Polarized Lambda-Calculus at Runtime, Dependent Types at Compile Time

András Kovács

4 June 2024, CS retreat

András Kovács

Postdoc in Logic and Types under Thierry Coquand since 2023 September.

Started in economics & finance in Budapest, switched to CS, did PhD in type theory.

Some current interests:

- Fitness, nutrition.
- "Harsh vocals" (screaming in metal music).

Research:

- Type theory: theory of inductive types, universes.
- Making proof assistants run fast (annoyed by Agda & Coq).
- High-level high-performance programming (annoyed by Haskell).

Compiling monads today in Haskell

GHC's input:

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

GHC's -O0 output:

```
dict :: Monad (Reader Int)
dict = MkDict bindReader returnReader

f :: Reader Bool Int
f x = (>>=) dict (ask dict) (\b →
  case b of
    True  → return dict 10
    False → return dict 20)
```

Compiling monads today in Haskell

GHC's -O1 output:

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

- Elaboration to `-O0` is deterministic and relatively cheap.
- Going from `-O0` to `-O1` is hard and needs a lot of machinery.

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

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

GHC has to guess the programmer's intent.

Doing it differently

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 *deterministically*, *without general-purpose optimization*.

Doing it differently

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 *deterministically, without general-purpose optimization*.

Main idea

- 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.

The 2LTT

Two type universes for the two levels.

1. **`MetaTy`**: universe of meta-level types. Supports Π, Σ, inductive families.
2. **`Ty`**: universe of object-level types.
    - `Ty` is itself an element of `MetaTy`.
    - No polymorphism or type dependency in `Ty`.
    - Two sub-universes:
        - `CompTy` contains *computation types*: functions, computational products.
        - `ValTy` contains *value types*: ADT-s and closure types.
        - ADT constructors only store values, functions only take value inputs.

The 2LTT

A metaprogram:

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

An object 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 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>
  pure <~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

Example for `Reader`:

```
newtype Identity (A : ValTy) := Identity {runIdentity : A}

newtype ReaderT (R : ValTy) (M : ValTy → Ty)(A : ValTy) :=
  Reader {runReader : R → A}

newtype ReaderTᴹ(R : MetaTy) (M : MetaTy → MetaTy)(A : MetaTy) =
  Readerᴹ {runReaderᴹ : R → A}
```

Staging monads

Instead of programming at type `ReaderT∘ R Identity∘` (which is not a monad!), we program at `ReaderT (⇑R) Gen`, and define back-and-forth conversions:

```
up : ⇑(ReaderT R Identity A) → ReaderT^M (⇑R) Gen (⇑A)
up f = ReaderT^M $ λ r. pure <runIdentity (runReaderT ~f ~r)>

down : ReaderT^M (⇑R) Gen (⇑A) → ⇑(ReaderT∘ R Identity A)
down (ReaderT^M f) = <ReaderT∘ (λ r. Identity (~runGen (f <r>)))>
```

In general: up/down is defined by recursion on a transformer stack. The bottom `Identity` is swapped to `Gen` at the meta-level.

Staging monads

Somewhat explicit source code:

```
f : ReaderT Int Int
f := ~(down $ do
   x <- ask
   pure <~x + 100>)
```

With more inference:

```
f : ReaderT Int Int
f := do
   x <- ask
   pure (x + 100)
```

Generated output:

```
f : ReaderT Int Identity Int
f := ReaderT (λ n. Identity (n + 100))
```

Polarization & Closure-Freedom

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

There's a value type former for *closures*, that we **have not yet used** in this talk.

The computational function type guarantees compilation without closures, with only statically known calls!

Essentially usage of closures is surprisingly rare in programming.

- Conditionally accepted at ICFP 24: *Closure-Free Functional Programming in a Two-Level Type Theory*.
- More things in paper: case splitting on object-level data, join points, stream fusion, more about polarized types.
- Implementations:
  - In Agda and typed Template Haskell with some limitations.
  - Standalone implementation planned, help from Ondrej Kubánek (MSc project).

Thank you!