A mechanical formalization of Hidnley-Damas-Milner type inference

Roger Bosman
roger.bosman@kuleuven.be

April 13, 2021

Why even formalize?

Why formalize type systems?

Why even formalize?

Why formalize type systems?

Programmers rely on guarantees of the type system

> *Well-typed programs cannot ”go wrong”*
> *— Robin Milner [2]*

Why even formalize?

Why formalize type systems?

Programmers rely on guarantees of the type system

> *Well-typed programs cannot "go wrong"*
> *— Robin Milner [2]*

Great benefit in verifying said guarantees

Informal proofs (on paper) often contain mistakes [1]

Mechanical proofs preferred

- Using a proof assistant such as Coq, Agda, Idris, Isabelle, ...

Type checking

Verifying that a program is well-typed

`1 + 234` ✓

`1 + 'a'` ✗

Inference, Type checking, and Elaboration

Type checking

Verifying that a program is well-typed

`1 + 234` ✓

`1 + 'a'` ✗

Type Inference

Inferring the type of expressions, lifting the need for explicit typing information (annotations)

```
id x = x
```

Inference, Type checking, and Elaboration

Type checking

Verifying that a program is well-typed

`1 + 234` ✓

`1 + 'a'` ✗

Type Inference

Inferring the type of expressions, lifting the need for explicit typing information (annotations)

```
id x = x :: a -> a
```

Inference, Type checking, and Elaboration

Type checking

Verifying that a program is well-typed

`1 + 234` ✓

`1 + 'a'` ✗

Type Inference

Inferring the type of expressions, lifting the need for explicit typing information (annotations)

```
id x = x :: a -> a
```

Elaboration

Something more ...

Example: Haskell's Typeclasses

Typeclasses add ad-hoc polymorphism to Haskell

```haskell
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Bool where
  p == q = (p && q) || (!p && !q)

instance Eq Int where
  i == j = i >= j && j >= i
  -- weird formulation just for demonstration

f :: Eq a => a -> Bool
f x = x == x
```

Implementation

How to implement typeclasses?

- Ideally, our internal language is as small as possible
  - For example, easier optimizations

Implementation

How to implement typeclasses?

- Ideally, our internal language is as small as possible
  - For example, easier optimizations
- Idea: express typeclasses in the existing language
  - Key structure: dictionary

Expanding Typeclasses

```haskell
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Bool where
  p == q = (p && q) ||
           (!p && !q)



f :: Eq a => a -> Bool
f x = x == x
```

```haskell
data EqDict a = EqDict
  { (==) :: a -> a -> Bool }

eqDictBool :: EqDict Bool
eqDictBool =
  let eq p q = (p && q) ||
               (!p && !q)
  in EqDict { (==) = eq }

f :: EqDict a -> a -> Bool
f dict x = ((==) dict) x x
```

Expanding Typeclasses

```haskell
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Bool where
  p == q = (p && q) ||
           (!p && !q)



f :: Eq a => a -> Bool
f x = x == x
```

```haskell
data EqDict a = EqDict
  { (==) :: a -> a -> Bool }

eqDictBool :: EqDict Bool
eqDictBool =
  let eq p q = (p && q) ||
               (!p && !q)
  in EqDict { (==) = eq }

f :: EqDict a -> a -> Bool
f dict x = ((==) dict) x x
```

Expanding Typeclasses

```haskell
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Bool where
  p == q = (p && q) ||
           (!p && !q)



f :: Eq a => a -> Bool
f x = x == x
```

```haskell
data EqDict a = EqDict
  { (==) :: a -> a -> Bool }

eqDictBool :: EqDict Bool
eqDictBool =
  let eq p q = (p && q) ||
               (!p && !q)
  in EqDict { (==) = eq }

f :: EqDict a -> a -> Bool
f dict x = ((==) dict) x x
```

Expanding Typeclasses

```haskell
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Bool where
  p == q = (p && q) ||
           (!p && !q)



f :: Eq a => a -> Bool
f x = x == x
```

```haskell
data EqDict a = EqDict
  { (==) :: a -> a -> Bool }

eqDictBool :: EqDict Bool
eqDictBool =
  let eq p q = (p && q) ||
               (!p && !q)
  in EqDict { (==) = eq }

f :: EqDict a -> a -> Bool
f dict x = ((==) dict) x x
```

Type checking vs elaboration

Convert the input language to an expanded, more explicit internal language

Type checking vs elaboration

Convert the input language to an expanded, more explicit internal language: we need to elaborate

Type checking vs elaboration

Convert the input language to an expanded, more explicit internal language: we need to elaborate

Type checking $\subset$ elaboration

Type checking:

- Emit set of constraints
- If none conflict the program is well-typed

```
const x y = x
const 1 (\x -> x)
```

Type checking never needs the type of `(\x -> x)` because it does not affect well-typedness.

Current work

Many more elaboration-based type system features

- Implicits (Scala, Agda)
- Intersection types (Java, Scala, TypeScript)
- Implicit type conversion (Java, Scala)

Existing end-to-end mechanizations of Hindley–Milner focus on type checking

- Not easily extended with elaboration
- Let alone with features *implemented using* elaboration

Current work

Many more elaboration-based type system features

- Implicits (Scala, Agda)
- Intersection types (Java, Scala, TypeScript)
- Implicit type conversion (Java, Scala)

Existing end-to-end mechanizations of Hindley–Milner focus on type checking

- Not easily extended with elaboration
- Let alone with features *implemented using* elaboration

Can we mechanize an elaboration algorithm for Hindley–Milner type system?

Email me!

Not enough time to go into details of the (ongoing) proof, but happy to talk about it during a coffee break

Email me!

Not enough time to go into details of the (ongoing) proof, but happy to talk about it during a coffee break

If you have any tips or questions: roger.bosman@kuleuven.be

- Coq
    - Proof automation
    - Rewriting setoids
- Any other relevant topic

References

- Casey Klein et al. “Run your research: on the effectiveness of lightweight mechanization”. In: *ACM SIGPLAN Notices* 47.1 (2012), pp. 285–296.
- Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

The idea

1. Move from a single environment to an in- and output environment

$$\Gamma \vdash e : \sigma \quad \longrightarrow \quad \Psi_{in} \vdash e : [A]\tau \dashv \Psi_{out}$$

2. Maintain a list A of existential type variables α
3. (partially) solve constraints when encountered
4. If at the end existential variables remain, convert to Skolem (”normal”) type variables.

Declarative vs algorithmic

- Declarative system "makes up" types without specifying *how* to actually determine these types
- An algorithmic system explicitly specifies an algorithm for determining all types.

$$\frac{\boxed{\Gamma \vdash_{\mathtt{ty}} \tau_1} \qquad \Gamma; x : \tau_1 \Vdash_{\mathsf{DM}} e : \tau_2}{\Gamma \Vdash_{\mathsf{DM}} \lambda x.e : \tau_1 \to \tau_2} \textsc{Abs}$$

$$\frac{\widehat{\alpha}\#\Psi_{in} \qquad (\Psi_{in}; (\widehat{\alpha}); x : \widehat{\alpha}) \vdash e : [A_2]\tau_2 \dashv (\Psi_{out}; A_1; x : \tau_1)}{\Psi_{in} \vdash \lambda x.e : [A_1, A_2](\tau_1 \to \tau_2) \dashv \Psi_{out}} \textsc{Abs}$$

Challenges

List of list

Ψ is a list, as is A. Ψ therefore is a list of lists

Splitting up a list into two gives unequal, but equivalent environments

$$\Psi_1; (A_1 + A_2); \Psi_2 \neq \Psi_1; A_1; A_2; \Psi_2$$

$$\Psi_1; (A_1 + A_2); \Psi_2 \approx \Psi_1; A_1; A_2; \Psi_2$$

We want to be able to rewrite environment for equivalent ones everywhere applicable

- Which is pretty much everywhere *except* inference judgments