

# Closure-Free Functional Programming in a Two-Level Type Theory

András Kovács

University of Gothenburg

3rd Sept 2024, ICFP 2024, Milan

# Compiling monads in GHC Haskell

Source:

```
f :: Reader Bool Int
f = do
  b <- ask
  if b then return 10
      else return 20
```

# Compiling monads in GHC Haskell

Source:

```
f :: Reader Bool Int
f = do
  b <- ask
  if b then return 10
      else return 20
```

-OO Core output:

```
dict1 :: Monad (Reader Int)
dict1 = MkMonad ...

dict2 :: MonadReader (Reader Int)
dict2 = MkMonadReader ...

f :: Reader Bool Int
f = (>=) dict1 (ask dict2) (\b ->
  case b of
    True  -> return dict1 10
    False -> return dict1 20)
```

# Compiling monads in GHC Haskell

-01 output:

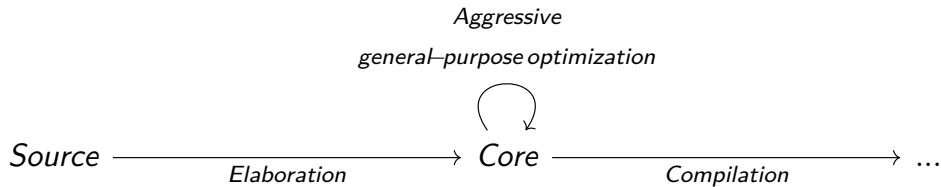
```
f :: Bool -> Int
f b = case b of
  True  -> 10
  False -> 20
```

*Optimization is hard!*

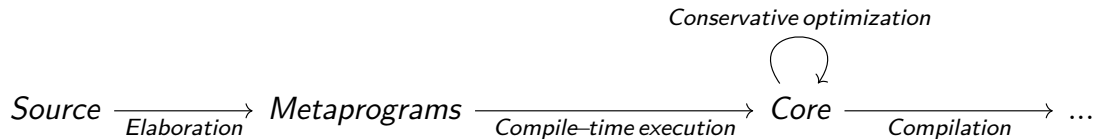
Example: `mapM` is third-order & rank-2 polymorphic, but almost all use cases should compile to first-order monomorphic code.

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

# Compiling monads in GHC Haskell



# Proposal



The paper explores **monad transformers** and stream fusion using this design.

# Proposal

Source in WIP language:

```
f : Reader Bool Int
f := do
  b <- ask
  if b then return 10
      else return 20
```

- Looks similar to Haskell.
- Desugaring & elaboration does slightly more work.
- Compilation to efficient code is formally guaranteed.

# Proposal

## Source in WIP language:

```
f : Reader Bool Int
f := do
  b <- ask
  if b then return 10
    else return 20
```

- Looks similar to Haskell.
- Desugaring & elaboration does slightly more work.
- Compilation to efficient code is formally guaranteed.

## Setup

- *Two-level type theory (2LTT)*:
  - Metalanguage (compile time): dependently typed.
  - Object language (runtime): simply typed, *polarized*.
  - The two are smoothly integrated.



# The 2LTT

- **MetaTy**: universe of meta-level types.
- **Ty**: universe of object-level types. **Polarization**: **ValTy** and **CompTy** are sub-universes of **Ty**.

A meta-level program:

```
id : {A : MetaTy} -> A -> A
id x = x
```

An object-level program:

```
data List (A : ValTy) := Nil | Cons A (List A)

myMap : List Int -> List Int
myMap ns := case xs of
  Nil      -> Nil
  Cons n ns -> Cons (n + 10) (myMap ns)
```

*Polarization lets us control closure creation through types.*

# The 2LTT - interaction between stages

- **Lifting**: for  $A : \text{Ty}$ , we have  $\uparrow A : \text{MetaTy}$ , as the type of metaprograms that produce  $A$ -typed object programs.
- **Quoting**: for  $t : A$  and  $A : \text{Ty}$ , we have  $\langle t \rangle : \uparrow A$  as the metaprogram which immediately returns  $t$ .
- **Splicing**: for  $t : \uparrow A$ , we have  $\sim t : A$  which runs the metaprogram  $t$  and inserts its output in some object-level code.
- Definitional equalities:  $\sim \langle t \rangle \equiv t$  and  $\langle \sim t \rangle \equiv t$ .

## Staged example

```
map : {A B : ValTy} -> (↑A -> ↑B) -> ↑(List A) -> ↑(List B)
```

```
map f as = <letrec go as := case as of  
    Nil      -> Nil  
    Cons a as -> Cons ~(f <a>) (go as)  
in go ~as>
```

```
myMap : List Int -> List Int
```

```
myMap ns := ~(map (\x. <~x + 10>) <ns>)
```

## Staged example - with stage inference

```
map : {A B : ValTy} -> (A -> B) -> List A -> List B
map f = letrec go as := case as of
    Nil      -> Nil
    Cons a as -> Cons (f a) (go as)
in go
```

```
myMap : List Int -> List Int
myMap := map (\x. x + 10)
```

# A monad for code generation

Type classes (and monads) only exist in the metalanguage.

```
class Monad (M : MetaTy -> MetaTy) where
  return : A -> M A
  (>>=)   : M A -> (A -> M B) -> M B
```

# A monad for code generation

Type classes (and monads) only exist in the metalanguage.

```
class Monad (M : MetaTy -> MetaTy) where
  return : A -> M A
  (>>=)   : M A -> (A -> M B) -> M B
```

**Gen** is a Monad whose effect is **generating object code**:

```
newtype Gen A = Gen {unGen : {R : Ty} -> (A -> ↑R) -> ↑R}
instance Monad Gen where ...
```

```
runGen : Gen (↑A) -> ↑A
runGen (Gen f) = f id
```

# A monad for code generation

Type classes (and monads) only exist in the metalanguage.

```
class Monad (M : MetaTy -> MetaTy) where
  return : A -> M A
  (>=>)   : M A -> (A -> M B) -> M B
```

**Gen** is a Monad whose effect is **generating object code**:

```
newtype Gen A = Gen {unGen : {R : Ty} -> (A -> ↑R) -> ↑R}
instance Monad Gen where ...
```

```
runGen : Gen (↑A) -> ↑A
runGen (Gen f) = f id
```

Generating an object-level **let**-definition:

```
gen : {A : Ty} -> ↑A -> Gen (↑A)
gen {A} a = Gen $ \k. <let x : A := ~a in ~(k <x>)>
```

# A monad for code generation

## Metaprogram:

```
foo : Int
foo := ~(runGen $ do
  x <- gen <10 + 10>
  y <- gen <~x * ~x>
  return <~x * ~y>)
```

## Code output:

```
foo : Int
foo := let x := 10 + 10 in
       let y := x * x in
       x * y
```



# Generating monadic code

We want to define efficient code generation for a monad  $M$ .

*$M$  extended with **Gen** at the bottom yields the corresponding code generator monad.*

For example:

- **ReaderT** ( $\uparrow R$ ) **Gen** ( $\uparrow A$ ) actions are code generators for **Reader**  $R$   $A$ .
- **StateT** ( $\uparrow S$ ) **Gen** ( $\uparrow A$ ) actions are code generators for **State**  $S$   $A$ .

# Generating monadic code

We want to define efficient code generation for a monad **M**.

***M** extended with **Gen** at the bottom yields the corresponding code generator monad.*

For example:

- **ReaderT** ( $\uparrow$ **R**) **Gen** ( $\uparrow$ **A**) actions are code generators for **Reader** **R** **A**.
- **StateT** ( $\uparrow$ **S**) **Gen** ( $\uparrow$ **A**) actions are code generators for **State** **S** **A**.

In each case, we can convert back-and forth, e.g.

```
up    :  $\uparrow$ (Reader R A) -> ReaderT ( $\uparrow$ R) Gen ( $\uparrow$ A)  
down  : ReaderT ( $\uparrow$ R) Gen ( $\uparrow$ A) ->  $\uparrow$ (Reader R A)
```

# Generating monadic code

## Metaprogram:

```
action : Reader Int Int
action := ~(down $ do
  x <- ask
  x <- ask
  return <~x + ~x>)
```

## Output:

```
action : Reader Int Int
action := Reader (\x. x + x)
```

*We get “fusion” for monadic code.*

# Pattern matching on object-level values

How do we inspect the structure of object-level values at compile time?

We don't directly support looking inside object expressions (it breaks some things).

But we can generate object-level pattern matches in **Gen**:

```
split : ↑Bool -> Gen MetaBool
split b = Gen $ \k. <case ~b of
  True  -> ~(k MetaTrue)
  False -> ~(k MetaFalse)>
```

**split** generalizes to all object ADT-s and all **Gen**-based monads.

## Compiling monads - example

```
f : Reader Bool Int
f := do
  b <- ask
  if b then return 10
    else return 20
```

==>

```
f : Reader Bool Int
f := ~(down $ do
  b <- ask
  split b >=> \case
    MetaTrue  -> return <10>
    MetaFalse -> return <20>)
```

```
==> f : Reader Bool Int
f := Reader (\b. case b of
  True  -> 10
  False -> 20)
```

# More things

More in the paper and artifact:

- Handling join points in monads.
- Handling mutually recursive blocks.
- Stream fusion.
- More metatheory.
- Adaptation as Agda and Typed Template Haskell libraries.

Work in progress:

- Standalone prototype targeting LLVM.
- Deploying the Template Haskell library in the Agda source code, in high-performance generics.

# More things

More in the paper and artifact:

- Handling join points in monads.
- Handling mutually recursive blocks.
- Stream fusion.
- More metatheory.
- Adaptation as Agda and Typed Template Haskell libraries.

Work in progress:

- Standalone prototype targeting LLVM.
- Deploying the Template Haskell library in the Agda source code, in high-performance generics.

**Thank you!**