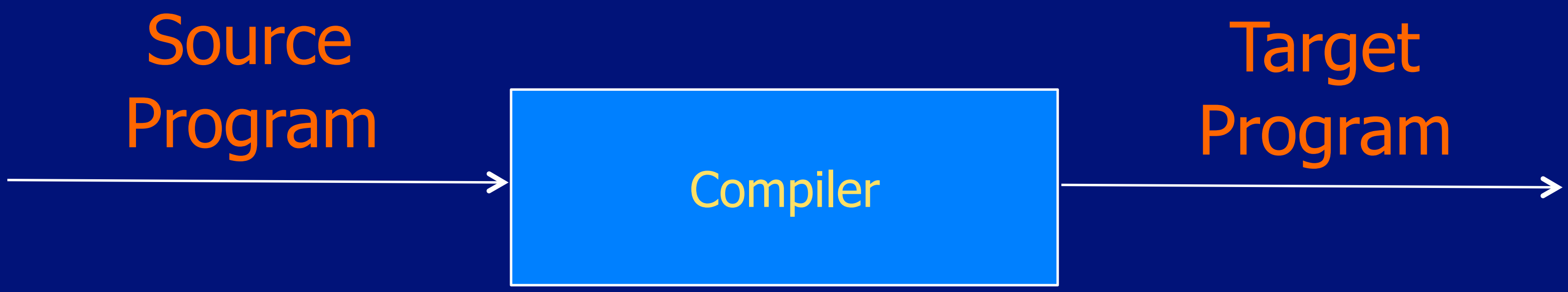
TOWARDS MODULAR COMPILERS FOR EFFECTS

Laurence E. Day
Functional Programming Laboratory
University of Nottingham

What Are Compilers?

Programs translating from high-level languages to low-level sequences of instructions.



Example – in Haskell

What Is Modularity?

The separation of individual features.

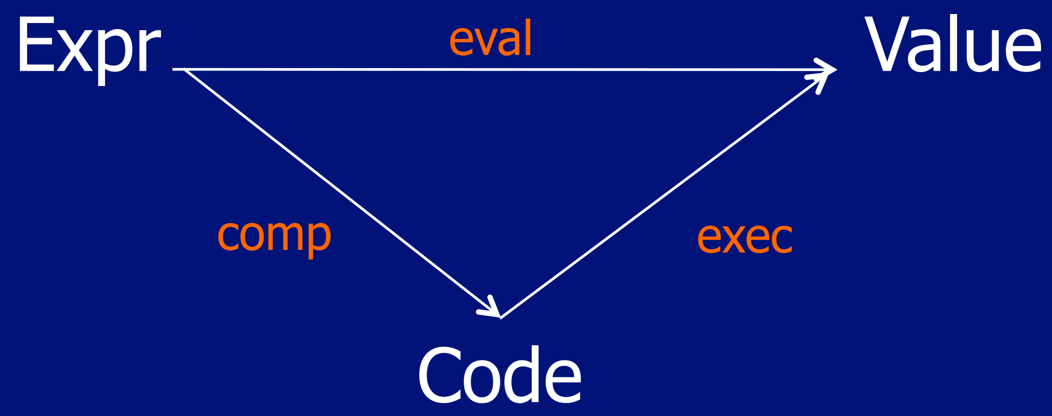
Compilers are usually factorised into stages:

Lexing, parsing, code generation…

Can they also be factorised by effects?

Exceptions, mutable state, I/O…

Modular Correctness



Can we combine correctness proofs
for each individual feature?

Example

Syntax:

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

Semantics:

```
eval               :: Expr → Int

eval (Val n)       = n
eval (Add x y)     = eval x + eval y
```

Compiler

Syntax:

```
type Code      = [Op]
data Op        = PUSH Int | ADD
```

Semantics:

```
comp              :: Expr → Code

comp (Val n)      = [PUSH n]
comp (Add x y)    = comp x ++ comp y
                              ++ [ADD]
```

Adding an Effect - Exceptions

```
data Expr = ... | Throw | Catch Expr Expr

eval              :: Expr → Maybe Int

eval (Val n)      = return n
eval (Add x y)    = do n ← eval x
                       m ← eval y
                       return (n + m)
eval (Throw)      = mzero
eval (Catch x h)  = eval x `mplus` eval h
```

Compiling with Catches

```
data Op = ... | THROW | MARK Code | UNMARK

comp                    :: Expr → Code
...
comp (Throw)            = [THROW]
comp (Catch x h)        = [MARK (comp h)] ++
                          [comp x] ++ [UNMARK]
```

Modularity In Haskell

Haskell syntax isn't modular!

Extending syntax $\rightarrow$ Editing functions

Modular syntax:

Data Types à La Carte (Swierstra)

Modular semantics:

Monad transformers and Modular Interpreters (Liang, Hudak and Jones)

Signatures

Consider the following two datatypes:

```
data Arith e    =  Val Int  |  Add e e
data Except e   =  Throw    |  Catch e e
```

Reveals the non-recursive nature of Expr.

Signatures as Functors

```
instance Functor Arith where
  fmap :: (a → b) → Arith a → Arith b
  fmap f (Val n)          = Val n
  fmap f (Add x y)        = Add (f x) (f y)
```

```
data Fix f = In (f (Fix f))
```

Induced recursive datatype for any functor f

Modular Syntax

```
ex1 :: Fix Arith
ex1 = val 27 `add` val 15

ex2 :: Fix Except
ex2 = throw `catch` throw

ex3 :: Fix (Arith ⊕ Except)
ex3 = throw `catch` (val 1336 `add` val 1)
```

Folding Functors for Fixpoints

```
fold :: Functor f ⇒ (f a → a) → Fix f → a
fold f (In t) = f (fmap (fold f) t)
```

The fold operator accepts an f-algebra (f a → a) as a directive for recursively processing expressions written using Fix.

Piecemeal Semantics

To define a semantics of the type

eval :: Fix f → m Value

using fold, we need an algebra:

evalAlg :: f (m Value) → m Value

Abstract this pattern out into a typeclass Eval

Modular Semantics (Arith)

```
instance Monad m ⇒ Eval Arith m where
  evalAlg :: Arith (m Value) → m Value
  evalAlg (Val n)       =     return n
  evalAlg (Add x y)     = do  n ← x
                              m ← y
                              return (n + m)
```

Semantics of Arith totally separate from those of Except (not even referenced in the code above!)

Modular Semantics (Except)

```
instance ErrorMonad m ⇒ Eval Except m where
   evalAlg :: Except (m Value) → m Value
   evalAlg (Throw)          = throw
   evalAlg (Catch x h)      = x `catch` h
```

Modular Compiler (Arith)

```
instance Comp Arith where
  compAlg :: Arith (Code → Code) → Code → Code
  compAlg (Val n)       = pushc n
  compAlg (Add x y)     = x . y . addc
```

Modular Compiler (Except)

```
instance Comp Except where
   compAlg :: Except (Code → Code) → Code → Code
   compAlg (Throw)       = throwc
   compAlg (Catch x h) = \c →
                          h c `markc` x (unmarkc c)
```

Modular Machine (Arith)

```
instance Monad m ⇒ Exec ARITH where
  execAlg :: ARITH (StackTrans m ()) → StackTrans m ()
  execAlg (PUSH n st)  = pushval n >> st
  execAlg (ADD st)     = addc      >> st
```

Modular Machine (Except)

```
instance ErrorMonad m ⇒ Exec EXCEPT where
  execAlg :: EXCEPT (StackTrans m ()) → StackTrans m ()
  execAlg (THROW _)          = unwind
  execAlg (MARK h st)        = pushcode h >> st
  execAlg (UNMARK st)        = unmark >> st
```

What's The Point?

Can define evaluators, compilers etc modularly:

Each effect handled separately!

Individual features can be proved correct:

Full proof is 'Semantic Lego' (Espinosa)

Modular syntax allows flexible languages:

Can describe what features are needed

The Road Ahead

Modularise semantics of virtual machine:

Some issues still to be resolved

Examine varying semantics of multiple effects:

Exceptions + State → Local / Global State

Moving on to modular correctness proofs:

A PhD will hopefully fall out at the end