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

András Kovács

University of Gothenburg

3 Sept 2024, ICFP 2024, Milan

Compiling monads in GHC Haskell

Input:

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

Compiling monads in GHC Haskell

Input:

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

-O0 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

-O1 output:

```haskell
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.

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

Compiling monads in GHC Haskell

$$\begin{tikzcd}[column sep=huge] \textit{Source} \arrow[r, "\textit{Elaboration}"'] & \textit{Core} \arrow[loop above, "\substack{\textit{Aggressive} \\ \textit{general–purpose optimization}}"] \arrow[r, "\textit{Compilation}"'] & \dots \end{tikzcd}$$

Proposal

$$
\begin{tikzcd}
\mathit{Source} \arrow[r, "\mathit{Elaboration}"'] & \mathit{Metaprograms} \arrow[r, "\mathit{Compile\text{--}time\ execution}"'] & \mathit{Core} \arrow[loop above, "\mathit{Conservative\ optimization}"] \arrow[r, "\mathit{Compilation}"'] & \ldots
\end{tikzcd}
$$

Proposal

Input 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.
- Compiles to efficient code *with a formal guarantee, without general-purpose optimization*.

Proposal

Input 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.
- Compiles to efficient code *with a formal guarantee, without general-purpose optimization*.

Setup

- We use a *two-level type theory (2LTT)*:
  - Metalanguage (compile time): dependently typed, fancy features.
  - Object language (runtime): simpler & lower-level.
  - The two are smoothly integrated.
- Monadic programs are *metaprograms* which generate efficient runtime code.
- Most optimizations are implemented in libraries instead of compiler internals.

Closure-freedom

How do we know that an abstraction feature is “low-cost”?

It’s a good indicator if the generated code is **free of closures**.

- Functional abstractions are usually implemented by closures: higher-order functions, classes, ML functors.
- A big part of GHC’s **`-O1`** work is to get rid of closures.

The 2LTT

- **`MetaTy`**: universe of meta-level types. Supports Π, Σ, inductive families.
- **`Ty`**: universe of object-level types. Only simple types. Polarized to *computation* & *value* types.

A meta-level program:

```
id : {A : MetaTy} → A → A
id x = x
```

An object-level program:

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

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

The 2LTT - interaction between stages

- **Lifting**: for `A : Ty`, we have `⇑A : MetaTy`, as the type of metaprograms that produce `A`-typed object programs.
- **Quoting**: for `t : A` and `A : Ty`, we have `<t>` as the metaprogram which immediately returns `t`.
- **Splicing**: for `t : ⇑A`, we have `~t : A` which runs the metaprogram `t` and inserts its output in some object-level code.
- Definitional equalities: `~<t> ≡ t` and `<~t> ≡ 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 (λ 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 (λ 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 $ λ k. <let x : A := ~a in ~(k <x>)>
```

A monad for code generation

Staged input:

```
myAction : ⇑Int → Gen ⇑Int
myAction x = do
  y ← gen <~x + ~x>
  z ← gen <~y * ~y>
  return <~y * ~z>

foo : Int
foo := ~(runGen $ myAction <10>)
```

Output:

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

Staging monads

We only program in meta-level monads, but also have back-and-forth translations between object-level types and metamonads.

```
down : ReaderT (⇑R) Gen (⇑A) → ⇑(ReaderT₀ R Identity₀ A)
up   : ⇑(ReaderT₀ R Identity₀ A) → ReaderT (⇑R) Gen (⇑A)

f : ReaderT₀ Bool Identity₀ Int
f := ~(down $ do
  b  ← ask
  b' ← split b
  case b' of
    MetaTrue  → return <10>
    MetaFalse → return <20>)
```

In general: up/down is defined by recursion on a transformer stack. **`Identity₀`** is related to **`Gen`**.

Case splitting on object values

```
split : MonadGen m => ⇑Bool → m MetaBool
split b = liftGen $ Gen $ λ k. <case ~b of
  True  → ~(k MetaTrue)
  False → ~(k MetaFalse)>

f : ReaderT₀ Bool Identity₀ Int
f := ~(down $ do
  b  ← ask
  b' ← split b
  case b' of
    MetaTrue  → return <10>
    MetaFalse → return <20>)
```

Polarization & Closure-Freedom

Computation and *value* types are tracked in the object language.

```
_→_     : ValTy → Ty → CompTy
Closure : CompTy → ValTy
List    : ValTy → ValTy
...
```

Closures only appear at runtime if we use **`Closure`**!

We have to use **`Closure (A → B)`** to store functions in ADTs or pass them as function arguments.

(It's rare that closures are *really needed* in programming!)

Polarization & Closure-Freedom

How to compile this?

```
f : Bool → Int → Int
f b = case b of True  → λ x. x + 10
                False → λ x. x * 10
```

And this?

```
f : Int → Int
f x :=
  let g y := x + y;
  g x + 10
```

More things

- Conditionally accepted at ICFP 24: *Closure-Free Functional Programming in a Two-Level Type Theory*.
- More things in paper: join points, stream fusion, semantics, more about polarized types.
- Implementations:
  - In Agda and typed Template Haskell with some limitations.
  - Standalone implementation early WIP.

Thank you!