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

Introduction

A) Implicit generalization

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

```
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

- Γ, Δ usually denote contexts

Mental map for each coherent document:

```
name₁ : type₁
name₂ : type₂
...
```

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

```
postulate
  Con : Set

variable          -- new keyword
  Γ Δ : 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.

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 → Con → Set

variable
  σ δ ν : Sub _ _   -- metavariables: σ.1, σ.2, δ.1, δ.1, ν.1, ν.2

postulate
  ass : (σ ∘ δ) ∘ ν ≡ σ ∘ (δ ∘ ν)

--  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)
-- note that δ.2 was solved with σ.1; ν.2 was solved with δ.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
```

n_1, the metavariable introduced by the underscore was not generalizable, but we generalized it because m, a generalizable meta was solved with suc n_1.

Nested variables

```
variable
  ℓ : Level      -- let ℓ denote a level
  A : Set ℓ      -- let A denote a set at (a) level ℓ (for all ℓ)

postulate
  f : A → Set ℓ
```

Three possible meanings:

A) "let A denote a set at level ℓ"

`f : {ℓ : Level}{A : Set ℓ} → A → Set ℓ`

B) "let A denote a set at *a* level ℓ"

`f : {ℓ ℓ' : Level}{A : Set ℓ'} → A → Set ℓ`

C) "let A denote a set at level ℓ *for all* ℓ"

`f : {ℓ : Level}{A : {ℓ : Level} → Set ℓ} → A → Set ℓ`

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:

```
postulate  Sub : (Γ : Con)(Δ : Con) → Set
```

- name hints for metas in the type of σ, δ and ν:

```
variable  σ δ ν : Sub Γ Δ
-- variables in type of variables are used for name hinting
```

```
postulate
   ass : (σ ∘ δ) ∘ ν ≡ σ ∘ (δ ∘ ν)
--  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)
```

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

Similar constructs in Agda #1

Module parameters with an anonymous module name:

```
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:

  ```
  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 argments uniformly
- similar to module parameters

Lean

'variable' in Lean is quite similar to 'variable' in Agda.

The Agda version seems to be strictly more powerful:

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

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

Documentation of 'variable' in Lean: [1], [2]

LF

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

Coq

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 [...] *)
```

Idris

Identifiers beginning with small letters are generalized.

One can give a type signature to generalized variables with `using`.

Other

- 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

variable in records

Motivating example:

```
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 : Set
  instance
    isGroup : IsGroup G

my-id : G
-- my-id : {G : Set} {{isGroup : IsGroup G}} → G
my-id = IsGroup.id
```

Macros of generalized variables

Motivating example:

```
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

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

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

Solution:

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

1. Typecheck A under R. The type and contents of R are not yet know (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
```