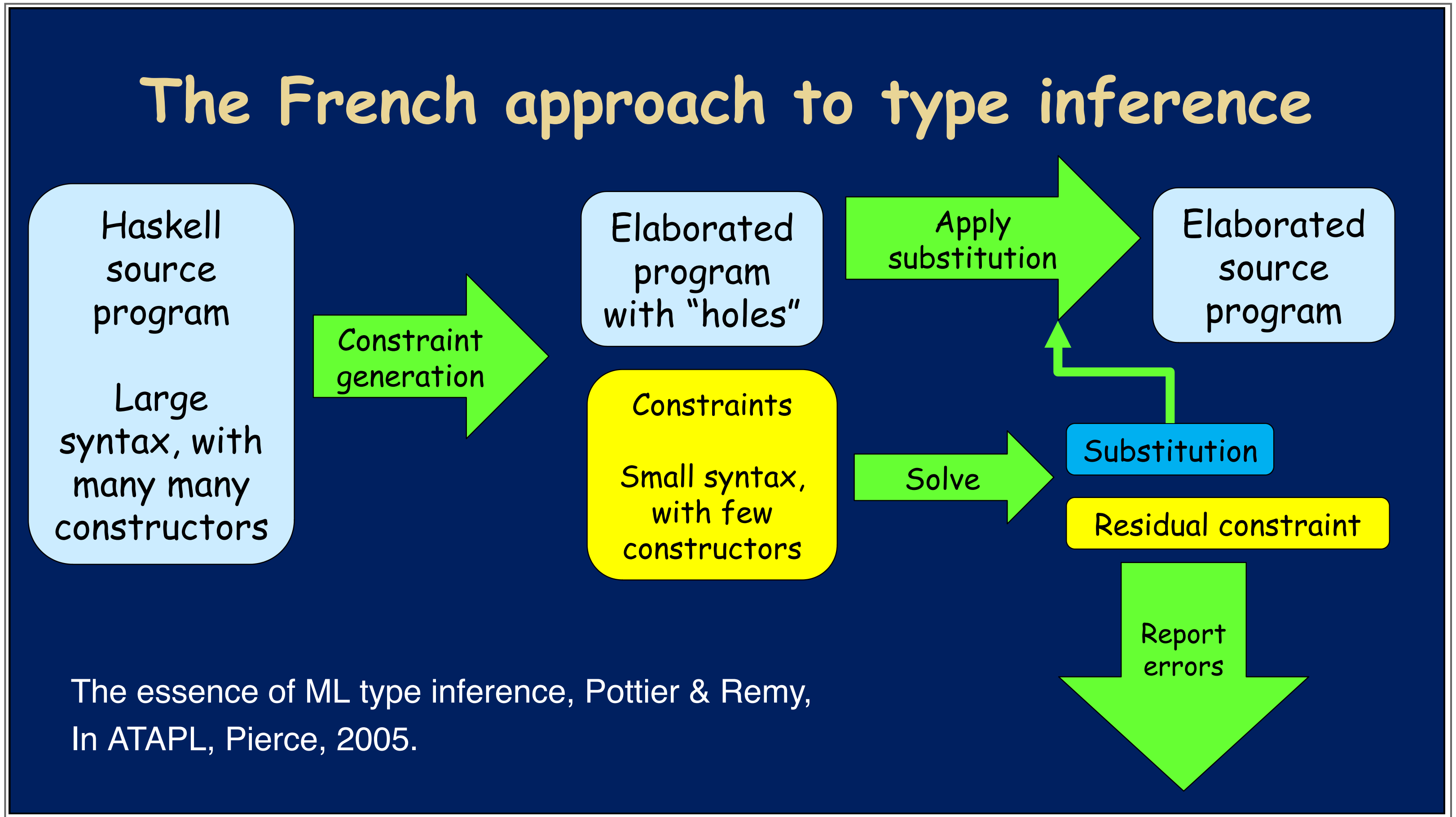
GHC’s constraint solver

Haskell eXchange

Sam Derbyshire, Well-Typed

November 16th, 2021


The French approach to type inference
Haskell source program
Large syntax, with many many constructors
Constraint generation
Elaborated program with "holes"
Constraints
Small syntax, with few constructors
Apply substitution
Elaborated source program
Solve
Substitution
Residual constraint
Report errors
The essence of ML type inference, Pottier & Remy,
In ATAPL, Pierce, 2005.

Damas–Hindley–Milner

```
foo xs = [ listToMaybe xs, Just (length xs) ]
```

```
foo :: γ
xs :: α
(:) :: δ -> [δ] -> [δ]
listToMaybe :: [ε] -> Maybe ε
Just :: ζ -> Maybe ζ
length :: [η] -> Int
```

Work list

```
δ ~ Maybe ζ
[η] ~ α
Maybe ε ~ Maybe ζ
[η] ~ [ε]
```

Inert set

This is **Algorithm W** from Damas–Hindley–Milner type theory.

It always infers the most general type.

```
δ ~ Maybe ε
```

ζ ~ Int

ε ~ ζ

Elaboration

```
foo xs = [ listToMaybe xs, Just (length xs) ]

                   e Int]

                     s

                     @Int xs )

    [] @(Maybe Int)
```

Scaling up to Haskell

```
                       { F (Maybe a) = a }
        -> a -> a

f x y = [ g x y, Just (not x) ]
```

```
f :: α -> β -> γ

(:) :: δ -> [δ] -> [δ]

g :: F ε -> ε -> ε
```

Work list

```
α ~ Bool

δ ~ Maybe Bool

α ~ F ε
```

Inert set

```
γ ~ [δ]

α ~ F ε

β ~ ε
```

ε ~ δ

α ~ Bool

Typeclasses, implications

```
palindrome ds = ds == reverse ds

= \ @a ($dEq_a :: Eq a) ->

in \ (ds :: [a]) -> (==) @[a] $dEq_List_a ds (reverse @a ds)
```

Given: `Eq a`

Wanted: `Eq [a]`

Solve the implication `[G] Eq a ⊢ [W] Eq [a]` using the dictionary function `$fEq_List :: Eq a -> Eq [a]`.

Nested implications

```
mily F a where { F Int = Int; F (f a) = a }

: Integral b => Maybe c -> b -> G (Maybe c)

G a -> F a

MkG2 m b -> fromMaybe (fromIntegral b) m

c
⊢ [W] Integral b, [W] Num c ]
```

Part II: constraint solving

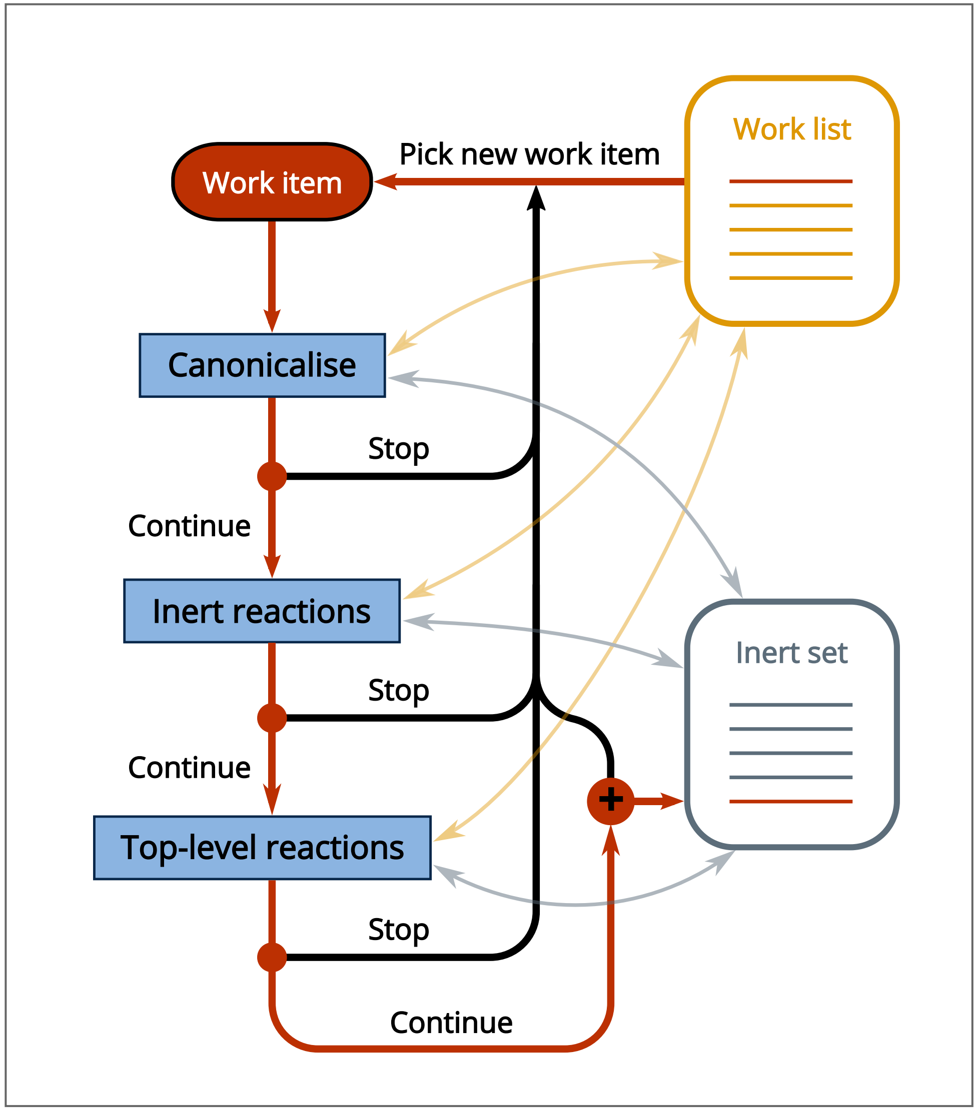
Predicates

Different kinds of constraints have different kinds of evidence:

- typeclass constraints have a dictionary of the methods,
- an equality is witnessed by a coercion (proof term).

Predicate	Examples	Evidence
Typeclass	`Ord a`, `Num a`, `(c1, c2)`, `()`, `a ~ b`	Dictionary
Equality	`a ~# b`, `a ~R# b`	Coercion
Quantified	`∀ a. Eq a => Eq (f a)`	Function
Irreducible	`c a`, `F x y`	Not yet known

Solving flat constraints



Rewriting

When we add a new equality `co :: old_ty ~ new_ty` to the inert set, we kick out constraints that can be rewritten using `co`, adding them back to the work list to be processed again.

```
[W] a ~ Maybe b
[W] Eq a
```

```
[W] $dEq :: Eq a
[W] $dNum :: Num b
[W] co :: a ~ Maybe b
[W] $dEq |> Eq co :: Eq (Maybe b)
```

Decomposition

```
                         ))

⇝ a ~ x, b ~ y, c ~ z

⇝ a ~R# b -- if the newtype constructor for Nt is in scope
```

Canonicalisation

A **canonical** constraint is one that is in **atomic** form: it can't be decomposed or rewritten in any way.

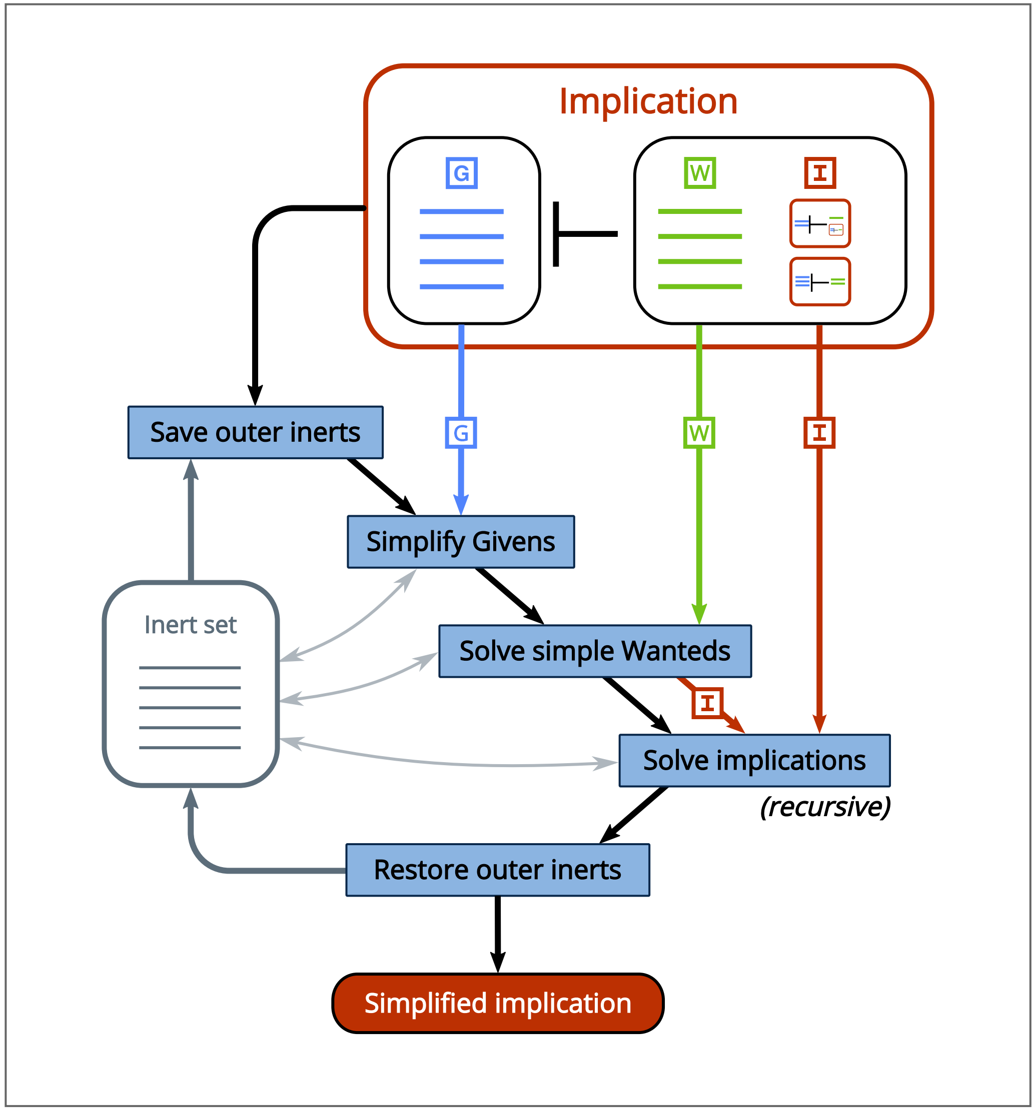
```
int
re

TyFamCt (a,b) = a ~ b

, g b)

⇝ f ~# g, a ~# b
```

Solving implications



```
type family F a where { F Int = Int, F (f a) = a }
```

```
                                                        c
          ⊢ [W] Integral b, [W] Num c ]
```

Work list

```
[G] co :: a ~ Int
[W] F a ~ Int
[G] $dNum :: Num (F a)
[W] (F co ; F[0]) :: F a ~ Int
[G] $dIntegral :: Integral b
[G] co :: a ~ Maybe c
[W] Integral b
```

```
[W] Num c
[G] $dNum :: Num (F a)
```

Questions?

Slides available online:

Inert set

```
[G] $dNum :: Num (F a)
[G] co :: a ~ Int
[G] ($dNum |> Num co) :: Num Int
[G] $dNum :: Num (F a)
[G] $dIntegral :: Integral b
[G] co :: a ~ Maybe c
[W] Num c
[G] ($dNum |> Num (F co ; F[1])) :: Num c
```

sheaf.github.io/ghc-constraint-solver