# Eta conversion for the unit type (is still not that simple)

András Kovács

University of Gothenburg & Chalmers University of Technology

25 January 2025, WITS, Denver

$$\frac{\Gamma \vdash t : \mathsf{Unit} \qquad \Gamma \vdash u : \mathsf{Unit}}{\Gamma \vdash t \equiv u}$$

- Problem: $t$ and $u$ can be anything, including distinct bound variables.
- Problem: if we have $\eta$ for $\Pi$ and/or $\Sigma$, many more types are definitionally uniquely inhabited! E.g. $(\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Unit}) \times \mathsf{Unit}$.

Conversion checking has to compute some types.

Unit $\eta$ is not *essential*...

$$\frac{\Gamma \vdash t : \mathsf{Unit} \qquad \Gamma \vdash u : \mathsf{Unit}}{\Gamma \vdash t \equiv u}$$

- Problem: $t$ and $u$ can be anything, including distinct bound variables.
- Problem: if we have $\eta$ for $\Pi$ and/or $\Sigma$, many more types are definitionally uniquely inhabited! E.g. $(\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Unit}) \times \mathsf{Unit}$.

Conversion checking has to compute some types.

Unit $\eta$ is not *essential*...

... but a good implementation can be reused more generally (e.g. for singleton types, cubical extension types, strict propositions).

# Unit $\eta$ in current practice

- Rocq, Idris 2: no attempt, purely syntax-directed conversion.

[1] https://github.com/leanprover/lean4/issues/2258

# Unit $\eta$ in current practice

- Rocq, Idris 2: no attempt, purely syntax-directed conversion.
- Lean 4:
  - Checks: `def eta (x y : Unit) : x = y := Eq.refl x`.

[1] https://github.com/leanprover/lean4/issues/2258

# Unit $\eta$ in current practice

- Rocq, Idris 2: no attempt, purely syntax-directed conversion.
- Lean 4:
  - Checks: `def eta (x y : Unit) : x = y := Eq.refl x`.
  - Fails[1]: `def eta (x y : Unit -> Unit) : x = y := Eq.refl x`

[1] https://github.com/leanprover/lean4/issues/2258

# Unit $\eta$ in current practice

- Rocq, Idris 2: no attempt, purely syntax-directed conversion.
- Lean 4:
  - Checks: `def eta (x y : Unit) : x = y := Eq.refl x`.
  - Fails[1]: `def eta (x y : Unit -> Unit) : x = y := Eq.refl x`

  In the kernel: calling `infer` on terms to get their types and check if they're unit.

[1] https://github.com/leanprover/lean4/issues/2258

# Unit $\eta$ in current practice

- Rocq, Idris 2: no attempt, purely syntax-directed conversion.
- Lean 4:
  - Checks: `def eta (x y : Unit) : x = y := Eq.refl x`.
  - Fails[1]: `def eta (x y : Unit -> Unit) : x = y := Eq.refl x`

  In the kernel: calling `infer` on terms to get their types and check if they're unit.
- Agda: type-directed conversion, good but not quite complete, inefficient (computes types even if they don't make a difference).

[1] https://github.com/leanprover/lean4/issues/2258

# Overview

In this talk:

1. Simple setup: bidirectional elaboration, no metavariables. Code examples.
2. Metavariables: not simple, no code examples.

Partially implemented, not benchmarked. Not the final word on anything!

# Basic setup

Distinction of terms and runtime values.[2]

```
data Tm
  = Var Name
  | Pi Name Tm Tm
  | Lam Name Tm
  | App Tm Tm
  | U
  | Unit
  | Tt
```

```
data Ne
  = Var Name
  | App Ne Val

type Ty = Val

data Val
  = Ne Ne
  | Pi Name (Lazy Ty) (Val -> Ty)
  | Lam Name (Lazy Ty) (Val -> Val)
  | U
  | Unit
  | Tt
```

[2] Thierry Coquand, 1996: *An algorithm for type-checking dependent types*

# Version 1: type-annotated neutrals

Distinction of terms and runtime values.[3]

```
data Tm
  = Var Name
  | Pi Name Tm Tm
  | Lam Name Tm
  | App Tm Tm
  | U
  | Unit
  | Tt
```

```
data Ne
  = Var Name
  | App Ne Val

type Ty = Val

data Val
  = Ne Ne (Lazy Ty)
  | Pi Name (Lazy Ty) (Val -> Ty)
  | Lam Name (Lazy Ty) (Val -> Val)
  | U
  | Unit
  | Tt
```

[3] Thierry Coquand, 1996: *An algorithm for type-checking dependent types*

# Version 1: type-annotated neutrals

```
type Env = Map Name Val
eval    : Env -> Tm -> Val
convert : Val -> Val -> ()
infer   : Cxt -> RawTm -> (Tm, Ty)
check   : Cxt -> RawTm -> Ty -> Tm
```

I use side-effectful pseudocode. **`eval`** is total, the other functions are partial.

# Version 1: type-annotated neutrals

```
typeOfApp : Val -> Val -> Val
typeOfApp (Pi _ _ b) u = b u

app : Val -> Val -> Val
app t u = case t of
  Ne n a    -> Ne (App n u) (typeOfApp a u)
  Lam _ _ t -> t u

eval : Env -> Tm -> Val
eval e t = case t of
  ...
  App t u -> app (eval e t) (eval e u)
  ...
```

# Version 1: type-annotated neutrals

```
isIrrelevant : Ty -> Bool
isIrrelevant a = case a of
  Unit     -> True
  Pi x a b -> let v = fresh x a; isIrrelevant (b v)
  _        -> False

convert : Val -> Val -> ()
convert t t' = case (t, t') of
  ...
  (Ne n a, Ne n' _) -> try (convert n n') (guard (isIrrelevant a))
  ...
```

- Conversion is still-syntax directed.
- Types are *only* computed if conversion depends on unit $\eta$.
- Types are computed reasonably efficiently.

# Enhancement: exploiting elaboration

The elaborator already computes many types - let's compute relevances at the same time!

```
data Val
  = Ne Ne (Lazy Ty) (Maybe Bool)           -- "Just True" is irrelevant
  ...                                      -- "Just False" is relevant
  ...                                      -- "Nothing" is "no info"

appIrr :: Maybe Bool -> Maybe Bool
appIrr (Just True) = Just True
appIrr _           = Nothing

app : Val -> Val -> Val
app t u = case t of
  Ne n a irr -> Ne (App n u) (appTy a u) (appIrr irr)
  ...
```

# Enhancement: exploiting elaboration

```
convTy : Ty -> Ty -> Maybe Bool
convTy a a' = case (a, a') of
  (U       , U         ) -> Just False
  (Unit    , Unit      ) -> Just True
  (Pi x a b, Pi _ a' b') -> convert a a';
                            let v = fresh x a; convTy (b v) (b' v)
  (Ne n _ _, Ne n' _ _ ) -> convert n n'; Nothing
  _                      -> throw CantConvert

data Tm = ... | Relevance Tm Bool

eval : Env -> Tm -> Val
eval e t = case t of
  Relevance t irr -> case eval e t of
    Ne n a _ -> Ne n a (Just irr)
    t        -> t
```

# Enhancement: exploiting elaboration

```
convert : Val -> Val -> ()
convert t t' = case (t, t') of
  ...
  (Ne n a irr, Ne n' _ irr') ->
     try (guard (irr == Just True || irr' == Just True)) $
     try (convert n n') $
     guard (irr == Nothing && irr' == Nothing && isIrrelevant a)
  ...
```

In **elaboration**: when comparing an expected and inferred type, we use **convTy** to annotate the output with relevance.

# More fancy enhancements

1. Memoize relevances computed during conversion.
2. Don't return `Nothing` from **`convTy`**, instead return a syntactic representation of a blocked computation.
   - Example: we have a big record type where all fields are irrelevant, except one with neutral type. Only the neutral type should re-evaluated at conversion time.

*Should be benchmarked! Could be pointless in practice.*

# Metavariables

Many complications.

Agda issue https://github.com/agda/agda/issues/5837:

```
test : (g : ⊤ → Bool)(h : Bool → ∀ b → if b then ⊤ else Bool) → ⊤
test g h =
  let m = _

      p : m ≡ g (h m true)
      p = refl in
  tt
```

# Task 1: detect irrelevant unification contexts

Assume bound variables $f$ and $g$:

$$f\,(g\,\alpha) =^? f\,(g\,t)$$

If $f$'s or $g$'s return type is irrelevant, we cannot uniquely solve the metavariable $\alpha$ to $t$.

During unification, if any enclosing neutral has an irrelevant type:

- Thrown exceptions are caught at the innermost such neutral.
- Attempting to solve a relevant metavariable instead throws an exception.

# Task 2: detect contractible contexts in meta solution candidates

Assume bound variables $f$ and $g$:

$$\alpha =? \, f(g\,\alpha)$$

This is an *occurs* error, except if $\alpha$ occurs in a contractible subterm. E.g. we may produce the solution:

$$\alpha := \; f \, \mathsf{tt}$$

Again we need to catch errors at contractible enclosing neutrals.

# Task 3: detect irrelevance in higher-order pattern checking

Assume bound variable $x$:

$$\alpha\, x\, x =?\, x$$

This has two solutions:

$$\alpha := \lambda\, x\, \_.\, x$$

$$\alpha := \lambda\, \_\, x.\, x$$

But if $x$'s type is irrelevant, we can pick either as the unique solution.

We need to catch *linearity errors* by looking at pattern variable types.

# Summary

I propose:

- Computing types only on demand, but efficiently.
- Piggybacking relevance computation on conversion checking in elaboration.
- Systematically catching errors and converting them to successes, based on the relevance of computational contexts.

Thank you!