Programming language techniques for proof assistants

Lecture 3
Holes and unification

Andrej Bauer
University of Ljubljana

International School on Logical Frameworks and Proof Systems Interoperability
Université Paris–Saclay, September 8–12, 2025

Overview

- Lecture 1: From declarative to algorithmic type theory
- Lecture 2: A monadic type checker
- Lecture 3: Holes and unification
  - Postponed computations as holes
  - Unification
  - A holey type checker
- Lecture 4: Variables as computational effects

Lecture 3

Holes and unification

What are holes?

- A **hole** is an unfinished or missing piece of formalization.
- A hole is eventually **filled in**.
- A hole does *not* block the proof checker.
- Examples:
  - Interactive holes: created and filled in by the user
  - Implicit arguments: created by type checker, filled in by unification
- Holes may be created by the user or by the machine:
  - The user types `?` in Agda to create an interactive hole
  - The user types `_` to create a hole that should be filled automatically
  - The type-checking algorithm inserts holes in place of implicit arguments

Holes as meta-variables

- In type theory, we represent holes as **meta-variables**.
- When a hole is made we introduce a fresh meta-variable:
  - When a meta-variable is created, we must know its type.
  - Consequently, holes may appear only in *checking* positions.
- When a hole is filled with a term, we substitute it for the meta-variable:
  - Caveat: the filling term may be discovered in a context that is different from that of the meta-variable.

Example

1. Can we fill the hole `?H`?

   `λ (A : Type) (a : A) ⇒ (λ (x : ?H) ⇒ x) a`

   Answer: yes, `?H` must be `A`.

2. Can we fill the hole `?G`?

   `λ (x : ?G) (A : Type) (f : A → Type) ⇒ f x`

   Answer: no, `?G` must be `A`, which is out of scope.

Meta-variables and contexts

Out strategy:

- Meta-variables have *closed* types.
- They must be filled with *closed* terms.

Example:

- Consider `λ (A : Type)(a : A)⇒ (λ (x : ?H) ⇒ x) a`.
- The hole `?H`, appears in context $A : \mathsf{Type}, a : A$ and has type Type.
- We introduce a meta-variable H of type $\Pi_{(A' : Type)} \Pi_{(a' : A')} \mathsf{Type}$.
- We replace `?H` with $H\,A\,a$ to obtain $\lambda(A : \mathsf{Type}).\, \lambda(a : A).\, (\lambda(x : H\,A\,a).\, x)a$
- Unification will solve $H\,A\,a \equiv_{\mathsf{Type}} A$ to give $H \equiv \lambda(A : \mathsf{Type}).\, \lambda(a : A).\, A$.

Implementation

- lib/core/TT.ml:

```
type tm =
  | Var of var
  | Meta of var
  | Let of tm * ty * tm binder
  | Type
  | Prod of ty * ty binder
  | Lambda of ty * tm binder
  | Apply of tm * tm
```

- lib/core/context.ml:

```
type t =
  { idents : TT.var IdentMap.t
  ; vars : (TT.tm option * TT.ty) VarMap.t
  ; locals : TT.var list
  ; metas : (TT.tm option * TT.ty) VarMap.t
  }
```

The context monad

- Variables behave like a reader monad
- Meta-variables behave like the state monad
- The new monad:

```
type 'a m = t -> t * 'a

module Monad =
struct
  let ( let* ) c1 c2 ctx =
    let ctx, v1 = c1 ctx in
    c2 v1 ctx

  let ( >>= ) = ( let* )

  let return v t = (t, v)
```

Unification

- Proceed as in our original equality checking algorithm.
- During normalization phase, suppose we encounter $M\, e_1 \cdots e_n \equiv_A e$ where:
  - M is an unsolved meta-variable,
  - e_i's normalize to *distinct* variables x_i's,
  - $\mathsf{FV}(e) \subseteq \{x_1, \ldots, x_n\}$.
- Then we may set $M := \lambda(x_1 : A_1). \cdots \lambda(x_n : A_n).\, e$.

Dirty details

- Top level must complain if not all meta-variables are solved. Or does it?
- What about local definitions?
- A meta-variable need not be closed, as it may refer to previously declared and defined constants.
- We get ugly answers with β-redexes:

```
# infer λ (A : Type) (a : A) ⇒ (λ (x : ?H) ⇒ x) a
λ (A : Type) ⇒ λ (a : A) ⇒
  (λ (x : (λ (A1 : Type) ⇒ λ (a1 : A1) ⇒ A1) A a) ⇒ x) a
    : Π (A : Type), Π (a : A), (λ (A1 : Type) ⇒ λ (a1 : A1) ⇒ A1) A
        a
```

Improvements

- Solve more equations.
- Normalize away the ugly β-redexes in solutions.
- Refine meta-variables to product types when necessary:

```
# infer λ (A : Type) (a : A) (f : ?H) ⇒ f (f a)
Typechecking error at line 1, characters 38-45:
this expression should be a function but has type ?H A a
```

Where to go from here?

Learn from the masters:

- András Kovács: https://github.com/AndrasKovacs/elaboration-zoo