TYPE INFERENCE
AS CONSTRAINT SOLVING

Simon Peyton Jones
Microsoft Research
August 2013
**Instantiate** ‘reverse’ with a unification variable \( \alpha \), standing for an as-yet-unknown type. So this occurrence of reverse has type \([\alpha] \rightarrow [\alpha]\).

**Constrain** expected arg type \([\alpha]\) equal to actual arg type \([\text{Bool}]\), thus \(\alpha \sim \text{Bool}\).

**Solve by** [unification](#): \(\alpha := \text{Bool}\)
Modify for type classes

\( (> ) :: \forall \alpha. \text{Ord} \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{Bool} \)

instance \text{Ord} \alpha \Rightarrow \text{Ord} [\alpha] \text{ where } ...

\( \text{foo} :: \forall \alpha. \text{Ord} \alpha \Rightarrow [\alpha] \rightarrow [\alpha] \rightarrow \text{Bool} \)
\( \text{foo} \; \text{xs} \; \text{ys} = \text{not} \; (\text{xs} > \text{ys}) \)

- Instantiate ‘(>)’ to \( \alpha \rightarrow \alpha \rightarrow \text{Bool} \), and emit a wanted constraint (\text{Ord} \alpha)

- **Constrain** \( \alpha \sim [\alpha] \), since \( \text{xs} :: [\alpha] \), and solve by unification

- **Solve** wanted constraint (\text{Ord} \alpha), i.e. (\text{Ord} [\alpha]), from **given constraint** (\text{Ord} \alpha)

- Here ‘\( \alpha \)’ plays the role of a **skolem constant**.
Another view

\( (> ) \) :: \( \forall a. \text{Ord}\ a \Rightarrow a \to a \to \text{Bool} \)
instance \( \text{Ord}\ a \Rightarrow \text{Ord}\ [a] \) where ...

\[ \text{foo} :: \forall a. \text{Ord}\ a \Rightarrow [a] \to [a] \to \text{Bool} \]
\[ \text{foo}\ xs\ ys = \text{not}\ (xs > ys) \]

- Instantiate \( (> ) \)' to \( \alpha \to \alpha \to \text{Bool} \), and emit a **wanted constraint** (Ord \( \alpha \))
- **Constrain** \( \alpha \sim [a] \), since \( xs :: [a] \), and solve by unification
- **Solve** wanted constraint (Ord \( \alpha \)) from **given constraint** (Ord \( a \))
- Here 'a' plays the role of a **skolem constant**.
instance Ord a => Ord [a] where ...

foo :: ∀a. Ord a => [a] -> [a] -> Bool
foo xs ys = not (xs > ys)

dfOrdList :: ∀a. Ord a -> Ord [a]

foo :: ∀a. Ord a -> [a] -> [a] -> Bool
foo a (d::Ord a) (xs::[a]) (ys::[a])
  = let d2::Ord [a] = dfOrdList a d
      in not ((>) [a] d2 xs ys)
How GHC works

Source Haskell → Source language → Typecheck → Elaborated source → Desugar → Typed intermediate language

Rest of GHC
dfOrdList :: ∀a. Ord a -> Ord [a]

foo :: ∀a. Ord a -> [a] -> [a] -> Bool
foo a (d::Ord a) (xs::[a]) (ys::[a])
    = let d2::Ord [a] = dfOrdList a d
        in not ((>) [a] d2 xs ys)
Type and dictionary applications
(inserted when we instantiate)

Type and dictionary abstractions
(inserted when we generalise)

Dictionary bindings
(inserted when we solve constraints)

```
dfOrdList :: \forall a. Ord a -> Ord [a]

foo :: \forall a. Ord a -> [a] -> [a] -> Bool
foo a (d::Ord a) (xs::[a]) (ys::[a])
  = let d2::Ord [a] = dfOrdList a d
      in not (> [a] d xs ys)
```
Another view

\[ \text{dfOrdList} :: \forall a. \text{Ord} \ a \rightarrow \text{Ord} \ [a] \]

\[ \text{foo} :: \forall a. \text{Ord} \ a \rightarrow [a] \rightarrow [a] \rightarrow \text{Bool} \]

\[ \text{foo} \ a \ (d::\text{Ord} \ a) \ (xs::[a]) \ (ys::[a]) \]
\[ = \text{let} \ d2::\text{Ord} \ [a] = \text{dfOrdList} \ a \ d \]
\[ \text{in} \ \text{not} \ ((>) \ [a] \ d2 \ xs \ ys) \]

- Instantiate '(>)' to \( \alpha \rightarrow \alpha \rightarrow \text{Bool} \), and emit a **wanted constraint** (Ord \( \alpha \))
- **Constrain** \( \alpha \sim [a] \), since \( xs :: [a] \), and solve by unification
- **Solve** wanted constraint (Ord \( \alpha \)) from **given constraint** (Ord \( a \))
- Here 'a' plays the role of a **skolem constant**.

\[ \forall a. \ d::\text{Ord} \ a \Rightarrow \]
\[ d2::\text{Ord} \ \alpha \land \alpha \sim [a] \]

Solve this, creating a binding for d2, mentioning d
Elaboration in practice

type Id = Var
data Var = Id Name Type | ....

data HsExpr n
  = HsVar n | HsApp (HsExpr n) (HsExpr n) | ..

tcExpr :: HsExpr Name -> TcRhoType -> TcM (HsExpr Id)
DEFERRED SOLVING
Deferring solving

- Old school
  - Find a unification problem
  - Solve it
  - If fails, report error
  - Otherwise, proceed

- This will not work any more

```haskell
g :: F a -> a -> Int
type instance F Bool = Bool
f x = (g x x, not x)
```
Deferring solving

\[ f \mathbf{x} = (g \times x, \ldots, \neg x) \]

\[ x :: \beta \]

\[ \text{Instantiate } g \text{ at } \alpha \]

We have to solve this first

Order of encounter
Deferring solving

Cannot solve constraint \((C \, a \, \beta)\) until we “later” discover that \((\beta \sim \text{Bool})\)

Need to **defer** constraint solving, rather than doing it all “on the fly”
Deferring solving

\[ \text{op} :: C \, a \, x \Rightarrow a \to x \to \text{Int} \]
\[ \text{instance} \quad \text{Eq} \, a \Rightarrow C \, a \Rightarrow \text{Bool} \]

\[ f \, x = \text{let} \ g :: \forall a \ \text{Eq} \, a \Rightarrow a \to a \]
\[ g \, a = \text{op} \, a \, x \]
\[ \text{in} \ g \, (\text{not} \, x) \]

\[ x : \beta \]
\[ \text{Constraint: } C \, a \, \beta \]

Solve this first

\[ (\forall a. \ \text{Eq} \, a \Rightarrow C \, a \, \beta) \]
\[ \land \]
\[ \beta \sim \text{Bool} \]

And then this
The French approach to type inference

Haskell source program
  Large syntax, with many many constructors

Constraint generation

A constraint, \( W \)
  Small syntax, with few constructors

Step 1:
  Easy

Step 2:
  Hard

Solve

Report errors

Residual constraint
The French approach to type inference

Haskell source program

A constraint, $W$

Constraint generation

Small syntax, with few constructors

$F ::= C \tau_1 \ldots \tau_n \mid \tau_1 \sim \tau_2 \mid F_1 \land F_2$

$W ::= F \mid W_1 \land W_2 \mid \forall a_1 \ldots a_n. F \implies W$

Residual constraint

Report errors

Level
**GHC uses the French approach**

- **Modular**: Totally separate
  - constraint generation (7 modules, 3000 loc)
  - constraint solving (5 modules, 3000 loc)
  - error message generation (1 module, 800 loc)

- **Independent of the order** in which you traverse the source program.

- Can solve the constraint however you like (outside-in is good), including iteratively.
GHC uses the French approach

- **Efficient**: constraint generator does a bit of “on the fly” unification to solve simple cases, but generates a constraint whenever anything looks tricky
- All **type error messages** generated from the final, residual unsolved constraint. (And hence type errors incorporate results of all solved constraints. Eg “Can’t match [Int] with Bool”, rather than “Can’t match [a] with Bool”)
- Cured a raft of **type inference bugs**
The language of constraints

\[ F ::= C \tau_1 \ldots \tau_n \]
\[ | \tau_1 \sim \tau_2 \]
\[ | F_1 \land F_2 \]
\[ | \text{True} \]

\[ W ::= F \]
\[ | W_1 \land W_2 \]
\[ | \forall a_1 \ldots a_n . F \Rightarrow W \]
The language of constraints

$$F ::= d :: C \; \tau_1 \ldots \tau_n$$
Class constraint

$$c :: \tau_1 \sim \tau_2$$
Equality constraint

$$F_1 \land F_2$$
Conjunction

$$\text{True}$$

$$W ::= F$$
Flat constraint

$$W_1 \land W_2$$
Conjunction

$$\forall a_1 \ldots a_n. \; F \Rightarrow W$$
Implication
data T a where
  T1 :: Bool -> T Bool
  T2 :: T a

f :: T a -> Maybe a
f x = case x of
  T1 z -> Just z
  T2   -> False

T1 :: ∀a. (a~Bool) -> Bool -> T a
Equality constraints generate evidence too!

T1 :: \( \forall a. (a \sim \text{Bool}) \rightarrow \text{Bool} \rightarrow T \ a \)

\[
f :: T \ a \rightarrow \text{Maybe} \ a
f \ (a:* \ x:T \ a) = \text{case} \ x \ \text{of}
\]
\[
\quad T1 \ (c:a \sim \text{Bool}) \ (z:\text{Bool})
\quad \rightarrow \ \text{let}
\quad \quad \quad \Box
\quad \quad \quad \text{in} \ \text{Just} \ z \ \triangleright c2
\]
\[
\quad T2 \rightarrow \text{False}
\]

(c :: a \sim \text{Bool}) \Rightarrow c2 :: \text{Maybe} \ \text{Bool} \sim \text{Maybe} \ a
Equality constraints generate evidence too!

\[(c :: a \sim \text{Bool}) \Rightarrow c2 :: \text{Maybe} \text{ Bool} \sim \text{Maybe} a\]
\[c2 = \text{Maybe} c3\]

\[(c :: a \sim \text{Bool}) \Rightarrow c3 :: \text{Bool} \sim a\]
\[c3 = \text{sym} c4\]

\[(c :: a \sim \text{Bool}) \Rightarrow c4 :: a \sim \text{Bool}\]
\[c4 = c\]

\[(c :: a \sim \text{Bool}) \Rightarrow \text{True}\]
Plug the evidence back into the term

\[ f :: \text{T a} \rightarrow \text{Maybe a} \]

\[ f (a:*)(x:\text{T a}) = \begin{cases} 
\text{T1 (c:a~Bool) (z:Bool)} & \rightarrow \text{let} \ c4:a~\text{Bool} = c \\
& \quad c3:\text{Bool~a} = \text{sym} \ c4 \\
& \quad c2:\text{Maybe Bool ~ Maybe a} = \text{Maybe} \ c3 \\
& \quad \text{in Just} \ z \triangleright c2 \\
\text{T2} & \rightarrow \text{False} 
\end{cases} \]
Things to notice

- Constraint solving takes place by successive rewrites of the constraint.
- Each rewrite generates a binding, for:
  - a type variable (fixing a unification variable)
  - a dictionary (class constraints)
  - a coercion (equality constraint)

As we go:

- Bindings record the proof steps.
- Bindings get injected back into the term.
Care with GADTs

```haskell
data T a where
  T1 :: Bool -> T Bool
  T2 :: T a

f x y = case x of
  T1 z -> True
  T2   -> y
```

What type shall we infer for f?
What type shall we infer for f?

- **f :: \( \forall b. \, T \, b \rightarrow b \rightarrow b \)**
- **f :: \( \forall b. \, T \, b \rightarrow \text{Bool} \rightarrow \text{Bool} \)**

Neither is more general than the other!
In the language of constraints

data T a where
    T1 :: Bool -> T Bool
    T2 :: T a

f x y = case x of
    T1 z -> True
    T2   -> y

f :: T α -> β -> γ

(α ~ Bool => γ ~ Bool) ∧ (β ~ γ)
In the language of constraints

\[ f :: T \alpha \to \beta \to \gamma \]

\[(\alpha \sim \text{Bool} \Rightarrow \gamma \sim \text{Bool}) \land (\beta \sim \gamma)\]

Two solutions, neither principal

- \(\gamma := \text{Bool}\)
- \(\gamma := a\)

** GHC's conclusion**
No principal solution, so reject the program
In the language of constraints

\[(\alpha \sim \text{Bool} \Rightarrow \gamma \sim \text{Bool}) \land (\beta \sim \gamma)\]

- Treat \(\gamma\) as **untouchable** under the \((\alpha\sim\text{Bool})\) equality; i.e. \((\gamma\sim\text{Bool})\) is not solvable
- Equality information propagates outside-in
- So \((\alpha \sim \text{Bool} \Rightarrow \gamma \sim \text{Bool}) \land (\alpha \sim \gamma)\) is soluble
This is THE way to do type inference

- Generalises beautifully to more complex constraints:
  - Functional dependencies
  - Implicit parameters
  - Type families
  - Kind constraints
  - Deferred type errors and holes

- Robust foundation for new crazy type stuff.

- Provides a great “sanity check” for the type system: is it easy to generate constraints, or do we need a new form of constraint?

- All brought together in an epic 80-page JFP paper “Modular type inference with local assumptions”
DEFERRED TYPE ERRORS
Type errors considered harmful

- The rise of dynamic languages
- “The type errors are getting in my way”
- Feedback to programmer
  - Static: type system
  - Dynamic: run tests
  “Programmer is denied dynamic feedback in the periods when the program is not globally type correct” [DuctileJ, ICSE’11]
Type errors considered harmful

- Underlying problem: forces programmer to fix all type errors before running any code.

Goal: Damn the torpedos

Compile even type-incorrect programs to executable code, without losing type soundness
Not just the command line: can load modules with type errors --- and run them

Type errors occur at run-time if (and only if) they are actually encountered
Type holes: incomplete programs

{-# LANGUAGE TypeHoles #-}
module Holes where
f x = (reverse . _) x

Quick, what type does the "_" have?

Holes.hs:2:18:

   Found hole ' _ ' with type: a -> [a1]
   Relevant bindings include
      f :: a -> [a1] (bound at Holes.hs:2:1)
      x :: a (bound at Holes.hs:2:3)
   In the second argument of (.), namely ' _ '
   In the expression: reverse . _
   In the expression: (reverse . _) x

Agda does this, via Emacs IDE
Multiple, named holes

\[ f \ x = [_\ a, \ x :: [\text{Char}], _\ b :: _\ c ] \]

Holes:2:12:
  Found hole `\_a' with type: [Char]
  In the expression: \_a
  In the expression: [_\_a, \_x :: [Char], _\_b : _\_c]
  In an equation for `f': \ f \ x = [_\_a, \_x :: [Char], _\_b : _\_c]

Holes:2:27:
  Found hole `\_b' with type: Char
  In the first argument of `(:)', namely `\_b'
  In the expression: _\_b : _\_c
  In the expression: [_\_a, \_x :: [Char], _\_b : _\_c]

Holes:2:30:
  Found hole `\_c' with type: [Char]
  In the second argument of `(:)', namely `\_c'
  In the expression: _\_b : _\_c
  In the expression: [_\_a, \_x :: [Char], _\_b : _\_c]
Combining the two

- `-XTypleHoles` and `-fdefer-type-errors` work together

- With both,
  - you get warnings for holes,
  - but you can still run the program

- If you evaluate a hole you get a runtime error.
Presumably, we generate a program with suitable run-time checks.

How can we be sure that the run-time checks are in the right place, and *stay* in the right places after optimisation?

Answer: not a hack at all, but a thing of beauty!

Zero runtime cost
When equality is insoluble...

Haskell term
(\text{True, } 'a' \land \land \text{False})

Constraints
\text{c7 : Int } \sim \text{Bool}

Elaborated program (mentioning constraint variables)
(\text{True, } ('a' \triangleright \text{c7}) \land \land \text{False})

Generate constraints
elaborated program
Step 2: solve constraints

- Use lazily evaluated “error” evidence
- Cast evaluates its evidence
- Error triggered when (and only when) 'a' must have type Bool

Constraints

let c7: Int~Bool
= error "Can't match ..."

Elaborated program (mentioning constraint variables)

(True, ('a' ▷ c7) && False)
Step 2:

- Use lazily evaluated "error" evidence
- Cast evaluates evidence
- Error triggered when (and only when) 'a' must have type Bool

Solve

\[
\text{let } c7 : \text{Int } \sim \text{Bool} = \text{error "Can't match ..."}
\]

Constraints

**c7 : Int ~ Bool**

Elaborated program (mentioning constraint variables)

**((True, ('a' ▼ c7) && False))**

Uh oh! What became of coercion erasure?
Hole constraints (a new form of constraint)

Haskell term: `True && _`

- **Generate constraints**:
  - `h7 : Hole β`
  - `β ~ Bool`

- **Elaborated program**:
  - `(True && h7)`

Constraints:

Elaborated program (mentioning constraint variables)
Hole constraints...

- Again use lazily evaluated “error” evidence
- Error triggered when (and only when) the hole is evaluated

Constraints:

- `h7 : Hole Bool`

Elaborated program (mentioning constraint variables):

```plaintext
let h7: Bool = error "Evaluated hole"

(True && h7)
```
A FLY IN THE OINTMENT
Generalisation (Hindley-Milner)

f :: Int -> Float -> (Int, Float)
f x y = let g v = v+v
       in (g x, g y)

- We need to infer the most general type for
  g :: ∀a. Num a => a -> a
  so that it can be called at Int and Float

- Generate constraints for g's RHS, simplify
  them, quantify over variables not free in the
  environment

- BUT: what happened to "generate then solve"?
A more extreme example

data T a where
  C :: T Bool
  D :: a -> T a

f :: T a -> a -> Bool
f v x = case v of
  C -> let y = not x
  in y
  D x -> True

Should this typecheck?

In the C alternative, we know a~Bool
data T a where
  C :: T Bool
  D :: a -> T a

f :: T a -> a -> Bool
f v x = let y = not x
  in case v of
    C -> y
    D x -> True

What about this?
Constraint a~Bool arises from RHS
A more extreme example

data T a where
  C :: T Bool
  D :: a -> T a

f :: T a -> a -> Bool
f v x = let y () = not x
     in case v of
         C -> y ()
         D x -> True
A more extreme example

data T a where
  C :: T Bool
  D :: a -> T a

f :: T a -> a -> Bool
f v x = let
  y :: (a~Bool) => () -> Bool
  y () = not x
  in case v of
    C -> y ()
    D x -> True

But this surely should!

Here we abstract over the a~Bool constraint
Abstract over all unsolved constraints from RHS

- Big types, unexpected to programmer
- Errors postponed to usage sites
- Have to postpone ALL unification
- (Serious) Sharing loss for thunks
- (Killer) Can’t abstract over implications

\[ f :: (\forall a. (a \sim [b]) \Rightarrow b \sim \text{Int}) \Rightarrow \text{blah} \]
A much easier path

Do not generalise local let-bindings at all!

- Simple, straightforward, efficient
- Polymorphism is almost never used in local bindings (see “Modular type inference with local constraints”, JFP)
- GHC actually generalises local bindings that could have been top-level, so there is no penalty for localising a definition.
EFFICIENT EQUALITIES
Questions you might like to ask

- Is this all this coercion faff efficient?
- ML typechecking has zero runtime cost; so anything involving these casts and coercions looks inefficient, doesn't it?


Remember deferred type errors: cast must evaluate its coercion argument.

What became of erasure?

let c7: Bool~Bool = refl Bool in (x ▷ c7) && False)
data Int = I# Int#

plusInt :: Int -> Int -> Int
plusInt x y
  = case x of I# a ->
    case y of I# b ->
      I# (a +# b)
So (~#) is the primitive type constructor

(▷#) is the primitive language construct

And (▷#) is erasable
Implementing ~#

data T where
  T1 :: ∀a. (a~#Bool) -> Double# -> Bool -> T a

A T1 value allocated in the heap looks like this

<table>
<thead>
<tr>
<th>T1</th>
<th>???</th>
<th>3.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bits</td>
<td>32 bits</td>
<td></td>
</tr>
</tbody>
</table>

**Question**: what is the representation for (a~#Bool)?

True
data T where
  T1 :: ∀a. (a~#Bool) -> Double# -> Bool -> T a

A T1 value allocated in the heap looks like this

<table>
<thead>
<tr>
<th>0 bits</th>
<th>64 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>3.8</td>
<td></td>
</tr>
</tbody>
</table>

**Question**: what is the representation for (a~#Bool)?

**Answer**: a 0-bit value
Boxed and primitive equality

data a ~ b = Eq# (a ~# b)

- User API and type inference deal exclusively in boxed equality (a~b)
- Hence all evidence (equalities, type classes, implicit parameters...) is uniformly boxed
- Ordinary, already-implemented optimisation unwrap almost all boxed equalities.
- Unboxed equality (a~#b) is represented by 0-bit values. Casts are erased.
- Possibility of residual computations to check termination
Modular type inference with local assumptions (JFP 2011). Epic paper.

Practical type inference for arbitrary-rank types (JFP 2007). Full executable code; but does not use the Glorious French Approach.