Implicit generalization in Agda

Péter Diviánszky

The project was supported by the European Union, co-financed by the European Social Fund
EFOP-3.6.3-VEKOP-16-2017-00002

AIMXXVIII
Nottingham, 17 October 2018
Implicit generalization in Agda

Péter Diviánszky

Introduction

Preparations

Existing solutions

Extensions

Implementation

Other
Introduction
A) Implicit generalization

It would be nice if this was valid Agda code (issue #1706):

```agda
data _∈_ : A → List A → Set where
  hd : x ∈ x :: xs
  tl : x ∈ xs → x ∈ (y :: xs)

▶ looks so natural that the issue was misread as a non-issue
▶ Coq, Lean, Idris, Isabelle, Haskell, ML, ... already support this
```
B) Declared variables

Notation is not arbitrary

- $\Gamma$, $\Delta$ usually denote contexts

Mental map for each coherent document:

\[
\begin{align*}
\text{name}_1 : \text{type}_1 \\
\text{name}_2 : \text{type}_2 \\
\text{...}
\end{align*}
\]

Let’s (partially) declare this mapping in Agda!

```
postulate
  Con : Set
```

```
variable -- new keyword
  $\Gamma$ $\Delta$ : Con
```
A+B) Implicit generalization of declared variables

1. declare variables
2. implicitly generalize over declared variables

This has a long tradition in scientific papers.

⇒ we can get closer to human language in a *formal* way
Example

postulate
   Con : Set
   Sub : Con → Con → Set

variable
   Γ Δ Θ : Con

postulate
   id : Sub Γ Γ
   _∘_ : Sub Θ Δ → Sub Γ Θ → Sub Γ Δ

-- id : {Γ : Con} → Sub Γ Γ
-- _∘_ : {Γ Δ Θ : Con} → Sub Θ Δ → Sub Γ Θ → Sub Γ Δ

Note that separate variables are introduced for each type signature.
General user experience

Trade-off between compactness and details:

▶ one can focus on the essentials
▶ the definitions should be “decompressed”

Easy to get used to it because it has a long history in publications.
Implicit generalization in Agda

Péter Diviánszky

Introduction

Preparations

Existing solutions

Extensions

Implementation

Other

Preparations
Considerations (most difficult first)

- treatment of metavariables
- nested variables
  (variables in the type signature of variables)
- naming of generalized metavariables
- ordering and placing of generalized parameters
- how it behaves across module boundaries
Treatment of metavariables #1

postulate
  Con : Set
  Ty : Con → Set
  Sub : Con → Con → Set
  _▹_ : (Γ : Con) → Ty Γ → Con

variable
  Γ Δ : Con
  A : Ty _  -- note the underscore here

postulate
  π₁ : Sub Γ (Δ▹A) → Sub Γ Δ
  -- π₁ : {Γ Δ : Con}{A : Ty Δ} → Sub Γ (Δ▹A) → Sub Γ Δ
  -- -- the metavariable was solved with Δ

Note that separate metavariables are introduced for each type signature.
Treatment of metavariables #2

Unsolved metavariables coming from variable are generalized too:

postulate
   Con : Set
   Sub : Con \rightarrow Con \rightarrow Set

variable
   \sigma \delta \nu : Sub \_ \_ \_ -- metavariables: \sigma.1, \sigma.2, \delta.1, \delta.1, \nu.1, \nu.2

postulate
   ass : (\sigma \cdot \delta) \cdot \nu \equiv \sigma \cdot (\delta \cdot \nu)

   -- ass : \{\sigma.1 \sigma.2 \delta.1 \nu.1 : Con\}
   -- \{\sigma : Sub \sigma.1 \sigma.2\}{\delta : Sub \delta.1 \sigma.1}\{\nu : Sub \nu.1 \delta.1\}
   -- \rightarrow (\sigma \cdot \delta) \cdot \nu \equiv \sigma \cdot (\delta \cdot \nu)
   -- note that \delta.2 was solved with \sigma.1; \nu.2 was solved with \delta.1

Let’s call *generalizable metavariables* the metavariables coming from variable.
Treatment of metavariables #3

data Vec (A : Set) : Nat → Set where

variable
  A : Set
  x : A
  n m : Nat
  xs : Vec A m

postulate
  IsHead : A → Vec A (suc n) → Set
  -- IsHead : {A : Set}{n : Nat} → A → Vec A (suc n) → Set

  foo : IsHead {n = _} x xs → Nat
  -- foo : {A : Set} {x : A} {n₁ : Nat} {xs : Vec A (suc n₁)}
  --   → IsHead x xs → Nat

n₁, the metavariable introduced by the underscore was not
generalizable, but we generalized it because m, a generalizable
meta was solved with suc n₁.
Nested variables

variable
  \( \ell : \text{Level} \quad \text{-- let } \ell \text{ denote a level} \)
  \( A : \text{Set } \ell \quad \text{-- let } A \text{ denote a set at (a) level } \ell \text{ (for all } \ell) \)

postulate
  \( f : A \rightarrow \text{Set } \ell \)

Three possible meanings:

A) “let A denote a set at level \( \ell \)”
  \( f : \{\ell : \text{Level}\} \{A : \text{Set } \ell\} \rightarrow A \rightarrow \text{Set } \ell \)

B) “let A denote a set at a level \( \ell \)”
  \( f : \{\ell \, \ell' : \text{Level}\} \{A : \text{Set } \ell'\} \rightarrow A \rightarrow \text{Set } \ell \)

C) “let A denote a set at level \( \ell \) for all \( \ell \)”
  \( f : \{\ell : \text{Level}\} \{A : \{\ell : \text{Level}\} \rightarrow \text{Set } \ell\} \rightarrow A \rightarrow \text{Set } \ell \)

The current implementation follows B)
Naming of generalized metavariables

Name hints (either of them works, the second is stronger):

- general name hints for the parameters of Sub:

  \texttt{postulate} \quad \text{Sub} : (\Gamma : \text{Con})(\Delta : \text{Con}) \rightarrow \text{Set}

- name hints for metas in the type of \( \sigma, \delta \) and \( \nu \):

  \text{variable} \quad \sigma \ \delta \ \nu : \text{Sub} \ \Gamma \ \Delta

  -- variables in type of variables are used for name hinting

\texttt{postulate}

  \text{ass} : (\sigma \circ \delta) \circ \nu \equiv \sigma \circ (\delta \circ \nu)

  -- \text{ass} : \{\sigma.\Gamma \ \sigma.\Delta \ \delta.\Gamma \ \nu.\Gamma : \text{Con}\}

  -- \{\sigma : \text{Sub} \ \sigma.\Gamma \ \sigma.\Delta\} \{\delta : \text{Sub} \ \delta.\Gamma \ \sigma.\Gamma\} \{\nu : \text{Sub} \ \nu.\Gamma \ \delta.\Gamma\}

  -- \rightarrow (\sigma \circ \delta) \circ \nu \equiv \sigma \circ (\delta \circ \nu)

Hierarchical names are used to track the “source” of the metavariables.
Questions about naming

-- ass : {σ.Γ σ.Δ δ.Γ ν.Γ : Con}
-- {σ : Sub σ.Γ σ.Δ} {δ : Sub δ.Γ σ.Γ} {ν : Sub ν.Γ δ.Γ}
-- → (σ ∘ δ) ∘ ν ≡ σ ∘ (δ ∘ ν)

Questions:

▶ Should it be possible to give generalised metavariables by name?

ass {σ.Γ = Γ₁} e  -- not allowed currently

ass {_} {_} {_} {Γ₂} e -- giving `ν.Γ` by position is too brittle

▶ The algorithm currently chooses one hierarchical name. Should all of them be allowed when giving arguments by name?

ass {δ.Δ = Γ₁} e  -- instead of {σ.Γ = Γ₁}
Ordering of generalized parameters

-- ass : {σ.Γ σ.Δ δ.Γ ν.Γ : Con}
-- {σ : Sub σ.Γ σ.Δ} {δ : Sub δ.Γ σ.Γ} {ν : Sub ν.Γ δ.Γ}
-- → (σ ∘ δ) ∘ ν ≡ σ ∘ (δ ∘ ν)

Hard dependencies between the parameters:

σ.Γ < σ, σ.Δ < σ, δ.Γ < δ, σ.Γ < δ, ν.Γ < ν, δ.Γ < ν

*Soft dependencies* help to complete the ordering:

- metavariables are smaller than variables
- variables/metavariables defined sooner are smaller

σ.Γ < σ.Δ < δ.Γ < ν.Γ < σ < δ < ν

Final ordering by *“smallest-numbered available vertex first”* topological sorting:

σ.Γ < σ.Δ < δ.Γ < ν.Γ < σ < δ < ν
Placement of generalized parameters

variable A B : Set
postulate const : A → B → A

Where to place the quantifications?

A) as early as possible

const : {A B : Set} → A → B → A

B) as late as possible

const : {A : Set} → A → {B : Set} → B → A

C) something else

The current implementation follows A, so all generalized parameters are at the beginning of the type.
Stability regarding code changes

▶ Metavariable resolution

-- ass : {σ.1 σ.2 δ.1 ν.1 : Con}
-- {σ : Sub σ.1 σ.2}{δ : Sub δ.1 σ.1}{ν : Sub ν.1 δ.1}
-- → (σ ∘ δ) ∘ ν ≡ σ ∘ (δ ∘ ν)

δ.2 is solved with σ.1 and not the other way around, because σ.1 was introduced earlier.
Existing solutions
Module parameters with an anonymous module name:

```agda
module _ {A : Set}{B : Set} where
  id : A → A
  const : A → B → A
```

Differences between variable and module:

- module will add all module parameters to the signatures:

  ```agda
  id : {A : Set}{B : Set} → A → A
  ```

- variable introduces separate variables and metavariables for each definition. This matters if the definitions depend on each-other.

- variable generalizes unsolved metavariables too (in a controlled way)
Similar constructs in Agda #2

```
data Exp : ∀{ℓ} → Env ℓ → Ω ℓ → Set where
  lit : ∀{ℓ Γ} → ℕ → Exp {ℓ} Γ Nat
```

vs.

```
data Exp {ℓ} {Γ} : Env ℓ → Ω ℓ → Set where
  lit : ℕ → Exp Γ Nat
```

- works only if all constructors use the same hidden arguments uniformly
- similar to module parameters
‘variable’ in Lean is quite similar to ‘variable’ in Agda.

The Agda version seems to be strictly more powerful:

```lean
variable any : _ -- possible in Lean
```

```lean
variable A : Ty _ -- not possible/not documented in Lean
```

Documentation of ‘variable’ in Lean: [1], [2]
A detailed description of the associated unification algorithm is here:

Brigitte Pientka. An insider’s look at LF type reconstruction: Everything you (n)ever wanted to know, Journal of Functional Programming, Jan 2013
There is **implicit generalization in Coq**. Coq also has a forall-generalisation.

Main difference:
not possible to specify the types of the variables to be generalized

An example:

Generalizable Variables A.

Definition id `(x : A) : A := x.

About id.

(* id : forall A : Type, A -> A
  Argument A is implicit and maximally inserted [...] *)
Identifiers beginning with small letters are generalized.

One can give a type signature to generalized variables with using.
ML, Isabelle, Haskell: forall generalization without any pragma for the variables needed
Twelf: capitalized identifiers are quantified over
CASL: keywords vars, var
PVS: see this and this
Extensions
Motivating example:

```agda
record Semigroup : Set₁ where
  field
    A : Set
    _⊙_: A → A → A

variable
  x y z : A

field
  assoc : x ⊙ (y ⊙ z) ≡ (x ⊙ y) ⊙ z
```
Attached instances

Motivating example:

variable

\[ G : \text{Set} \]

instance

\[ \text{isGroup} : \text{IsGroup} \ G \]

my-id : G

-- my-id : \{ G : \text{Set} \} \ { \{ \text{isGroup} : \text{IsGroup} \ G \} \} \rightarrow G

my-id = \text{IsGroup}.\text{id}
Macros of generalized variables

Motivating example:

```agda
record V : Set₁ where
  field FieldOfV : Set

variable
  v : V
  w = V.FieldOfV v

postulate
  f : w → w
  -- f : {v : V} → let w = V.FieldOfV v in w → w
```
Implementation
Contributors

Original issue (2015): Jesper Cockx

Coding (2018): Ulf Norell, Péter Diviánszky

Discussions & testing (2018): lots of people
Things implemented

- parsing of ‘variable’ statements
- hiding / export / import of ‘variable’ declarations
- scope checking (recognize generalizable variables)
- type checking
  1. create fresh generalizable variables;
  2. create fresh metas for their types;
  3. name metas
  4. put these into the context
  5. type check the original type
  6. collect unsolved metas
  7. decide which metavariables should be generalized
  8. make a pre-order of the variables to be generalized
  9. complete the ordering
  10. build the generalized type
  11. adjust the context of non-generalized metavariables
Context handling

Let A be the type to be generalized (A is a scope checked expression).

Let the final generalized type be

\{x_1 : B_1\} \{x_2 : B_2\} \ldots \{x_n : B_n\} \rightarrow A'

A should be typechecked under \{x_1 : B_1\} \{x_2 : B_2\} \ldots \{x_n : B_n\}, but this is known only after type checking.

Solution:

- Let R be the record of x_1, x_2, ..., x_n.

1. Typecheck A under R. The type and contents of R are not yet known (they are metavariables).
2. After typechecking of A, solve the type and contents of R with the proper values.
Other tricks

- Generalizable variables are handled as *frozen* metavariables.
  (This makes the implementation more uniform.)
Other
Complex example

Type theory in type theory

- Original code
- Code using generalize (demo)
Another example

postulate
  Class : Set -> Set
  method : {X : Set} {{_} : Class X} -> X -> Set

variable
  n : ℕ
  x : Fin _

postulate
  instance ClassFin : Class (Fin n)
  -- instance ClassFin : {n : ℕ} -> Class (Fin n)
  test : method x
  -- test : {n : ℕ} {x : Fin n} -> method x