A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading

Beta Ziliani

Max Planck Institute for Software Systems
(MPI-SWS)

Matthieu Sozeau

Inria & PPS, Université Paris Diderot

ICFP'15, Vancouver, September 1st 2015

Prof. Zed and his students Alice and Bob

Ass 1: Write in Coq a function that maps 0 to 0.

Definition alices : $(\lambda x.\ \_)\ 0 = 0$:= eq_refl.

$\Rightarrow$ Type-checks, setting _ to x.

Prof. Zed and his students Alice and Bob

Ass 1: Write in Coq a function that maps 0 to 0.

Definition alices : $(\lambda x.\ \_)\ 0 = 0$:= eq_refl.

$\Rightarrow$ Type-checks, setting _ to x.

Definition bobs : $\_\ 0 = 0$:= eq_refl.

$\Rightarrow$ Fails

Both problems have NO most general unifier.

Another assignment

Ass 2: Given:

in_head : $a \in (a :: l)$

in_tail : $a \in l \to a \in (b :: l)$

inR : $a \in r \to a \in \mathsf{append}\ l\ r$

inthelist := $\forall\, x\ y\ z : \mathsf{nat}.\ x \in \mathsf{append}\ [y]\ [z; x]$

Another assignment

Ass 2: Given:

$$\text{in\_head} : a \in (a :: l)$$
$$\text{in\_tail} : a \in l \to a \in (b :: l)$$
$$\text{inR} : a \in r \to a \in \text{append } l\ r$$

$$\text{inthelist} := \forall\, x\ y\ z : \text{nat}.\ x \in \text{append } [y]\ [z; x]$$

Prove inthelist in Coq or Agda:

Definition alices2 : inthelist := $\lambda x\ y\ z.$ inR (in_tail in_head)

Another assignment

Ass 2: Given:

$$\text{in\_head} : a \in (a :: l)$$
$$\text{in\_tail} : a \in l \to a \in (b :: l)$$
$$\text{inR} : a \in r \to a \in \text{append } l\ r$$

$$\text{inthelist} := \forall\, x\ y\ z : \text{nat}.\ x \in \text{append } [y]\ [z; x]$$

Prove inthelist in Coq or Agda:

Definition alices2 : inthelist := $\lambda x\ y\ z.$ inR (in_tail in_head)

$\Rightarrow$ Type-checks in Coq, fails in Agda

??

Unification

- A central component in interactive proof assistants.
  1. Type inference (overloading, coercions), refinement.
  2. Tactic application, rewriting...
- Not discussed in Coq's documentation!
- A hard (undecidable) problem. Tackled with heuristics in Coq.

An old problem

- Unification modulo $\beta\eta$ (Elliot '89).
- HO-Pattern unification: decidable fragment (Huet '91).
- HO-Pattern for CoC (Pfenning '91).
- Unification modulo δ (Pfenning & Schrmann '98).
- Unification modulo $\beta\eta$ for Π and Σ (Abel & Pientka '11).

Our contribution

Theory:

- **Formalization** of a new unification algorithm.
- Including the **whole** language and Coq's features.
  - Overloading.
  - Universe polymorphism.
  - No constraint postponement (Reed '09).
  - Refinements of existing heuristics.

Implementation:

- A plugin: **Unicoq**.
- Solving 99.9% of cases in existing libraries – "Mathematical Components" included (≈10M queries).
  - In all benchmarks, a negligible amount of typing annotations needed.
  - Not optimized, obviously faithful to the rules.

A taste of the algorithm

- First-order approximation:

$$\frac{t \approx u}{f\ t \approx f\ u} FO$$

  Even when f is a defined, unfoldable constant.

- Overloading:

$$\mathsf{eq} : \forall e : \mathsf{eqType}, \mathsf{sort}\ e \to \mathsf{sort}\ e \to \mathsf{bool}$$

$$\mathsf{eq}\ \_\ \mathsf{true}\ \mathsf{false} \rightsquigarrow\ \ \mathsf{eq}\ \mathsf{eqType\_bool}\ \mathsf{true}\ \mathsf{false}$$

- Universe polymorphism, dependency deletion (see paper).

FO-Approximation

$$\text{inR} : a \in r \to a \in \text{append } l\ r$$

$$\text{inthelist} := \forall\, x\ y\ z : \text{nat}.\ x \in \text{append } [y]\ [z; x]$$

Definition alices2 : inthelist := $\lambda x\ y\ z.$ inR (in_tail in_head)

$$x \in \text{append } [y]\ [z; x] \approx ?a \in \text{append } ?l\ (?b :: ?a :: ?r)$$

"Natural" solution:

$$?a := x, ?l := [y], ?b := z, ?r := []$$

.

FO-Approximation with backtracking

$$\mathsf{inR} : a \in r \to a \in \mathsf{append}\ l\ r$$

$$\mathsf{inthelist} := \forall\, x\ y\ z : \mathsf{nat}.\ x \in \mathsf{append}\ [y]\ [z; x]$$

Definition alices2.2 : inthelist :=

$$\lambda x\ y\ z.\ \mathsf{inR}\ (l := [])\ (\mathsf{in\_tail}\ (\mathsf{in\_tail}\ \mathsf{in\_head}))$$

$$?a \in \mathsf{append}\ []\ (?c :: ?b :: ?a :: ?r) \approx x \in \mathsf{append}\ [y]\ [z; x]$$

Canonical Structures 101

- Designed to support overloading à la Haskell type classes.
- Equality class (called a structure, i.e. dependent record):

$$\textbf{Structure}\ \overbrace{\mathsf{eqType}}^{\text{name}} := \overbrace{\mathsf{EqType}}^{\text{constructor}}\ \overbrace{\{\ \mathsf{sort} : \mathsf{Type};\ \mathsf{equal} : \mathsf{sort} \to \mathsf{sort} \to \mathsf{bool}\}}^{\text{fields}}$$

- Generates several projection functions as well:

$$\begin{array}{lcl} \mathsf{sort} & : & \mathsf{eqType} \to \mathsf{Type} \\ \mathsf{equal} & : & \forall T : \mathsf{eqType}.\ \mathsf{sort}\ T \to \mathsf{sort}\ T \to \mathsf{bool} \end{array}$$

Canonical Instances

$$\textbf{Canonical}\ \mathsf{eqType\_bool} := \mathsf{EqType}\ \overbrace{\mathsf{bool}}^{\mathsf{sort}}\ \overbrace{\mathsf{eq\_bool}}^{\mathsf{equal}}$$

$$\textbf{Canonical}\ \mathsf{eqType\_pair}\ (A\ B : \mathsf{eqType}) :=$$
$$\mathsf{EqType}\ \underbrace{(\mathsf{sort}\ A \times \mathsf{sort}\ B)}_{\mathsf{sort}}\ \underbrace{(\mathsf{eq\_pair}\ A\ B)}_{\mathsf{equal}}$$

where

$$\mathsf{eq\_bool}\ x\ y := (x\ \&\&\ y)\ ||\ (!x\ \&\&\ !y)$$

$$\mathsf{eq\_pair}\ (A\ B : \mathsf{eqType})\ (u\ v : \mathsf{sort}\ A \times \mathsf{sort}\ B) :=$$
$$\mathsf{equal}\ (\pi_1\ u)\ (\pi_1\ v)\ \&\&\ \mathsf{equal}\ (\pi_2\ u)\ (\pi_2\ v)$$

Triggering canonical instance resolution

$$\text{sort}\ ?X \approx T$$

Match T against the sort field of canonical instances of eqType, recursively.

Triggering canonical instance resolution

$$\mathsf{sort}\ ?X \approx T$$

Match T against the sort field of canonical instances of eqType, recursively.

Example:

$$\mathsf{equal}\ (b_1, b_2)\ (c_1, c_2)$$

where b_1, b_2, c_1, c_2 : bool.
This generates:

$$\mathsf{sort}\ ?T \approx (\mathsf{bool} \times \mathsf{bool})$$

Triggering canonical instance resolution

$$\text{sort } ?X \approx T$$

Match T against the sort field of canonical instances of eqType, recursively.

Example:

$$\text{equal } (b_1, b_2)\ (c_1, c_2)$$

where b_1, b_2, c_1, c_2 : bool.
This generates:

$$\text{sort } ?T \approx (\text{bool} \times \text{bool})$$

Using the canonical instances, the solution is:

$$?T := \text{eqType\_pair eqType\_bool eqType\_bool}$$

Programming inference with canonical structures

- Unification and hence resolution *backtracks* on unfolding.
- A documented implementation was lacking.
- Essential use in Ssreflect for controlled overloading, disambiguation, small scale proof search, where the *unfolding strategy* matters.

Constraint postponement: why not?

Produces more m.g.u's, BUT:

- We embraced useful heuristics, this precludes m.g.u's.
- Postponement implies less predictable algorithmic behavior when programming unification.
- Global unification failures are harder to explain and understand.
- Complex correctness proof.

The Greek

Summary

Contributions:

- A formalized unification algorithm, combining overloading, universe polymorphism and FO approximation.
- A faithful, realistic prototype implementation.

Future work:

- Bi-directional type inference.
- Mechanized proof of soundness.
- Optimize: caching, unfolding heuristics.

Thanks

github.com/unicoq