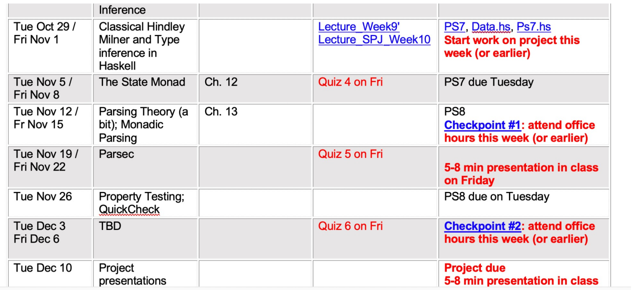
Type Inference in Haskell

1

Schedule



	Inference			
Tue Oct 29 / Fri Nov 1	Classical Hindley Milner and Type inference in Haskell		Lecture_Week9' Lecture_SPJ_Week10	PS7, Data.hs, Ps7.hs **Start work on project this week (or earlier)**
Tue Nov 5 / Fri Nov 8	The State Monad	Ch. 12	Quiz 4 on Fri	PS7 due Tuesday
Tue Nov 12 / Fr Nov 15	Parsing Theory (a bit); Monadic Parsing	Ch. 13		PS8 **Checkpoint #1: attend office hours this week (or earlier)**
Tue Nov 19 / Fri Nov 22	Parsec		Quiz 5 on Fri	**5-8 min presentation in class on Friday**
Tue Nov 26	Property Testing; QuickCheck			PS8 due on Tuesday
Tue Dec 3 Fri Dec 6	TBD		Quiz 6 on Fri	**Checkpoint #2: attend office hours this week (or earlier)**
Tue Dec 10	Project presentations			**Project due** **5-8 min presentation in class**

2

Outline

- Simple type inference
  - Expressions, types and type environment
  - Goal and intuition
  - Equality constraints ✓
  - Substitution ✓
  - Robinson's unification ✓
  - Type inference strategies
    - Algorithm V (Strategy One) and
    - Algorithm V (Strategy Two)

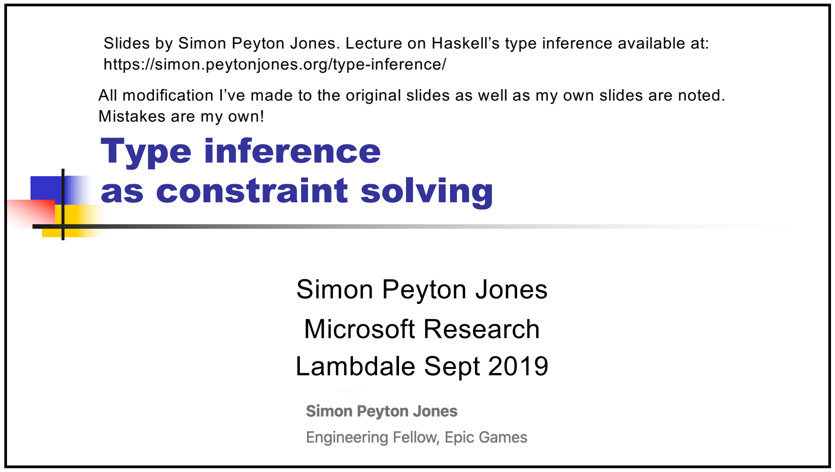
3

Outline

- Hindley Milner (also known as Milner Damas)
  - Monotypes (types) and polytypes (type schemes)
  - Instantiation and generalization
  - Algorithm W
  - Observations
- Now that we've seen classical Hindley Milner… Haskell!

  How to extend classical system to account for
  - Type signatures
  - Pattern matching
  - Type classes
  - Strategy One vs Hindley Milner's Strategy Two

4



Slides by Simon Peyton Jones. Lecture on Haskell's type inference available at:
https://simon.peytonjones.org/type-inference/

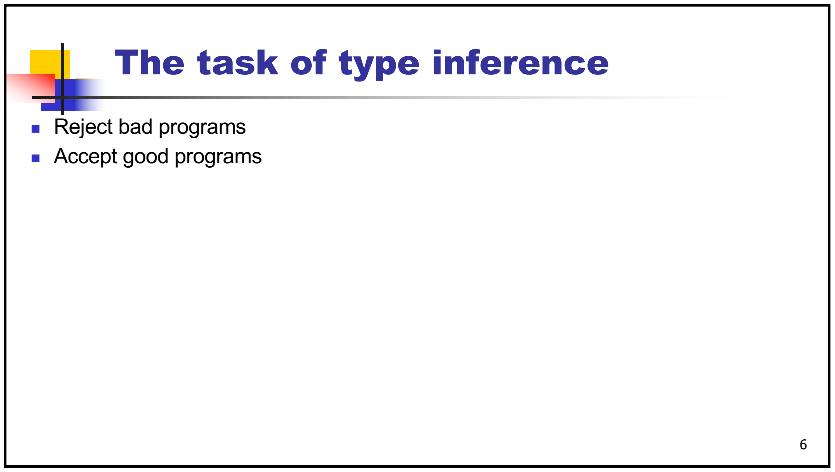
All modification I've made to the original slides as well as my own slides are noted. Mistakes are my own!

Type inference as constraint solving

Simon Peyton Jones
Microsoft Research
Lambdale Sept 2019

Simon Peyton Jones
Engineering Fellow, Epic Games

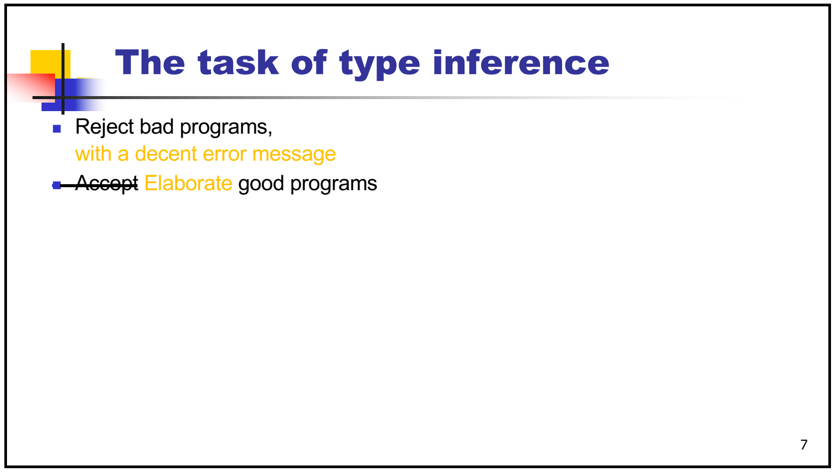
5



The task of type inference

- Reject bad programs
- Accept good programs

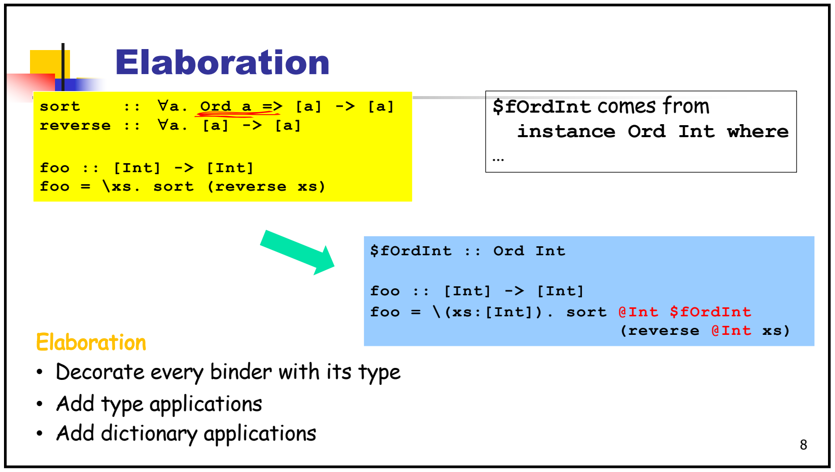
6



The task of type inference

- Reject bad programs,
  with a decent error message
- ~~Accept~~ Elaborate good programs

7



Elaboration

```
sort    :: ∀a. Ord a => [a] -> [a]
reverse :: ∀a. [a] -> [a]

foo :: [Int] -> [Int]
foo = \xs. sort (reverse xs)
```

$fOrdInt comes from

```
  instance Ord Int where
…
```

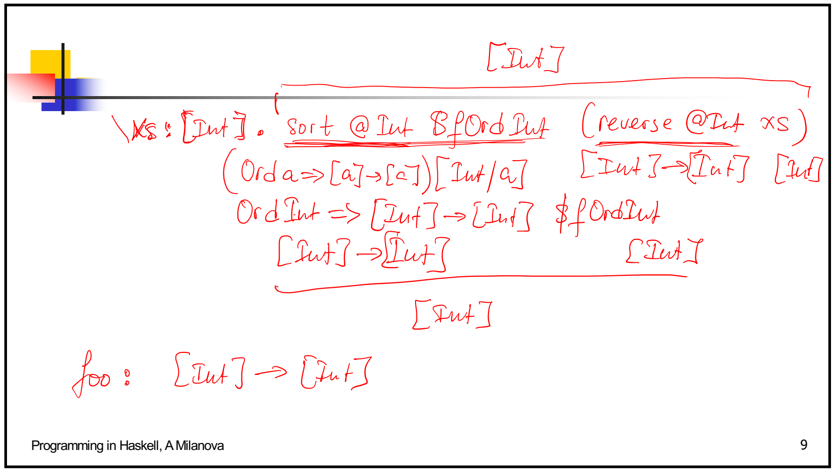
```
$fOrdInt :: Ord Int

foo :: [Int] -> [Int]
foo = \(xs:[Int]). sort @Int $fOrdInt
                        (reverse @Int xs)
```

Elaboration

- Decorate every binder with its type
- Add type applications
- Add dictionary applications

8



[Int]

\xs : [Int] . sort @Int $fOrdInt (reverse @Int xs)

(Ord a => [a] -> [a]) [Int/a]   [Int] -> [Int]   [Int]

Ord Int => [Int] -> [Int] $fOrdInt

[Int] -> [Int]   [Int]

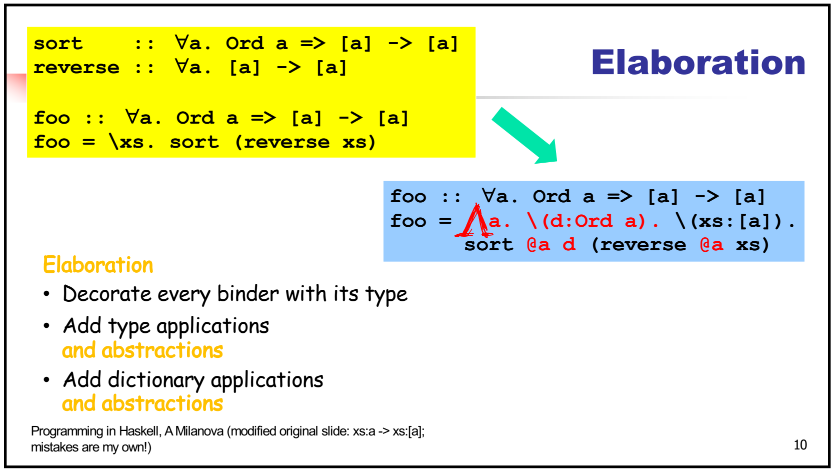
[Int]

foo : [Int] -> [Int]

Programming in Haskell, A Milanova

9

9



Elaboration

```
sort    :: ∀a. Ord a => [a] -> [a]
reverse :: ∀a. [a] -> [a]

foo :: ∀a. Ord a => [a] -> [a]
foo = \xs. sort (reverse xs)
```

```
foo :: ∀a. Ord a => [a] -> [a]
foo = Λa. \(d:Ord a). \(xs:[a]).
      sort @a d (reverse @a xs)
```

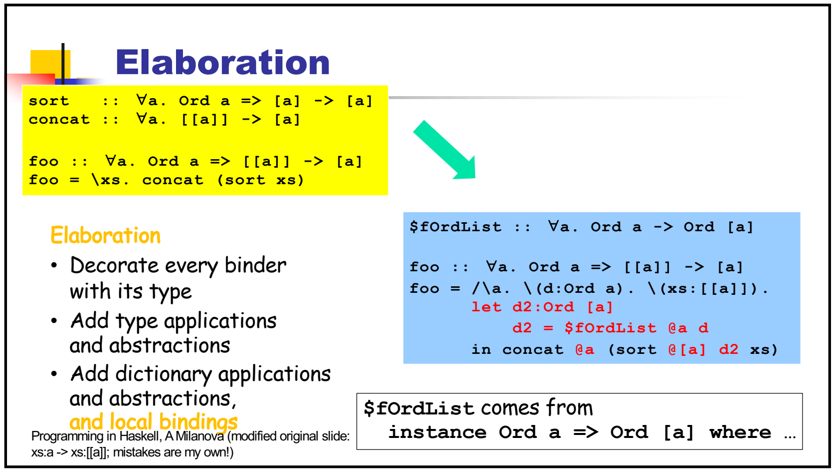
Elaboration

- Decorate every binder with its type
- Add type applications and abstractions
- Add dictionary applications and abstractions

Programming in Haskell, A Milanova (modified original slide: xs:a -> xs:[a]; mistakes are my own!)

10

10



Elaboration

```
sort   :: ∀a. Ord a => [a] -> [a]
concat :: ∀a. [[a]] -> [a]

foo :: ∀a. Ord a => [[a]] -> [a]
foo = \xs. concat (sort xs)
```

```
$fOrdList :: ∀a. Ord a -> Ord [a]

foo :: ∀a. Ord a => [[a]] -> [a]
foo = /\a. \(d:Ord a). \(xs:[[a]]).
    let d2:Ord [a]
        d2 = $fOrdList @a d
    in concat @a (sort @[a] d2 xs)
```

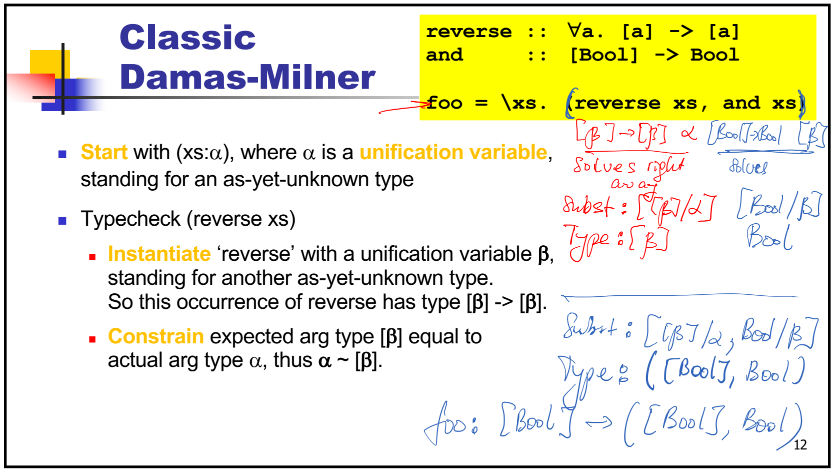
Elaboration

- Decorate every binder with its type
- Add type applications and abstractions
- Add dictionary applications and abstractions, and local bindings

$fOrdList comes from
```
  instance Ord a => Ord [a] where ...
```

Programming in Haskell, A Milanova (modified original slide: xs:a -> xs:[[a]]; mistakes are my own!)

11



Classic Damas-Milner

```
reverse :: ∀a. [a] -> [a]
and     :: [Bool] -> Bool

foo = \xs. (reverse xs, and xs)
```

[β] -> [β]   α   [Bool]->Bool   [β]

Solves right away   Solves

Subst : [[β]/α]   [Bool/β]

Type : [β]   Bool

- **Start** with (xs:α), where α is a **unification variable**, standing for an as-yet-unknown type
- Typecheck (reverse xs)
  - **Instantiate** 'reverse' with a unification variable **β**, standing for another as-yet-unknown type. So this occurrence of reverse has type [β] -> [β].
  - **Constrain** expected arg type [β] equal to actual arg type α, thus **α ~ [β]**.

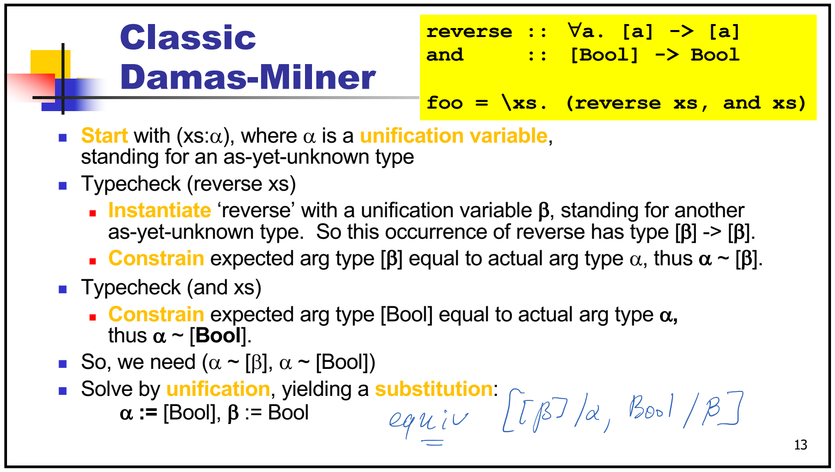
Subst : [[β]/α, Bool/β]

Type : ([Bool], Bool)

foo : [Bool] -> ([Bool], Bool)

12

12



Classic Damas-Milner

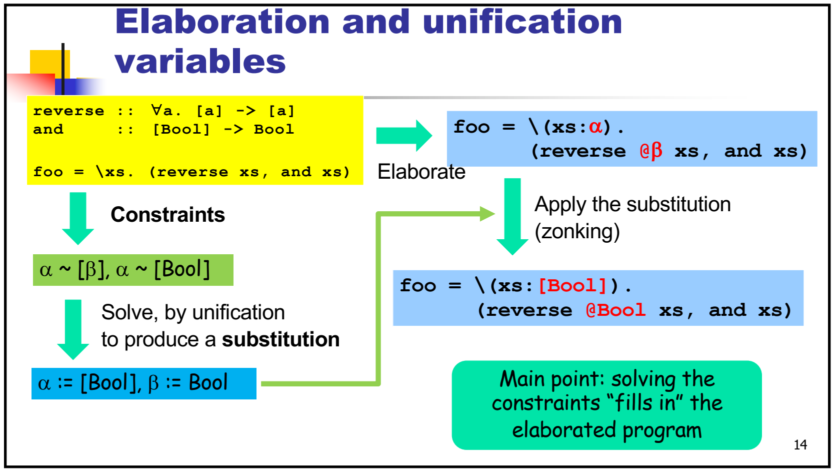
```
reverse :: ∀a. [a] -> [a]
and     :: [Bool] -> Bool

foo = \xs. (reverse xs, and xs)
```

- Start with (xs:α), where α is a unification variable, standing for an as-yet-unknown type
- Typecheck (reverse xs)
  - Instantiate ‘reverse’ with a unification variable **β**, standing for another as-yet-unknown type. So this occurrence of reverse has type [β] -> [β].
  - Constrain expected arg type [β] equal to actual arg type α, thus **α** ~ [β].
- Typecheck (and xs)
  - Constrain expected arg type [Bool] equal to actual arg type **α,** thus **α** ~ [**Bool**].
- So, we need (α ~ [β], α ~ [Bool])
- Solve by unification, yielding a substitution:
  α := [Bool], **β** := Bool equiv [[β]/α, Bool/β]

13

13



Elaboration and unification variables

```
reverse :: ∀a. [a] -> [a]
and     :: [Bool] -> Bool

foo = \xs. (reverse xs, and xs)
```

Elaborate

```
foo = \(xs:α).
        (reverse @β xs, and xs)
```

Constraints

α ~ [β], α ~ [Bool]

Solve, by unification
to produce a **substitution**

α := [Bool], β := Bool

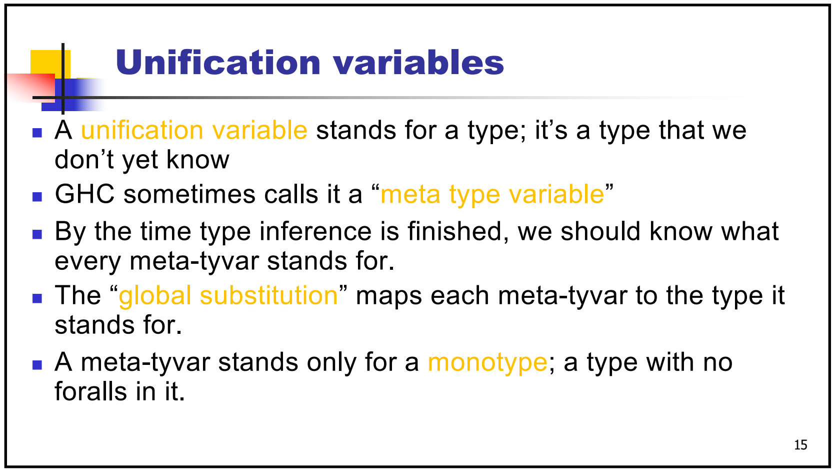
Apply the substitution
(zonking)

```
foo = \(xs:[Bool]).
        (reverse @Bool xs, and xs)
```

Main point: solving the constraints “fills in” the elaborated program

14

14

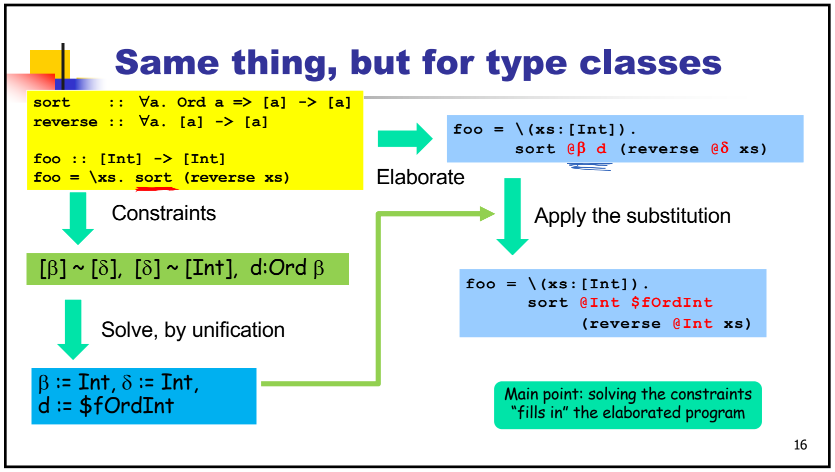


Unification variables

- A unification variable stands for a type; it’s a type that we don’t yet know
- GHC sometimes calls it a “meta type variable”
- By the time type inference is finished, we should know what every meta-tyvar stands for.
- The “global substitution” maps each meta-tyvar to the type it stands for.
- A meta-tyvar stands only for a monotype; a type with no foralls in it.

15

15



Same thing, but for type classes

```
sort    :: ∀a. Ord a => [a] -> [a]
reverse :: ∀a. [a] -> [a]

foo :: [Int] -> [Int]
foo = \xs. sort (reverse xs)
```

Elaborate

```
foo = \(xs:[Int]).
        sort @β d (reverse @δ xs)
```

Constraints

[β] ~ [δ], [δ] ~ [Int], d:Ord β

Solve, by unification

β := Int, δ := Int,
d := $fOrdInt

Apply the substitution

```
foo = \(xs:[Int]).
        sort @Int $fOrdInt
             (reverse @Int xs)
```

Main point: solving the constraints “fills in” the elaborated program

16

16

Deferring solving

- Old school: "on the fly solving"
  - Encounter a unification problem
  - Solve it
  - If fails, report error
  - Otherwise, proceed
- This will not work any more

[β] ~ [δ], [δ] ~ [Int], d:Ord β

We have to solve β := Int,
before we can solve d:Ord β

Main point

The order in which we encounter constraints

≠

The order in which we solve them

Deferring solving

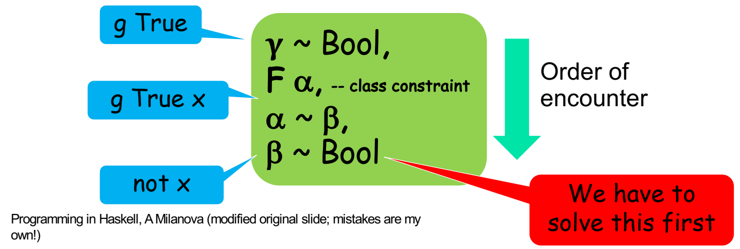
```
g :: ∀a,b. F a => b -> a -> Int
instance F Bool

f x = (g True x, ...., not x)
```

✓ F: Bool → $ FBool

- x::β
- Instantiate g: γ -> α -> Int

b ↑ (γ) a ↑ (α)



Programming in Haskell, A Milanova (modified original slide; mistakes are my own!)

An aside

```
f y = let twice f x = f (f x)
      in twice twice (+1) y
```

f: α , x : β
Constraints:
α ~ β → γ From (f x)
α ~ γ → δ From f (f x)
Type:
α -> β -> δ

- How do we generalize and instantiate with Strategy One?

Generalize both constraints and type?

∀ α, β, γ, δ. =>
{ α ~ β → γ
α ~ γ → δ }

Type: ∀ α, β, δ . α -> β -> δ

Then instantiate constraints and type?

α' ~ β' → γ' Type: α' -> β' -> δ'
α' ~ γ' → δ'

α''~ β'' → γ'' Type: α'' -> β'' -> δ''
α'' ~ γ'' → δ''

Programming in Haskell, A Milanova (my slide, mistakes are my own!)

My guess

```
f y = let twice f x = f (f x)
      in twice twice (+1) y
```

f: α , x : β
Constraints:
α ~ β → γ From (f x)
α ~ γ → δ From f (f x)
Type:
α -> β -> δ

- How do we generalize and instantiate with Strategy One?

Solve as much as possible before generalization (or otherwise will be solving same constraints twice):

∀ α, β, γ, δ. => { }

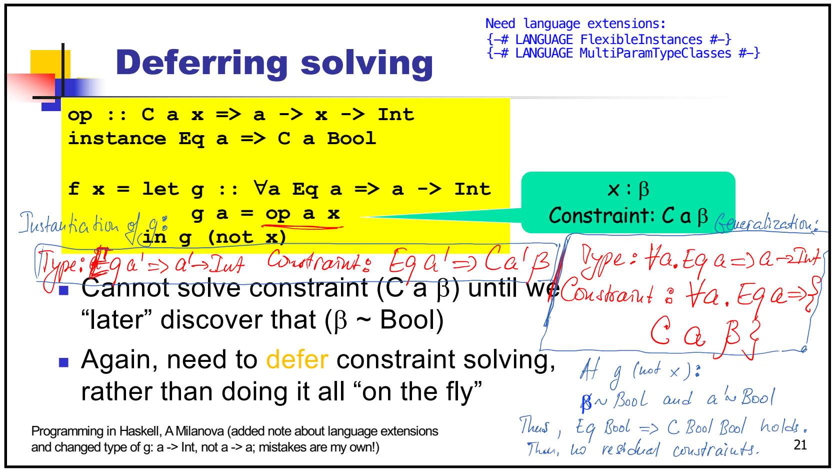
Type: ∀ β. (β -> β) -> β -> β

Then instantiate constraints and type:

Type: (β' -> β') -> β' -> β'

Type: (β'' -> β'') -> β'' -> β''

Programming in Haskell, A Milanova (my slide, mistakes are my own!)



Deferring solving

Need language extensions:
`{-# LANGUAGE FlexibleInstances #-}`
`{-# LANGUAGE MultiParamTypeClasses #-}`

```
op :: C a x => a -> x -> Int
instance Eq a => C a Bool

f x = let g :: ∀a Eq a => a -> Int
          g a = op a x
      in g (not x)
```

x : β
Constraint: C a β

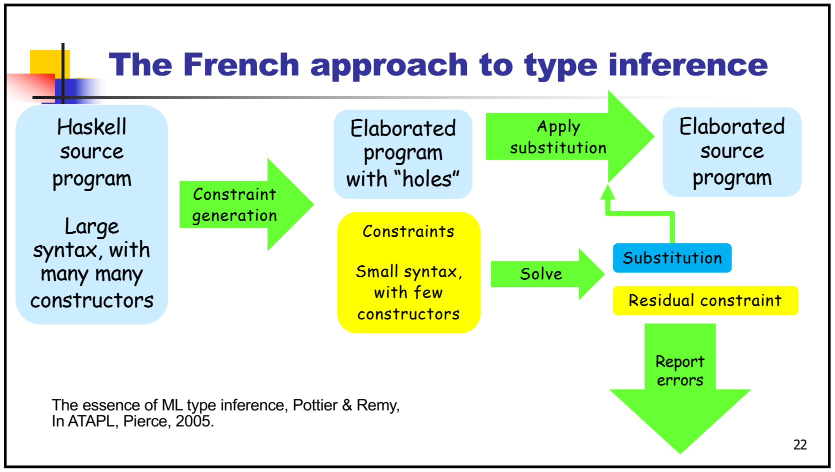
Instantiation of g:
Type: Eq a' => a'→Int Constraint: Eq a' => C a' β

Generalization:
Type: ∀a. Eq a => a→Int
Constraint: ∀a. Eq a => {C a β}

- Cannot solve constraint (C a β) until we "later" discover that (β ~ Bool)
- Again, need to defer constraint solving, rather than doing it all "on the fly"

At g (not x):
β ~ Bool and a' ~ Bool
Thus, Eq Bool => C Bool Bool holds.
Thus, no residual constraints.

Programming in Haskell, A Milanova (added note about language extensions and changed type of g: a -> Int, not a -> a; mistakes are my own!)



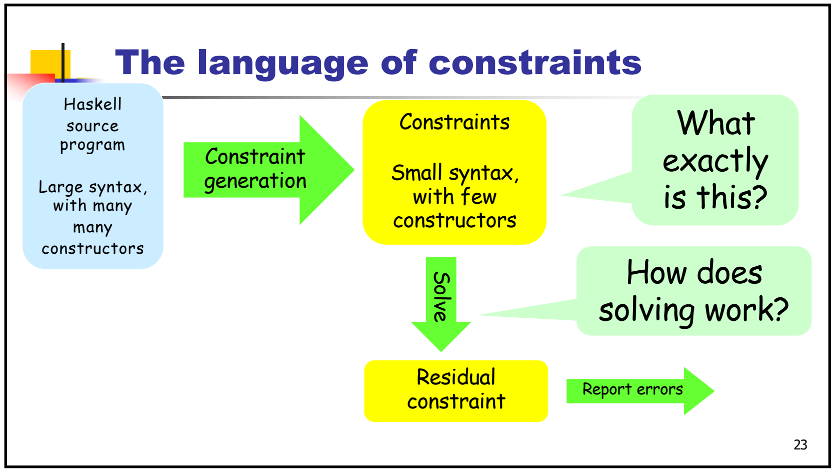
The French approach to type inference

Haskell source program

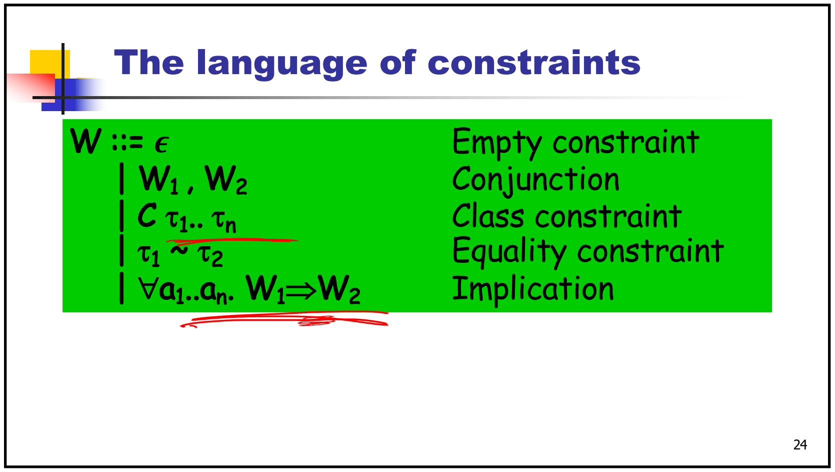
Large syntax, with many many constructors
Constraint generation
Elaborated program with "holes"
Constraints

Small syntax, with few constructors
Solve
Substitution
Residual constraint
Apply substitution
Elaborated source program
Report errors

The essence of ML type inference, Pottier & Remy, In ATAPL, Pierce, 2005.



The language of constraints



The language of constraints

$W ::= \epsilon$	Empty constraint
$\mid W_1 , W_2$	Conjunction
$\mid C\ \tau_1 .. \tau_n$	Class constraint
$\mid \tau_1 \sim \tau_2$	Equality constraint
$\mid \forall a_1..a_n.\ W_1 \Rightarrow W_2$	Implication

The language of constraints

W ::= ϵ	Empty constraint
| W_1 , W_2	Conjunction
| d : C $\tau_1 .. \tau_n$	Class constraint
| g : $\tau_1 \sim \tau_2$	Equality constraint
| $\forall a_1..a_n.\ W_1 \Rightarrow W_2$	Implication

Evidence

How solving works

1. Take the constraints
2. Do one rewrite
3. Repeat from 1

- Each step takes a set of constraints and returns a logically-equivalent set of constraints.
- When you can't do any more, that's the "residual constraint"

$[\beta] \sim [\delta],\ [\delta] \sim [\mathrm{Int}],\ d{:}\mathrm{Ord}\ \beta$

Decompose $[\beta] \sim [\delta]$

$\beta \sim \delta,\ [\delta] \sim [\mathrm{Int}],\ d{:}\mathrm{Ord}\ \beta$

Substitute $\beta \coloneqq \delta$

$[\delta] \sim [\mathrm{Int}],\ d{:}\mathrm{Ord}\ \delta$ | $\beta \coloneqq \delta$

Decompose $[\delta] \sim [Int]$

$\delta \sim \mathrm{Int},\ d{:}\mathrm{Ord}\ \delta$

Substitute $\delta \coloneqq Int$

$d{:}\mathrm{Ord}\ Int$ | $\beta \coloneqq \delta$, $\delta \coloneqq Int$

Solve $d{:}\,Ord\ Int$ from instance declaration

ϵ

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

len [] = 0
len (x:xs) = len xs +1

Pattern match

```
data [a] = (:) a [a] | []
(:) :: ∀a. a -> [a] -> [a]
```

```
len =
  \xs. case xs of
     (:) x xs' -> len xs' + 1
           [] -> 0
```

```
len =
  \(xs:α). case xs of
     (:) a' (x:a') (xs':[a'])
         -> len xs' + 1
```

- At pattern match, instantiate **a** with fresh a'
- len :: α -> β
- xs :: α

Constraints

α ~ [a']	From case, and from call len xs'
Num β	From len xs' + 1

Solve and substitute

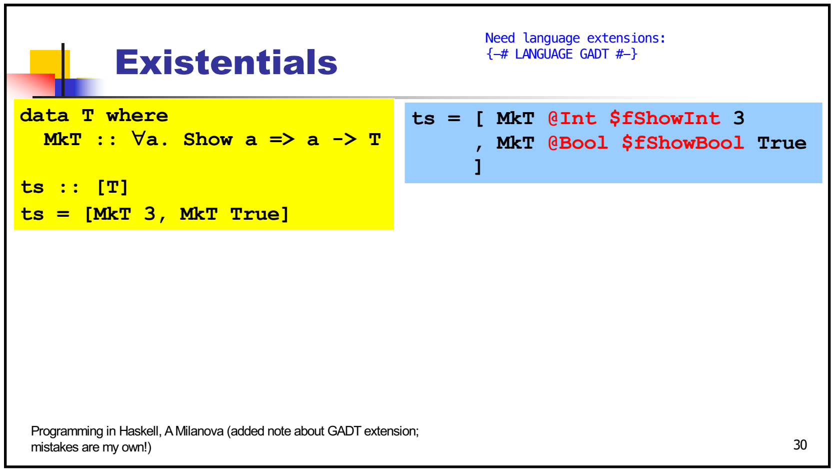
len :: ∀a', β. Num β => [a'] -> β

What if there are class constraints on component types? Then we type rhs of -> under assumptions.

Programming in Haskell, A Milanova (my slide; mistakes are my own!)

IMPLICATION CONSTRAINTS

29



Existentials

Need language extensions:
{-# LANGUAGE GADT #-}

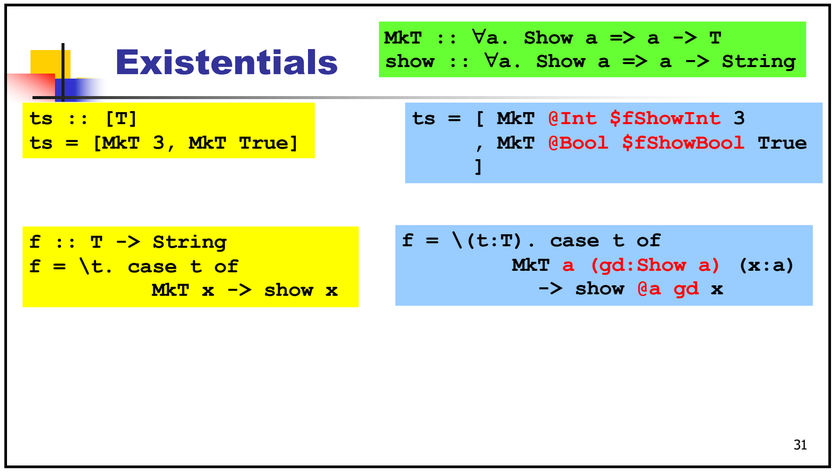
```
data T where
  MkT :: ∀a. Show a => a -> T

ts :: [T]
ts = [MkT 3, MkT True]
```

```
ts = [ MkT @Int $fShowInt 3
     , MkT @Bool $fShowBool True
     ]
```

Programming in Haskell, A Milanova (added note about GADT extension;
mistakes are my own!)

30



Existentials

```
MkT  :: ∀a. Show a => a -> T
show :: ∀a. Show a => a -> String
```

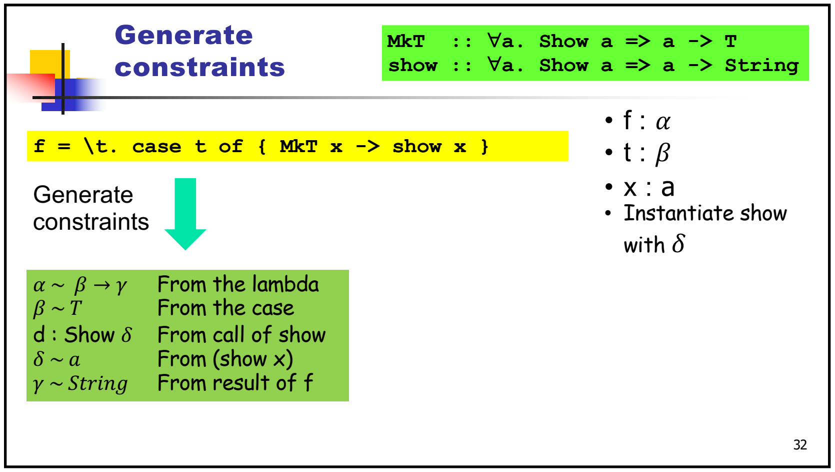
```
ts :: [T]
ts = [MkT 3, MkT True]
```

```
ts = [ MkT @Int $fShowInt 3
     , MkT @Bool $fShowBool True
     ]
```

```
f :: T -> String
f = \t. case t of
          MkT x -> show x
```

```
f = \(t:T). case t of
            MkT a (gd:Show a) (x:a)
              -> show @a gd x
```

31



Generate constraints

```
MkT  :: ∀a. Show a => a -> T
show :: ∀a. Show a => a -> String
```

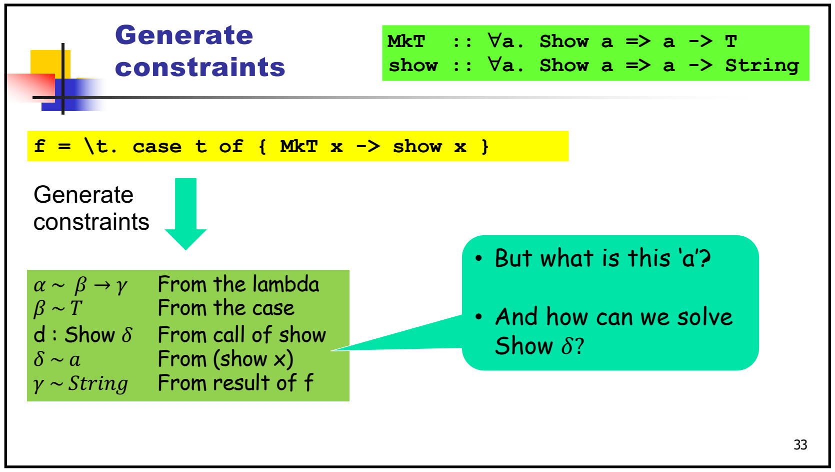
```
f = \t. case t of { MkT x -> show x }
```

- f : α
- t : β
- x : a
- Instantiate show with δ

Generate constraints

$\alpha \sim \beta \to \gamma$	From the lambda
$\beta \sim T$	From the case
d : Show δ	From call of show
$\delta \sim a$	From (show x)
$\gamma \sim String$	From result of f

32



Generate constraints

```
MkT  :: ∀a. Show a => a -> T
show :: ∀a. Show a => a -> String
```

```
f = \t. case t of { MkT x -> show x }
```

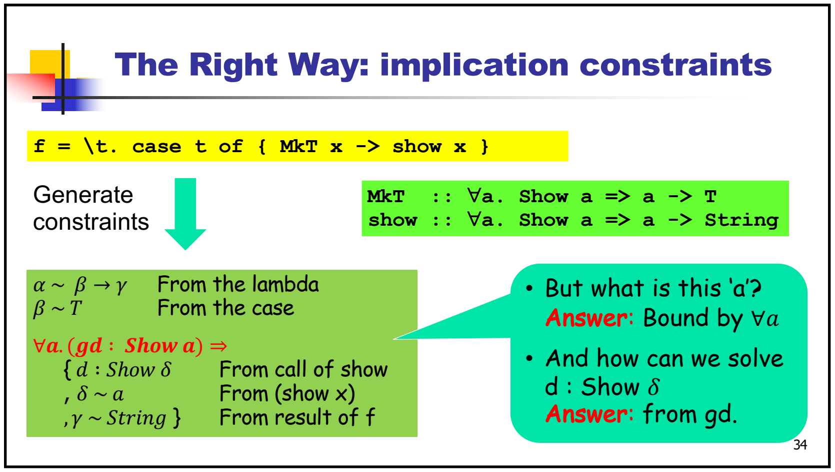
Generate constraints

$\alpha \sim \beta \to \gamma$	From the lambda
$\beta \sim T$	From the case
d : Show δ	From call of show
$\delta \sim a$	From (show x)
$\gamma \sim String$	From result of f

- But what is this 'a'?
- And how can we solve Show δ?

33

33



The Right Way: implication constraints

```
f = \t. case t of { MkT x -> show x }
```

Generate constraints

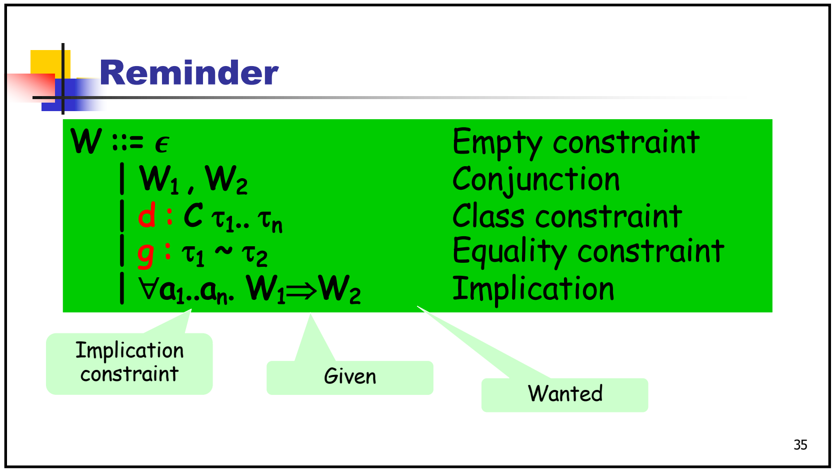
```
MkT  :: ∀a. Show a => a -> T
show :: ∀a. Show a => a -> String
```

$\alpha \sim \beta \to \gamma$	From the lambda
$\beta \sim T$	From the case
$\forall a.(gd : Show\ a) \Rightarrow$	
$\{ d : Show\ \delta$	From call of show
$, \delta \sim a$	From (show x)
$, \gamma \sim String \}$	From result of f

- But what is this 'a'? Answer: Bound by $\forall a$
- And how can we solve d : Show δ Answer: from gd.

34

34



Reminder

W ::= ϵ	Empty constraint
| W_1 , W_2	Conjunction
| d : C $\tau_1 .. \tau_n$	Class constraint
| g : $\tau_1 \sim \tau_2$	Equality constraint
| $\forall a_1..a_n.\ W_1 \Rightarrow W_2$	Implication

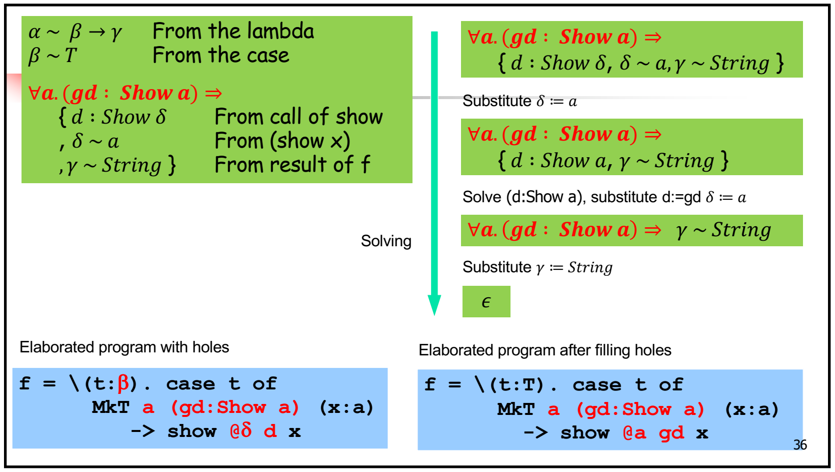
Implication constraint

Given

Wanted

35

35



$\alpha \sim \beta \to \gamma$	From the lambda
$\beta \sim T$	From the case
$\forall a.(gd : Show\ a) \Rightarrow$	
$\{ d : Show\ \delta$	From call of show
$, \delta \sim a$	From (show x)
$, \gamma \sim String \}$	From result of f

Solving

$\forall a.(gd : Show\ a) \Rightarrow \{ d : Show\ \delta, \delta \sim a, \gamma \sim String \}$

Substitute $\delta := a$

$\forall a.(gd : Show\ a) \Rightarrow \{ d : Show\ a, \gamma \sim String \}$

Solve (d:Show a), substitute d:=gd $\delta := a$

$\forall a.(gd : Show\ a) \Rightarrow \gamma \sim String$

Substitute $\gamma := String$

ϵ

Elaborated program with holes

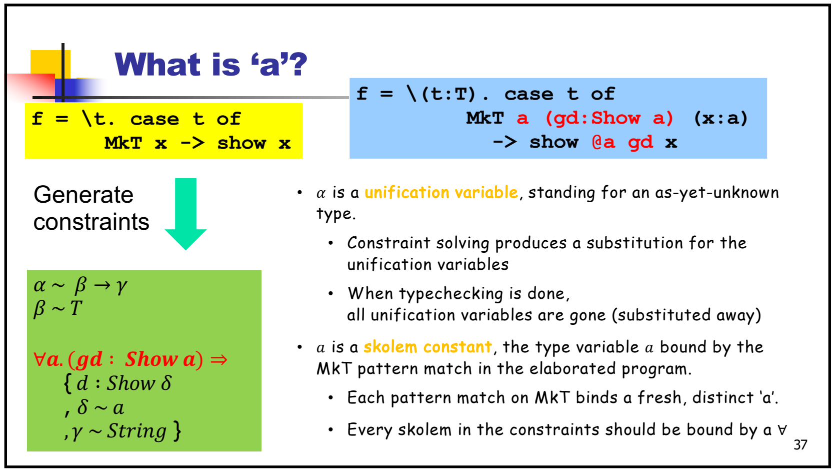
```
f = \(t:β). case t of
        MkT a (gd:Show a) (x:a)
          -> show @δ d x
```

Elaborated program after filling holes

```
f = \(t:T). case t of
        MkT a (gd:Show a) (x:a)
          -> show @a gd x
```

36

36



What is 'a'?

```
f = \t. case t of
      MkT x -> show x
```

```
f = \(t:T). case t of
      MkT a (gd:Show a) (x:a)
        -> show @a gd x
```

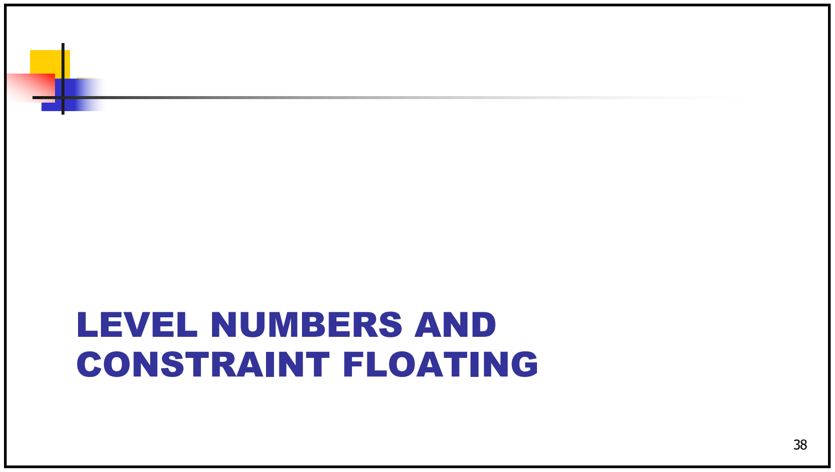
Generate constraints

$\alpha \sim \beta \to \gamma$
$\beta \sim T$

$\forall a.(gd : \mathit{Show}\ a) \Rightarrow$
$\{\, d : \mathit{Show}\ \delta$
$,\ \delta \sim a$
$,\gamma \sim \mathit{String}\,\}$

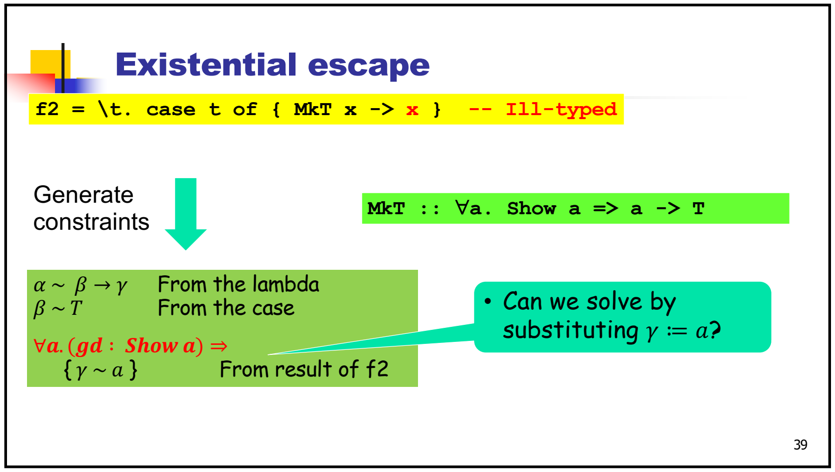
- α is a unification variable, standing for an as-yet-unknown type.
  - Constraint solving produces a substitution for the unification variables
  - When typechecking is done, all unification variables are gone (substituted away)
- a is a skolem constant, the type variable a bound by the MkT pattern match in the elaborated program.
  - Each pattern match on MkT binds a fresh, distinct 'a'.
  - Every skolem in the constraints should be bound by a ∀

37



LEVEL NUMBERS AND CONSTRAINT FLOATING

38



Existential escape

```
f2 = \t. case t of { MkT x -> x }   -- Ill-typed
```

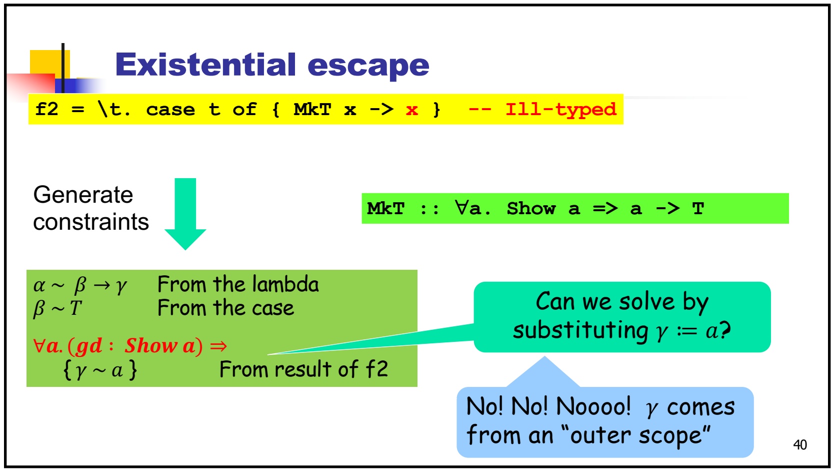
Generate constraints

```
MkT :: ∀a. Show a => a -> T
```

$\alpha \sim \beta \to \gamma$ From the lambda
$\beta \sim T$ From the case
$\forall a.(gd : \mathit{Show}\ a) \Rightarrow$
$\{\,\gamma \sim a\,\}$ From result of f2

- Can we solve by substituting $\gamma := a$?

39



Existential escape

```
f2 = \t. case t of { MkT x -> x }   -- Ill-typed
```

Generate constraints

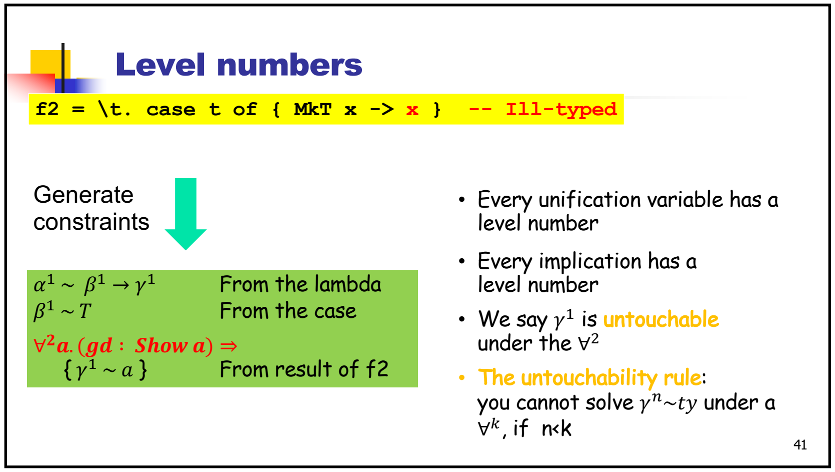
```
MkT :: ∀a. Show a => a -> T
```

$\alpha \sim \beta \to \gamma$ From the lambda
$\beta \sim T$ From the case
$\forall a.(gd : \mathit{Show}\ a) \Rightarrow$
$\{\,\gamma \sim a\,\}$ From result of f2

Can we solve by substituting $\gamma := a$?

No! No! Noooo! γ comes from an "outer scope"

40



Level numbers

```
f2 = \t. case t of { MkT x -> x }  -- Ill-typed
```

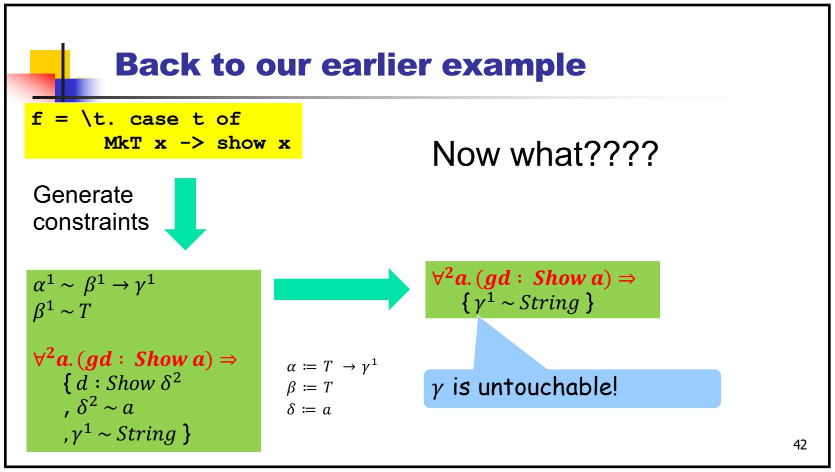
Generate constraints

$\alpha^1 \sim \beta^1 \to \gamma^1$ From the lambda
$\beta^1 \sim T$ From the case
$\forall^2 a.(gd : Show\ a) \Rightarrow \{\gamma^1 \sim a\}$ From result of f2

- Every unification variable has a level number
- Every implication has a level number
- We say γ^1 is untouchable under the $\forall^2$
- The untouchability rule: you cannot solve $\gamma^n \sim ty$ under a $\forall^k$, if n<k

41

41



Back to our earlier example

```
f = \t. case t of
        MkT x -> show x
```

Now what????

Generate constraints

$\alpha^1 \sim \beta^1 \to \gamma^1$
$\beta^1 \sim T$

$\forall^2 a.(gd : Show\ a) \Rightarrow$
$\{ d : Show\ \delta^2$
$, \delta^2 \sim a$
$, \gamma^1 \sim String \}$

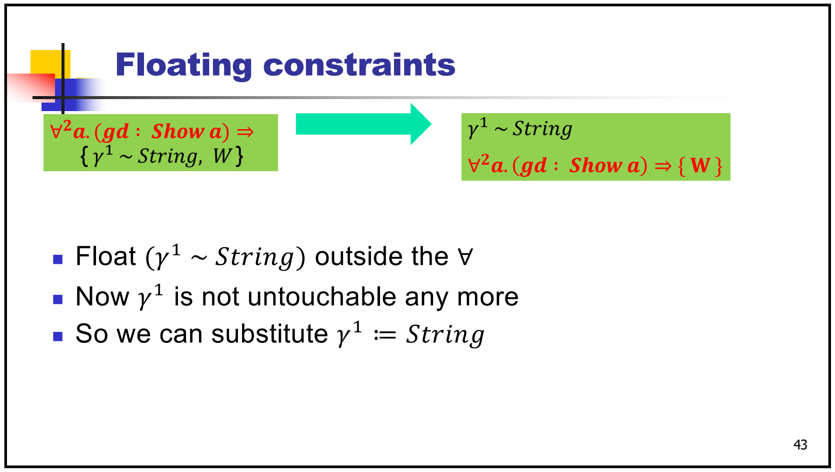
$\alpha := T \to \gamma^1$
$\beta := T$
$\delta := a$

$\forall^2 a.(gd : Show\ a) \Rightarrow \{\gamma^1 \sim String\}$

γ is untouchable!

42

42



Floating constraints

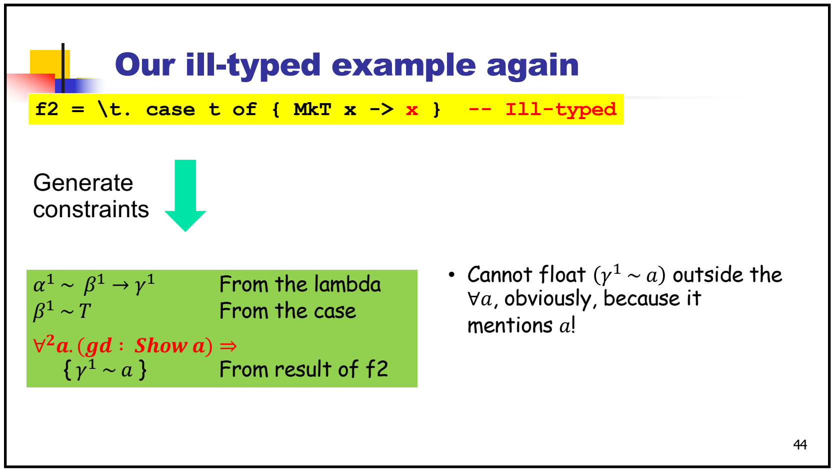
$\forall^2 a.(gd : Show\ a) \Rightarrow \{\gamma^1 \sim String,\ W\}$

$\gamma^1 \sim String$
$\forall^2 a.(gd : Show\ a) \Rightarrow \{ W \}$

- Float $(\gamma^1 \sim String)$ outside the ∀
- Now γ^1 is not untouchable any more
- So we can substitute $\gamma^1 := String$

43

43



Our ill-typed example again

```
f2 = \t. case t of { MkT x -> x }  -- Ill-typed
```

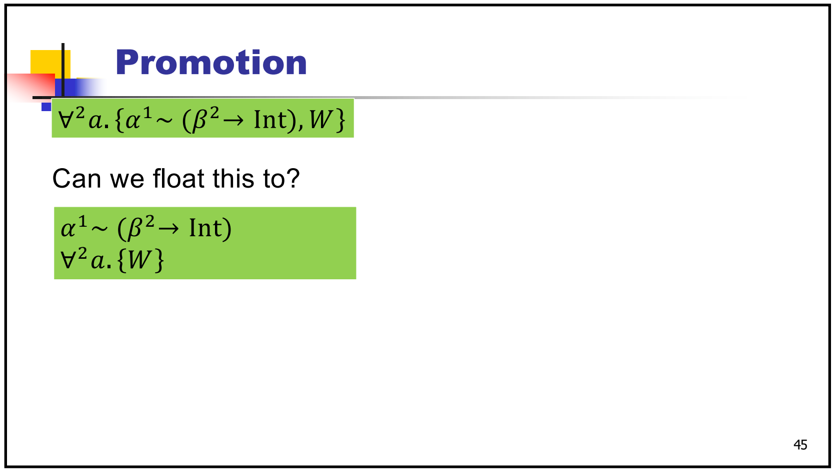
Generate constraints

$\alpha^1 \sim \beta^1 \to \gamma^1$ From the lambda
$\beta^1 \sim T$ From the case
$\forall^2 a.(gd : Show\ a) \Rightarrow \{\gamma^1 \sim a\}$ From result of f2

- Cannot float $(\gamma^1 \sim a)$ outside the $\forall a$, obviously, because it mentions a!

44

44

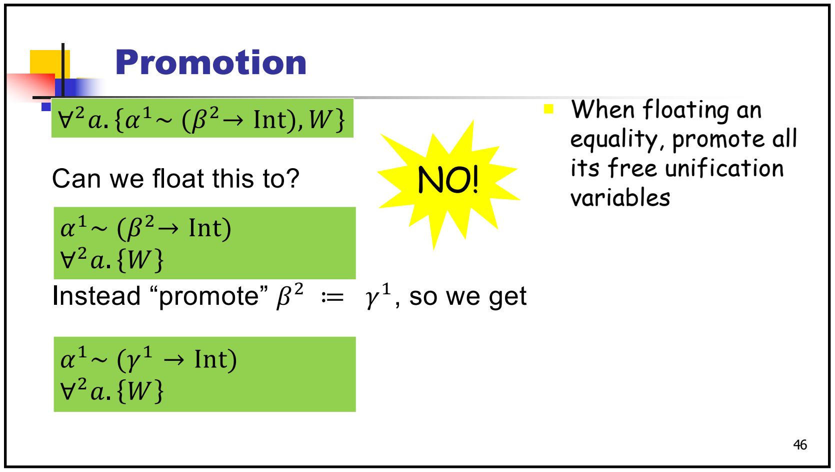


Promotion

$\forall^2 a.\{\alpha^1 \sim (\beta^2 \rightarrow \mathrm{Int}), W\}$

Can we float this to?

$\alpha^1 \sim (\beta^2 \rightarrow \mathrm{Int})$
$\forall^2 a.\{W\}$



Promotion

$\forall^2 a.\{\alpha^1 \sim (\beta^2 \rightarrow \mathrm{Int}), W\}$

Can we float this to?

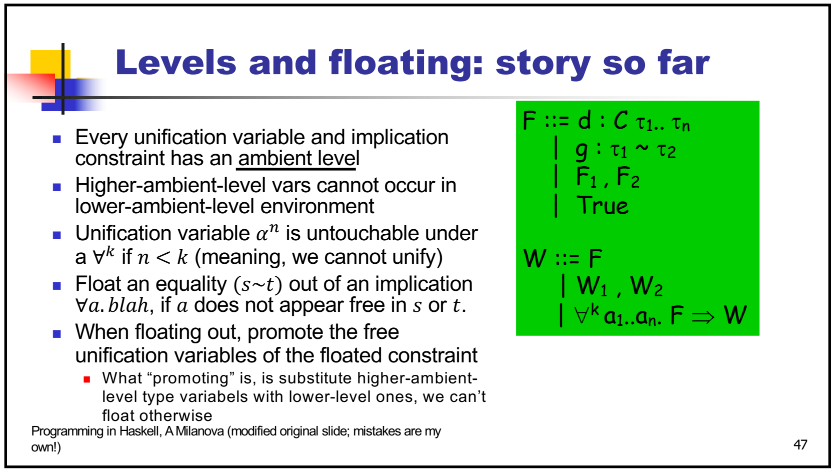
NO!

$\alpha^1 \sim (\beta^2 \rightarrow \mathrm{Int})$
$\forall^2 a.\{W\}$

Instead “promote” $\beta^2 \ := \ \gamma^1$, so we get

$\alpha^1 \sim (\gamma^1 \rightarrow \mathrm{Int})$
$\forall^2 a.\{W\}$

- When floating an equality, promote all its free unification variables



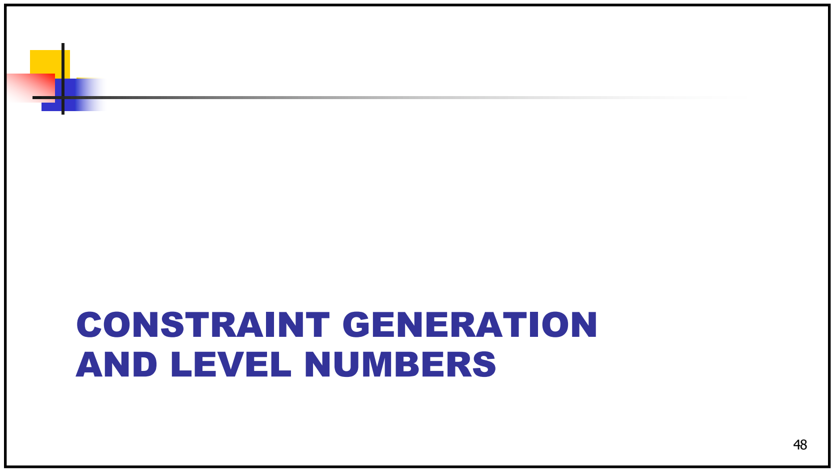
Levels and floating: story so far

- Every unification variable and implication constraint has an ambient level
- Higher-ambient-level vars cannot occur in lower-ambient-level environment
- Unification variable α^n is untouchable under a $\forall^k$ if $n < k$ (meaning, we cannot unify)
- Float an equality $(s \sim t)$ out of an implication $\forall a.\, blah$, if a does not appear free in s or t.
- When floating out, promote the free unification variables of the floated constraint
  - What “promoting” is, is substitute higher-ambient-level type variables with lower-level ones, we can’t float otherwise

F ::= d : *C* $\tau_1 .. \tau_n$
  | g : $\tau_1 \sim \tau_2$
  | F_1 , F_2
  | True

W ::= F
  | W_1 , W_2
  | $\forall^k a_1 .. a_n .\ F \Rightarrow W$

Programming in Haskell, A Milanova (modified original slide; mistakes are my own!)



CONSTRAINT GENERATION AND LEVEL NUMBERS

The “ambient” level

- When generating constraints for a term, the generator has an “ambient” level
- Fresh unification variables are born at this level
- At a pattern match e.g. case x of { K x y -> rhs }
  - Increment the ambient level
  - Generate constraints for the rhs
  - Wrap them in an implication constraint binding the existentials and constraints of K
  - No need for this wrapping if no existentials or constraints
    e.g.   case x of { Just y -> rhs; … }

Type signatures

```
reverse :: ∀a. [a] -> [a]
sort    :: ∀a. Ord a => [a] -> [a]
```

```
f :: ∀a. Ord a => [a] -> [a]
f = \xs -> reverse (sort xs)
```

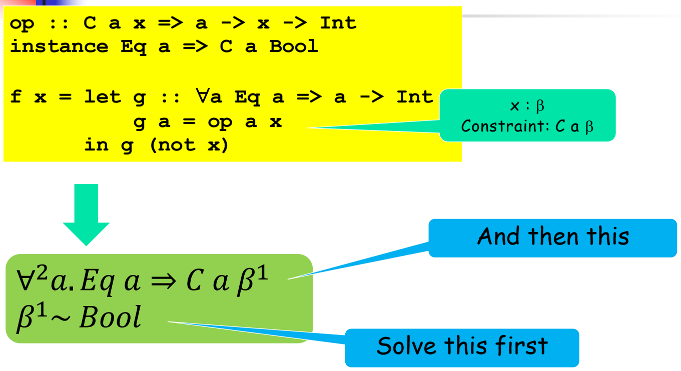
- xs : $[a]$
- Instantiate reverse with α
- Instantiate sort with β

$\forall^1 a. (gd : Ord\ a) \Rightarrow$

$\{\, d : Ord\ \beta^1$	From call of sort
$,\ [\beta^1] \sim [\alpha^1]$	Result of sort
$,\ [\alpha^1] \sim [a]\ \}$	From result of f

- Type signature gives rise to an implication constraint
- Constraints of the signature become “givens” of the implication
- Increment the ambient level before generating constraints for the RHS

Works equally well for nested signatures



```
op :: C a x => a -> x -> Int
instance Eq a => C a Bool

f x = let g :: ∀a Eq a => a -> Int
          g a = op a x
      in g (not x)
```

x : β
Constraint: C a β

$\forall^2 a. Eq\ a \Rightarrow C\ a\ \beta^1$

$\beta^1 \sim Bool$

And then this

Solve this first

CONSTRAINT SOLVING: HITHER AND YON

Story so far

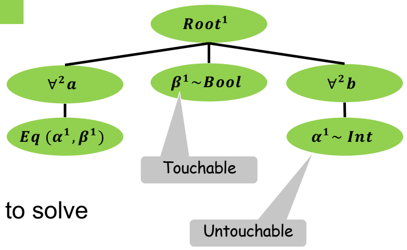
- Perform repeated rewrites on the constraints
- Each rewrite preserves logical meaning
- Each rewrite is recorded by adding an evidence binding, in the elaborated program
- The constraint language is very small
- But solving is quite subtle

F ::= d : $C\ \tau_1 .. \tau_n$
  | g : $\tau_1 \sim \tau_2$
  | F_1 , F_2
  | True

W ::= F
  | W_1 , W_2
  | $\forall^k a_1..a_n.\ F \Rightarrow W$

Solving hither and yon

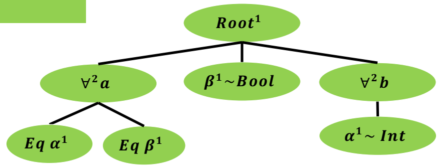
$\forall^2 a.\epsilon \Rightarrow Eq\ (\alpha^1, \beta^1)$
$\beta^1 \sim Bool$
$\forall^2 b.\epsilon \Rightarrow \alpha^1 \sim Int$



A tree of constraints to solve

Solving hither and yon

$\forall^2 a.\epsilon \Rightarrow \{\ Eq\ \alpha^1, Eq\ \beta^1\}$
$\beta^1 \sim Bool$
$\forall^2 b.\epsilon \Rightarrow \alpha^1 \sim Int$

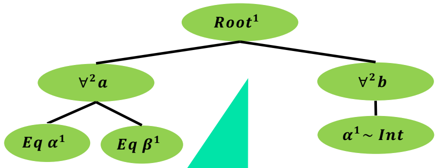


Use
instance (Eq a, Eq b) => Eq (a,b)

Solving hither and yon

$\beta := Bool$

$\forall^2 a.\epsilon \Rightarrow \{\ Eq\ \alpha^1, Eq\ \beta^1\}$
$\forall^2 b.\epsilon \Rightarrow \alpha^1 \sim Int$

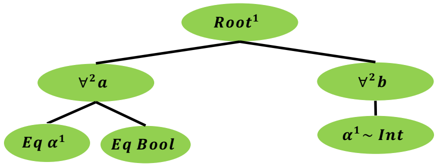


Solve $\beta^1 \sim Bool$

Solving hither and yon

$\beta \coloneqq Bool$

$\forall^2 a.\epsilon \Rightarrow \{\, Eq\ \alpha^1, Eq\ Bool\}$
$\forall^2 b.\epsilon \Rightarrow \alpha^1 \sim Int$

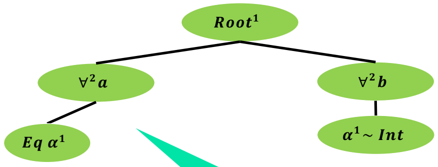


Apply subst to $Eq\ \beta$

Solving hither and yon

$\beta \coloneqq Bool$

$\forall^2 a.\epsilon \Rightarrow \{\, Eq\ \alpha^1\}$
$\forall^2 b.\epsilon \Rightarrow \alpha^1 \sim Int$

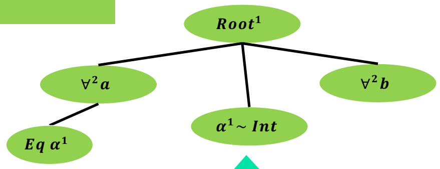


Use instance Eq Bool

Solving hither and yon

$\beta \coloneqq Bool$

$\forall^2 a.\epsilon \Rightarrow \{\, Eq\ \alpha^1\}$
$\alpha^1 \sim Int$
$\forall^2 b.\epsilon \Rightarrow \epsilon$

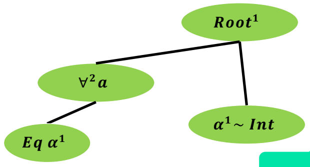


Float $(\alpha^1 \sim Int)$ out of $\forall b$

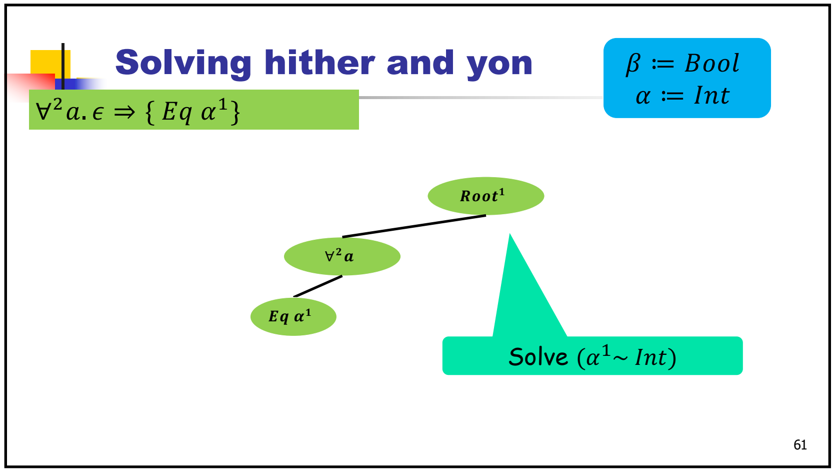
Solving hither and yon

$\beta \coloneqq Bool$

$\forall^2 a.\epsilon \Rightarrow \{\, Eq\ \alpha^1\}$
$\alpha^1 \sim Int$



Discard empty $\forall b$



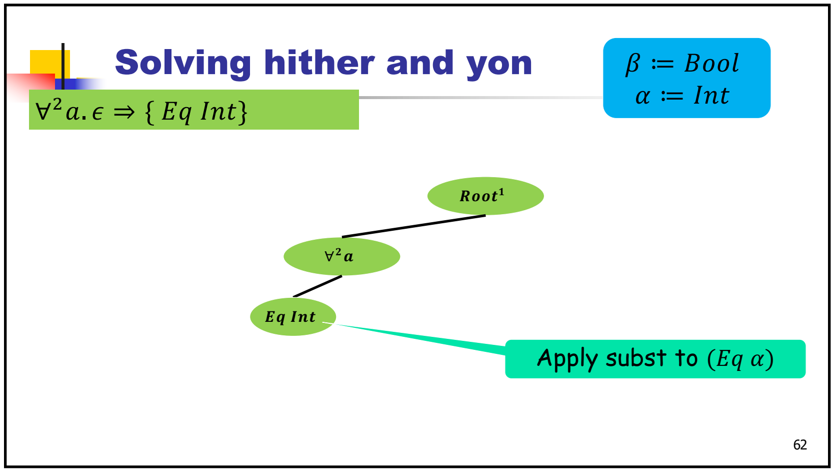
Solving hither and yon

$\beta := Bool$
$\alpha := Int$

$\forall^2 a.\epsilon \Rightarrow \{\, Eq\ \alpha^1\}$

Solve $(\alpha^1 \sim Int)$

61



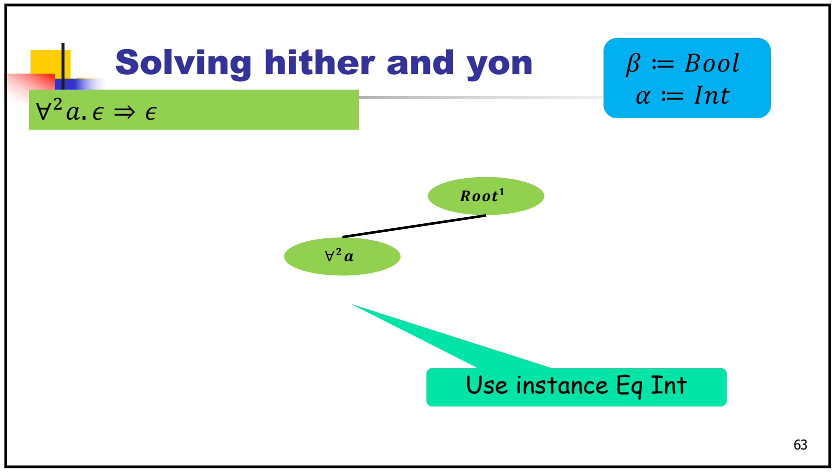
Solving hither and yon

$\beta := Bool$
$\alpha := Int$

$\forall^2 a.\epsilon \Rightarrow \{\, Eq\ Int\}$

Apply subst to $(Eq\ \alpha)$

62



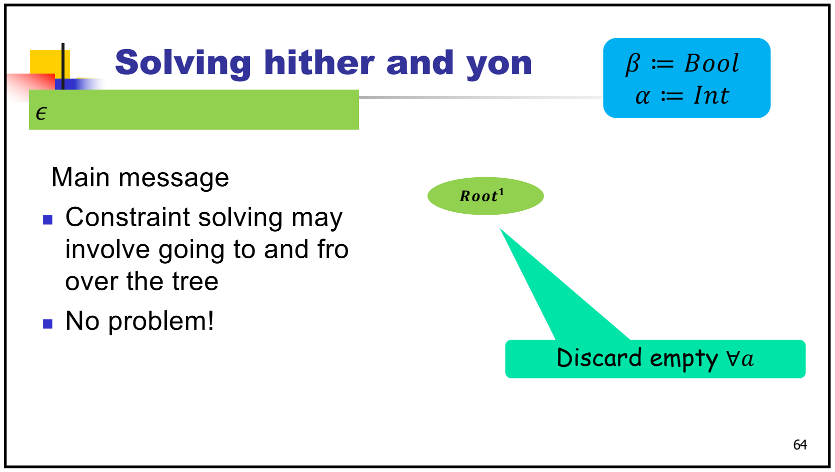
Solving hither and yon

$\beta := Bool$
$\alpha := Int$

$\forall^2 a.\epsilon \Rightarrow \epsilon$

Use instance Eq Int

63



Solving hither and yon

$\beta := Bool$
$\alpha := Int$

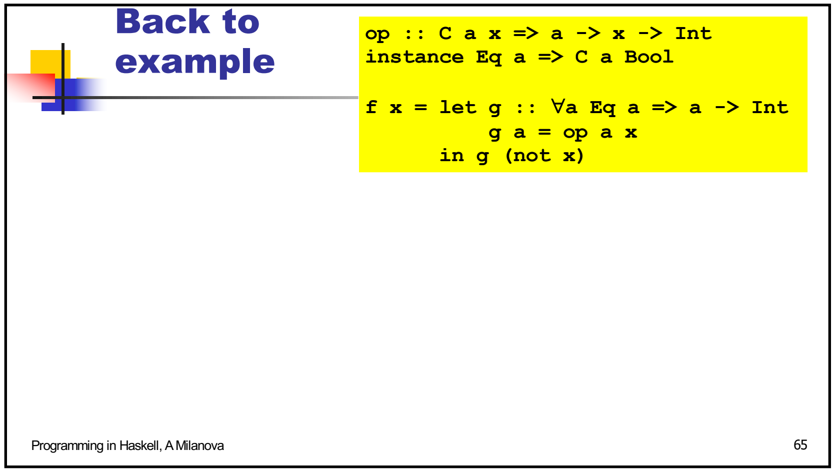
ϵ

Main message

- Constraint solving may involve going to and fro over the tree
- No problem!

Discard empty $\forall a$

64

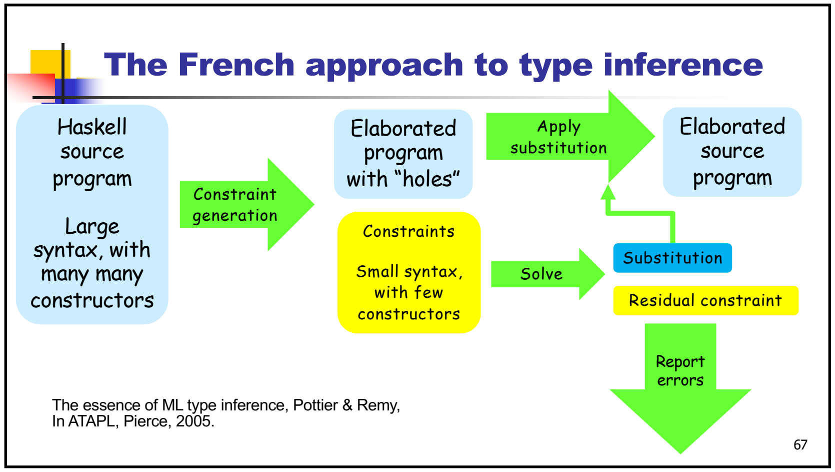


Back to example

```
op :: C a x => a -> x -> Int
instance Eq a => C a Bool

f x = let g :: ∀a Eq a => a -> Int
          g a = op a x
      in g (not x)
```

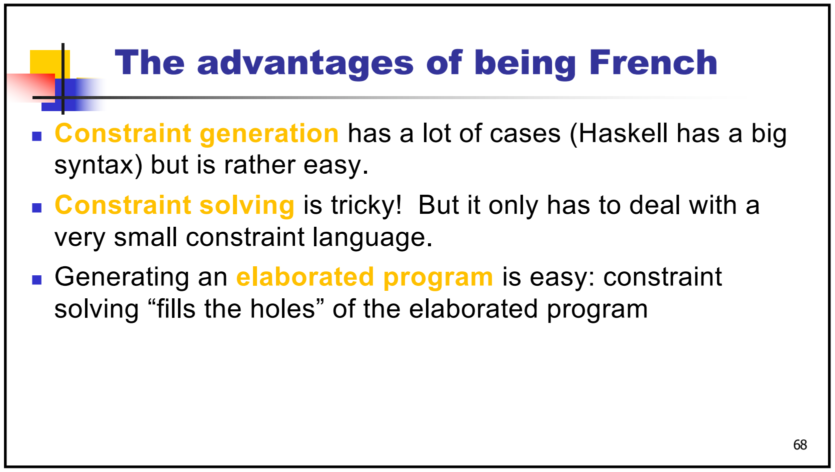
BACK TO THE BIG PICTURE



The French approach to type inference

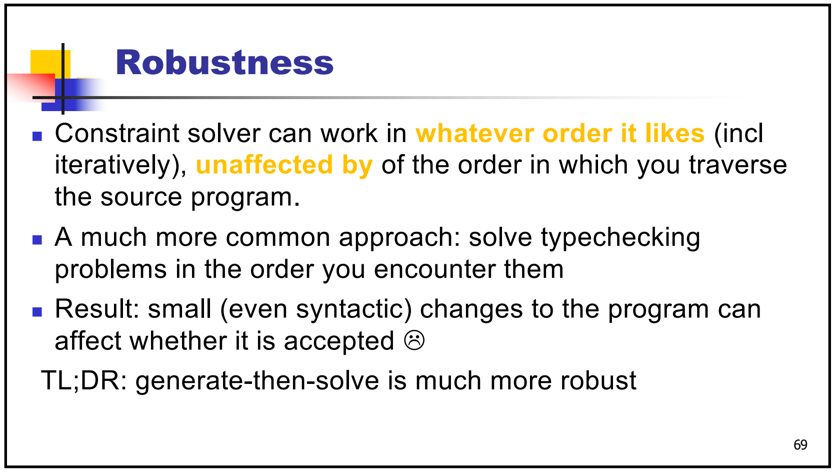
Haskell source program
Large syntax, with many many constructors
Constraint generation
Elaborated program with "holes"
Apply substitution
Elaborated source program
Constraints
Small syntax, with few constructors
Solve
Substitution
Residual constraint
Report errors

The essence of ML type inference, Pottier & Remy,
In ATAPL, Pierce, 2005.



The advantages of being French

- Constraint generation has a lot of cases (Haskell has a big syntax) but is rather easy.
- Constraint solving is tricky! But it only has to deal with a very small constraint language.
- Generating an elaborated program is easy: constraint solving "fills the holes" of the elaborated program

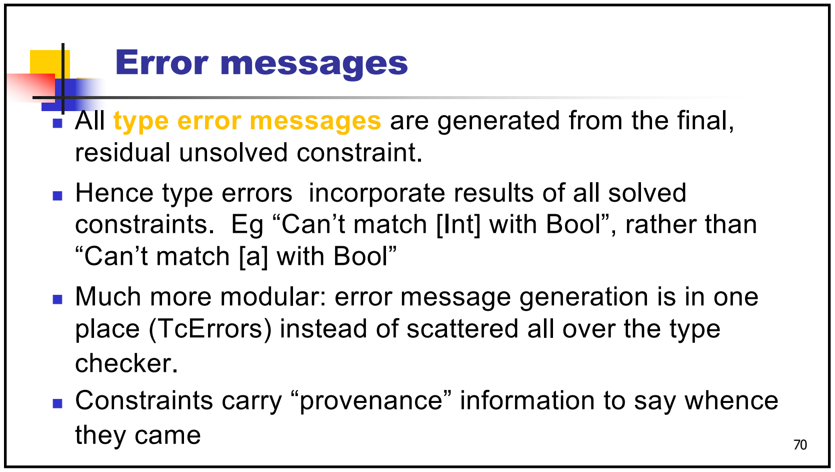


Robustness

- Constraint solver can work in **whatever order it likes** (incl iteratively), **unaffected by** of the order in which you traverse the source program.
- A much more common approach: solve typechecking problems in the order you encounter them
- Result: small (even syntactic) changes to the program can affect whether it is accepted ☹

TL;DR: generate-then-solve is much more robust

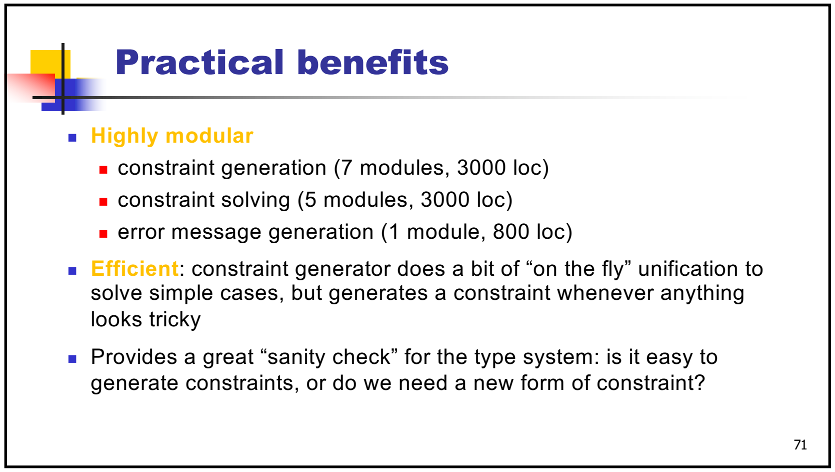
69



Error messages

- All **type error messages** are generated from the final, residual unsolved constraint.
- Hence type errors incorporate results of all solved constraints. Eg “Can’t match [Int] with Bool”, rather than “Can’t match [a] with Bool”
- Much more modular: error message generation is in one place (TcErrors) instead of scattered all over the type checker.
- Constraints carry “provenance” information to say whence they came

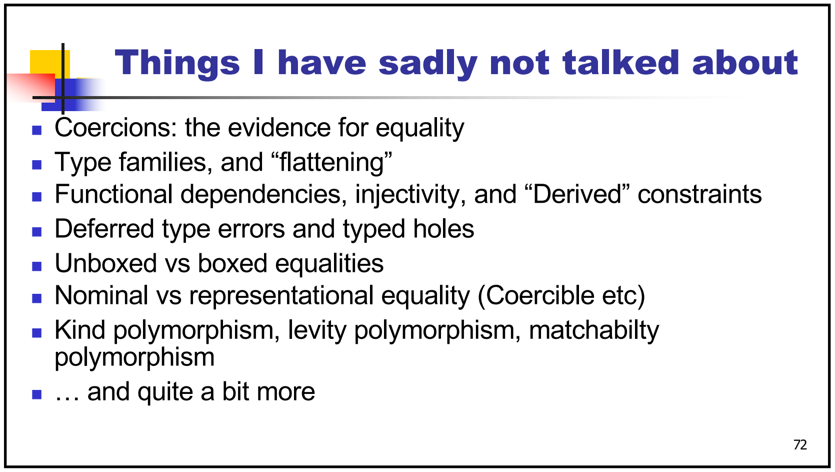
70



Practical benefits

- **Highly modular**
  - constraint generation (7 modules, 3000 loc)
  - constraint solving (5 modules, 3000 loc)
  - error message generation (1 module, 800 loc)
- **Efficient**: constraint generator does a bit of “on the fly” unification to solve simple cases, but generates a constraint whenever anything looks tricky
- Provides a great “sanity check” for the type system: is it easy to generate constraints, or do we need a new form of constraint?

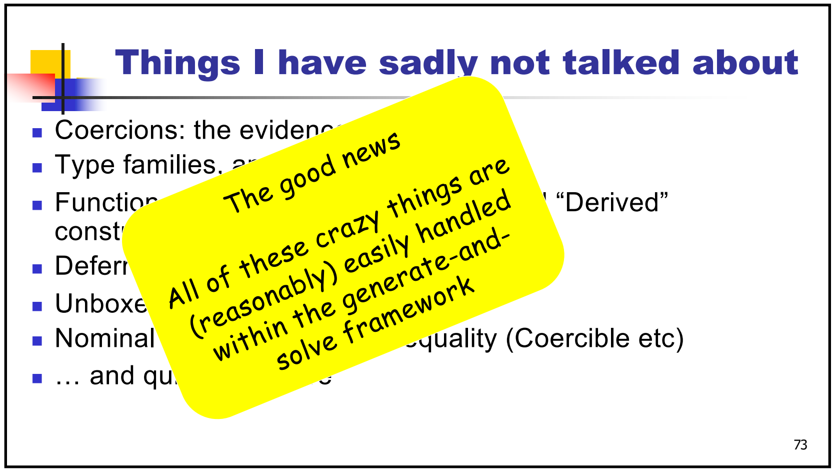
71



Things I have sadly not talked about

- Coercions: the evidence for equality
- Type families, and “flattening”
- Functional dependencies, injectivity, and “Derived” constraints
- Deferred type errors and typed holes
- Unboxed vs boxed equalities
- Nominal vs representational equality (Coercible etc)
- Kind polymorphism, levity polymorphism, matchabilty polymorphism
- … and quite a bit more

72



Things I have sadly not talked about

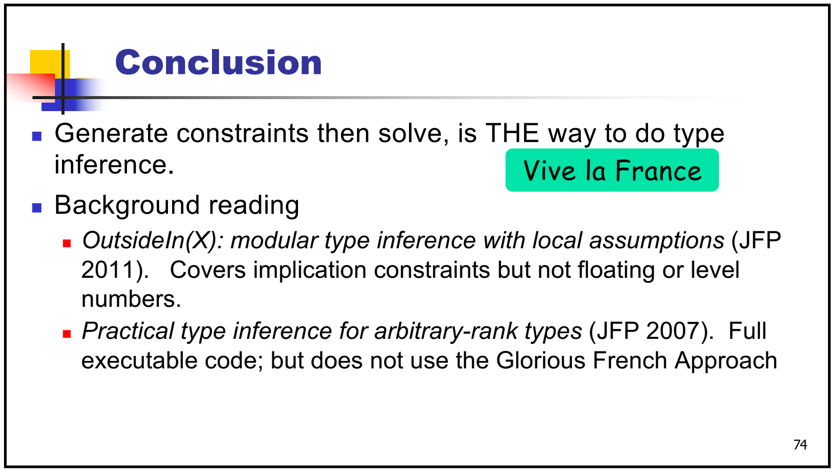
- Coercions: the evidenc
- Type families, a
- Functio ' “Derived” const
- Deferr
- Unboxe
- Nominal equality (Coercible etc)
- … and qu e

The good news

All of these crazy things are (reasonably) easily handled within the generate-and-solve framework

73

73



Conclusion

- Generate constraints then solve, is THE way to do type inference.

Vive la France

- Background reading
  - *OutsideIn(X): modular type inference with local assumptions* (JFP 2011). Covers implication constraints but not floating or level numbers.
  - *Practical type inference for arbitrary-rank types* (JFP 2007). Full executable code; but does not use the Glorious French Approach

74

74