Programming With Two-Level Type Theory

András Kovács

University of Gothenburg & Chalmers University of Technology

29th Jan 2026, TFP 26, Odense

What is this about

Concrete, unnamed, WIP language for high-level high-performance programming.

- High-level: FP abstractions, generic programming, strong types.
- High-performance: control over code generation, memory layout, allocation.

Non-goal: “systems” programming.

- We have substantial RTS with GC & full memory safety.

Past implementations: smaller demo [Kov22], Agda & Typed TH embedding [Kov24]
Currently in early stage of development: **`https://github.com/AndrasKovacs/2ltt-impl`**

Overview

1. Motivation
2. 2LTT intro
3. Monads
4. Fusion
5. Memory layout control
6. Region allocation

Motivation

I'm interested in high-performance type theory implementations.

GHC Haskell has been the clear best choice:

- High-throughput GC, decent code generation, unboxed types, compact regions, type classes, efficient laziness.

Motivation

I'm interested in high-performance type theory implementations.

GHC Haskell has been the clear best choice:

- High-throughput GC, decent code generation, unboxed types, compact regions, type classes, efficient laziness.

Why not use

- Rust: no native GC, library solutions are slow & noisy.
- OCaml: GC leans more towards latency than throughput, less memory layout control.

Motivation

I'm interested in high-performance type theory implementations.

GHC Haskell has been the clear best choice:

- High-throughput GC, decent code generation, unboxed types, compact regions, type classes, efficient laziness.

Why not use

- Rust: no native GC, library solutions are slow & noisy.
- OCaml: GC leans more towards latency than throughput, less memory layout control.

However, major performance problems with GHC.

Motivation

I'm interested in high-performance type theory implementations.

GHC Haskell has been the clear best choice:

- High-throughput GC, decent code generation, unboxed types, compact regions, type classes, efficient laziness.

Why not use

- Rust: no native GC, library solutions are slow & noisy.
- OCaml: GC leans more towards latency than throughput, less memory layout control.

However, major performance problems with GHC.

OxCaml: good progress but immature ecosystem ATM, legacy design constraints with many of the same problems as in GHC.

The GHC pipeline

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

The core simplifier is

- Complex.
- Unstable across GHC versions.
- Poorly controllable by users.

A lot of idiomatic Haskell relies on it for acceptable performance.

GHC example 1

Source:

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

GHC example 1

Source:

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

-O0 Core output:

```haskell
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)
```

GHC example 2

`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) -> m [a] -> m [b]
```

High-performance Haskell programming requires a read-eval-print-look-at-Core loop.

Revised pipeline

$$\begin{tikzcd} \textit{Source} \arrow[r, "\textit{Elaboration}"'] & \textit{Metaprograms} \arrow[r, "\substack{\textit{Compile–time}\\ \textit{execution}}"'] & \textit{Object core} \arrow[loop above, "\textit{Optimization}"] \arrow[r, "\textit{Compilation}"'] & \dots \end{tikzcd}$$

The *metalanguage* and the *object language* should be different.

- Simple object language supports better compilation & performance.
- Dependent type theory as metalanguage.

Main design question: explicit control in object language vs. optimizations in the compiler.

- Impractical manually: tail calls, dead code elimination, etc.

The 2LTT

Universes for stages:

- `Set : Set` contains **dependent** meta-level types.
- `Ty : Set` contains **simple** object-level types.
- `ValTy : Set` and `CompTy : Set` are subtypes of `Ty`.

Interaction between stages:

- **Lifting**: for `A : Ty`, we have `⇑A : Set`, as the type of metaprograms that produce `A`-typed object programs.
- **Quoting**: for `t : A` and `A : Ty`, we have `<t> : ⇑A` 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`.

The object level

An object-level program:

```
data List (A : ValTy) := nil | cons A (List A)

f : List Int → List Int
f xs := case xs of
  nil       → nil
  cons x xs → cons (x + 10) (f xs)
```

Polarization:

- Functions have value arguments and are computations.
- Data types have value fields and are values.

The object level

Explicit type former for closures:

```
Close : CompTy → ValTy
close : A → Close A
open  : Close A → A
```

Mapping with closures:

```
map : Close (Int → Int) → List Int → List Int
map f xs = case xs of
  nil       → nil
  cons x xs → cons (open f x) (map f xs)
```

Closures are surprisingly rarely needed in practical programming!

Staging

Fully explicitly:

```
map : {A B : ValTy} → (⇑A → ⇑B) -> ⇑(List A) → ⇑(List B)
map {A}{B} f as = <
  let go : List ~A → List ~B
      go as := case as of
        nil       → nil {~B}
        cons a as → cons {~B} ~(f <a>) (go as)
  go ~as>

monoMap : List Int -> List Int
monoMap xs := ~(map (λ x. <~x + 10>) <xs>)
```

Staging

Unstaged output:

```
monoMap : List Int → List Int
monoMap xs :=
  let go : List Int → List Int
      go as := case as of
        nil       → nil {Int}
        cons a as → cons {Int} (a + 10) (go as)
  go xs
```

With inference & elaboration

```
map : {A B : ValTy} → (A → B) → List A → List B
map f as =
  let go as := case as of
        nil        → nil
        cons a as → cons (f a) (go as)
  go as

monoMap : List Int → List Int
monoMap := map (λ x. x + 10)
```

How to compile: monads

Not easy! We want

- guaranteed closure-freedom for everything except CPS monads
- guaranteed fusion for straight-line code (e.g. jumps instead of constructor allocation in **`Maybe`**)
- proper handling of join points and tail calls
- modest code noise relative to Haskell

How to compile: monads

Not easy! We want

- guaranteed closure-freedom for everything except CPS monads
- guaranteed fusion for straight-line code (e.g. jumps instead of constructor allocation in **`Maybe`**)
- proper handling of join points and tail calls
- modest code noise relative to Haskell

Overview of the solution:

- The bulk of the logic is in a plain *library*.
- We use extra desugaring logic in **`do`**-blocks.
- It's good to have implicit coercions as an extra feature.
- *Tail calls* are guaranteed in the general-purpose optimizer.

Monads: the bulk of the logic

Monads only exist at compile time.

```
class Monad (M : Set → Set) where
  pure  : {A : Set} → A → M A
  (>>=) : {A B : Set} → M A → (A → M B) → M B
```

Recipe:

1. We want to port a transformer stack from Haskell.
2. We have an object-level type, same as in Haskell (but with polarities).
3. We have a meta-level transformer stack, which has an extra monad at the bottom, having *code generation as an effect*.
4. We define back-and-forth conversion between the object-level type and the metamonad.

The Gen monad

```
record Gen (A : Set) : Set = gen {unGen : {R : Ty} → (A → ⇑R) → ⇑R}

instance Monad Gen where ...

runGen : Gen ⇑A → ⇑A
runGen (gen f) = f id

class Monad M => MonadGen M where
  liftGen : Gen A → M A

genLet : MonadGen M => ⇑A → M ⇑A
genLet a = liftGen λ k. <let x := ~a; ~(k <x>)>
```

The Gen monad

```
f : Int
f := ~(runGen do
  x ← genLet <10 + 20>
  y ← genLet <~x * 10>
  pure <~x + ~y>)
```

unstage
==>

```
f : Int
f :=
  let x := 10 + 20
  let y := x * 10
  x + y
```

Case splitting in Gen

```
data BoolM : Set = trueM | falseM
data Bool : ValTy := true | false

down : BoolM → ⇑Bool
down x = case x of trueM → <true>; falseM → <false>

up : ⇑Bool → BoolM
up = IMPOSSIBLE
```

However:

```
up : MonadGen M => ⇑Bool → M BoolM
up b = liftGen λ k. <case ~b of true → ~(k trueM); false → ~(k falseM)>
```

Case splitting in MonadGen

We add **extra desugaring** in **`MonadGen do`**-blocks for case splitting.

```
f : Bool → Bool
f b := runGen do
  case b of
    true  → pure false
    false → pure true
```

```
elaborate
   ==>
```

```
f : Bool → Bool
f b := ~(runGen do
  b ← up <b>
  case b of
    trueM  → pure <false>
    falseM → pure <true>)
```

Monads in general

Implicit conversion between metamonads and runtime types, defined by recursion on the transformer stack (details in [Kov24]). Overloading in **`mtl`**-style. Example:

```
M : Ty
M = StateT Int (ReaderT Bool Identity)

f : M ()                                         f : M ()
f := do                                          f = stateT λ s. readerT λ r.
  b <- ask                        unstage          case r of
  n <- get                          ==>              true  → let s := s + 10; ((), s)
  case b of                                          false → let s := s * 10; ((), s)
    true  → put $ n + 10
    false → put $ n * 10
```

Monads in general

Only a modest amount of extra noise compared to Haskell.
(But no native implementation yet!)

All of **`mtl`** works. Closures are only needed in **`ContT`**.

`Reader` and **`State`** are computation types! We need to wrap them in **`Close`** to store them in data structures.

Memory layout control

All constructors are unboxed by default.

```
data Pair A B := pair A B
data Sum A B := left A | right B
```

Recursive constructors must be guarded by a *pointer to a region*. **`Hp`** is the general GC-d heap.

```
data List A := nil | cons@Hp A (List A)
```

Weird sum type with just one unboxed constructor:

```
data Sum A B := left A | right@Hp B
```

Tag-free GC & bit-stealing

GC is *almost tag-free*: only 1 bit metadata per heap object.

Arbitrary data can be opportunistically stored into unused bits in pointers.

On x64: we use 16 bits in pointers for storage, 1 reserved for GC.

Huge space savings compared to GHC!

Tag-free GC & bit-stealing

Example: pure lambda terms with 32-bit variables.

```
data Tm := var UInt32 | app@Hp Tm Tm | lam@Hp Tm
```

Layout of **`app (var 0) (var 1)`**

```
  | app | ptr |                 -- 1 word
          ↓
      | var | 0 | var | 1 |     -- 2 words
```

Same in GHC:

```
  | app | ptr |                             -- 1 word
          ↓
      | app    | ptr          | ptr |       -- 3 words
                 ↓              ↓
             | var | 0 |    | var | 1 |     -- 4 words
```

Tag-free GC & bit-stealing

Implementation: explored back in the 90s [**?**].

In a simple type theory, it's enough to know the types (memory layouts) of GC roots.

For each monotype, we generate code for GC scanning & copying.

Only *stack frames* need to store runtime type information about roots.

Regions

```
Location : Set
Hp       : Location
Region   : Set
```

There's implicit coercion from **`Region`** to **`Location`**. The object language supports dependent functions of the form **`(R : Region) → ...`**.

Lists with cons cells in a specified location:

```
data List (L : Location) (A : ValTy) := nil | cons@L A (List L A)
```

Regions

Example: list in a local region.

```
sum : {R : Region} → List R Int → Int
sum xs := case xs of nil → 0; cons x xs → x + sum xs

countDown : {R : Region} → Int → List R Int
countDown x := case x of 0 → nil
                         n → cons x (countDown (x - 1))

f : Int → Int
f x :=
  let R : Region
  let xs : List R Int := countDown x
  sum xs
```

András Kovács.
Staged compilation with two-level type theory.
Proc. ACM Program. Lang., 6(ICFP):540–569, 2022.

András Kovács.
Closure-free functional programming in a two-level type theory.
Proc. ACM Program. Lang., 8(ICFP):659–692, 2024.