Getting Into An Exceptional State

Laurence E. Day

Functional Programming Laboratory Away Day

Buxton, England

July 8, 2011

Contents of Forthcoming Rant

- Mutable state as a computational effect
  - associated and necessary operations

- Implementation of state in our modular compiler
  - at least, how we'd like to do it

- Problems with fixing a target language:
  - sadly, causes a lot of unnecessary hassle

Modular Compilation: The Defence

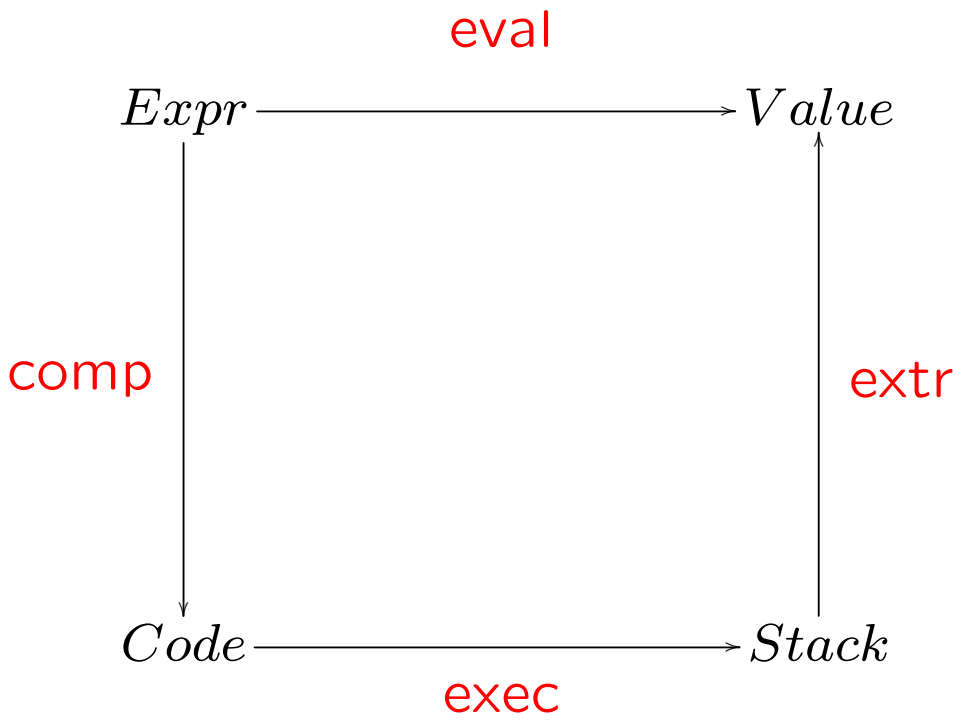
So far as the literature is concerned...

- A modular syntax, semantics, compiler and virtual machine exists supporting arithmetic and exceptions

- This isn't our only foray into compiling effects!

- Mixing things up with state proves to be both simple to conceptualise and a **nightmare** to implement

A Thousand Words



Modularising Languages

```
data (f :+: g) e = Inl (f e) | Inr (g e)
data Fix f = In (f (Fix f)) // f :: Functor
```

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

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

Fix (Arith :+: Except) $\simeq$ Expr $\simeq$ Fix (Except :+: Arith)

The Idea In A Nutshell

```
class Functor f => Comp f where
  compAlg ::  f (Endo (Fix g)) -> Endo (Fix g)


instance Comp EffectFunctor where
  compAlg (Constructor1) = ...
  compAlg (Constructor2) = ...


comp ::  Functor f => Fix f -> ~~Code~~ Fix g
comp = fold compAlg emptyAccumulator
```

Engineering Modular Effects

Monad transformers and effect typeclasses!

```
class MonadTrans t where
   lift ::  Monad m => m a -> t m a
```

```
class Monad m => EffectMonad m where
   operation1 = ...
   operation2 = ...
```

Effect operations lifted through transformer 'layers'.

Lifting Effects: An Example

```
class Monad m => ErrMonad m where
  throw ::  m a
  catch ::  m a -> m a -> m a
```

```
instance ErrMonad m => ErrMonad (EffectT m) where
  throw       = EffectT $ ...
  x `catch` h = EffectT $ ...
```

Isn't this all child's play so far?

State as an Effect

Can either set a value or retrieve one for usage.

```
data State e = Get e | Set StType e
```

```
data STATE e = GET e  | SET StType e
             | SAVE e | RESTORE e
```

Save and Restore commands for the virtual machine

The StateT(ransformer)

```
newtype StateT s m a = StateT {run::  s -> m (a, s)}

instance Monad m => Monad (StateT s m) where
  return a          = StateT $
                          λs -> return (a, s)
  (StateT x) >>= f = StateT $ λs -> do
                          (a, t) <- x s
                          (b, u) <- run (f a) t
                          return (b, u)

instance Monad m => StateMonad (StateT s m) where
  update f ::  StateT $ λs -> return (s, f s)
```

In An Ideal World...

The compilation of state would be...

```
instance StateMonad m => Comp State m where
  compAlg (Get)   = getc   // smart constructor
  compAlg (Set v) = setc v // same, parameterised
```

...and that would be the end of it.

[Ergo the whole 'modular' thing we've got going]

So What Goes Wrong?

StateT s (ErrorT m) $\not\equiv$ ErrorT (StateT s m)

In the interests of being **painfully** explicit:

s $\rightarrow$ m (Maybe (a, s)) $\not\equiv$ s $\rightarrow$ m (Maybe a, s)

A **fundamental** difference in observable behaviour!

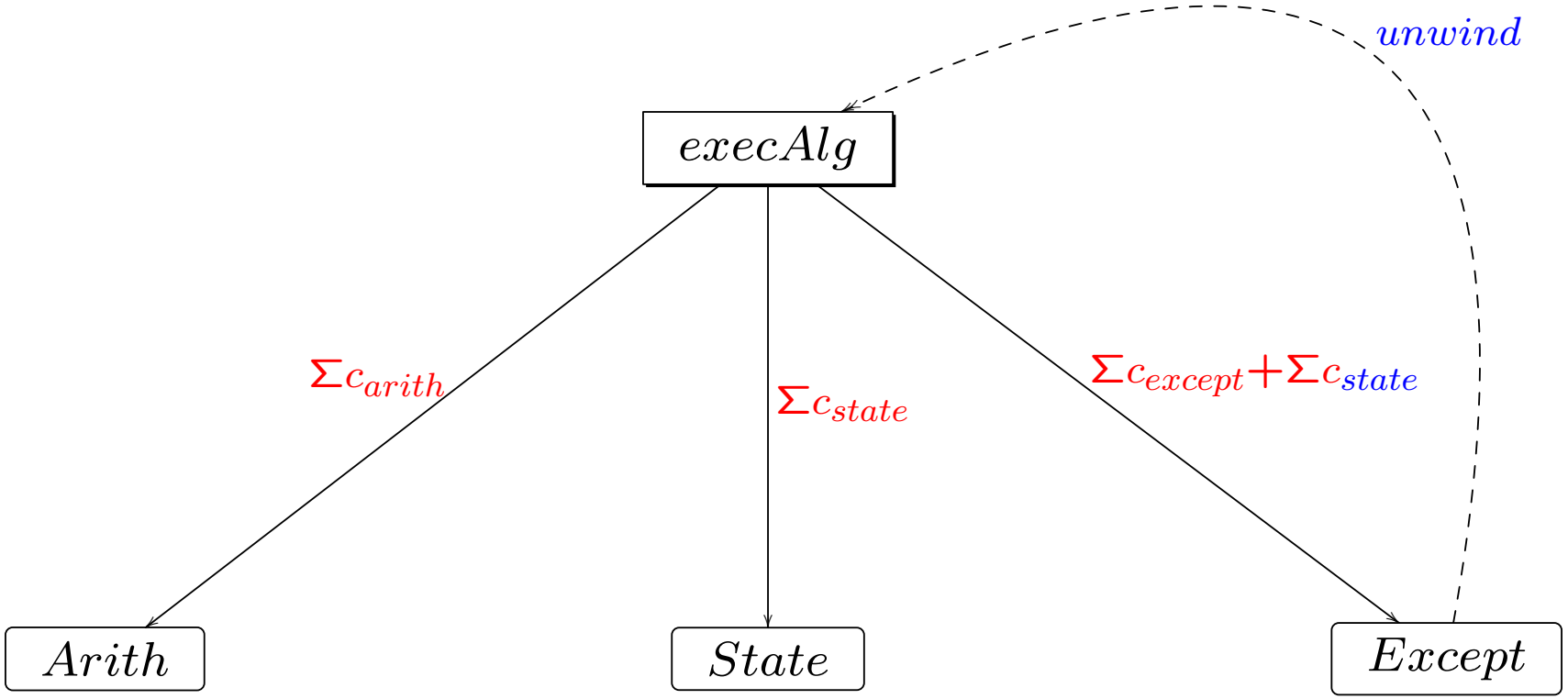
Those Pesky Effects

The compilation algebra for **exceptions** changes:

```
instance Comp Except where
  compAlg (Throw)     = throwc
  compAlg (Catch x h) = x `catchc` h
                        λc ->
                          h c `markc`
                          (savec $ x $ restorec $
                          unmarkc c)
```

But we only **need** this change for local state!

Execution and Fixed Code Types



Solutions?

1. **Different compiler algebras** predicated on the presence of certain functor signatures:
   - No state? Don't need save and restore!

2. **Different execution algebras** based on the signature of the code it operates on:
   - e.g. global state ignores state stack records

3. **Modularising the target types** for these algebras to avoid unnecessary VM constraints:
   - the monad for exceptions $\notin$ StateMonad!

Critique: Different Compiler Algebras

- The same problem as we're trying to solve!
  - dressed up in catamorphisms and coproducts
  - once we write an algebra instance we should never have to touch it again
- Overlapping instances abound:
  - how do you pick an evaluation order?
  - likely wouldn't even have the grace to compile

Critique: Different Virtual Machines

- Some effects can be 'compiled away':
  - Maybe doesn't currently appear mid-execution
  - Could obviously change in a different context!
- Missing a notion of modular case analysis:
  - New constructors force code restructuring
  - Pattern matching as a catamorphism?

Why Is This Worthwhile?

- Exceptions + State the most palatable combination of effects going:
  - simple but powerfully expressive problem
- True separation of the two a sign of progress:
  - can move onto more complex combinations
- Target type modularisation a likely panacea:
  - most of our issues will fall to the wayside

What We're Doing Next

- Bahr and Hvitved:
  - have modularised target types! [will be collaborating with them post-transfer report]

- Syntactic extensions and other semantic approaches:
  - languages with binders, some categories

- Convinced monad transformers aren't for us:
  - Monatron? Coproducts of monads?

Obligatory Summary Slide

Brief synopsis of the issues in the first year of my PhD

Effects are tricky to pry apart semantically:
even if they're quite basic on their own

Catamorphisms and effect typeclasses are excellent
separation techniques for language design

There is no love lost between me and the `mtl` library.