Sharing-Preserving Elaboration for Dependent Type Theories

András Kovács

ELTE, Department of Programming Languages and Compilers

Jan 12, 2018

What is elaboration?

- Input: program with omitted parts.
- Output: program with omitted parts inferred from provided parts.
- Omitted parts are usually boilerplate or "boring" in some ways.
- Usually type inference is the main part of elaboration.
- In dependently typed settings, it's necessarily type inference + program inference.

Example

```
-- Input
id : {A : Set} → A → A
id = λ x → x

id2 : {A : Set} → A → A
id2 = λ x → id x

-- Output
id : {A : Set} → A → A
id = λ {A} x → x

id2 : {A : Set} → A → A
id2 = λ {A} x → id {A} x
```

Traditional implementation

1. Create a metavariable (“meta”) for each program hole.
2. Solve equations involving metas.
3. Plug meta solutions into the output program.

Traditionally, metas are represented as topmost-level functions, and their solutions can depend on local bound variables but not on any defined names or other metas.

Problems with unscoped metas

- Classic Hindley-Milner exponential-time cases

```
-- Input
badId : {A : Set} → A → A
badId = id id id id

-- Output
badId : {A : Set} → A → A
badId = λ {A} → (id {((A → A) → A → A) → (A → A) → A → A})
                (id {(A → A) → A → A})
                (id {A → A})
                (id {A})
```

A better (linear) output

```
notSoBadId : {A : Set} → A → A
notSoBadId = λ {A} →
  let m1 = A
      m2 = m1 → m1
      m3 = m2 → m2
      m4 = m3 → m3
  in (id {m4}) (id {m3}) (id {m2}) (id {m1})
```

- In plain Hindley-Milner, exponential cases are rare in practice, types are small, and very little computation happens in types.
- With dependent types, sharing of structure and computation becomes more important.
- The general solution needs to track binding locations of metas and move them around in scope if necessary.
- Algorithm is a generalization of
  - Rémy's level-based inference for OCaml (Hindley-Milner) (see e. g. [1])
  - Krishnaswami & Dunfield's [2] bidirectional System F checker
- Also allows efficient let-generalization with dependent types.
- Could be also a principled basis for numerous other optimizations.

[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.