Type Inference as Constraint Solving

Simon Peyton Jones
Microsoft Research
August 2013

Classic Damas-Milner

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

foo :: [Bool]
foo = reverse xs
```

- **Instantiate** 'reverse' with a unification variable α, standing for an as-yet-unknown type. So this occurrence of reverse has type [α] -> [α].
- **Constrain** expected arg type [α] equal to actual arg type [Bool], thus α ~ Bool.
- Solve by **unification**: α := Bool

Modify for type classes

```
(>) :: ∀a. Ord a => a -> a -> Bool
instance Ord a => Ord [a] where ...

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

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

Another view

```
(>) ::  ∀a. Ord a => a -> a -> Bool
instance Ord a => Ord [a] where ...

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

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

$\forall$a. Ord a =>
Ord $\alpha \wedge \alpha \sim$ [a]

Solve this

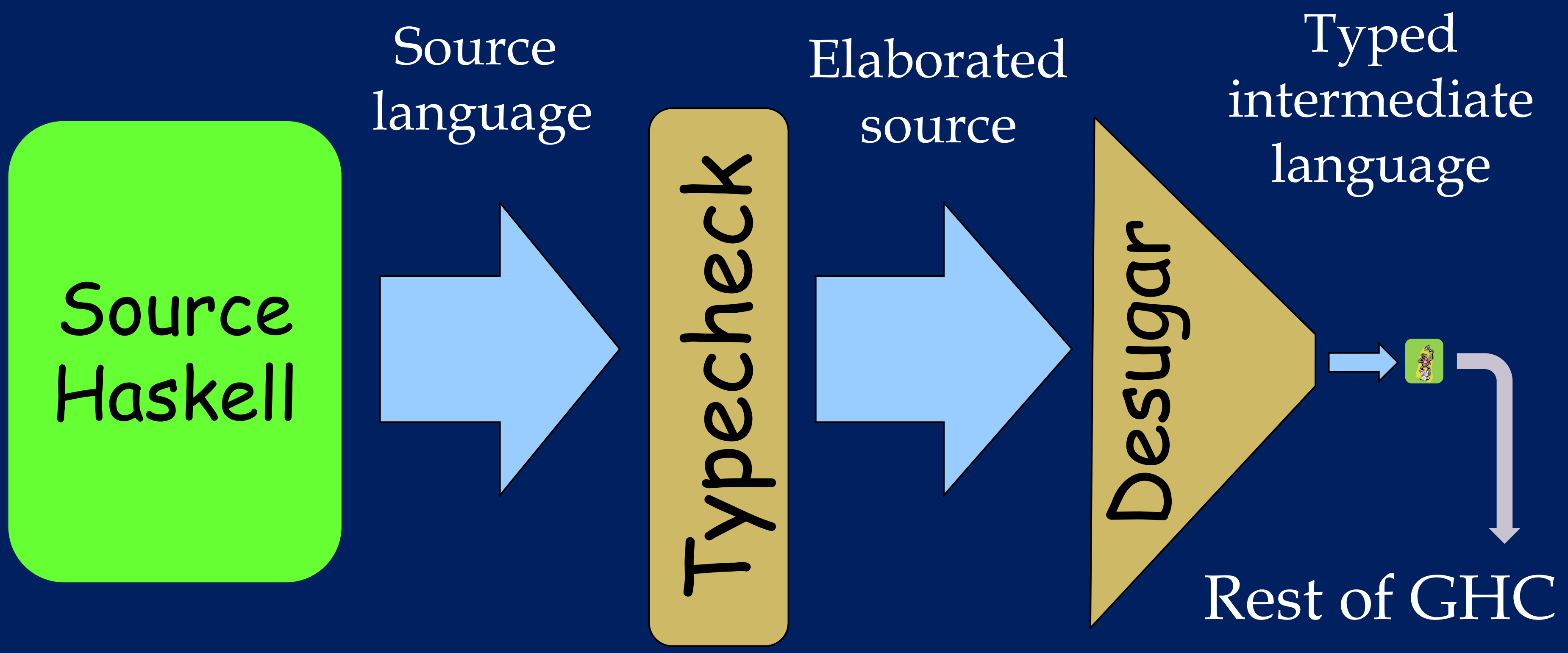
Additional complication: evidence

```
instance Ord a => Ord [a] where ...

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

Elaborate

```
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



Elaboration

```
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)
```

Elaboration inserts

- Type and dictionary applications
- Type and dictionary abstractions
- Dictionary bindings

Elaboration

```
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] d 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)

Another view

```
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)
```

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

∀a. d::Ord a =>
  d2::Ord α ∧ α ~ [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)
```

Term to typecheck

Expected type

Elaborated term

DEFERRED SOLVING

Deferring solving

- Old school
  - Find a unfication problem
  - Solve it
  - If fails, report error
  - Otherwise, proceed
- This will not work any more

```
g :: F a -> a -> Int
type instance F Bool = Bool

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

Deferring solving

```
g :: F a -> a -> Int
type instance F Bool = Bool

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

- x::β
- Instantiate *g* at α

g x

g x 'v'

not x

F α ~ β ∧
α ~ β ∧
β ~ Bool

Order of encounter

We have to solve this first

Deferring solving

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

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

x : β
Constraint: C a β

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

Deferring solving

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

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

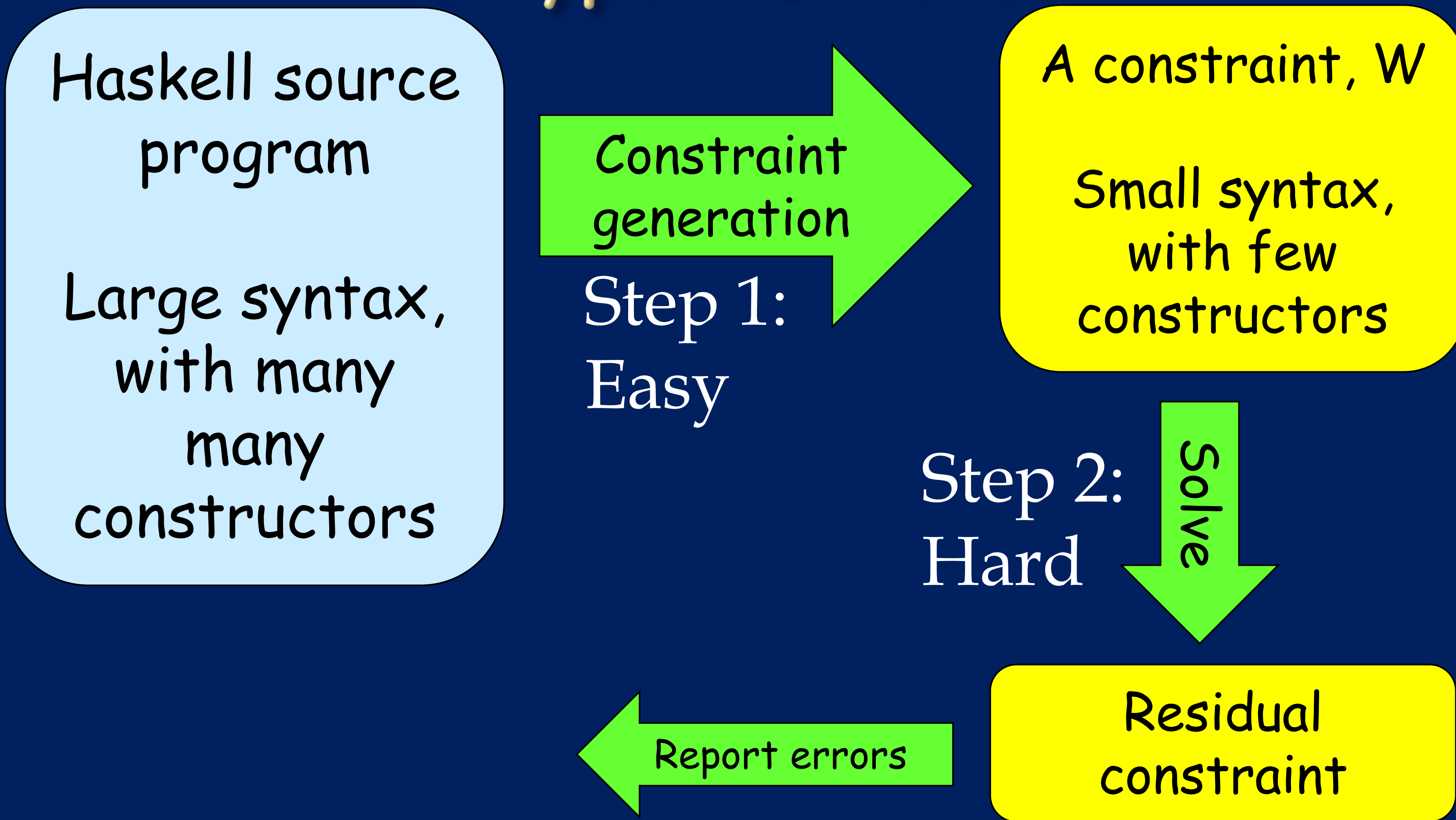
$x : \beta$
Constraint: $C\ a\ \beta$

$(\forall a.\ Eq\ a \Rightarrow C\ a\ \beta)$
$\wedge$
$\beta \sim Bool$

Solve this first

And then this

The French approach to type inference



The French approach to type inference

Haskell source program

constructors

Constraint generation

A constraint, W

Small syntax,

$$F ::= C\ \tau_1 .. \tau_n \mid \tau_1 \sim \tau_2 \mid F_1 \wedge F_2$$

$$W ::= F \mid W_1 \wedge W_2 \mid \forall a_1..a_n.\ F \Rightarrow W$$

lve

Residual constraint

Report errors

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

$F ::= C\ \tau_1 .. \tau_n$	Class constraint
$\mid\ \tau_1 \sim \tau_2$	Equality constraint
$\mid\ F_1 \wedge F_2$	Conjunction
$\mid$ True	
$W ::= F$	Flat constraint
$\mid\ W_1 \wedge W_2$	Conjunction
$\mid\ \forall a_1 .. a_n .\ F \Rightarrow W$	Implication

The language of constraints

$F ::= d :: C\ \tau_1 .. \tau_n$	Class constraint
$\mid\ c :: \tau_1 \sim \tau_2$	Equality constraint
$\mid\ F_1 \wedge F_2$	Conjunction
$\mid$ True	
$W ::= F$	Flat constraint
$\mid\ W_1 \wedge W_2$	Conjunction
$\mid\ \forall a_1 .. a_n .\ F \Rightarrow W$	Implication

Equality constraints generate evidence too!

```
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 :: ∀a. (a~Bool) -> Bool -> T a
```

```
f :: T a -> Maybe a
f (a:*) (x:T a)
   = case x of
         T1 (c:a~Bool) (z:Bool)
            -> let
                     □
               in Just z ▷ c2
         T2 -> False
```

Elaborated program

plus constraint to solve

(c :: a~Bool) => c2 :: Maybe Bool ~ Maybe a

Equality constraints generate evidence too!

(c :: a~Bool) => c2 :: Maybe Bool ~ Maybe a

c2 = Maybe c3

(c :: a~Bool) => c3 :: Bool ~ a

c3 = sym c4

(c :: a~Bool) => c4 :: a ~ Bool

c4 = c

(c :: a~Bool) => True

Plug the evidence back into the term

```
f :: T a -> Maybe a
f (a:*) (x:T a)
  = case x of
      T1 (c:a~Bool) (z:Bool)
        -> let  c4:a~Bool                  = c
                c3:Bool~a                  = sym c4
                c2:Maybe Bool ~ Maybe a = Maybe c3
           in Just z ▷ c2
      T2   -> False
```

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

```
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?

Care with GADTs

```
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?

- f :: ∀b. T b -> b -> b
- f :: ∀b. T b -> Bool -> 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
```

$f :: T\,\alpha \to \beta \to \gamma$

$(\alpha \sim Bool \Rightarrow \gamma \sim Bool) \wedge (\beta \sim \gamma)$

From T1 branch

From T2 branch

In the language of constraints

$f :: T\ \alpha \to \beta \to \gamma$

$(\alpha \sim Bool \Rightarrow \gamma \sim Bool) \wedge (\beta \sim \gamma)$

Two solutions, neither principal

- $\gamma := Bool$
- $\gamma := a$

GHC's conclusion
No principal solution, so reject the program

In the language of constraints

$(\alpha \sim Bool \Rightarrow \gamma \sim Bool) \wedge (\beta \sim \gamma)$

- Treat γ as **untouchable** under the (α~Bool) equality; i.e. (γ~Bool) is not solvable
- Equality information propagates outside-in
- So $(\alpha \sim Bool \Rightarrow \gamma \sim Bool) \wedge (\alpha \sim \gamma)$ **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

Vive la France

- 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**

How it looks

```
bash$ ghci -fdefer-type-errors
ghci> let foo = (True, ‘a’ && False)
Warning: can’t match Char with Bool
gici> fst foo
True
ghci> snd foo
Error: can’t match Char with Bool
```

- 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::[Char], _b:_c ]
```

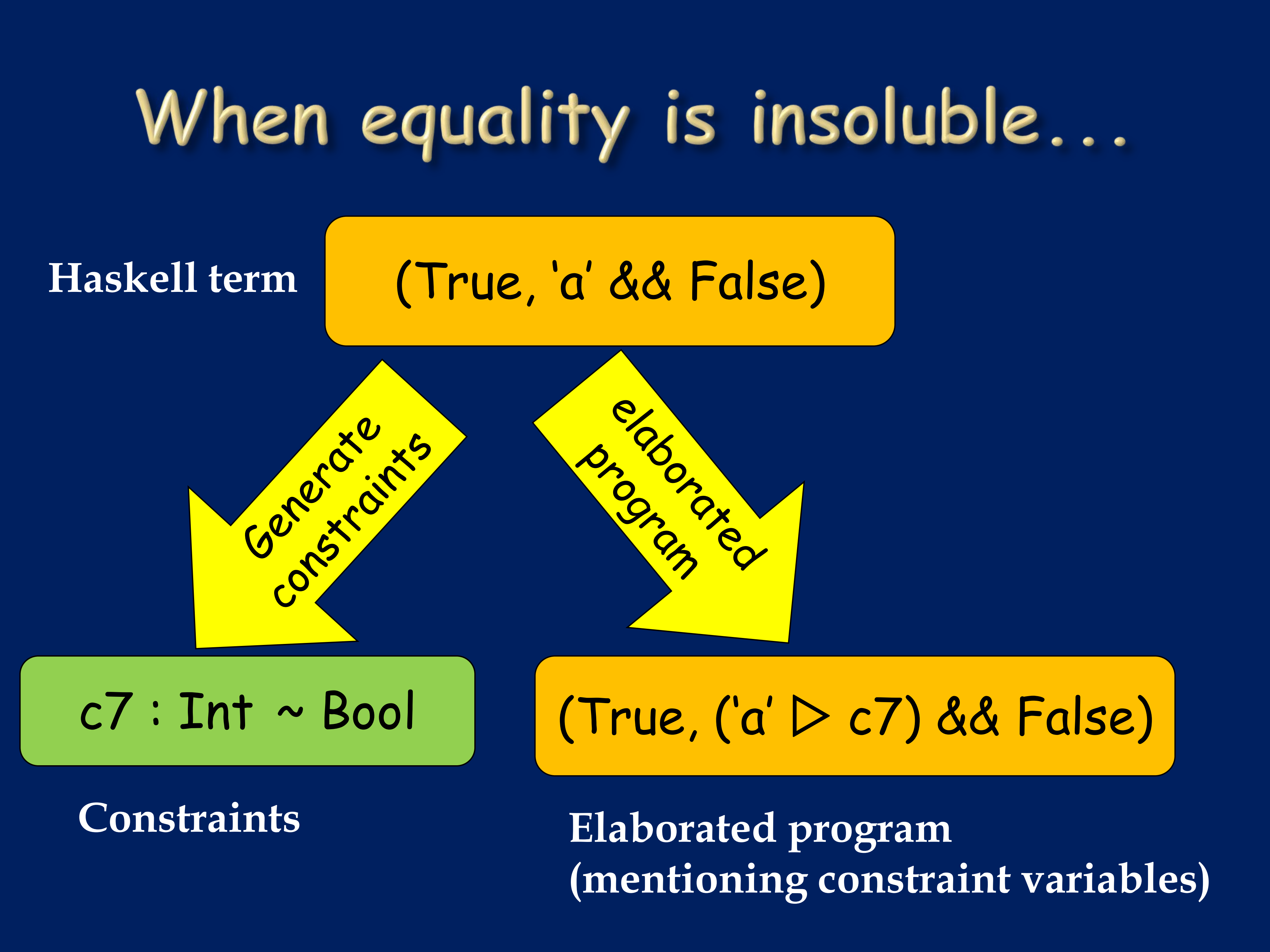
```
Holes:2:12:
    Found hole `_a' with type: [Char]
    In the expression: _a
    In the expression: [_a, x :: [Char], _b : _c]
    In an equation for `f': f x = [_a, x :: [Char], _b : _c]

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

- -XTypeHoles 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.

Just a hack?

- 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
(True, 'a' && False)
Generate constraints
elaborated program
c7 : Int ~ Bool
(True, ('a' ▷ c7) && False)
Constraints
Elaborated program
(mentioning constraint variables)

Step 2: solve constraints

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

Solve

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

```
c7 : Int ~ Bool
```

```
(True, ('a' ▷ c7) && False)
```

Constraints

Elaborated program (mentioning constraint variables)

Step 2:

- Use lazily evalu
- Cast evaluates
- Error triggered whe d only when) 'a' must have type Bool

Uh oh! What became of coercion erasure?

Solve

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

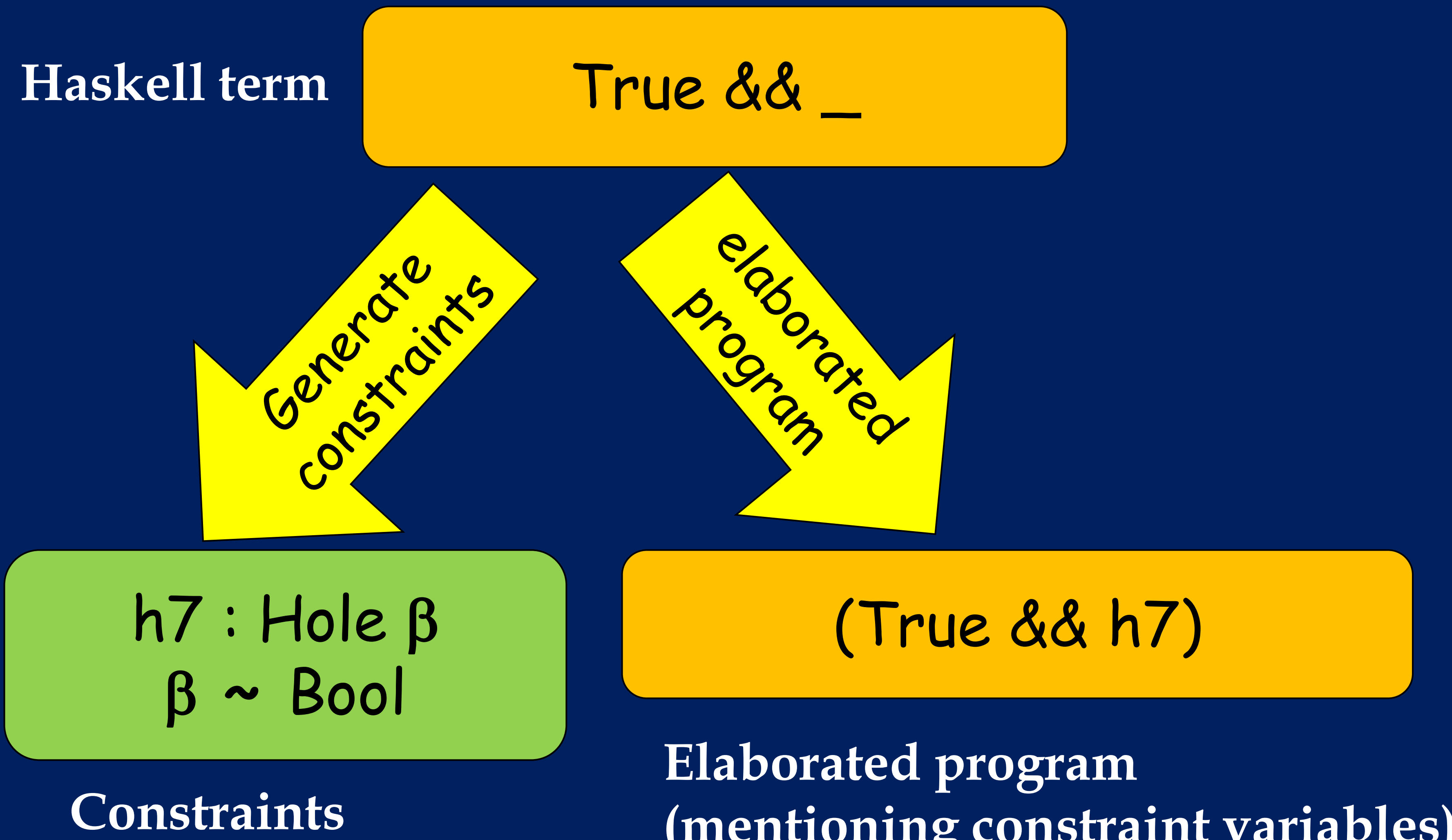
c7 : Int ~ Bool

(True, ('a' ▷ c7) && False)

Constraints

Elaborated program (mentioning constraint variables)

Hole constraints
(a new form of constraint)



Hole constraints...

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

Solve

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

```
h7 : Hole Bool
```

```
(True && h7)
```

Constraints

Elaborated program (mentioning constraint variables)

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

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
```

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
```

Or this?

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

A possible path [Pottier et al]

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~[b]) => b~Int) => 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?

Making it efficient

```
let c7: Bool~Bool = refl Bool
in (x ▷ c7) && False)
```

- Remember deferred type errors: cast must evaluate its coercion argument.
- What became of erasure?

Take a clue from unboxed values

```
data Int = I#  Int#

plusInt :: Int -> Int -> Int
plusInt x y
  = case x of I# a ->
     case y of I# b ->
      I#  (a +# b)
```

Library code

```
x `plusInt` x

= case x of I# a ->
   case x of I# b ->
    I#  (a +# b)

= case x of I# a ->
    I#  (a +# a)
```

Inline + optimise

- Expose evaluation to optimiser

Take a clue from unboxed values

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

(▷) :: (a~b) -> a -> b
x ▷  c = case c of
            Eq# d -> x ▷# d

refl :: t~t
refl = /\t. Eq# (refl# t)
```

Library code

```
let c7 = refl Bool
in (x ▷ c7) && False

 ...inline refl, ▷
=  (x ▷#  (refl# Bool))
     && False
```

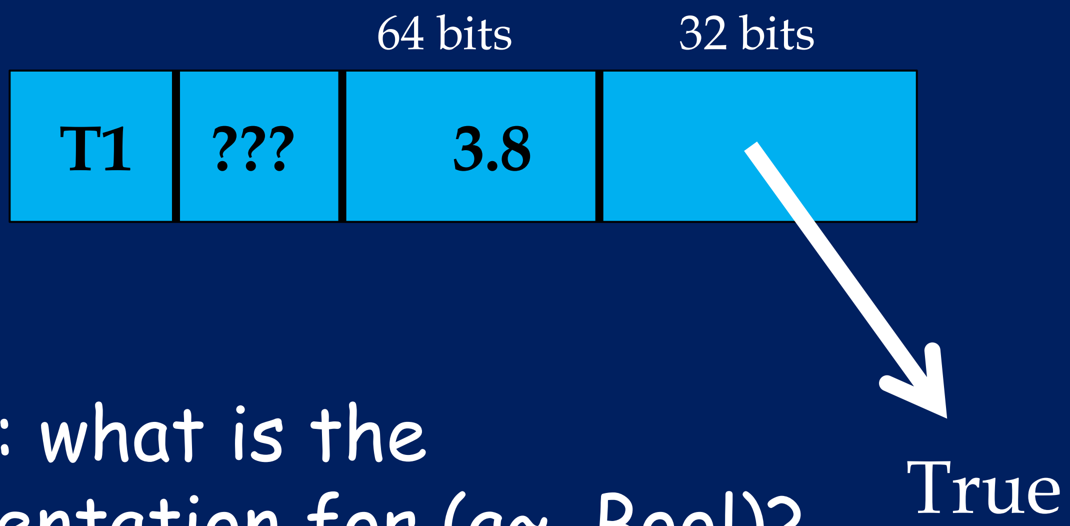
Inline + optimise

- So ($\sim_\#$) is the primitive type constructor
- ($\triangleright_\#$) is the primitive language construct
- And ($\triangleright_\#$) is erasable

Implementing $\sim_{\#}$

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

A T1 value allocated in the heap looks like this

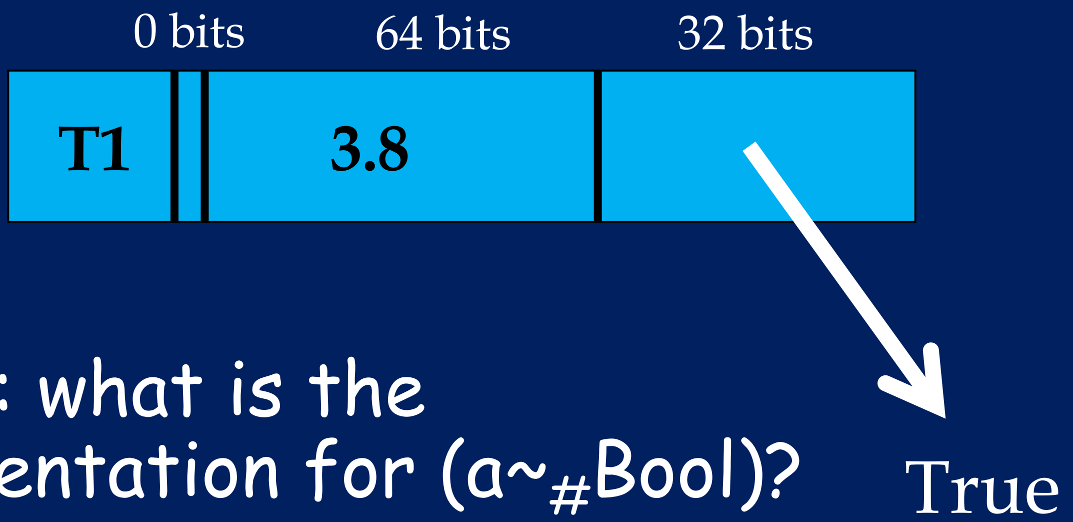


Question: what is the representation for $(a\sim_{\#}Bool)$?

Implementing $\sim_{\#}$

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

A T1 value allocated in the heap looks like this



Question: what is the representation for $(a\sim_{\#}Bool)$?

Answer: a 0-bit value

Boxed and primitive equality

data a ~ b = Eq# (a $\sim_{\#}$ 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

Background reading

- *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