Fast Elaboration for Dependent Type Theories

András Kovács

Eötvös Loránd University, Budapest

EUTypes WG Meeting, Krakow, 24 February 2019

Motivation, overview

Performance issues in current proof assistants.

Motivation, overview

Performance issues in current proof assistants.

Even greater performance demands on future proof assistants.

Motivation, overview

Performance issues in current proof assistants.

Even greater performance demands on future proof assistants.

Cubical type theories, univalence, HITs, QITs: more moving parts, computing transports, possibly huge computationally relevant proof terms.

Motivation, overview

Performance issues in current proof assistants.

Even greater performance demands on future proof assistants.

Cubical type theories, univalence, HITs, QITs: more moving parts, computing transports, possibly huge computationally relevant proof terms.

Current goals:

- Considering elaboration from ground-up, with performance as priority.
- Benchmarking a prototype against Coq and Agda.

Elaboration

Computing (explicit, well-typed) core from (implicit, incomplete) source language. Includes type checking, unification, desugaring, tactics, etc.

Minimal example for filling holes:

```
id : (A : Set) → A → A
id A x = x

id' : (A : Set) → A → A
id' A x = id _ x
```

Output:

```
id : (A : Set) → A → A
id A x = x

id' : (A : Set) → A → A
id' A x = id A x
```

Two core computational tasks in elaboration:

Two core computational tasks in elaboration:

1. $\beta\eta$-conversion checking.

Two core computational tasks in elaboration:

1. $\beta\eta$-conversion checking.
2. Generating solutions for holes (metavariables).

Solving metas in the standard way

1: Source:

```
id : (A : Set) → A → A
id A x = x

id' : (A : Set) → A → A
id' A x = id _ x
```

2: Plug hole with fresh meta:

```
α = λ A x. ?

id : (A : Set) → A → A
id A x = x

id' : (A : Set) → A → A
id' A x = id (α A x) x
```

3: Solve meta:

```
α = λ A x. A

id : (A : Set) → A → A
id A x = x

id' : (A : Set) → A → A
id' A x = id (α A x) x
```

4: Unfold meta in output:

```
id : (A : Set) → A → A
id A x = x

id' : (A : Set) → A → A
id' A x = id A x
```

Problems with the standard way

Metas are essentially unscoped: solutions can't refer to other definitions and meta solutions. Hence: everything must be unfolded.

Problems with the standard way

Metas are essentially unscoped: solutions can't refer to other definitions and meta solutions. Hence: everything must be unfolded.

Input:

```
id' : {A : Set} → A → A
id' = id id id id
```

Output:

```
id' : {A : Set} → A → A
id' = λ {A} →
          (id {((A → A) → A → A) → (A → A) → A → A})
          (id {(A → A) → A → A})
          (id {A → A})
          (id {A})
```

A better elaboration output

```
id' : {A : Set} → A → A
id' {A} =
  let α : Set = A
      β : Set = α → α
      γ : Set = β → β
      δ : Set = γ → γ
  in (id {δ}) (id {γ}) (id {β}) (id {α})
```

This is a classic example for exponential time Hindley-Milner inference.

This is a classic example for exponential time Hindley-Milner inference.

In Hindley-Milner, such silly programs are rare, and meta solutions don't get large in practice.

This is a classic example for exponential time Hindley-Milner inference.

In Hindley-Milner, such silly programs are rare, and meta solutions don't get large in practice.

In dependent TT, the size of solved metas often *dominates* the elaboration output. Hence, poor meta solutions imply poor elaboration output, and also cause slowdowns whenever we need to compute with these solutions during further elaboration.

This is a classic example for exponential time Hindley-Milner inference.

In Hindley-Milner, such silly programs are rare, and meta solutions don't get large in practice.

In dependent TT, the size of solved metas often *dominates* the elaboration output. Hence, poor meta solutions imply poor elaboration output, and also cause slowdowns whenever we need to compute with these solutions during further elaboration.

Quadratic and worse solution sizes are quite easy to get.

This is a classic example for exponential time Hindley-Milner inference.

In Hindley-Milner, such silly programs are rare, and meta solutions don't get large in practice.

In dependent TT, the size of solved metas often *dominates* the elaboration output. Hence, poor meta solutions imply poor elaboration output, and also cause slowdowns whenever we need to compute with these solutions during further elaboration.

Quadratic and worse solution sizes are quite easy to get.

Example: elaborating length-indexed vector expressions with implicit length indices is quadratic in Agda and Coq: each `cons` contains a unary natural number index with the size of the vector tail.

This is a classic example for exponential time Hindley-Milner inference.

In Hindley-Milner, such silly programs are rare, and meta solutions don't get large in practice.

In dependent TT, the size of solved metas often *dominates* the elaboration output. Hence, poor meta solutions imply poor elaboration output, and also cause slowdowns whenever we need to compute with these solutions during further elaboration.

Quadratic and worse solution sizes are quite easy to get.

Example: elaborating length-indexed vector expressions with implicit length indices is quadratic in Agda and Coq: each `cons` contains a unary natural number index with the size of the vector tail.

(Can hash consing help? Not really: overheads and failure to handle beta redexes.)

Scoping for metavariables

Metavariables must be scoped more precisely than in Agda/Coq, in order to have high-quality solutions.

Scoping for metavariables

Metavariables must be scoped more precisely than in Agda/Coq, in order to have high-quality solutions.

How precise?

Scoping for metavariables

Metavariables must be scoped more precisely than in Agda/Coq, in order to have high-quality solutions.

How precise?

1. Full precision: metas are elaborated into `let`-definitions in arbitrary local scopes.
   - Dependently typed upgrade of Krishnawami and Dunfield's mixed-prefix bidirectional checkers.
   - Allows fast let-generalization.
   - More efficient, better output.
   - Challenging to implement.

Scoping for metavariables

Metavariables must be scoped more precisely than in Agda/Coq, in order to have high-quality solutions.

How precise?

1. Full precision: metas are elaborated into `let`-definitions in arbitrary local scopes.
   - Dependently typed upgrade of Krishnawami and Dunfield's mixed-prefix bidirectional checkers.
   - Allows fast let-generalization.
   - More efficient, better output.
   - Challenging to implement.
2. Limited precision: metas only have top-level scope, and are elaborated into top-level mutual (unordered) definition blocks.
   - Easy to implement.
   - Less efficient and captures less sharing.
   - Implemented in prototype.

Evaluators for elaboration

Recall the two computational tasks: **conversion checking**, **meta solution generation**.

Evaluators for elaboration

Recall the two computational tasks: **conversion checking**, **meta solution generation**.

We need to perform both at the same time efficiently.

Evaluators for elaboration

Recall the two computational tasks: **conversion checking**, **meta solution generation**.

We need to perform both at the same time efficiently.

Conversion checking requires fast call-by-need evaluation to fully unfolded values.

Evaluators for elaboration

Recall the two computational tasks: **conversion checking**, **meta solution generation**.

We need to perform both at the same time efficiently.

Conversion checking requires fast call-by-need evaluation to fully unfolded values.

But for meta solutions, unfolded/normalized terms are very bad.

Evaluators for elaboration

Recall the two computational tasks: **conversion checking**, **meta solution generation**.

We need to perform both at the same time efficiently.

Conversion checking requires fast call-by-need evaluation to fully unfolded values.

But for meta solutions, unfolded/normalized terms are very bad.

Solution: a "glued" evaluator, which computes two different semantic values at the same time.

Evaluators for elaboration

Recall the two computational tasks: **conversion checking**, **meta solution generation**.

We need to perform both at the same time efficiently.

Conversion checking requires fast call-by-need evaluation to fully unfolded values.

But for meta solutions, unfolded/normalized terms are very bad.

Solution: a "glued" evaluator, which computes two different semantic values at the same time.

1. *Glued values*: fully unfolded values, which also carry local values around.
2. *Local values*: these are computed to some head normal form while **not** unfolding some class of definitions.

Minimal glued evaluator in Haskell

Glues call-by-need and call-by-name machines together.

```
data Tm  = Var Int | App Tm Tm | Lam Tm
data Val = VNe Int [Val] [Cl] | VLam [Val] [Cl] Tm
data Cl  = Cl [Cl] Tm

eval ∷ [Val] → [Cl] → Tm → Val
eval vs cs t = case t of
  Var i → case lookup i vs of
    Just v  -> v
    Nothing -> VNe (length vs - i - 1) [] []
  App t u → case (eval vs cs t, eval vs cs u) of
    (VLam vs' cs' t', u') → eval (u':vs') (Cl cs u :cs') t'
    (VNe i vs' cs'  , u') → VNe i (u':vs') (Cl cs u :cs')
  Lam t → VLam vs cs t
```

Glued evaluation

Unification is an operation on glued values.

Glued evaluation

Unification is an operation on glued values.

Whenever we have a meta on one side of an equation, we get a local value for a compact solution, and a glued value for fast occurs checking and (Miller) pattern condition checking.

Glued evaluation

Unification is an operation on glued values.

Whenever we have a meta on one side of an equation, we get a local value for a compact solution, and a glued value for fast occurs checking and (Miller) pattern condition checking.

We can also reuse local values for fast approximate checks: first try to unify local values, if that doesn't yield a definite result, retry full unification on glued values.

Glued evaluation

Unification is an operation on glued values.

Whenever we have a meta on one side of an equation, we get a local value for a compact solution, and a glued value for fast occurs checking and (Miller) pattern condition checking.

We can also reuse local values for fast approximate checks: first try to unify local values, if that doesn't yield a definite result, retry full unification on glued values.

In principle, one could glue together any number of different evaluators, each optimized for a specific task. Gluing just two machines seems to strike a good balance of complexity and constant overheads.

Takeaways so far

Kernel should consist of core syntax **and** a carefully chosen semantic domain (in our case, a particular environment machine). (Early example: Coquand (1996), since then: mini-TT, cubicaltt).

Takeaways so far

Kernel should consist of core syntax **and** a carefully chosen semantic domain (in our case, a particular environment machine). (Early example: Coquand (1996), since then: mini-TT, cubicaltt).

We **should not** compute directly on core syntax!

Takeaways so far

Kernel should consist of core syntax **and** a carefully chosen semantic domain (in our case, a particular environment machine). (Early example: Coquand (1996), since then: mini-TT, cubicaltt).

We **should not** compute directly on core syntax!

Core syntax should be treated as immutable machine code, used mainly for on thing: evaluating into semantic domain. Unification, scope checking etc. are all operations on semantic values.

Takeaways so far

Kernel should consist of core syntax **and** a carefully chosen semantic domain (in our case, a particular environment machine). (Early example: Coquand (1996), since then: mini-TT, cubicaltt).

We **should not** compute directly on core syntax!

Core syntax should be treated as immutable machine code, used mainly for on thing: evaluating into semantic domain. Unification, scope checking etc. are all operations on semantic values.

We can go from semantics to syntax by a readback operation, which generates meta solutions.

Takeaways so far

Kernel should consist of core syntax **and** a carefully chosen semantic domain (in our case, a particular environment machine). (Early example: Coquand (1996), since then: mini-TT, cubicaltt).

We **should not** compute directly on core syntax!

Core syntax should be treated as immutable machine code, used mainly for on thing: evaluating into semantic domain. Unification, scope checking etc. are all operations on semantic values.

We can go from semantics to syntax by a readback operation, which generates meta solutions.

Computing in the presence of metas is critically important, so metas should be in the kernel as well.

Takeaways so far

Kernel should consist of core syntax **and** a carefully chosen semantic domain (in our case, a particular environment machine). (Early example: Coquand (1996), since then: mini-TT, cubicaltt).

We **should not** compute directly on core syntax!

Core syntax should be treated as immutable machine code, used mainly for on thing: evaluating into semantic domain. Unification, scope checking etc. are all operations on semantic values.

We can go from semantics to syntax by a readback operation, which generates meta solutions.

Computing in the presence of metas is critically important, so metas should be in the kernel as well.

We get a larger kernel than in the Coq-style, but benefits seem to be significant.

Prototype implementation

Available: `https://github.com/AndrasKovacs/smalltt`

Prototype implementation

Available: `https://github.com/AndrasKovacs/smalltt`

Obviously more limited than Agda/Coq ...

Prototype implementation

Available: `https://github.com/AndrasKovacs/smalltt`

Obviously more limited than Agda/Coq ...

... but I see no reason why an extended version to Agda/Coq capabilities would be significantly slower.

Prototype implementation

Available: `https://github.com/AndrasKovacs/smalltt`

Obviously more limited than Agda/Coq ...

... but I see no reason why an extended version to Agda/Coq capabilities would be significantly slower.

Thank you!