Term Inference in Dedukti

Gaëtan Gilbert
supervised by Arnaud Spiwack
Inria

September 8, 2015

Introduction

$\lambda-$calculus:

$$\lambda x\ y.x$$

With simple types:

$$\vdash \lambda x\ y.x : A \to B \to A$$

Introduction

$\lambda-$calculus:

$$\lambda x\ y.x$$

With simple types:

$$\vdash \lambda x\ y.x : A \to B \to A$$

Dependent types ($\lambda\Pi-$calculus):

$$n : nat \vdash cons\ 0\ n\ nil : vector\ (0 + 1)$$

Introduction

$\lambda-$calculus:

$$\lambda x\ y.x$$

With simple types:

$$\vdash \lambda x\ y.x : A \to B \to A$$

Dependent types ($\lambda\Pi-$calculus):

$$n : nat \vdash cons\ 0\ n\ nil : vector\ (0 + 1)$$

$\lambda\Pi-$calculus modulo:
Declare $(n + m) + p \equiv n + (m + p)$ then with

$$append : \forall n\ m.vector\ n \to vector\ m \to vector\ (n + m)$$

the following have the same type:

$$append\ (n + m)\ p\ (append\ n\ m\ a\ b)\ c : vector\ ((n + m) + p)$$

$$append\ n\ (m + p)\ a\ (append\ m\ p\ b\ c) : vector\ (n + (m + p))$$

Introduction

Conversion:

$$vector\ ((n + m) + p) \equiv ?vector\ (n + (m + p)$$

Unification:

$$vector\ (\_ + p) \equiv ?vector\ ((n + m) + \_)$$

Inference:

$$append\ \_\ \_\ a\ nil : \_$$

becomes

$$append\ n\ 0\ a\ nil : vector\ (n + 0)$$

Introduction

Conversion:

$$vector\ ((n + m) + p) \equiv ?vector\ (n + (m + p)$$

Unification:

$$vector\ (\_ + p) \equiv ?vector\ ((n + m) + \_)$$

Inference:

$$append\ \_\ \_\ a\ nil : \_$$

becomes

$$append\ n\ 0\ a\ nil : vector\ (n + 0)$$

- Simple types: all decidable.
- Dependent types: unification undecidable, type inference decidable with annotations ($\lambda x : A.t$).
- Goal: investigate separation of concerns in inference algorithms, through adding one to Dedukti.

Table of Contents

1 Introduction

2 $\lambda\Pi$ Calculus modulo

3 Monads

4 Elaboration

5 Unification primitives

6 Heuristics

7 Conclusion

Conclusion

$\lambda\Pi$ Calculus modulo

$$\frac{}{\Gamma \vdash \mathit{Type} : \mathit{Kind}}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{c : A \in \Sigma}{\Gamma \vdash c : A}$$

$$\frac{\Gamma \vdash t : T \qquad T \rhd \Pi x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x \leftarrow u]}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash t : B \qquad B \neq \mathit{Kind}}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.B : s}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \equiv B}{\Gamma \vdash t : B}$$

Monads

```
type 'a t
val return : 'a -> 'a t
val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
val run : 'a t -> 'a
```

Backtracking State Monad

Operations:

```
val get : state t
val set : state -> unit t

val plus : 'a t list -> 'a t
val fail : 'a t
```

Example:

```
set 1 >>= fun () ->
plus
  [ set 0 >>= fun () -> fail
  ; get >>= fun n -> print n ]
```

Partial terms

Definition (Partial terms)

Partial terms are terms where subterms can be the placeholder term _.

Examples:

- $\lambda x : \_.x$ an identity function for some type left to be inferred.
- $(\lambda x : \_.x\ x)\ (\lambda x : \_.x\ x)$ the classical bad term.
- *plus* _ 1 0 a use case for a polymorphic group operator
  $plus : \Pi(G : group).gtype\ G \to gtype\ G \to gtype\ G$.

Extended terms

Definition (Extended terms)

Extended terms are terms where subterms may be

- a metavariable $?_i[\sigma]$ where $?_i$ is its name and $\sigma = [x_1 \leftarrow t_1, ..., x_n \leftarrow t_n]$ a substitution.
- a guarded term $p[\sigma]\ t$ where p is its name, σ a substitution and t an extended term.

Metavariables are placeholders with additional information.
Guards are used for typing to deal with constraints.

Elaboration

Elaboration is the process taking a partial term and

- replacing placeholders with metavariables
- generating unification constraints attached to guards
- generating a prospective type (which is an extended term)

Judgements:

$$\Gamma \vdash t \Uparrow u : T$$

$$\Gamma \vdash t : T \Downarrow u$$

In pseudocode they're monadic functions

```
val infer : context -> partial term ->
  (extended term * extended term) t
val check : context -> partial term ->
  extended term -> extended term t
```

The monad provides a state Θ (unification problem), with information about metavariables and guards, with backtracking.

Elaboration

Translate the typing rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Uparrow x : A}$$

$$\frac{}{\Gamma \vdash \mathit{Type} \Uparrow \mathit{Type} : \mathit{Kind}}$$

$$\frac{c : A \in \Sigma}{\Gamma \vdash c \Uparrow c : A}$$

$$\frac{\Gamma \vdash a : \mathit{Type} \Downarrow A \qquad \Gamma, x : A \vdash t \Uparrow u : B \qquad B \neq \mathit{Kind}}{\Gamma \vdash \lambda x : a.t \Uparrow \lambda x : A.u : \Pi x : A.B}$$

$$\frac{\Gamma \vdash a : \mathit{Type} \Downarrow A \qquad \Gamma, x : A \vdash b \Uparrow B : s \qquad s \text{ sort}}{\Gamma \vdash \Pi x : a.b \Uparrow \Pi x : A.B : s}$$

$$\frac{\Gamma \vdash f \Uparrow g : T \qquad T \rhd \Pi x : A.B \qquad \Gamma \vdash t : A \Downarrow u}{\Gamma \vdash f\ t \Uparrow g\ u : B[x \leftarrow u]}$$

Elaboration

$$\frac{\Gamma \vdash f \Uparrow g : T \qquad T \rhd \Pi x : A.B \qquad \Gamma \vdash t : A \Downarrow u}{\Gamma \vdash f\ t \Uparrow g\ u : B[x \leftarrow u]}$$

In pseudocode:

```
let infer ctx (f t) = infer ctx f >>= fun (g,T) ->
  whnf T >>= function
  | Pi (x:A) B -> check ctx t A >>= fun u ->
      return ((g u), B[x <- u])
  | _ -> fail
```

Guards

Guards let us translate the conversion rule into an elaboration rule:

$$\frac{\Gamma \vdash t \Uparrow t' : A \qquad \mathit{NewGuard}(\Gamma \vdash p : A \to B)}{\Gamma \vdash t : B \Downarrow p[id_\Gamma]\ t'}$$

Metavariables

Placeholders become metavariables:

$$\frac{\mathit{NewMeta}(\Gamma \vdash ?_j : *) \qquad \mathit{NewMeta}(\Gamma \vdash ?_i : ?_j[id_\Gamma])}{\Gamma \vdash \_ \Uparrow ?_i[id_\Gamma] : ?_j[id_\Gamma]}$$

In pseudocode:

```
let infer ctx _ = new_meta ctx MType >>= fun mty ->
  new_meta ctx (MTyped mty) >>= fun m ->
  return (m, mty)
```

Unification problem

Definition

A unification problem is given by

- metavariable declarations $\Gamma \vdash ?_i : T$ (also type metavariable and sort metavariable versions)
- guard declarations $\Gamma \vdash p : A \to B$ each associated with a list of constraints $\Delta \vdash t \equiv u$
- metavariable definitions $?_i := t$

Guards are pass-through when they have no associated constraints: $p[\sigma]\ t \equiv t$.
Metavariables are convertible with their definitions: $?_i[\sigma] \equiv \sigma(t)$ when $?_i := t$.

Unification primitives

Adding to the problem:

- *NewMeta*($\Gamma \vdash ?_i : T$) when $\Gamma \vdash T : *$ (also type metavariable and sort metavariable versions).
- *NewGuard*($\Gamma \vdash p : A \to B$) when A and B are types or sorts under Γ, with initial constraint list $[\Gamma \vdash A \equiv B]$.
- *Define*($?_i := t$) when $?_i$ is declared as $\Gamma \vdash ?_i : T$ without a definition and $\Gamma \vdash t : T$,
  AND occurs check: no defining $?_i := ?_j + ?_k$ and $?_j := ?_i$.

Unification primitives

Refinement:

- $\mathit{Narrow}(?_i, \Delta)$: when $\Gamma \vdash ?_i : T$ declared not defined, and $\Delta \subseteq \Gamma$ such that $\exists \Delta', \Delta' \subseteq \Delta$ and $\Delta' \vdash T : *$, $\mathit{NewMeta}(\Delta' \vdash ?_j : T)$ then $\mathit{Define}(?_i := ?_j[id_{\Delta'}])$.
- $\mathit{Refine}(?_i := t)$: when $\Gamma \vdash ?_i : B$ declared not defined and $\Gamma \vdash t : A$, $\mathit{NewGuard}(\Gamma \vdash p : A \to B)$ then $\mathit{Define}(?_i := p[id_\Gamma]\ t)$.

Unification primitives

Transform constraints:

- $\Gamma \vdash A \equiv B$ when $A \equiv B$: can be removed.
- $\Gamma \vdash A \equiv B$ with $A \rhd A'$ and $B \rhd B'$: can be replaced with $\Gamma \vdash A' \equiv B'$.
- $\Gamma \vdash \lambda x : A.t \equiv \lambda y : B.u$: can be replaced with $\Gamma \vdash A \equiv B$ AND $\Gamma, x : A \vdash t \equiv u[y \leftarrow x]$.
- as above with Π and applications.

Heuristics

```
val get_constraint : constraint t
val refine : meta -> extended term -> unit t
val constraint_solved : unit t

let trivial_rule = function
  | ctx, ?i[] == t -> refine ?i t
  | _ -> fail

let rec solve = get_constraint >>= function
  | Some c -> plus
      [ constraint_solved
      ; (trivial_rule c)
      ; ... ] >>= fun () ->
      solve
  | None -> return ()
```

Heuristics

Some rules:

$$\frac{\Gamma \vdash t \equiv u \qquad \Gamma \vdash t_1 \equiv u_1 \qquad \ldots \qquad \Gamma \vdash t_n \equiv u_n}{\Gamma \vdash t\ t_1 \ldots t_n \equiv u\ u_1 \ldots u_n}$$

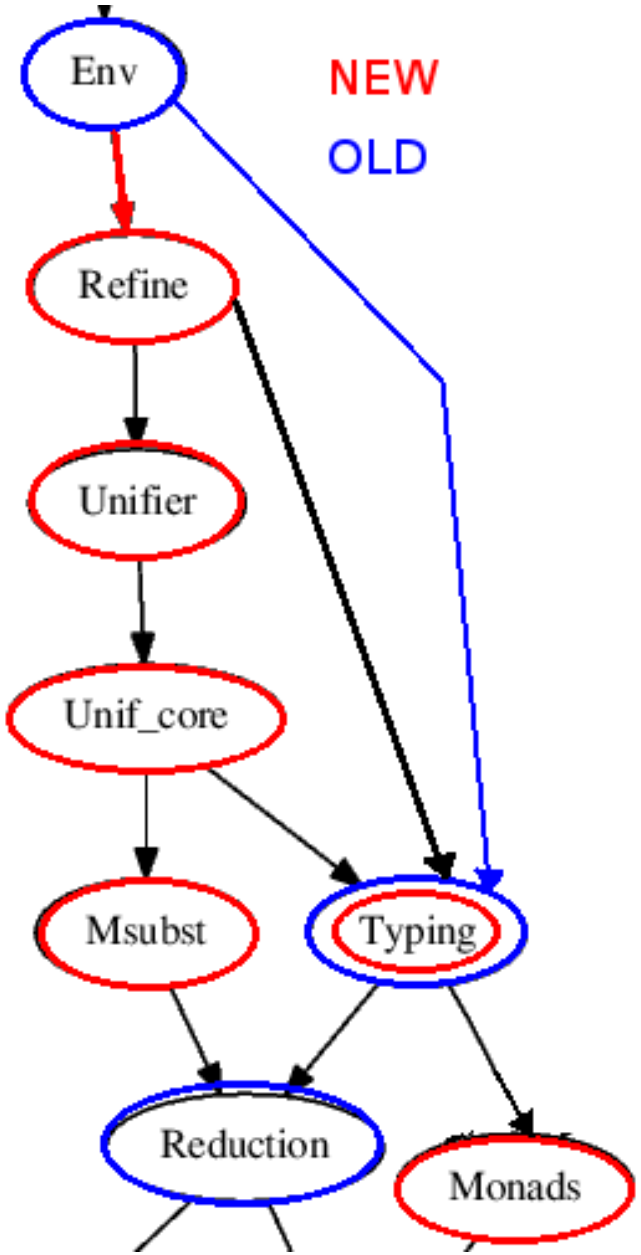
$$\frac{\mathit{Narrow}(?_i, \{x, \sigma(x) = \rho(x)\})}{\Gamma \vdash ?_i[\sigma] \equiv ?_i[\rho]}$$

$$\frac{\sigma \text{ invertible} \qquad \mathit{Refine}(?_i, \sigma^{-}1(t))}{\Gamma \vdash ?_i[\sigma] \equiv t}$$

(other rules not displayed)

Rules adapted from Beta Ziliani and Matthieu Sozeau, *A Predictable Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading*

Code Organisation



Conclusion

- Separation of concern embodied
  - in the theory: elaboration / safe primitives / heuristics
  - in the code: only safe interfaces exported
- Experimental validation:

```
transport (A : _) (P : _)
  (x : U A) (y : _) (e : _)
  (H : U (P x)) : U (P y)
 := eq_rect _ x (a : _ => p : _ => P a) H _ e.
```

  Possible termination bug to investigate.
- Future work:
  - placeholders on the left hand side of a rewrite rule: $P\ \_ \longrightarrow t$
  - high level features: implicit arguments, coercions, user provided unification hints, etc.