Mechanizing an elaboration algorithm for the Hindley-Damas-Milner system

Roger Bosman, PhD @ KU Leuven

August 27, 2021

KU LEUVEN

Type checking, inference & elaboration

Type checking

```
bool p = let bool q = True in q \/ q
```

$\Rightarrow$

✓ (well-typed)

Type checking, inference & elaboration

Type checking

```
bool p = let bool q = True in q \/ q
```

$\Rightarrow$

✓ (well-typed)

Type inference

```
bool p = let bool q = True in q \/ q
```

$\Rightarrow$

`Bool` (type of `p`)

Type checking, inference & elaboration

Type checking

```
bool p = let bool q = True in q \/ q
```

$\Rightarrow$

✓ (well-typed)

Type inference

```
p = let q = True in q \/ q
```

$\Rightarrow$

`Bool` (type of `p`)

Type checking, inference & elaboration

Type checking

```
bool p = let bool q = True in q \/ q
```

$\Rightarrow$

✓ (well-typed)

Type inference

```
p = let q = True in q \/ q
```

$\Rightarrow$

`Bool` (type of `p`)

Elaboration

```
p = let q = True in q \/ q
```

$\Rightarrow$

```
(p :: Bool) =
  let (q :: Bool) = True in q \/ q
```

Why not study elaboration

Plenty of research on type checking and type inference [GMM10; McB03; NN99; DM99; RC15]

Why not study elaboration

Plenty of research on type checking and type inference [GMM10; McB03; NN99; DM99; RC15]

Yet, compilers like GHC use an elaboration algorithm

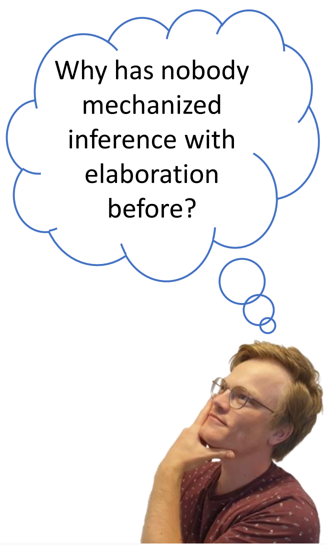
- Used to implement features like typeclasses [WB89]

Why not study elaboration

Plenty of research on type checking and type inference [GMM10; McB03; NN99; DM99; RC15]

Yet, compilers like GHC use an elaboration algorithm

- Used to implement features like typeclasses [WB89]



Limitation

Claim: type checking algorithms like algorithm $\mathcal{W}$ [DM82] cannot (fully) do elaboration

Example

```
let x =
      (λ y. unit)
      (λ z. z) in ...
```

Example

$$\texttt{let (x} :: \widehat{\alpha}\texttt{) =}$$
$$\texttt{(}\lambda\texttt{(y} :: \widehat{\beta}\texttt{). unit)}$$
$$\texttt{(}\lambda\texttt{(z} :: \widehat{\gamma}\texttt{). z) in ...}$$

To elaborate we must

- Assign existential type variables

Example

```
let (x :: Unit) =
      (λ(y :: γ̂ → γ̂). unit)
      (λ(z :: γ̂). z) in ...
```

To elaborate we must

- Assign existential type variables
- Solve constraints

Example

```
let (x :: Unit) =
      (λ(y :: γ̂ → γ̂). unit)
      (λ(z :: γ̂). z) in ...
```

To elaborate we must

- Assign existential type variables
- Solve constraints

Example

```
let (x :: ∀a. Unit) =
  Λa. (λ(y :: a → a). unit)
        (λ(z :: a). z) in ...
```

To elaborate we must

- Assign existential type variables
- Solve constraints
- Generalize

Example

```
let (x :: ∀a. Unit) =
  Λa. (λ(y :: a → a). unit)
      (λ(z :: a). z) in ...
```

To elaborate we must

- Assign existential type variables
- Solve constraints
- Generalize

Since type inference algorithms like algorithm $\mathcal{W}$ are constraint based, they cannot perform this last step. The scope of existential type variables is lost.

Our algorithm

Solution? Maintain explicit scopes!

Our algorithm

Solution? Maintain explicit scopes!

$$\boxed{\Psi_{in} \vdash e : [A]\tau \leadsto t \dashv \Psi_{out}}$$

Our algorithm

Solution? Maintain explicit scopes!

$$\boxed{\overbrace{\Psi_{in} \vdash e}^{inputs} : \underbrace{[A]\tau \rightsquigarrow t \dashv \Psi_{out}}_{outputs}}$$

Our algorithm

Solution? Maintain explicit scopes!

$$\boxed{\Psi_{in} \vdash e : [A]\tau \rightsquigarrow t \dashv \Psi_{out}}$$

Our algorithm

Solution? Maintain explicit scopes!

$$\boxed{\Psi_{in} \vdash e : [A]\tau \rightsquigarrow t \dashv \Psi_{out}}$$

Now with explicit scope information!

Our algorithm

Now with explicit scope information!

Solution? Maintain explicit scopes!

$$\boxed{\Psi_{in} \vdash e : [A]\tau \rightsquigarrow t \dashv \Psi_{out}}$$

A is a list of existential type variables $\overline{\widehat{\alpha}}$ in scope of inferred type τ

Our algorithm

Now with explicit scope information!

Solution? Maintain explicit scopes!

$$\boxed{\Psi_{in} \vdash e : [A]\tau \rightsquigarrow t \dashv \Psi_{out}}$$

A is a list of existential type variables $\overline{\widehat{\alpha}}$ in scope of inferred type τ

This makes generalization almost trivial:

$$\frac{\Psi_{in} \vdash e : [A]\tau \rightsquigarrow t \dashv \Psi_{out}}{\Psi_{in} \vdash e : \forall A.\tau \rightsquigarrow \Lambda A.t \dashv \Psi_{out}} \textsc{Gen}$$

State of affairs

Mechanization ongoing

- Coq proof assistant [BC13]
- Generalized rewriting [Soz]
- Locally nameless [Cha12]
- Ott/LNgen [Sew+10; AW10]

State of affairs

Mechanization ongoing

- Coq proof assistant [BC13]
- Generalized rewriting [Soz]
- Locally nameless [Cha12]
- Ott/LNgen [Sew+10; AW10]

Future work: extend with typeclasses

References

Brian Aydemir and Stephanie Weirich. “LNgen: Tool support for locally nameless representations”. In: (2010).

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq’Art: The Calculus of Inductive Constructions.* Springer Science & Business Media, 2013.

References ii

Arthur Charguéraud. “The locally nameless representation”. In: *Journal of automated reasoning* 49.3 (2012), pp. 363–408.

Luis Damas and Robin Milner. “Principal Type-Schemes for Functional Programs”. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1982, pp. 207–212.

Catherine Dubois and Valerie Menissier-Morain. “Certification of a Type Inference Tool for ML: Damas–Milner within Coq”. In: *Journal of Automated Reasoning* 23.3 (1999), pp. 319–346. ISSN: 1573-0670.

Adam Gundry, Conor McBride, and James McKinna. "Type Inference in Context". In: *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming*. 2010, pp. 43–54.

Conor McBride. "First-Order Unification by Structural Recursion". In: *Journal of functional programming* 13.6 (2003), pp. 1061–1075. ISSN: 1469-7653.

References iv

Wolfgang Naraschewski and Tobias Nipkow. “Type Inference Verified: Algorithm W in Isabelle/HOL”. In: *Journal of Automated Reasoning* 23.3 (1999), pp. 299–318. ISSN: 1573-0670.

Rodrigo Ribeiro and Carlos Camarao. “A Mechanized Textbook Proof of a Type Unification Algorithm”. In: *Brazilian Symposium on Formal Methods*. Springer, 2015, pp. 127–141.

Peter Sewell et al. “Ott: Effective tool support for the working semanticist”. In: *Journal of functional programming* 20.1 (2010), pp. 71–122.

Matthieu Sozeau. *Generalized rewriting*. URL: https://coq.inria.fr/refman/addendum/generalized-rewriting.html.

Philip Wadler and Stephen Blott. “How to Make Ad-Hoc Polymorphism Less Ad Hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1989, pp. 60–76.