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

Unnamed WIP language for high-level high-performance programming.

What is this about

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.

What is this about

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`**

Motivation

I'm interested in high-performance implementations of dependent type systems.

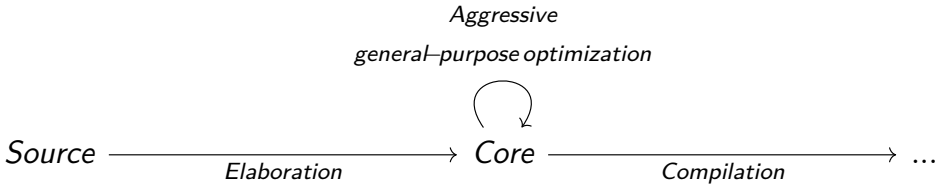
GHC Haskell has been my choice for performance:

- GC focused on throughput, OK machine code output, unboxed types, compact regions, efficient laziness (for the few cases when it's needed).

There are some performance issues or significant inconveniences in every other language I know.

Still: *performance issues with GHC too* - motivating this research.

The GHC pipeline



The core simplifier is

- Complex.
- Unstable across GHC versions.
- Supports limited user control.

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:

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

`-O0` 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)
```

GHC example 2

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

Revised pipeline

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.

Revised pipeline

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.

- The object language **can** be tedious as long as we can address tedium with metaprogramming!

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`** (polarization!).

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`** (polarization!).

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@Hp 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.
- ADTs 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

```
inlineInt : ⇑Int
inlineInt = <100>

myInt : Int
myInt := 200

id : {A : Ty} → ⇑A → ⇑A
id x = x

f : Int → Int
f x := x + ~inlineInt

g : Int → Int
g x := ~(id <x>)
```

unstage

==>

```
myInt : Int
myInt := 200

f : Int → Int
f x := x + 100

g : Int → Int
g x := x
```

Staging

Fully explicit **map**:

```
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
```

Inference & elaboration

Quotes and splices are almost always inferable!

```
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

Ingredients of the solution:

- The bulk of the logic is in a plain library.
- We use type classes & *implicit coercions*.
- Extra desugaring logic in **`do`**-blocks.
- Modest work in the downstream optimizer for tail calls.

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 for porting over a transformer stack from Haskell:

1. We have an object-level type, same as in Haskell (but with polarities).
2. We have a meta-level transformer stack, which has an extra monad at the bottom, having *code generation as an effect*.
3. 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 : Int
f := ~(runGen do            unstage      f :=
  x ← genLet <10 + 20>        ==>          let x := 10 + 20
  y ← genLet <~x * 10>                     let y := x * 10
  pure <~x + ~y>)                          x + y
```

Case splitting in MonadGen

```
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>)
```

Every object-level case split can be handled analogously!

Monads in general

Example: conversion between object-level type and a meta-monad.

```
⇑(StateT Int (ReaderT Bool Identity) A)
                    ≈
StateTM (⇑Int) (ReaderTM (⇑Bool) Gen) (⇑A)
```

Generally: this conversion can be defined by recursion on the transformer stack (using e.g. typeclasses) [Kov24].

Monads in general

Code example:

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

f : M ()
f := do
  b <- ask
  n <- get
  case b of
    true  → put $ n + 10
    false → put $ n * 10
```

elab + unstage ==>

```
f : M ()
f := stateT λ s. readerT λ r.
  case r of
    true  → let s' := s + 10
            identity ((), s')
    false → let s' := s * 10
            identity ((), s')
```

Monads in general

- A modest amount of extra noise compared to Haskell (but no native implementation yet!)
- All of **`mtl`** works. Closures are only needed in **`ContT`**.
- Note: **`Reader`** and **`State`** are computation types, so 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 : ValTy) := pair A B
data Sum (A B : ValTy) := 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.

Bit-stealing: any data can be stored in unused bits in pointers or constructor fields.

Large space savings compared to pretty much any managed RTS language.

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 [Tol94].

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
```

Type-directed GC scanning

Example 1: if a list contains no heap pointers, GC doesn't touch it!

Type-directed GC scanning

Example 1: if a list contains no heap pointers, GC doesn't touch it!

Why? Assume we have `xs : List R Int` in scope where `R : Region`.

Type-directed GC scanning

Example 1: if a list contains no heap pointers, GC doesn't touch it!

Why? Assume we have `xs : List R Int` in scope where `R : Region`.

The region `R` itself is present at runtime, and it serves as a GC root for the whole region.

Type-directed GC scanning

Example 1: if a list contains no heap pointers, GC doesn't touch it!

Why? Assume we have `xs : List R Int` in scope where `R : Region`.

The region `R` itself is present at runtime, and it serves as a GC root for the whole region.

Since the whole region is kept alive by the `R` reference, there's no need to scan the list.

Type-directed GC scanning

Example 2: let's have `data HpPtr A := box@Hp A` and consider `List R (HpPtr Int)`.

Type-directed GC scanning

Example 2: let's have `data HpPtr A := box@Hp A` and consider `List R (HpPtr Int)`.

GC scans values of `List R (HpPtr Int)` because it needs to scan the general heap pointers inside. But it doesn't copy the list cells.

Type-directed GC scanning

Example 2: let's have **`data HpPtr A := box@Hp A`** and consider **`List R (HpPtr Int)`**.

GC scans values of **`List R (HpPtr Int)`** because it needs to scan the general heap pointers inside. But it doesn't copy the list cells.

Example 3: **`List R (Close (Int → Int))`**. Closures can capture arbitrary data, including heap pointers, so we also need to scan them!

Type-directed GC scanning

Example 2: let's have `data HpPtr A := box@Hp A` and consider `List R (HpPtr Int)`.

GC scans values of `List R (HpPtr Int)` because it needs to scan the general heap pointers inside. But it doesn't copy the list cells.

Example 3: `List R (Close (Int → Int))`. Closures can capture arbitrary data, including heap pointers, so we also need to scan them!

Example 4: `List Hp Int`. The cons cells are on the general heap, so we scan and possibly copy everything.

Existential regions

```
data InSomeRegion (L : Location) (F : Region → ValTy) :=
  inSomeRegion@L (R : Region) (F R)
```

We can also *hash regions* and compare for them equality.

So we can store regions in data structures and manage them at runtime.

The more we know about lifetimes, the more we can speed up GC by moving data into regions.

If we don't care, we can ignore regions with *no burden* on programmers.

Related

1. **GHC compact regions** [YCA+15]
   - We can create compact regions.
   - There's a primop to *deeply copy* GHC objects into the region.
   - Hence: a compact region only contains internal pointers.
   - GC doesn't scan regions. A region is alive as long as any object in it is alive.
   - Very important in Agda!

Related

1. **GHC compact regions** [YCA+15]
   - We can create compact regions.
   - There's a primop to *deeply copy* GHC objects into the region.
   - Hence: a compact region only contains internal pointers.
   - GC doesn't scan regions. A region is alive as long as any object in it is alive.
   - Very important in Agda!
2. **Rust lifetimes**
   - Deterministic & precise tracking of where objects get destroyed.
   - Sub-structural typing.

The problem with sub-structural typing

We don't know how to make it work nicely with staging.

Consider an application rule for linear functions:

$$\frac{\Gamma \vdash t : A \multimap B \qquad \Delta \vdash u : A}{\Gamma,\Delta \vdash t\ u : B}$$

The semantic (operational) meaning of $\Uparrow A \to \Uparrow B$ in 2LTT:

- We map object term to object terms in *arbitrary* object contexts.
- The mapping *commutes with substitutions*.

We can't inhabit $\Uparrow(A \multimap B) \to \Uparrow A \to \Uparrow B$. No idea if the free variables are disjoint.

Structuralizing things

1. **Massage sub-structural features into structural shape**
   - Erasure control by Constantine Theocharis (unpublished).
   - Closure capture control by AK (unpublished).

2. **Other structural features**
   - ST monad, monadic regions.
   - Disentanglement typing for thread-local GC [MBXW26].

Other things

- Staged fusion.
- IR optimizations.
- Backend compilation to LLVM.

Other things

- Staged fusion.
- IR optimizations.
- Backend compilation to LLVM.

Thank you!

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.

Alexandre Moine, Stephanie Balzer, Alex Xu, and Sam Westrick.
Typedis: A type system for disentanglement.
Proc. ACM Program. Lang., 10(POPL), January 2026.

Andrew P. Tolmach.
Tag-free garbage collection using explicit type parameters.
In Robert R. Kessler, editor, *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994*, pages 1–11. ACM, 1994.

Edward Z. Yang, Giovanni Campagna, Ömer S. Agacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton.
Efficient communication and collection with compact normal forms.
In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 362–374. ACM, 2015.