of printing an integer:

```
class Monad m ⇒ MonadPrint m where
  printInt :: Int → m ()
```

Using this declaration, the function *eval* remains unchanged syntactically, but now has a more general type:

```
eval :: MonadPrint m ⇒ Expr → m Int
eval (Val n)     = return n
eval (Add x y) = do n ← eval x; m ← eval y; return (n + m)
eval (Print x)   = do n ← eval x; printInt n; return n
```

In turn, if the type for *exec* is similarly generalised,

```
exec :: MonadPrint m ⇒ Code → Stack → m Stack
```

then our compiler specification and calculation for the printing language in Section 2 can be reframed without modification using the general *MonadPrint* class, and results in the same definitions. We are then free to instantiate the *MonadPrint* class with different monads as and when required, such as using the *PrintSeq* or *IO* monads:

```
instance MonadPrint PrintSeq where
  printInt :: Int → PrintSeq ()
  printInt n = PrintInt n (Ret ())
instance MonadPrint IO where
  printInt :: Int → IO ()
  printInt n = print n
```

Taking this approach conveniently side-steps the need for a semantic function *run* to interpret the *PrintSeq* type. On the other hand, we lose some flexibility as we can no longer manipulate the *structure* of the effects, only the basic operations. For example, we cannot give an instance that reverses the printing order. Furthermore, Haskell permits only one type class instance for each type, so we lose the ability to provide multiple differing interpretations so easily.

For state, it is also easy to declare a class that abstracts the basic operations for manipulating an integer state:

```
class Monad m ⇒ MonadState m where
  getInt :: m Int
  setInt :: Int → m ()
```

However, we need to be more careful with non-determinism, because we do make use of the behaviour of $\gg=$ during the calculations. Once again, the starting point is to declare a class that captures the basic operations required:

```
class Monad m ⇒ MonadChoice m where
  zero :: m a
  plus :: m a → m a → m a
```

This time, the class declaration alone is not enough to complete our compiler calculations. In particular, the case for *Fail* uses a left-zero law, and the case for *Or* uses left distributivity. Hence, we need to require that any instance also satisfies these two additional monadic laws:

$$zero \gg= f \;=\; zero$$
$$(plus\ x\ y) \gg= f \;=\; plus\ (x \gg= f)\ (y \gg= f)$$

The *MonadChoice* class with these two additional laws is essentially the same (modulo renaming) as the *MonadPlus* class in Haskell, except that the latter also requires values of type $m\ a$ to form a monoid under the two operations of the class. While choice trees themselves do not form a monoid because they are just syntax, as noted in Section 4 they do form a monoid under the semantic function *run* that interprets such trees in the list monad. That is, if we define a congruence relation $\cong$ on choice trees by $x \cong y$ iff $run\ x = run\ y$, then we have the following monoid laws:

$$Or\ Fail\ y \;\cong\; y$$
$$Or\ x\ Fail \;\cong\; x$$
$$Or\ (Or\ x\ y)\ z \;\cong\; Or\ x\ (Or\ y\ z)$$

While the above type class generalisations do allow us to present the compiler calculations in a more abstract manner, we chose to introduce the classes here rather than using them throughout the article to keep the presentation simple. Taking a more abstract approach would also have delayed in the introduction of the effect types *PrintSeq*, *StateSeq* and *ChoiceTree*, which we found to be invaluable in understanding and explaining how to calculate compilers for languages with effects. Furthermore, such a generalisation also limits the ways in which effects can be interpreted, as illustrated with the reverse printing example.

5.2 Free monads

The form of effect types that we have considered naturally arise as *free monads* [Lüth and Ghani 2002]. The starting point for this construction is to capture the signatures of the operations of an effect type by a *signature functor*. For example, if we recall the type of state sequences,

```
data StateSeq a where
  Ret    :: a → StateSeq a
  GetInt :: (Int → StateSeq a) → StateSeq a
  SetInt :: Int → StateSeq a → StateSeq a
```

then the operations that get and set an integer state can be captured by the following signature functor:

```
data StateSig a where
  GetSig :: (Int → a) → StateSig a
  SetSig :: Int → a → StateSig a
```

Given a signature functor f, the free monad on f at type a comprises syntactic terms constructed using operations from f and variables of type a, and is defined as follows:

data *Free f a* = *Var a* | *Con* (*f* (*Free f a*))

It is straightforward to show that *Free f* is indeed a monad, with $\gg=$ providing a notion of substitution for variables.

Using these ideas, the state sequence type *StateSeq a* can be shown to be isomorphic to the free monad *Free StateSig a*. In a similar manner, the types of print sequences *PrintSeq a* and choice trees *ChoiceTree a* are isomorphic to free monads on their underlying signature functors.

The utility of viewing effect types as free monads becomes apparent if we wish to combine multiple effects together. For example, suppose we wanted to define an effect type that captures the operations of both *PrintSeq* and *StateSeq*. One approach is to manually combine the constructors:

```
data PrintStateSeq a where
  Ret      :: a → PrintStateSeq a
  PrintInt :: Int → PrintStateSeq a → PrintStateSeq a
  GetInt   :: (Int → PrintStateSeq a) → PrintStateSeq a
  SetInt   :: Int → PrintStateSeq a → PrintStateSeq a
```

Another, more modular, approach is to consider the signature functors of the two effect types. In particular, if we additionally define the signature functor for printing,

```
data PrintSig a where
  PrintSig :: Int → a → PrintSig a
```

then we can define the combined type by simply taking the coproduct of *PrintSig* and *StateSig* using Haskell's *Sum* constructor for functors, and then considering the free monad of the resulting signature functor:

```
type PrintStateSeq a = Free (Sum PrintSig StateSig) a
```

To date, however, the approach to compiler calculation on which this article is based [Bahr and Hutton 2015] hasn't considered modular approaches to syntax and semantics. It would be interesting to explore the idea of calculating compilers using effect types modelled as free monads.

6 Further work

In addition to the topics discussed in the previous section, possible directions for further work include attempting to calculate compilers in a generic manner for arbitrary effects, supporting first-class handlers that allow dynamic control over how effects are interpreted, exploiting the use of recursion operators and their properties rather than using explicit recursion and induction, and exploring how the approach can be applied to more sophisticated languages.

Acknowledgements

We thank the referees for their useful comments and suggestions. This work was partially funded by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/Y010744/1, *Semantics-Directed Compiler Construction: From Formal Semantics to Certified Compilers*.

References

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. *From Interpreter to Compiler and Virtual Machine: A Functional Derivation.* Technical Report RS-03-14. BRICS, Department of Computer Science, University of Aarhus.

Patrick Bahr and Graham Hutton. 2015. Calculating Correct Compilers. *Journal of Functional Programming* 25 (2015).

Patrick Bahr and Graham Hutton. 2020. Calculating Correct Compilers II: Return of the Register Machines. *Journal of Functional Programming* 30 (2020).

Patrick Bahr and Graham Hutton. 2023. Calculating Compilers for Concurrency. *Proceedings of the ACM on Programming Languages* 7, ICFP, Article 213 (2023).

Conal Elliott. 2017. Compiling to Categories. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 27 (2017).

Jeremy Gibbons. 2022. Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity. *The Art, Science, and Engineering of Programming* 6, 2, Article 7 (2022).

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

Graham Hutton. 2023. Programming Language Semantics: It's Easy As 1,2,3. *Journal of Functional Programming* 33 (2023).

Graham Hutton and Patrick Bahr. 2016. Cutting Out Continuations. In *A List of Successes That Can Change the World.*

Christoph Lüth and Neil Ghani. 2002. Composing Monads Using Coproducts. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming.*

Erik Meijer. 1992. *Calculating Compilers.* Ph. D. Dissertation. Katholieke Universiteit Nijmegen.

Simon Peyton Jones. 2001. Tackling The Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. In *Engineering Theories of Software Construction.*

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. In *Proceedings of the 31st Conference on the Mathematical Foundations of Programming Semantics.*

Wouter Swierstra and Thorsten Altenkirch. 2007. Beauty in the Beast. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop.*

Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming.*

Mitchell Wand. 1982. Deriving Target Code as a Representation of Continuation Semantics. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982).

Received 2024-06-03; accepted 2024-07-05