Sharing-Preserving Elaboration with Precisely Scoped Metavariables

András Kovács

Agda Implementors' Meeting XXVI, Jan 2018

A classic example

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

After elaboration:

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

Exponential time in Agda, Coq, GHC (the last time I checked).

- In Hindley-Milner, such silly examples are rare. Not much happens on the type level, anyway.
- Conjecture: in dependent type theory, sharing matters a lot more.
- Meta solutions should be able to refer to other metas and (local) definitions.

A better 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})
```

- *Maximizing* sharing with e.g. hash-consing and CSE is too expensive.
- Goal: not destroying sharing present/implicit in source.
- Sharing: both in time and space.
- Call-by-need is a natural default: it's not possible to statically tell which expressions need to be evaluated during elaboration.
  - ... although evolving metacontexts preclude pure call-by-need.
- We'd like to have pervasive efficient (potentially bytecode/machine code based) evaluation, even in the presence of metas.

Two main parts of this talk

1. Simple setup for scoped metas, without saying anything about evaluation.
2. Evaluation & implementation considerations.

Scoped metas

Why scoped metas

- Necessary for decent sharing.
- Allows efficient let-generalization.
- Allows local first-order solution to lots of metas (probably: most metas).
- Dependency on `data` and `postulate` is non-awkward. Less or no meta freezing required.
- Conceptual simplicity (in my opinion).

Prior art

- Didier Rémy's level-based generalization (see e.g. [1]).
- Dunfield & Krishnaswami's System F checker [2] and later extensions.
- Adam Gundry's thesis [3]
- Richard Eisenberg's thesis [4]
- But these are:
  - Rémy, Dunfield & Krishnaswami: not dependent enough, no type-level "let".
  - Gundry, Eisenberg: burdened with enormous Haskell-related complexity: phase distinction, separate coercion language, polymorphic subsumptions, etc. Also: not discussing efficient elaboration-time evaluation (both) or not having higher-order unification (Eisenberg).

Our own minimal setup: syntax

- Russell-style, types and terms are in the same syntactic sort.

```
A, B, t, u ::=
  λx. t | t u | (x : A) → B | U | let x = t : A in u

Γ, Δ, Σ ::= •                -- empty context
          | Γ , x = t : A  -- defined name
          | Γ , x : A      -- bound variable
          | Γ , x = ? : A  -- metavariable binding
```

- Metavariables are only distinguished by their context entries.
- Presyntax has holes in addition, but no typing/conversion rules or contexts.

```
A, B, t, u ::= ... | _
```

Rules (only non-standard parts here)

$$\frac{}{\Gamma \vdash (\mathtt{let}\ x = t : A\ \mathtt{in}\ u) \equiv u[x \mapsto t]}\ \text{let conversion}$$

$$\frac{}{\Gamma, x = t : A, \Delta \vdash x \equiv t}\ \delta\text{-conversion}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x = t : A \vdash u : B}{\Gamma \vdash (\mathtt{let}\ x = t : A\ \mathtt{in}\ u) : B}\ \text{let typing}$$

$$\frac{}{\Gamma, x = ? : A, \Delta \vdash x : A}\ \text{meta typing}$$

Otherwise, we have standard typing and conversion rules with type-in-type, implicit weakening & substitution.

Strengthening

- If Γ`, Δ ⊢ t : A` then we may write `Γ ⊢ t : A` if such strengthening is possible.
- Strengthening may involve unfolding definitions from `Δ`, or adding them as `let`-s.
- For example: if `Γ, x = t : A ⊢ x : B`, then two possible `Γ`-strengthenings of `x` are:
  - `Γ ⊢ t : B[x ↦ t]`
  - `Γ ⊢ (let x = t : A in x) : (let x = t : A in B)`
- We do some notation abuse by denoting strengthened terms with the same symbol as the original.
- In checking/elaboration algorithms, strengthening may fail and trigger overall failure.
- Strengthening over `let`-definitions is always possible.

Meta-operations

- $\Gamma \leq \Delta$ means that Γ can be updated to Δ via basic operations on metas.
- Stability: $\Gamma \leq \Delta$ and $\Gamma \vdash t : A$ implies $\Delta \vdash t : A$.
- Reflexive-transitive closure of three basic operations: new meta, strengthening, solution.

Basic meta-operations

1. New meta

$$\frac{\Gamma \vdash A : U \qquad x \text{ fresh in } \Gamma}{\Gamma \leq (\Gamma,\ x = ?\ : A)}$$

2. Strengthening

$$\frac{\Gamma,\ \Delta \vdash A : U \qquad \Gamma \vdash A : U}{(\Gamma,\ \Delta,\ x = ?\ : A,\ \Sigma) \leq (\Gamma,\ x = ?\ : A,\ \Delta,\ \Sigma)}$$

3. Solution

$$\frac{\Gamma \vdash t : A}{(\Gamma,\ x = ?\ : A,\ \Delta) \leq (\Gamma,\ x = t : A,\ \Delta)}$$

Judgements for bidirectional elaboration

1. Checking preterm `t` with `A : U`, returning term `u` in `Δ`, such that `Γ ≤ Δ` and `Δ ⊢ u : A`.

   ```
   Γ ⊢ t <= A ~> u ⊣ Δ
   ```

2. Inferring `A : U` for preterm `t`, returning term `u` in `Δ`, such that `Γ ≤ Δ` and `Δ ⊢ u : A`.

   ```
   Γ ⊢ t => A ~> u ⊣ Δ
   ```

3. Unifying `t` and `u` terms, returning `Δ` such that `Γ ≤ Δ` and `Δ ⊢ t ≡ u`. The sides must have the same type, but unification is *not* type-directed.

   ```
   Γ ⊢ t =? u ⊣ Δ
   ```

We consider everything up to definitional (≡) equality (evaluation unspecified).

Unification

- Standard, but no constraint postponing or advanced unification (lowering, pruning, etc.)
- The only interesting case is meta solution.
- Looking at first-order case, for simplicity:

$$\frac{}{\Gamma_0,\ \alpha = ?\ :\ A,\ \Delta_0 \vdash \alpha =?\ t \dashv\ ???,\ \alpha = t\ :\ A,\ ???}$$

- We need to strengthen `t` to Γ_0.
- But: `t` may contain unsolved metas from Δ_0.
- We need to strengthen these as well, by moving them before α in the context and recursively strengthening their types.
- We get output context $\Gamma_1,\ \alpha = t\ :\ A,\ \Delta_1$ after performing these strengthenings.
- This “solution strengthening” subsumes occurs and scope checking, and performs part of the “pruning” operation as used in Agda/Coq.

Unification (2)

- Example solution:

```
(A = ? : U, x : A, B = ? : U) ⊢
  A =? (B → B) ⊣
(B = ? : U, A = B → B : U, x : A)
```

- Pattern unification works as usual, but we need to also consider bound vars from the meta's scope for linearity.
- E. g. the following is non-linear:

```
Γ, x : A, α = ? : A → A ⊢ α x =? x
```

Elaboration

- When checking a hole, push a new meta to the context.
- Applications, variables handled in standard way.
- The interesting things happen at λ and Π binders.
- Trying to check λ:

$$\text{we need to get rid of this} \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\downarrow$$

$$\frac{\Gamma_0, x : A \vdash t <= B \sim> t' \dashv \Gamma_1, x : A, \Delta}{\Gamma_0 \vdash (\lambda\ x.\ t) <= ((x : A) \to B) \sim> (\lambda x.\ ???) \dashv ???}\ \lambda{<=}$$

We know that $\Gamma_1, x : A, \Delta \vdash t' : A$ and $\Gamma_0 \le \Gamma_1$, and we need to return a context Γ_2 such that $\Gamma_0 \le \Gamma_2$. We also know that Δ consists of definitions and unsolved metas (these are implied by $\Gamma_0, x : A \le \Gamma_1, x : A, \Delta$).

A possible solution

1. For each unsolved meta $(\alpha = ?\ :\ B)$ in Δ, insert a fresh meta $(\alpha' = ?\ :\ (x : A) \to B)$ before $(x : A)$, then solve to $(\alpha = \alpha'\ x : B)$ in Δ.
2. This yields new Δ' and Γ_2 such that $\Gamma_2, x : A, \Delta' \vdash t' : B$.
3. Now, Δ' consists only of definitions, so t' can be strengthened to $\Gamma_2, x : A \vdash t' : B$, and so $\Gamma_2 \vdash (\lambda\ x.\ t') : ((x : A) \to B)$.

$$\frac{\begin{array}{c}\Gamma_0, x : A \vdash t \Leftarrow B \rightsquigarrow t' \dashv \Gamma_1, x : A, \Delta \\ \text{Get } \Gamma_2 \text{ by step 1 above}\end{array}}{\Gamma_0 \vdash (\lambda\ x.\ t) \Leftarrow ((x : A) \to B) \rightsquigarrow (\lambda x.\ t') \dashv \Gamma_2}\ \lambda\Leftarrow$$

Example

We push new meta under the binder. Since it's not solvable locally, we generalize it over the binder.

```
                 ─────────────────────────────────────── _<=
                 x : U ⊢ _ <= U ~> α ⊣ x : U, α = ? : U
─────────────────────────────────────────────────────────────── λ<=
• ⊢ (λ x. _) <= (U → U) ~> λ x. (let α : U = α' x in α)
                   ⊣ α' = ? : (x : U) → U
```

- Analogously for Π binders.
- Most metas are created and solved as values, then dropped from the context.
- Unsolved metas acquire new function parameters on each generalization.
- Optimization: avoid creating intermediate metas by considering multiple λ or Π binders at once.

Let generalization

- Requires implicit binders.
- After inferring type for a generalizable `let`, convert all new local unsolved metas to implicit Π-s in the inferred type and implicit λ-s in the output term.
- Many choices, no principal types
  - Do we infer dependent or non-dependent function types?
  - Do we lift all unsolved metas all the way up, or can we generalize them sooner? E. g. elaborate `(λ x y. x)`
    to `(λ {A B : U}(x : A)(y : B).x)`
    or `(λ {A : U}(x : A){B : U}(y : B).x)`?
  - (Hindley-Milner outlaws cases like the second)

Unforced choices in elaboration

- Which metas to inline, which to `let`-define.
- Where to put `let`-s.
- General-purpose optimization passes can tidy up output.

Evaluation & implementation

- We would like to compute as much as possible by fast environment machines.
- But: we need at least two different forms of values
    1. Full weak head normal values: for type/conversion/occurrence checking.
    2. Less-than-fully reduced values: for sharing-preserving meta solutions, approximate (“syntactic”) conversion checks and pretty printing

Glued evaluation

- Idea: use an evaluator which computes two different forms of values.
- For example: call-by-need which also produces unreduced ("call-by-name") closures.
- The more efficient strategy drives the overall computation, but we also produce the other kind of values.

In Haskell, with de Bruijn levels

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

eval :: [Val] → [C] → Tm → Val
eval vs cs t = case t of
  Var i   → vs !! (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') (C cs u:cs') t'
    (VNe i vs' cs'  , u') → VNe i (u':vs') (C cs u:cs')
  Lam t   → VLam vs cs t
```

- During unification, both a `C` and a `Val` can be available for meta solution canditates.
- We could do approximate checks on `C`-s, then force `Val`-s if needed.
- Glued evaluation has modest time overhead compared to plain call-by-need.

Plan for prototype implementation

- Glued evaluation, with:
  1. Full whnf values, or “values”.
  2. Whnf values where definitions coming from the elaboration context are not unfolded (“local values”).
- Local values still use call-by-need for local redexes.
- Some information is lost compared to fully unreduced closures.
  - I have no conclusion yet on which one is better.
- Elaboration context contains at least four sub-contexts:
  1. Values of definitions
  2. Local values of definitions
  3. Binder types (in values)
  4. Binder types (in local values)

- We first try approximate ("syntactic") conversion/scope checks on local values, then switch to full unification/scope checking.
- We make sure do very limited evaluation in syntactic mode, because any work we do there is *not* shared.
- This is in contrast to the strategy in Ziliani & Sozeau's new Coq unifier guide [5], where they try to unify, then reduce, then try to unify again, and so on.
- This interleaved style has bad performance when solvable forms are many reductions away.
- .. and good performance if solvable forms are near, but whnf-s are far.
- My conjecture: when evaluation needs to be done, it's better to stop heuristic fiddling and do the serious evaluation (benchmarking will be the judge).

(meta)context implementation

- The naive one (which I implemented so far): linked lists for contexts, metas interleaved with non-metas.
- The “production strength” one:
  - Contexts are persistent vectors without metas.
    - With interleaved metas, moving meta entries around would be a de Bruijn apocalypse.
  - Meta entries are stored in a persistent vector which tells us which metas are inserted at given points in the context.
  - Metas have unique ID-s, and yet another structure maps the ID-s to their current position in the metacontext.
  - In terms, metavars carry their ID-s.
  - We need an extra final pass on the elaborated output to convert meta Var-s into regular Var-s.

Call-by-need modulo metacontext

- Meta solutions may cause whnf values to become out-of-date.
- E. g. a neutral term headed by an unsolved meta isn't whnf anymore after the meta is solved.
- We need to bring values up to date when doing type/conversion checking.
- Updating: check if value is neutral with a solved meta for head.
  - If yes, instantiate the meta and evaluate, then update again.
  - If no, return the value unchanged.
- Updating is fast and constant time on meta-free values.
- Updating is not shared computation in a straightforward Haskell implementation (hence we have a mix of call-by-need and call-by-name).

Call-by-need modulo metacontext (2)

- There's an interesting optimization available when we don't backtrack on meta solutions.
- If we put metacontexts in a usual State monad, if we force a call-by-need thunk, it is evaluated in the metacontext which we had at the time the thunk was created. Thus, values may be computed in an out-of-date state to begin with.
- Instead, we can arrange the implementation so that if we force a thunk, it uses the *current* metacontext for evaluation.
- We can switch between the two evaluation modes depending on whether destructive updates are allowed.

References

[1] O. Kiselyov, “How ocaml type checker works – or what polymorphism and garbage collection have in common.” [Online]. Available: http://okmij.org/ftp/ML/generalization.html.

[2] J. Dunfield and N. R. Krishnaswami, “Complete and easy bidirectional typechecking for higher-rank polymorphism,” in *ACM sigplan notices*, 2013, vol. 48, pp. 429–442.

[3] A. M. Gundry, “Type inference, haskell and dependent types,” PhD thesis, University of Strathclyde, 2013.

[4] R. A. Eisenberg, “Dependent types in haskell: Theory and practice,” *arXiv preprint arXiv:1610.07978*, 2016.

[5] B. ZILIANI and M. SOZEAU, “A comprehensible guide to a new unifier for cic including universe polymorphism and overloading,” *Journal of Functional Programming*, vol. 27, 2017.