

Type checking in the presence of meta-variables (reprise)

Francesco Mazzoli

October 2014

Meta-variables and dependent types

Used for inference:

$$\begin{aligned} \text{length} &: \forall \{A\} \rightarrow \text{List } A \rightarrow \mathbb{N} \\ \text{length } [] &= 0 \\ \text{length } (x :: xs) &= 1 + \text{length } xs \end{aligned}$$

The type becomes

$$\text{length} : \{A : \text{Set}\} \rightarrow \text{List } A \rightarrow \mathbb{N}$$

And in invocations the type will be filled in automatically:

$$\text{length } [1, 2, 3] \Rightarrow \text{length } _ [1, 2, 3] \Rightarrow \text{length } \mathbb{N} [1, 2, 3]$$

How does it work?

We want to instantiate meta-variables as needed when type checking some terms.

This is usually accomplished in an on-demand fashion: when trying to check equality between two terms in type checking we will instantiate them accordingly if they are meta-variables.

How does it work?

$length : \{ A : _ \} \rightarrow List A \rightarrow \mathbb{N}$

_ will introduce a new meta-variable, say α .

The application of α to $List : Set \rightarrow Set$ will prompt the type checker to check $\alpha = Set$, thus instantiating the meta-variable.

Inconveniences

Given

$$Foo : Bool \rightarrow Set$$
$$Foo \mathbf{true} = Bool$$
$$Foo \mathbf{false} = \mathbb{N}$$

and

$$\alpha : Bool$$
$$\alpha = _$$

How should we proceed when faced with definition

$$test_1 : Foo \alpha$$
$$test_1 = \mathbf{true}$$

?

Why can't we just give up?

It's tempting to just stop when facing such problems.

However, consider

$$\begin{aligned} test_2 &: (Foo\ \alpha, \alpha \equiv \mathbf{true}) \\ test_2 &= (\mathbf{true}, \mathbf{refl}) \end{aligned}$$

We don't want to stop when type checking $\mathbf{true} : Foo\ \alpha$, because checking $\mathbf{refl} : \alpha \equiv \mathbf{true}$ will let us type check the whole definition.

Why can't we just pretend things are fine?

In the previous example:

$$test_2 : (Foo\ \alpha, \alpha \equiv \mathbf{true})$$
$$test_2 = (\mathbf{true}, \mathbf{refl})$$

Can't we just pretend we have checked $\mathbf{true} : Foo\ \alpha$ and continue?

It doesn't work in the general case, consider

$$test_3 : ((x : Foo\ \alpha) \rightarrow Foo\ (\neg x)) \rightarrow \mathbb{N}$$
$$test_3 = \lambda g \rightarrow g\ 0$$

We'll generate constraints $Foo\ \alpha \equiv Bool$, since $x : Bool$. If we just continue and try to type check the body, we'll get the ill-typed $\neg 0$ by instantiating x .

So, we need to be more careful.

Elaborate!

Type checking will take a lump of syntax (Expr) and give us back a term:

$$\text{check} : \text{Ctx} \rightarrow \text{Type} \rightarrow \text{Expr} \rightarrow \text{Term}$$

The idea is that in $\text{check } \Gamma A e \rightsquigarrow t$ the resulting t might be only an approximation of the original e .

Elaborate!

check will insert meta-variables when type checking is “stuck”, which will be instantiated when type checking will be able to resume.

Going back to

$$test_1 : Foo \alpha$$
$$test_1 = \mathbf{true}$$

Type checking the body of the definition we get

$$check \cdot (Foo \alpha) \mathbf{true} \rightsquigarrow \beta$$

Where

$$\beta : Foo \alpha$$

Elaborate!

check will add constraints to ensure that if possible we'll type check the original term fully.

Going back to

$$\text{check} \cdot (\text{Foo } \alpha) \text{ true} \rightsquigarrow \beta$$

We want to resume type checking when *Foo* α is unblocked, and if we succeed instantiate β to the original term:

$$\beta := \text{true}$$

Things get messy

How exactly to handle this elaboration works is the question.

Ulf's thesis, chapter 3, provides an account on how to do this – an approach then developed in Agda.

A lot of poorly documented work in this direction has been done in Epigram and Epigram2.

How it works in Agda

It's pretty messy, but the basic idea is to type check normally, but stopping and inserting meta-variables when unification is stuck.

We then need to put some quite complicated machinery in place to resume checking when needed, instantiate “placeholder” variables, etc.

Let the unifier do all the work

The idea is to convert a type checking problem

$$\Gamma \vdash e : A$$

Into a list of heterogeneous unification problems, of the form

$$\Gamma \vdash t : A \cong u : B$$

Intuitively, by looking at the expression to check we build up a series of constraints that make sure that A has the right shape.

Let the unifier do all the work

We can give this elaboration procedure a simple type:

-- I'll use $\Gamma \vdash t : A \cong u : B$ as a nicer notation

data Constraint = Constraint Ctxt Term Type Term Type

-- I'll use $\llbracket \Gamma \vdash e : A \rrbracket$ as a nicer notation

elaborate : Ctxt \rightarrow Type \rightarrow Expr \rightarrow TC (Term, [Constraint])

TC is some monad that lets us add meta-variables:

newMeta : Ctxt \rightarrow Type \rightarrow TC Term

Expr

We'll write the elaboration much like we'd write a type checker without meta-variables.

Given a simple type expression type

```
data Expr
  = x                -- Variable
  | _                -- Meta-variable
  | Set              -- Type of types
  | (x : Expr) → Expr -- Function type
  | λx → Expr        -- Abstraction
  | Expr Expr        -- Function application
```

Term

And terms

data Term

= x	-- Variable
α	-- Meta-variable
Set	-- Type of types
$(x : \text{Term}) \rightarrow \text{Term}$	-- Function type
$\lambda x \rightarrow \text{Term}$	-- Abstraction
Term Term	-- Function application

type Type = Term

Variables, meta-variables, Set

$\llbracket \Gamma \vdash x : A \rrbracket = \mathbf{do}$
 let $B = \text{lookup } x \ \Gamma$
 $t \leftarrow \text{newMeta } \Gamma \ A$
 return $(t, [\Gamma \vdash x : B \cong t : A])$

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

$\llbracket \Gamma \vdash _ : A \rrbracket = \mathbf{do}$
 $t \leftarrow \text{newMeta } \Gamma \ A$
 return $(t, [])$

$\llbracket \Gamma \vdash \text{Set} : A \rrbracket = \mathbf{do}$
 $t \leftarrow \text{newMeta } \Gamma \ A$
 return $(t, [\Gamma \vdash \text{Set} : \text{Set} \cong t : A])$

$$\frac{}{\Gamma \vdash \text{Set} : \text{Set}}$$

Dependent function type

$$\frac{\Gamma \vdash \text{Dom} : \text{Set} \quad \Gamma; x : \text{Dom} \vdash \text{Cod} : \text{Set}}{\Gamma \vdash (x : \text{Dom}) \rightarrow \text{Cod} : \text{Set}}$$

```
[[ $\Gamma \vdash (x : \text{Dom}) \rightarrow \text{Cod} : A$ ]] = do  
  ( $\text{Dom}'$ ,  $cs_1$ ) ← [[ $\Gamma \vdash \text{Dom} : \text{Set}$ ]]  
  ( $\text{Cod}'$ ,  $cs_2$ ) ← [[ $\Gamma; x : \text{Dom}' \vdash \text{Cod} : \text{Set}$ ]]  
   $t$  ← newMeta  $\Gamma$   $A$   
  let  $c = \Gamma \vdash ((x : \text{Dom}') \rightarrow \text{Cod}') : \text{Set} = t : A$   
  return ( $t$ ,  $c :: (cs_1 \uplus cs_2)$ )
```

Abstractions

$$\frac{\Gamma; x : Dom \vdash t : Cod}{\Gamma \vdash \lambda x \rightarrow t : (x : Dom) \rightarrow Cod}$$

$\llbracket \Gamma \vdash (\lambda x \rightarrow e) : A \rrbracket = \mathbf{do}$

$Dom \leftarrow \mathit{newMeta} \ \Gamma \ \mathit{Set}$

$Cod \leftarrow \mathit{newMeta} \ (\Gamma; x : Dom) \ \mathit{Set}$

$(body, cs) \leftarrow \llbracket \Gamma; x : Dom \vdash e : Cod \rrbracket$

$t \leftarrow \mathit{newMeta} \ \Gamma \ A$

let $c = \Gamma \vdash (\lambda x \rightarrow body) : ((x : Dom) \rightarrow Cod) \cong t : A$

return $(t, c :: cs)$

Application

$$\frac{\Gamma \vdash f : (x : Dom) \rightarrow Cod \quad \Gamma \vdash arg : Dom}{\Gamma \vdash f \ arg : Cod [x := arg]}$$

$\llbracket \Gamma \vdash e_1 \ e_2 : A \rrbracket = \mathbf{do}$

$Dom \leftarrow \mathit{newMeta} \ \Gamma \ \mathit{Set}$

$Cod \leftarrow \mathit{newMeta} \ (\Gamma; x : Dom) \ \mathit{Set}$

$(f, cs_1) \leftarrow \llbracket \Gamma \vdash e_1 : (x : Dom) \rightarrow Cod \rrbracket$

$(arg, cs_2) \leftarrow \llbracket \Gamma \vdash e_2 : Dom \rrbracket$

$t \leftarrow \mathit{newMeta} \ \Gamma \ A$

$\mathbf{let} \ c = \Gamma \vdash (f \ arg) : Cod [x := arg] = t : A$

$\mathbf{return} \ (t, c :: (cs_1 \ \# \ cs_2))$

What about unification?

We have constraints of the form

$$\Gamma \vdash t : A \cong u : B$$

We can convert them to homogeneous constraints

$$\Gamma \vdash A \cong B : \text{Set} \gg \Gamma \vdash t \cong u : A$$

Where \gg is a sequencing operator that has the unifier solve the first constraint before attempting the second.

But we lose solutions – the types might be similar enough to advance unification of the terms, while with homogeneous equality we demand the types to be equal first.

What about unification?

We can also leave the constraints as they are, and have the unifier solve heterogeneously, by making sure that the types have a rigid head before trying to compare the head of the terms.

$$\Gamma \vdash t : A \cong u : B$$

Becomes

$$\Gamma \vdash t : A \cong u : B \wedge \Gamma \vdash A : \text{Set} \cong B : \text{Set}$$

However, we still lose solutions.

What about unification?

Consider what happens when unifying

$$\Gamma \vdash (\lambda x \rightarrow t) : (x : A) \rightarrow B \cong (\lambda x \rightarrow u) : (x : S) \rightarrow T$$

We'd like to go ahead and compare the bodies...

$$\Gamma; x : ? \vdash t : B [x := ?] \cong u : T [x := ?]$$

With one context, we need to make sure that $A \cong B$ first.

What about unification?

Gundry & McBride solved this problem with “twin variables”.

I think it's simpler to just keep two contexts:

$$(\Gamma \vdash (\lambda x \rightarrow t) : (x : A) \rightarrow B) \cong (\Delta \vdash (\lambda x \rightarrow u) : (x : S) \rightarrow T)$$

Becomes

$$(\Gamma; x : A \vdash t : B) \cong (\Delta; x : S \vdash u : T)$$

What about bidirectional type checking?

What to do about

- ▶ Implicit arguments;
- ▶ Overloaded constructors;
- ▶ Type classes (canonical structures);
- ▶ ...