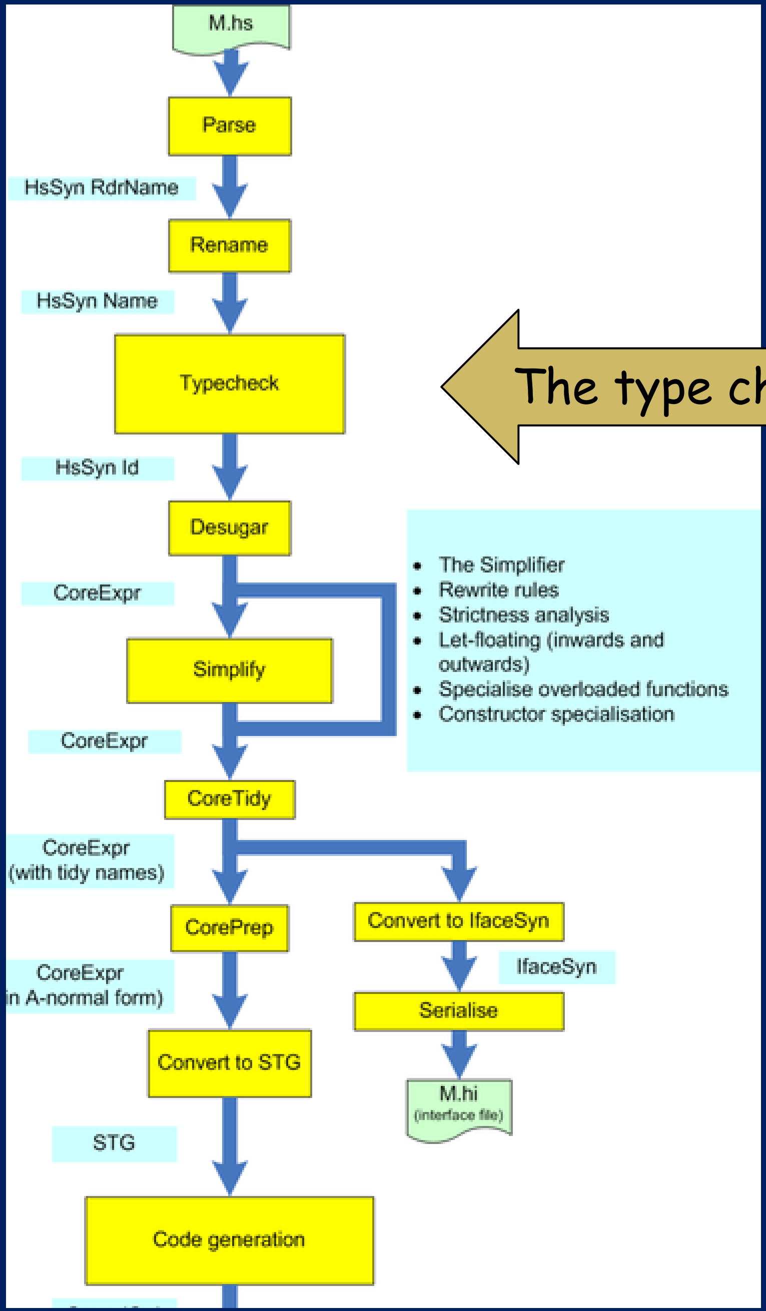
A HISTORY OF THE WORLD IN 100 OBJECTS

SECRETS OF THE GHC TYPECHECKER
(30 YEARS OLD, 50,000 LINES OF CODE)
IN 100 TYPE DECLARATIONS

Simon Peyton Jones
Epic Games
GHC Contributors workshop, June 2023

The big picture



The type checker

- Input: HsSyn GhcRn
- Output: HsSyn GhcTc

A very big data type

Elaboration

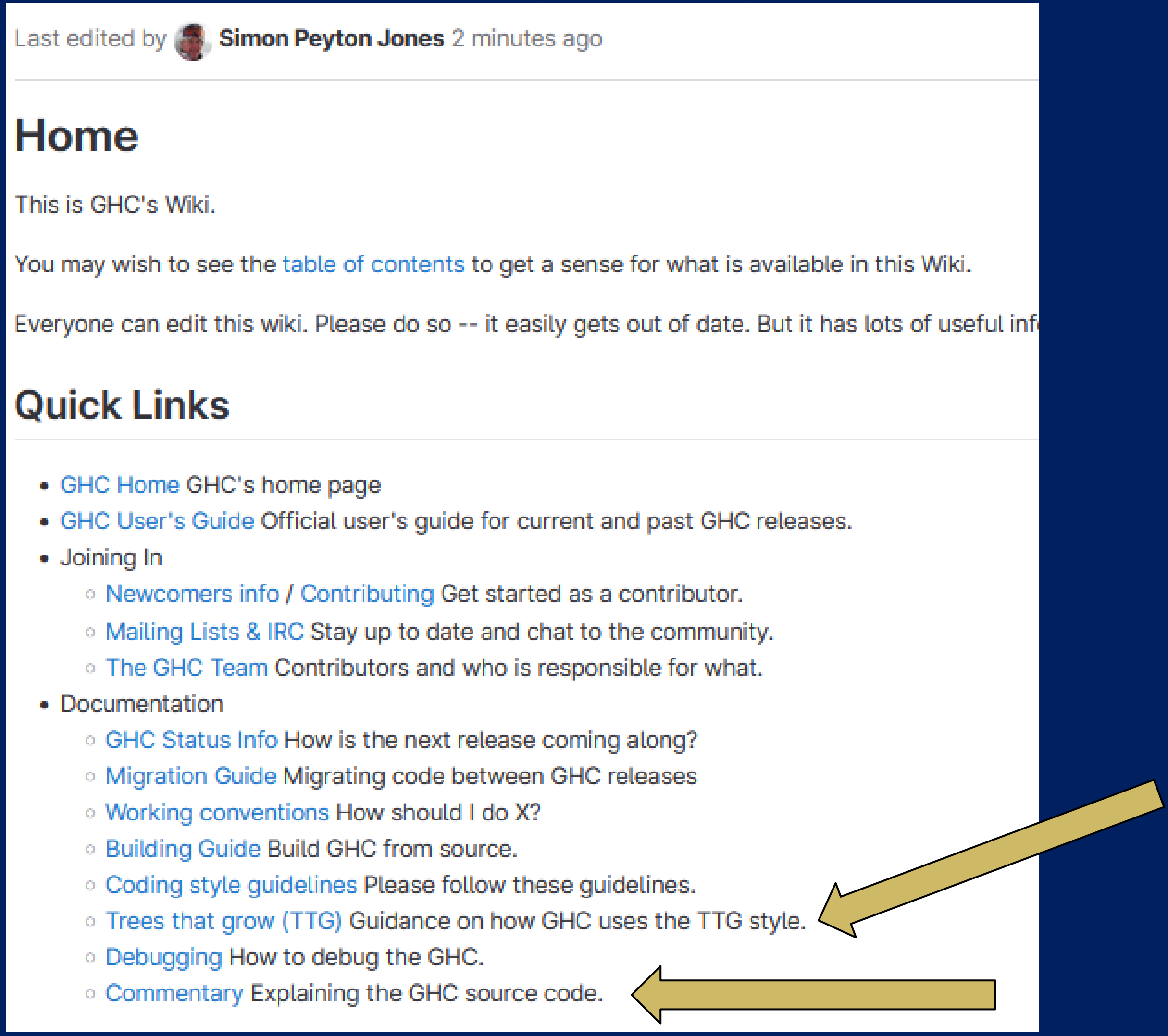
The HsSyn syntax tree, and Trees That Grow

HsSyn

- Language.Haskell.Syntax.* aka "HsSyn"
  - **GHC-independent** definition of syntax tree
  - Ultimately intended to be a separate package.
  - Intended to be useful for other tools (eg Template Haskell, haskell-src-exts).
- GHC.Hs.*
  - **GHC-specific instantiation** of HsSyn.
  - Uses Trees that Grow ideas a lot.
  - Wiki page: https://gitlab.haskell.org/ghc/ghc/-/wikis/implementing-trees-that-grow.

Side note: wiki:

https://gitlab.haskell.org/ghc/ghc/-/wikis


Last edited by Simon Peyton Jones 2 minutes ago

Home

This is GHC's Wiki.

You may wish to see the table of contents to get a sense for what is available in this Wiki.

Everyone can edit this wiki. Please do so -- it easily gets out of date. But it has lots of useful inf

Quick Links

- GHC Home GHC's home page
- GHC User's Guide Official user's guide for current and past GHC releases.
- Joining In
  - Newcomers info / Contributing Get started as a contributor.
  - Mailing Lists & IRC Stay up to date and chat to the community.
  - The GHC Team Contributors and who is responsible for what.
- Documentation
  - GHC Status Info How is the next release coming along?
  - Migration Guide Migrating code between GHC releases
  - Working conventions How should I do X?
  - Building Guide Build GHC from source.
  - Coding style guidelines Please follow these guidelines.
  - Trees that grow (TTG) Guidance on how GHC uses the TTG style.
  - Debugging How to debug the GHC.
  - Commentary Explaining the GHC source code.

- Key resource
- Tons of useful information
- Easily gets out of date
- Everyone can edit:
  please, please do so.
- Do not accept bogus or out of date info! Ask, redraft, fix.

Language.Haskell.Syntax

```
module Language.Haskell.Syntax.Expr where

data HsExpr p
  = HsVar      (XVar p)  (LIdP p)
  | HsLit      (XLitE p) (HsLit p)
  | OpApp      (XOpApp p)  (LHsExpr p)  (LHsExpr p)  (LHsExpr p)
  ...dozens of others...
  | XExpr     !(XXExpr p)
```

```
module Language.Haskell.Syntax.Extension where

type family XVar p
type family XLitE p
type family XLam p
type family XXExpr p
type family XRec p a

type family IdP p
type LIdP p = XRec p (IdP p)
```

- XVar, XLitE, XOpApp live are the constructor extensions
- XXExpr is the data type extension
- Instances for XVar, XXExpr etc are in the GHC-specific tree: GHC.Hs.*

XOpApp: per-pass extensions of OpApp

```
module Language.Haskell.Syntax.Expr where

data HsExpr p
  = HsVar       (XVar p)  (LIdP p)
  | HsLit       (XLitE p) (HsLit p)
  | OpApp       (XOpApp p)  (LHsExpr p)  (LHsExpr p)  (LHsExpr p)
  ...dozens of others...
  | XExpr      !(XXExpr p)
```

```
module GHC.Hs.Extension where

data Pass = Parsed | Renamed | Typechecked

data GhcPass (c :: Pass) where
  GhcPs :: GhcPass 'Parsed
  GhcRn :: GhcPass 'Renamed
  GhcTc :: GhcPass 'Typechecked

type GhcPs   = GhcPass `Parsed
type GhcRn   = GhcPass `Renamed
type GhcTc   = GhcPass 'Typechecked

type instance XOpApp GhcPs = EpAnn [AddEpAnn]
type instance XOpApp GhcRn = Fixity
type instance XOpApp GhcTc = DataConCantHappen
```

- HsExpr (GhcPass p): output of pass p of GHC
- XOpApp (GhcPass p): Extension field of HsOpApp is populated with different types, depending on which pass

XRec: adding source locations

```
module Language.Haskell.Syntax.Expr where

data HsExpr p
  = HsVar      (XVar p)  (LIdP p)
  | HsLit      (XLitE p) (HsLit p)
  | OpApp      (XOpApp p)  (LHsExpr p)  (LHsExpr p)  (LHsExpr p)
  ...dozens of others...
  | XExpr     !(XXExpr p)

type LHsExpr p = XRec p (HsExpr p)
```

```
module GHC.Hs.Extension where

data Pass = Parsed | Renamed | Typechecked

data GhcPass (c :: Pass) where
  GhcPs :: GhcPass 'Parsed
  GhcRn :: GhcPass 'Renamed
  GhcTc :: GhcPass 'Typechecked

type instance XRec (GhcPass p) a
  = GenLocated (Anno a) a

data GenLocated l e = L l e
type family Anno a
```

- GenLocated (Anno e) e: wraps e in decoration (Anno e)
- The 'Anno e' is a SrcSpan...
- ...maybe plus some extra stuff
- Most HsExprs are wrapped in LHsExpr, which gives a SrcSpan.

IdP: varying the payload

```
module Language.Haskell.Syntax.Expr where

data HsExpr p
  = HsVar      (XVar p)  (LIdP p)
  | HsLit      (XLitE p) (HsLit p)
  | OpApp      (XOpApp p)  (LHsExpr p)  (LHsExpr p)  (LHsExpr p
  ...dozens of others...
  | XExpr     !(XXExpr p)

module Language.Haskell.Syntax.Extension where

type LIdP p = XRec p (IdP p)
type family IdP p
```

```
module GHC.Types.Var where
  type Id = Var
  data Var = ...
    = TyVar { ... }
    | TcTyVar { ... }
    | Id {
        varName     :: !Name,
        realUnique  :: !Int,
        varType     :: Type,
        varMult     :: Mult,
        idScope     :: IdScope,
        id_details  :: IdDetails
        id_info     :: IdInfo }
```

```
module GHC.Hs.Extension where

type instance IdP (GhcPass p) = IdGhcP p

type family IdGhcP pass where
  IdGhcP 'Parsed      = RdrName
  IdGhcP 'Renamed     = Name
  IdGhcP 'Typechecked = Id
```

So HsVar contains

- A RdrName after parsing
- A Name after renaming
- An Id after type checking

XXExpr: extending HsExpr

```
module Language.Haskell.Syntax.Expr where

data HsExpr p
  = HsVar      (XVar p)  (LIdP p)
  | HsLit      (XLitE p) (HsLit p)
  | OpApp      (XOpApp p)  (LHsExpr p)  (LHsExpr p)  (LHsExpr p)
  ...dozens of others...
  | XExpr     !(XXExpr p)

module Language.Haskell.Syntax.Extension where

type LIdP p = XRec p (IdP p)
type family IdP p
```

- XXExpr (GhcPass p): says what extra constructors are needed in HsExpr after pass p.

```
module GHC.Hs.Expr where

type instance XXExpr GhcPs = DataConCantHappen
type instance XXExpr GhcRn = HsExpansion (HsExpr GhcRn)  (HsExpr GhcRn)
type instance XXExpr GhcTc = XXExprGhcTc

-- After renaming
data HsExpansion orig expanded = HsExpanded orig expanded

-- After typechecking
data XXExprGhcTc = WrapExpr ... | ExpansionExpr ... | ...
```

Typecheck then desugar?
Or desugar then typecheck?

The Original Plan

- Typecheck the original Haskell, as written by the user
- Desugar afterwards
- That way, the error messages make sense.

The Original Plan can be Jolly Hard Work

```
x :: T Int Char

y = x { name = "Simon", info1 = True }
-- y :: T Bool Char
```

```
data T a b = MkT { name  :: String
                 , info  :: a
                 , info2 :: b }
```

- Typechecking the original took a hundred lines of tricky code
- If we desugar first....

```
x :: T Int Char

y = case x of
      MkT { info2 = i2 }
        -> MkT { name = "Simon", info1 = True, info2 = i2 }
```

- ...it's all much easier

Rebindable syntax

- For –XRebindableSyntax, the definition of "well-typed" is "expand and typecheck the expansion".
- E.g. numeric literals with -XRebindableSyntax
  - 3 means fromInteger 3
  - where 'fromInteger' means whatever fromInteger is in scope, which might have a weird type like fromInteger :: Integer -> a -> Bool
- Most straightforward approach: desugar then typecheck.

- https://gitlab.haskell.org/ghc/ghc/-/wikis/Rebindable-syntax

HsExpansion and error messages

```
data HsExpr p
  = ...dozens of others...
  | XExpr     !(XXExpr p)
```

```
type instance XXExpr GhcRn
     = HsExpansion (HsExpr GhcRn) (HsExpr GhcRn)

-- After renaming
data HsExpansion orig expanded = HsExpanded orig expanded
```

- 'orig' retains the original, un-expanded, expression
- 'expanded' is the desugared version
- Typechecker pushes 'orig' on the context stack, for the "In the expression ..." location information
- SrcSpans on 'expanded' are "GeneratedSrcSpan", and are **not** put on the context stack by the typechecker
- Somewhere inside 'expanded' we'll get back to "original" expressions, with non-Generated SrcSpans, and will resume putting SrcSpans on the context stack

Two places to do this desguaring

- In the Renamer

```
{- Note [Handling overloaded and rebindable constructs]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
For overloaded constructs (overloaded literals, lists, strings), and
rebindable constructs (e.g. if-then-else), our general plan is this,
using overloaded labels #foo as an example:

* In the RENAMER: transform
      HsOverLabel "foo"
      ==> XExpr (HsExpansion (HsOverLabel #foo)
                             (fromLabel `HsAppType` "foo"))
  We write this more compactly in concrete-syntax form like this
      #foo  ==>  fromLabel @"foo"

  Recall that in (HsExpansion orig expanded), 'orig' is the original term
  the user wrote, and 'expanded' is the expanded or desugared version
  to be typechecked.

* In the TYPECHECKER: typecheck the expansion, in this case
      fromLabel @"foo"
  The typechecker (and desugarer) will never see HsOverLabel
```

Two places to do this desguaring

- In the Typechecker

```
tcExpr expr@(RecordUpd { rupd_expr = record_expr
                       , rupd_flds = RegularRecUpdFields
                                       { xRecUpdFields = possible_parents
                                       , recUpdFields  = rbnds } })
       res_ty
  = assert (notNull rbnds) $
    do  { -- Desugar the record update. See Note [Record Updates].
        ; (ds_expr, ds_res_ty, err_ctxt)
            <- desugarRecordUpd record_expr possible_parents rbnds res_ty

          -- Typecheck the desugared expression.
        ; expr' <- addErrCtxt err_ctxt $
                   tcExpr (mkExpandedExpr expr ds_expr) (Check ds_res_ty)
        ...
```

- Typecheck has a bit more information available

Back to type inference

Typechecking an expression

```
module GHC.Tc.Gen.Expr where

tcMonoExpr :: LHsExpr GhcRn
           -> ExpRhoType
           -> TcM (LHsExpr GhcTc)
```

Output of renamer

"Expected type" of the term

Typechecker monad

"Elaborated term"

- TcM carries
  - Type environment: what is in scope, with what type
  - Ambient level
  - Error context
  - Template Haskell stage
  - State to accumulate
    - emitted constraints
    - error messages

Typechecking an expression

- TcM carries
  - Type environment: what is in scope, with what type
  - Ambient level
  - State to accumulate emitted constraints
  - Error context

```
Bar2.hs:3:11: error:
    • Couldn't match expected type ‘Bool’ with actual type ‘Char’
    • In the first argument of ‘not’, namely ‘'c'’
      In the expression: not 'c'
      In an equation for ‘f’: f x = not 'c'
  |
3 | f x = not 'c'
  |           ^^^
```

Elaboration

Before typechecking: HsExpr GhcRn

```
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
...
```

After typechecking: HsExpr GhcTc

```
$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

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 = /\a. \(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

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 …`

Recording the elaboration

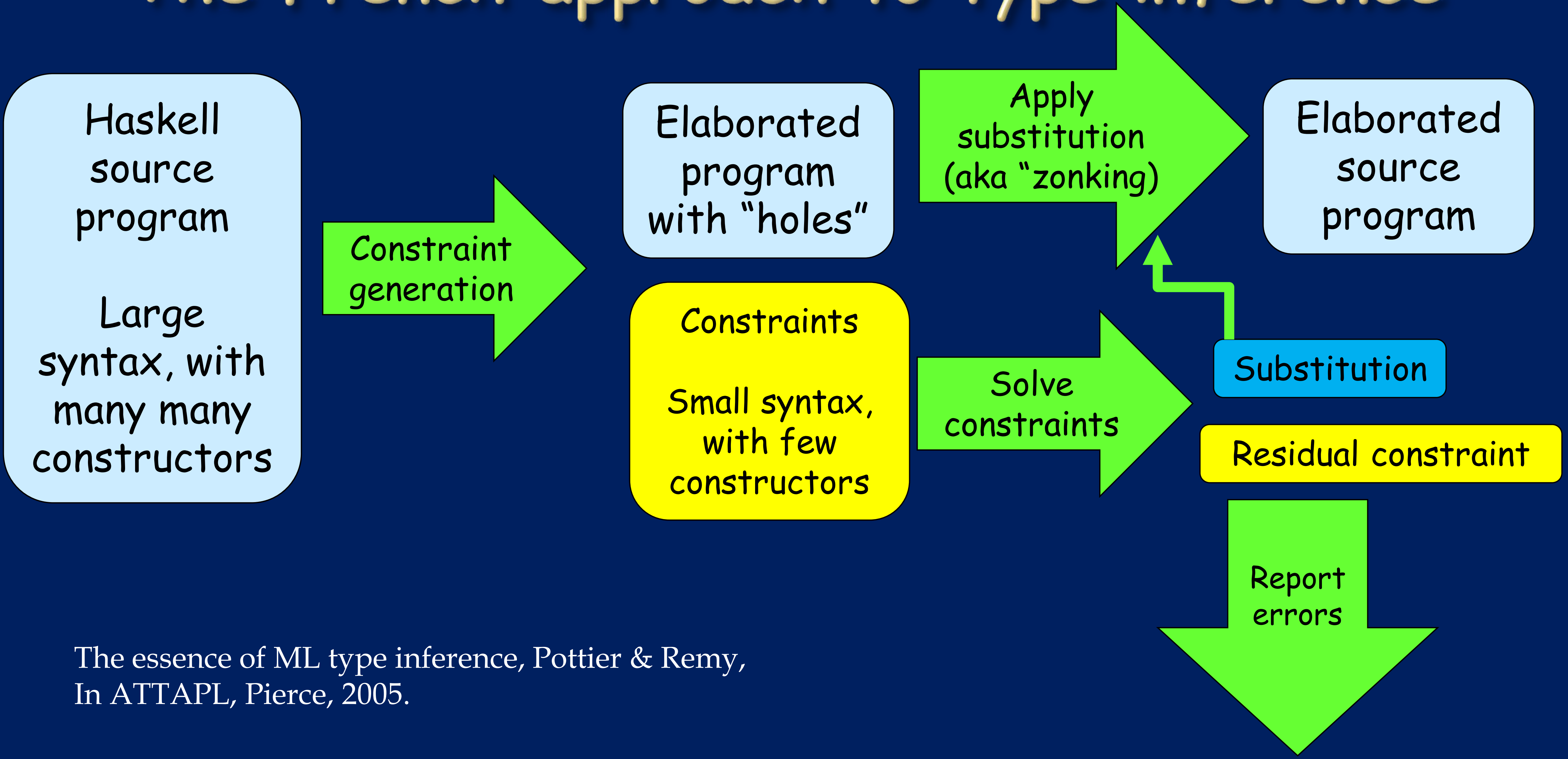
```
data HsExpr p
  = ...dozens of constructors...
  | XExpr     !(XXExpr p)
```

- Type applications, type abstractions, dictionary bindings, are all stored in a TTG extension constructor

```
type instance XXExpr GhcTc = XXExprGhcTc

data XXExprGhcTc = WrapExpr HsWrapper (HsExpr GhcTc)
                 | ...

data HsWrapper = WpHole
               | WpCompose HsWrapper HsWrapper
               | WpFun HsWrapper HsWrapper (Scaled TcTypeFRR)
               | WpCast TcCoercionR
               | WpEvLam  EvVar
               | WpEvApp  EvTerm
               | WpTyLam  TyVar
               | WpTyApp  KindOrType
               | WpLet    TcEvBinds
               | WpMultCoercion Coercion
```

Generating and solving constraints

The French approach to type inference



The essence of ML type inference, Pottier & Remy,
In ATTAPL, 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 (e.g. Damas-Milner): solve typechecking problems in the order you encounter them
- Result: small (even syntactic) changes to the program can affect whether it is accepted ☹

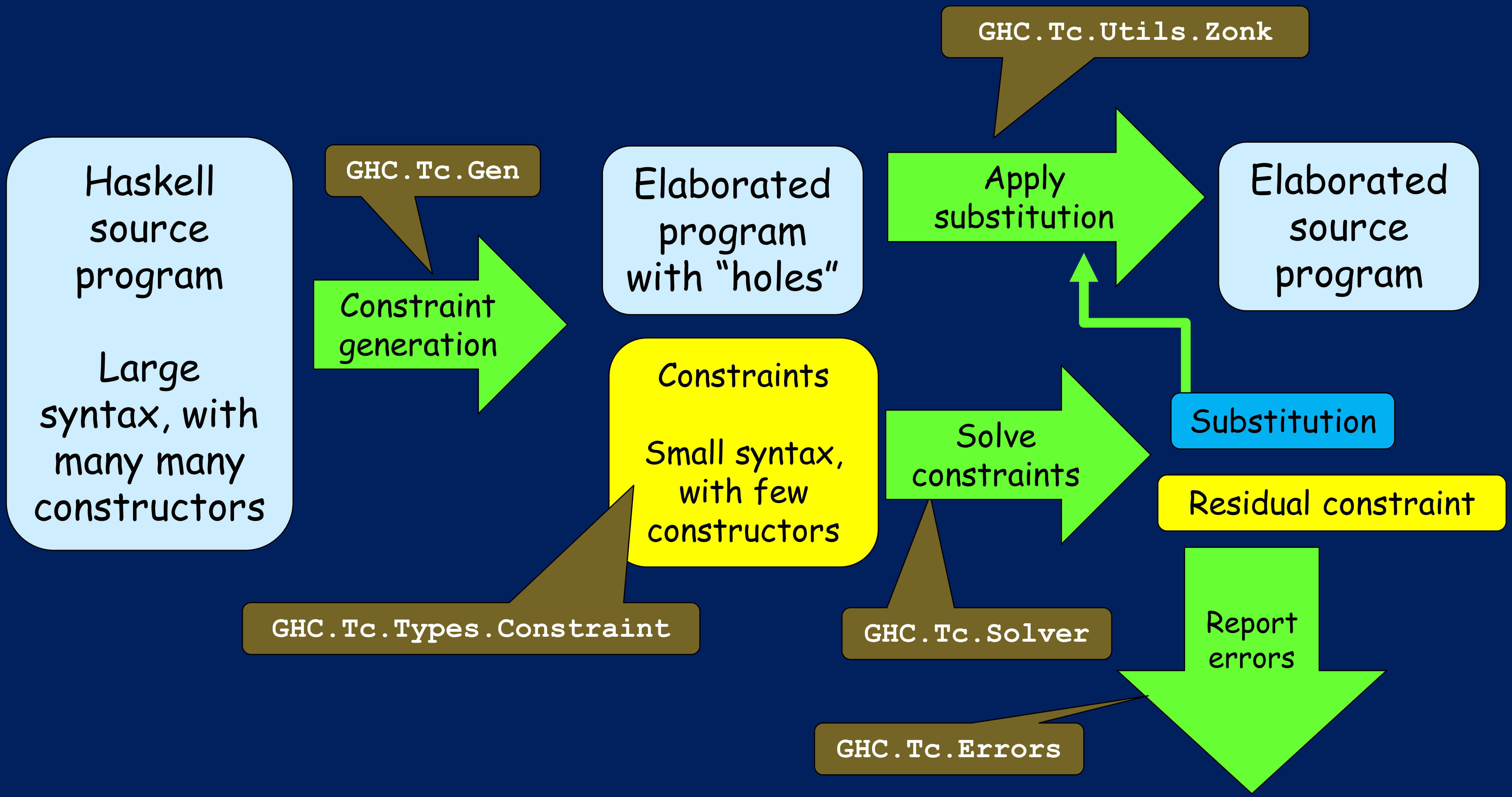
TL;DR: generate-then-solve is much more robust

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 (GHC.Tc.Errors) instead of scattered all over the type checker.
- Constraints carry "provenance" information to say whence they came

Practical benefits

- **Highly modular**
  - constraint generation (7 modules, 8,000 loc)
  - constraint solving (5 modules, 7,000 loc)
  - error message generation (1 module, 10,000 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 proposed type system extensions: is it easy to generate constraints, or do we need a new form of constraint?



Haskell source program
Large syntax, with many many constructors
GHC.Tc.Gen
Constraint generation
Elaborated program with "holes"
Constraints
Small syntax, with few constructors
GHC.Tc.Types.Constraint
GHC.Tc.Utils.Zonk
Apply substitution
Elaborated source program
Solve constraints
GHC.Tc.Solver
Substitution
Residual constraint
Report errors
GHC.Tc.Errors

Constraint generation

```
module GHC.Tc.Gen.Expr where

tcMonoExpr :: LHsExpr GhcRn
           -> ExpRhoType
           -> TcM (LHsExpr GhcTc)
```

Output of renamer

"Expected type" of the term

Elaborated term

Generated constraints accumulated by TcM (writer monad)

The type checker source code (June 2023)

		Code	Comments
Constraint generation	GHC.Tc.Gen	11,363	8,300
Constraint solver	GHC.Tc.Solver	5,944	7,152
Error checking and messages	GHC.Tc.Errors	9,242	4,987
“Deriving” (Ryan Scott)	GHC.Tc.Deriv	4,260	4,401
Type and class decls	GHC.Tc.TyCl	4,889	4,639
Instance decls	GHC.Tc.Instance	1,321	1,451
Utilities	GHC.Tc.Utils	6,497	5,447
Zonk	GHC.Tc.Zonk	1,848	741
Types	GHC.Tc.Types	3,111	2,891
TOTAL		**49,603**	**41,439**

Types and zonking

Types

```
module GHC.Core.TyCo.Rep where
type Kind = Type

data Type
   = TyVarTy Var
   | AppTy Type Type
   | TyConApp TyCon [Type]
   | ForAllTy ForAllTyBndr Type
   | FunTy FunTyFlag Mult Type Type
   | LitTy TyLit
   | CastTy Type Coercion
   | CoercionTy Coercion
```

Faithfully represents Haskell types

forall a. Eq a => a -> a

```
ForAllTy a (FunTy FTF_C_T
    (TyConApp Eq [TyVarTy a])
    (FunTy FTF_T_T (TyVarTy a)
                   (TyVarTy a)
```

```
data TyLit
  = NumTyLit Integer
  | StrTyLit FastString
  | CharTyLit Char

data FunTyFlag
  = FTF_T_T   -- (->)  Type -> Type -> Type
  | FTF_T_C   -- (-=>) Type -> Constraint -> Constraint
  | FTF_C_T   -- (=>)  Constraint -> Type -> Type
  | FTF_C_C   -- (==>) Constraint -> Constraint -> Constraint
```

```
module GHC.Types.Var where

type TyVar = Var
data Var = ...
    = TyVar { varName :: Name
            , realUnique :: !Int
            , varType :: Kind }
    | TcTyVar { ... }
    | Id {...}
```

Side note: Notes

```
data Var
  = TyVar {  -- Type and kind variables
             -- see Note [Kind and type variables]
        varName    :: !Name,
        realUnique :: {-# UNPACK #-} !Int,
                                     -- ^ Key for fast comparison
                                     -- Identical to the Unique in the name,
                                     -- cached here for speed
        varType    :: Kind           -- ^ The type or kind of the 'Var' in question
 }

  | TcTyVar {                           -- Used only during type inference
                                        -- Used for kind variables during
                                        -- inference, as well
        varName        :: !Name,
        realUnique     :: {-# UNPACK #-} !Int,
        varType        :: Kind,
        tc_tv_details  :: TcTyVarDetails
  }

  | Id {
        varName    :: !Name,
        realUnique :: {-# UNPACK #-} !Int,
        varType    :: Type,
        varMult    :: Mult,             -- See Note [Multiplicity of let binders]
        idScope    :: IdScope,
        id_details :: IdDetails,        -- Stable, doesn't change
        id_info    :: IdInfo }          -- Unstable, updated by simplif

-- | Identifier Scope
data IdScope    -- See Note [GlobalId/LocalId]
  = GlobalId
  | LocalId ExportFlag

data ExportFlag   -- See Note [ExportFlag on binders]
  = NotExported   -- ^ Not exported: may be discarded as dead code.
  | Exported      -- ^ Exported: kept alive
```

Cites Note without disturbing the code

```
{- Note [ExportFlag on binders]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
An ExportFlag of "Exported" on a top-level binder says "keep this
binding alive; do not drop it as dead code".  This transitively
keeps alive all the other top-level bindings that this binding refers
to.  This property is persisted all the way down the pipeline, so that
the binding will be compiled all the way to object code, and its
symbols will appear in the linker symbol table.

However, note that this use of "exported" is quite different to the
export list on a Haskell module.  Setting the ExportFlag on an Id does
/not/ mean that if you import the module (in Haskell source code) you
will see this Id.  Of course, things that appear in the export list
of the source Haskell module do indeed have their ExportFlag set.
But many other things, such as dictionary functions, are kept alive
by having their ExportFlag set, even though they are not exported
in the source-code sense.

We should probably use a different term for ExportFlag, like
KeepAlive.

Note [GlobalId/LocalId]
~~~~~~~~~~~~~~~~~~~~~~~
A GlobalId is
  * always a constant (top-level)
  * imported, or data constructor, or primop, or record selector
  * has a Unique that is globally unique across the whole
    GHC invocation (a single invocation may compile multiple modules)
  * never treated as a candidate by the free-variable finder;
        it's a constant!

A LocalId is
  * bound within an expression (lambda, case, local let(rec))
  * or defined at top level in the module being compiled
  * always treated as a candidate by the free-variable finder
```

Note gives the details. May be cited in many places

- Notes are a very simple device
- Heavily used in GHC (over 2,500 Notes)
- An absolute life saver
- Letters to our future selves
- See Wiki coding style guidance

Coding style

https://gitlab.haskell.org/ghc/ghc/-/wikis

Last edited by Simon Peyton Jones 2 minutes ago

Home

This is GHC's Wiki.

You may wish to see the table of contents to get a sense for what is available in this Wiki.

Everyone can edit this wiki. Please do so -- it easily gets out of date. But it has lots of useful inf

Quick Links

- GHC Home GHC's home page
- GHC User's Guide Official user's guide for current and past GHC releases.
- Joining In
  - Newcomers info / Contributing Get started as a contributor.
  - Mailing Lists & IRC Stay up to date and chat to the community.
  - The GHC Team Contributors and who is responsible for what.
- Documentation
  - GHC Status Info How is the next release coming along?
  - Migration Guide Migrating code between GHC releases
  - Working conventions How should I do X?
  - Building Guide Build GHC from source.
  - Coding style guidelines Please follow these guidelines.
  - Trees that grow (TTG) Guidance on how GHC uses the TTG style.
  - Debugging How to debug the GHC.
  - Commentary Explaining the GHC source code.

Coding style guidelines

coding style

This is a description of some of the coding practices and style that we use
Guidelines for RTS C code. Also see the wiki page on Contributing for issue

Table of contents

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" (aka state!) 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.

Unification variables

```
module GHC.Types.Var where

data Var
  = TyVar { varName    :: !Name
          , realUnique :: {-# UNPACK #-} !Int
          , varType    :: Kind }

  | TcTyVar { varName        :: !Name,
            , realUnique     :: {-# UNPACK #-} !Int,
            , varType        :: Kind,
            , tc_tv_details  :: TcTyVarDetails }

  | Id { ... }
```

```
module GHC.Tc.Utils.TcType where

type TcType = Type -- May have TcTyVars

data TcTyVarDetails
  = SkolemTv SkolemInfo TcLevel Bool

  | MetaTv { mtv_info   :: MetaInfo
           , mtv_ref    :: IORef MetaDetails
           , mtv_tclvl  :: TcLevel }

  | RuntimeUnk

data MetaDetails = Flexi | Indirect TcType

data MetaInfo
   = TauTv
   | TyVarTv
   | RuntimeUnkTv
   | CycleBreakerTv
   | ConcreteTv ConcreteTvOrigin
```

- No static distinction between TcType and Type,
- Sad, but has never proved to be a problem in practice

Zonking

- Zonking replaces a filled-in meta-tyvar with the type in the ref-cell.

```
module GHC.Tc.Zonk.TcType where
  zonkTcType :: TcType -> TcM TcType
```

- Saves requiring every function that examines types to be in the TcM monad; instead, zonk first.
- Tricky point: knowing when to zonk.
  - Zonking too much is inefficient
  - Zonking too little is wrong.

Two completely-separate zonkers

```
module GHC.Tc.Zonk.TcType where
  zonkTcType :: TcType -> TcM TcType
```

- Used **during** type inference
- Result can have TcTyVars
- Types and constraints only (hence small)

```
module GHC.Tc.Zonk.Type where
  zonkTcTypeToType :: TcType -> TcM Type
```

- Used **after** type inference
- Result has no TcTyVars
- Types and terms (hence big)
- Fills in "holes" in the elaborated term

Inference vs checking

```
module GHC.Tc.Gen.Expr where

tcMonoExpr :: LHsExpr GhcRn      -- Expression to type check
           -> ExpType            -- Expected type
           -> TcM (LHsExpr GhcTc)
```

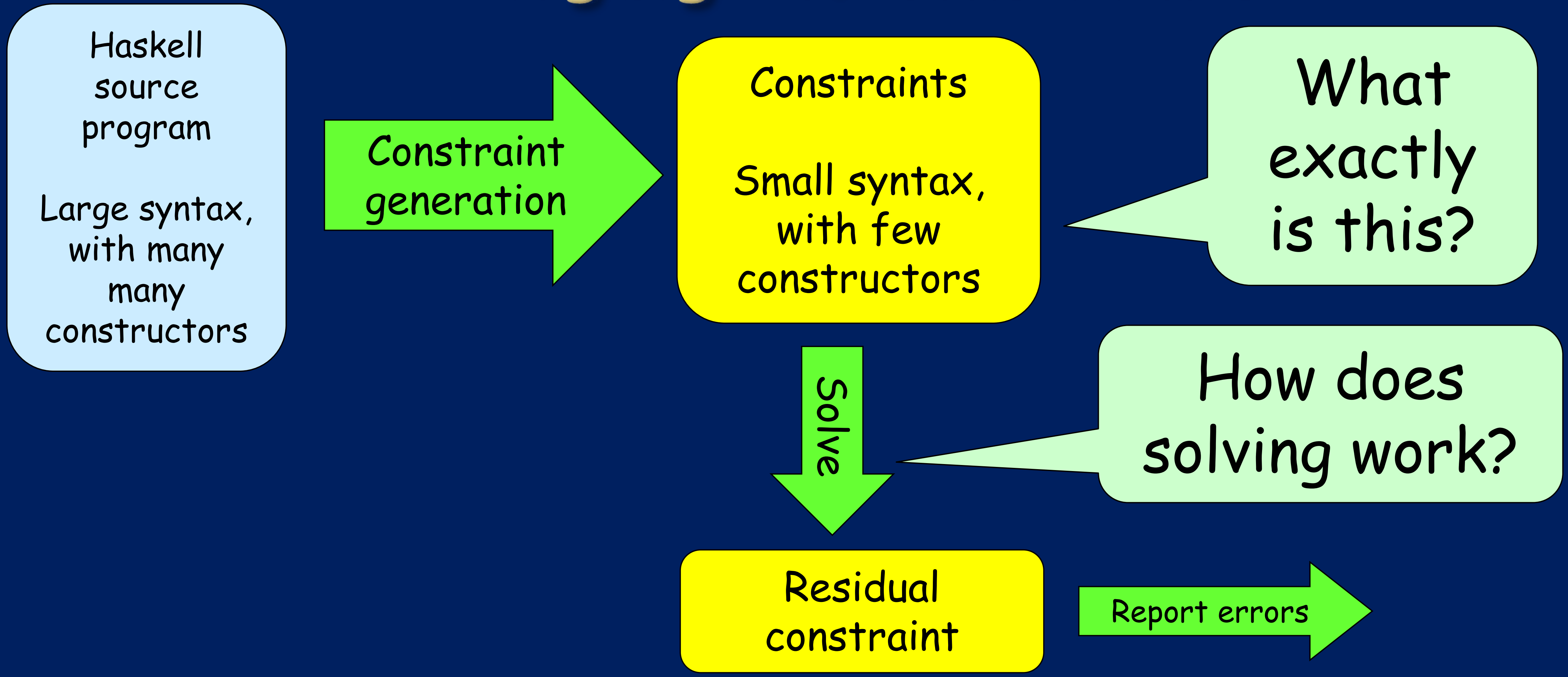
```
-- Checking
f :: Int -> Int
f x = x+1

-- Inference
g x = x+1
```

```
module GHC.Tc.Utils.TcType where

data ExpType = Check TcType
             | Infer !InferResult

data InferResult
  = IR { ir_uniq :: Unique
       , ir_lvl  :: TcLevel
       , ir_frr  :: Maybe FixedRuntimeRepContext
       , ir_ref  :: IORef (Maybe TcType) }
```

Very like a unification variable BUT can be filled in with a polytype

Back to constraints

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

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

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"

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

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

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

Substitute $\beta := \delta$

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

$\beta := \delta$

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

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

Substitute $\delta := Int$

$d{:}Ord\ Int$

$\beta := \delta$
$\delta := Int$

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 elaborated term

Tracing the solver

Tracing the type checker

```
module Foo where

f :: Eq a => [a] -> [a] -> Bool
f xs ys = not (xs == ys)
```

```
$ ghc -c -ddump-tc-trace Foo.hs >& foo.tc-trace
$ wc Foo.tc-trace
 1748  6058 48736 Foo.tc-trace
```

Tips

- Line 1115
  Starting to typecheck f

```
Bindings for { [f]
Generalisation plan
  CheckGen f :: forall a. Eq a => [a] -> [a] ->
Bool
tcPolyCheck
  f
  Foo.hs:3:1-31
```

- Line 1325
  Finished f
  (NB: matching braces)

```
} End of bindings for
  [f]
  NonRecursive
  f forall a. Eq a => [a] -> [a] -> Bool
tcExtendBinderStack [f[<TopLevel>]]
```

- Unification

```
writeMetaTyVar a_aKv[tau:1] := [a_aKr[sk:1]]
```

Tips

- Line 1359: solve constraints (again matching braces)

```
Tc6
Tc7
Tc7a
simplifyTop {
  wanted =  WC {wc_impl =
                  Implic {
                    TcLevel = 1
                    Skolems = a_aKr[sk:1]
                    Given-eqs = MaybeGivenEqs
                    Status = Unsolved
                    Given = $dEq_aKs :: Eq a_aKr[sk:1]
                    Wanted =
                      WC {wc_simple = [W] $dEq_aKw {0}:: Eq a_aKv[tau:1] (CNonCanonical)}
                    Binds = EvBindsVar<aKA>
                    the type signature for:
                      f :: forall a. Eq a => [a] -> [a] -> Bool }}
```

Build with -DDEBUG

- Always build your development compiler with -DDEBUG
- That enables a bunch of assertions, which sometimes catch bugs early

Implication constraints

Existentials

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

```
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
     ]
```

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
```

"gd" is short for "Given dictionary"

Generate constraints

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

- $f : \alpha$
- $t : \beta$
- $x : a$
- Instantiate show with δ

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

Generate constraints

$\alpha \sim \beta \rightarrow \gamma$	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 foo

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 foo

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

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
```

$\alpha \sim \beta \rightarrow \gamma$	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 foo

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

Reminder

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

Implication constraint ($\forall a_1..a_n.$)

Given (W_1)

Wanted (W_2)

Constraints

```
data WantedConstraints
  = WC { wc_simple :: Bag Ct
       , wc_impl   :: Bag Implication
       , wc_errors :: Bag DelayedError }
```

```
data Implication = Implic {
      ic_tclvl  :: TcLevel,
      ic_info   :: SkolemInfoAnon,
      ic_skols  :: [TcTyVar],
      ic_given  :: [EvVar],
      ic_wanted :: WantedConstraints,
      ic_binds  :: EvBindsVar,
      ...some more stuff... }
```

```
data Ct = CDictCan       DictCt
        | CEqCan         EqCt
        | CIrredCan      IrredCt
        | CQuantCan      QCInst
        | CNonCanonical  CtEvidence
```

```
data DictCt   -- e.g.  Num ty
  = DictCt { di_ev   :: CtEvidence
           , di_cls  :: Class
           , di_tys  :: [Xi]
           , di_pend_sc :: ExpansionFuel }
```

```
data EqCt = EqCt { eq_ev     :: CtEvidence
                 , eq_lhs    :: CanEqLHS
                 , eq_rhs    :: Type
                 , eq_eq_rel :: EqRel }
```

```
data CanEqLHS = TyVarLHS TcTyVar
              | TyFamLHS TyCon [Type]
```

$\alpha \sim \beta \to \gamma$ 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 foo

Solving

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

Substitute $\delta \coloneqq a$

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

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

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

Substitute $\gamma \coloneqq String$

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
```

What is 'a'?

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

Generate constraints

$$\alpha \sim \beta \rightarrow \gamma$$
$$\beta \sim T$$

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

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

- 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 ∀

Unification variables

```
module GHC.Types.Var where

data Var
  = TyVar { varName     :: !Name
          , realUnique :: {-# UNPACK #-} !Int
          , varType     :: Kind }

  | TcTyVar { varName         :: !Name,
            , realUnique      :: {-# UNPACK #-} !Int,
            , varType         :: Kind,
            , tc_tv_details   :: TcTyVarDetails }

  | Id { ... }
```

- SkolemTv: bound by type signature or existential pattern match
- MetaTv: a meta-tyvar (aka unification variable)

```
module GHC.Tc.Utils.TcType where

type TcType = Type -- May have TcTyVars

data TcTyVarDetails
  = SkolemTv SkolemInfo TcLevel Bool

  | MetaTv { mtv_info   :: MetaInfo
           , mtv_ref    :: IORef MetaDetails
           , mtv_tclvl  :: TcLevel }

  | RuntimeUnk

data MetaDetails = Flexi | Indirect TcType

data MetaInfo
   = TauTv
   | TyVarTv
   | RuntimeUnkTv
   | CycleBreakerTv
   | ConcreteTv ConcreteTvOrigin
```

Level numbers

Existential escape

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

Generate constraints

```
MkT :: ∀a. Show a => a -> T
```

$\alpha \sim \beta \rightarrow \gamma$ From the lambda
$\beta \sim T$ From the case

$\forall a. (gd : Show\ a) \Rightarrow$
$\{ \gamma \sim a \}$ From result of foo

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

Existential escape

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

Generate constraints

```
MkT :: ∀a. Show a => a -> T
```

$\alpha \sim \beta \rightarrow \gamma$ From the lambda

$\beta \sim T$ From the case

$\forall a.\,(gd : Show\ a) \Rightarrow$
$\{\gamma \sim a\}$ From result of foo

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

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

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
$\forall a^2.(gd : Show\ a) \Rightarrow$ $\{\gamma^1 \sim a^2\}$	From result of foo

- Every TcTyVar type variable has a level number
  - Unification variables like $'\alpha'$
  - Skolems like 'a'
- Cannot unify outer γ^1 with a type whose free vars include inner a^2

Unification variables

```
module GHC.Types.Var where

data Var
  = TyVar { varName     :: !Name
          , realUnique :: {-# UNPACK #-} !Int
          , varType    :: Kind }

  | TcTyVar { varName        :: !Name,
            , realUnique     :: {-# UNPACK #-} !Int,
            , varType        :: Kind,
            , tc_tv_details  :: TcTyVarDetails }

  | Id { ... }
```

- Both SkolemTv and MetaTv has a level number

```
module GHC.Tc.Utils.TcType where

type TcType = Type -- May have TcTyVars

data TcTyVarDetails
  = SkolemTv SkolemInfo TcLevel Bool

  | MetaTv { mtv_info   :: MetaInfo
           , mtv_ref    :: IORef MetaDetails
           , mtv_tclvl  :: TcLevel }

  | RuntimeUnk

data MetaDetails = Flexi | Indirect TcType

data MetaInfo
   = TauTv
   | TyVarTv
   | RuntimeUnkTv
   | CycleBreakerTv
   | ConcreteTv ConcreteTvOrigin

newtype TcLevel = TcLevel Int
```

Back to our earlier example

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

Generate constraints

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

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

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

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

This is fine; no free inner variables

Promotion

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

- Can we unify $\alpha^1 := (\beta^2 \to Int)$?
- No, it has a free inner variable β^2
- But we can **promote** β, thus $\beta^2 := \gamma^1$, where γ^1 is fresh
- Now we can unify $\alpha^1 := (\gamma^1 \to Int)$

GADTs and untoucability

```
data G a where
  GInt :: Bool -> G Char
  MkG  :: a -> G a

f x = case x of
        GInt v -> 'x'
        MkG v  -> v
```

```
GInt  :: ∀a. (a~Char) => Bool -> G a
```

$\mathrm{f} : G\ \alpha^1 \rightarrow \beta^1$

$\mathrm{x} : G\ \alpha^1$

$\forall.(g:\alpha^1 \sim Char) \Rightarrow \{\ \beta^1 \sim Char\}$	from GInt branch
$\beta^1 \sim \alpha^1$	from MkG branch

GADTs and untoucability

```
data G a where
  GInt :: Bool -> G Char
  MkG  :: a -> G a

f x = case x of
        GInt v -> 'x'
        MkG v  -> v
```

Must not solve by $\beta^1 := Char$!

β^1 is "untouchable" under the equality $\alpha^1 \sim Char$

$\forall.(g{:}\,\alpha^1 \sim Char) \Rightarrow \{\beta^1 \sim Char\}$	from GInt branch
$\beta^1 \sim \alpha^1$	from MkG branch

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 foo

- 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 -> a
          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

Given and Wanted constraints

Givens and Wanteds

- Given constraint
  - We have evidence for it
  - Use it to prove Wanteds
- Wanted constraint
  - We want to produce evidence for it

Evidence

```
data Ct = CDictCan        DictCt
        | CEqCan          EqCt
        | CIrredCan       IrredCt
        | CQuantCan       QCInst
        | CNonCanonical   CtEvidence
```

```
data DictCt
  = DictCt { di_ev   :: CtEvidence
           , di_cls  :: Class
           , di_tys  :: [Xi]
           , di_pend_sc :: ExpansionFuel }
```

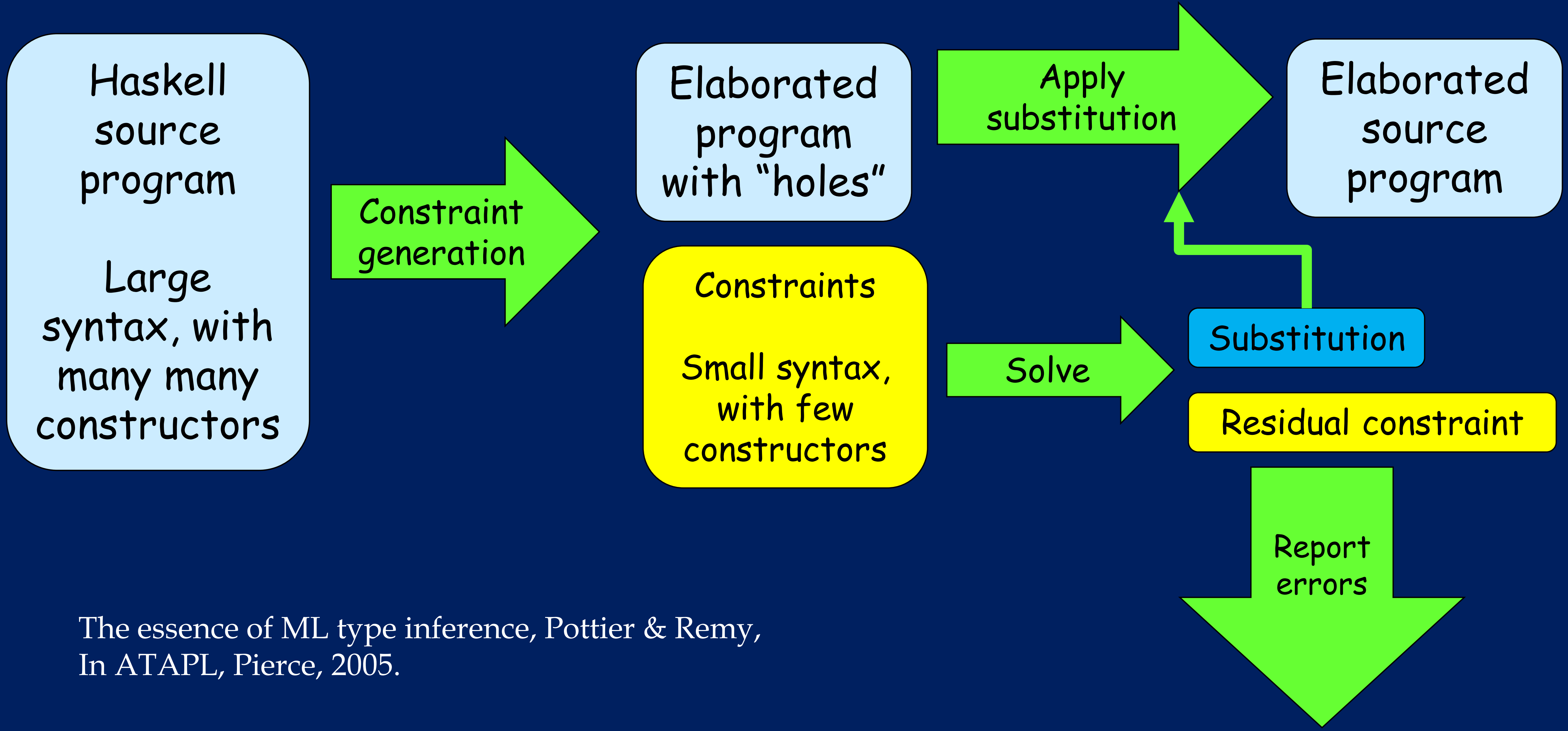
```
data EqCt = EqCt { eq_ev     :: CtEvidence
                 , eq_lhs    :: CanEqLHS
                 , eq_rhs    :: Type
                 , eq_eq_rel :: EqRel }
```

```
data CtEvidence
  = CtGiven    -- Truly given, not depending on subgoals
      { ctev_pred :: TcPredType      -- See Note [Ct/evidence invariant]
      , ctev_evar :: EvVar           -- See Note [CtEvidence invariants]
      , ctev_loc  :: CtLoc }


  | CtWanted   -- Wanted goal
      { ctev_pred      :: TcPredType    -- See Note [Ct/evidence invariant]
      , ctev_dest      :: TcEvDest      -- See Note [CtEvidence invariants]
      , ctev_loc       :: CtLoc
      , ctev_rewriters :: RewriterSet } -- See Note [Wanteds rewrite Wanteds]
```

```
data TcEvDest
  = EvVarDest EvVar
  | HoleDest  CoercionHole
```

Back to the big picture

The French approach to type inference



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

Things I have sadly not talked about

- Coercions: the evidence for equality
- Type families, and “flattening”
- Functional dependencies, injectivity
- 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

Things I have sadly not talked about

- Coercions: the evidenc
- Type familie
- Func "Derived"
  cor
- Def
- Unbo
- Nomina onal equality (Coercible etc)
- … and qu more

The good news

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

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

EXTRA SLIDES

There is lots more to say.

Far too much to fit in a 1-hr talk.

Some of these extra topics are in the following slides.

Evidence of equality

Equality constraints generate evidence too!

```
data T a where
  K1 :: Bool -> T Bool
  K2 :: T a

f :: T a -> Maybe a
f x = case x of
          K1 z -> Just z
          K2   -> Nothing
```

```
K1 :: ∀a. (a~Bool) =>
          Bool -> T a
```

Equality constraints generate evidence too!

```
f :: T a -> Maybe a
f = Λ(a:*) λ(x:T a).
    case x of
        K1 (c:a~Bool) (z:Bool)
            -> Just z ▷ c2
        K2 -> False
```

```
K1 :: ∀a. (a~Bool) =>
          Bool -> T a
```

Plus constraint to solve

$$\forall.(\mathrm{c} : \mathrm{a{\sim}Bool}) \Rightarrow (c2 : Maybe\ Bool \sim Maybe\ a)$$

Equality constraints generate evidence too!

$$\forall^2.(c : a \sim Bool) \Rightarrow (c2 : Maybe\ Bool \sim Maybe\ a)$$

Decompose

c2 := Maybe c3

$$\forall^2.(c : a \sim Bool) \Rightarrow (c3 : Bool \sim a)$$

Use given to substitute for a

c3 := c4 ; Sym c

$$\forall^2.(c : a \sim Bool) \Rightarrow (c3 : Bool \sim Bool)$$

Proving Bool~Bool is easy

c4 := Refl Bool

$$\forall^2.(c : a \sim Bool) \Rightarrow \epsilon$$

Plug the evidence back into the term

```
f :: T a -> Maybe a
f = Λ(a:*) λ(x:T a)
     case x of
        K1 (c:a~Bool) (z:Bool)
            -> Just z ▷ (Maybe (Refl Bool ; Sym c))
        K2 -> False
```

Floating with GADTs

```
data T a where
  K :: Bool -> T Bool

f x = case x of
        K z -> True
```

What type should we infer for f?

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

Neither is more general than (a substitution instance of) the other!

Floating with GADTs

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

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

f : $\alpha \to \gamma$
x : α

$\alpha^1 \sim T\,\beta^1$
$\forall^2.(\beta^1 \sim Bool) \Rightarrow \gamma^1 \sim Bool$

- Float, and solve?
  $\gamma^1 := Bool$

Get f :: ∀b. T b -> Bool

- Rewrite $\gamma^1 \sim Bool$ to $\gamma^1 \sim \beta^1$ using the given $\beta^1 \sim Bool$; then float and solve $\gamma^1 := \beta^1$

Get ∀b. T b -> b

Floating with GADTs

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

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

$f : \alpha \to \gamma$
$x : \alpha$

$\alpha^1 \sim T\ \beta^1$
$\forall^2 . (\beta^1 \sim Bool) \Rightarrow \gamma^1 \sim Bool$

Solution

Do not float anything out of an implication that has "given" equalities

Result (in this case):
"cannot unify untouchable γ with Bool"

Floating with GADTs

```
data T a where
  K1 :: Bool -> T Bool
  K2 :: T a
f2 x = case x of
         K1 z -> True
         K2   -> False
```

$f : \alpha \to \gamma$
$x : \alpha$

$\alpha^1 \sim T\ \beta^1$
$\forall^2.(\beta^1 \sim Bool) \Rightarrow \gamma^1 \sim Bool$
$\gamma^1 \sim Bool$

Another branch, with no given equalities, may resolve the ambiguity

From the K2 branch, no implication needed

Deferred type errors andtyped holes

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]
```

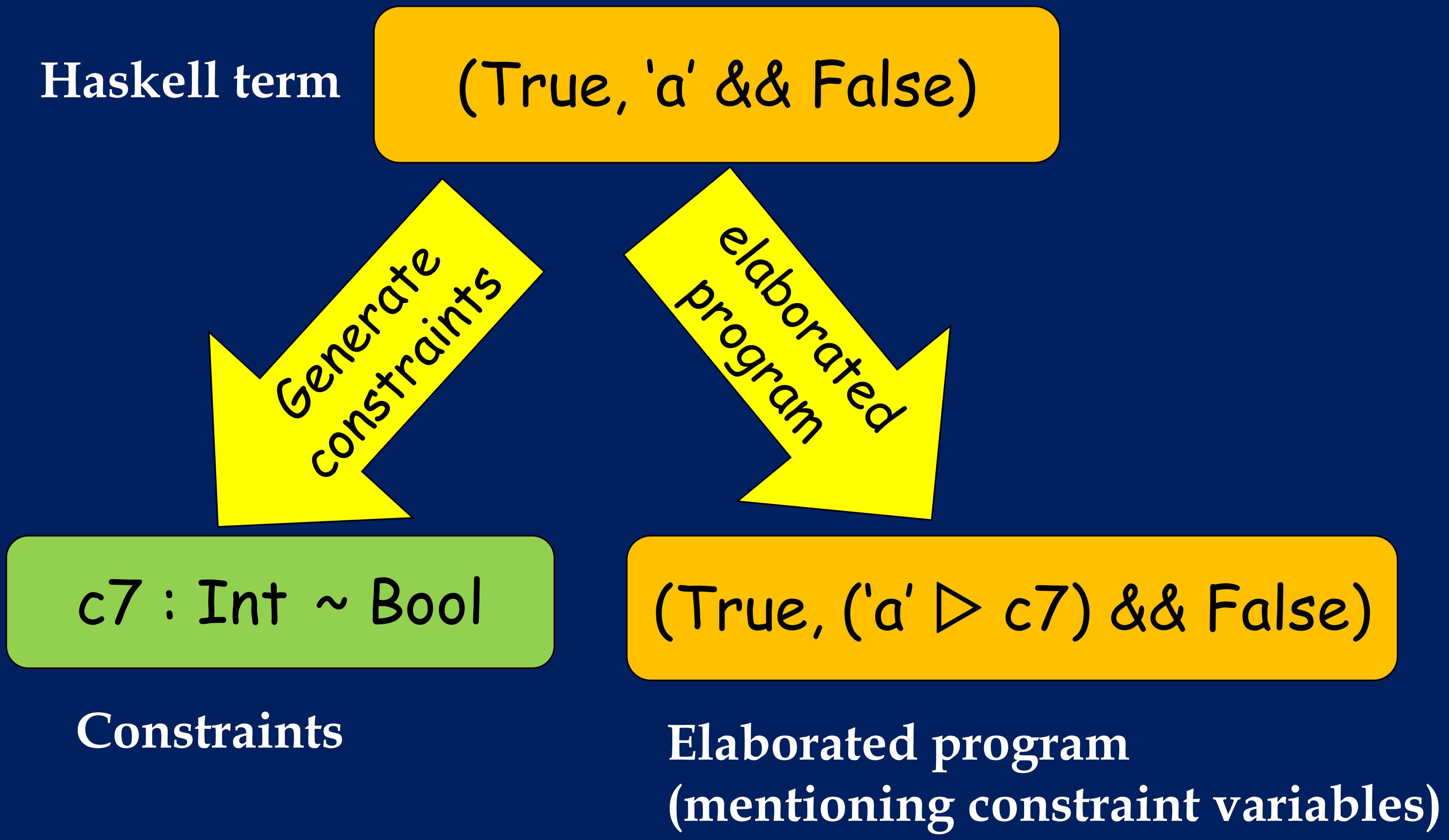
Combining the two

- -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...



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

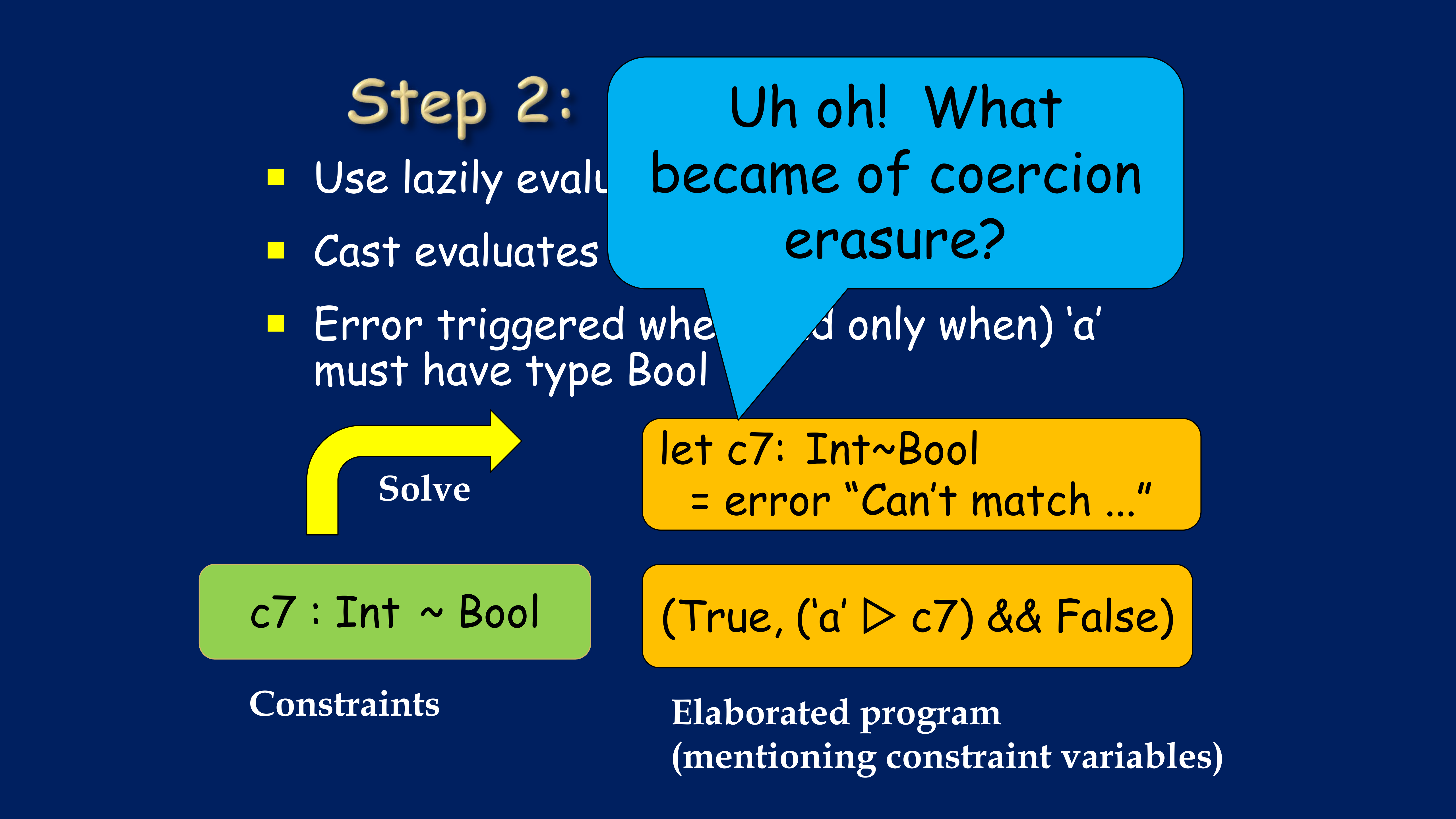
```
let c7: Int~Bool
  = error "Can't match ..."
```

```
c7 : Int ~ Bool
```

```
(True, ('a' ▷ c7) && False)
```

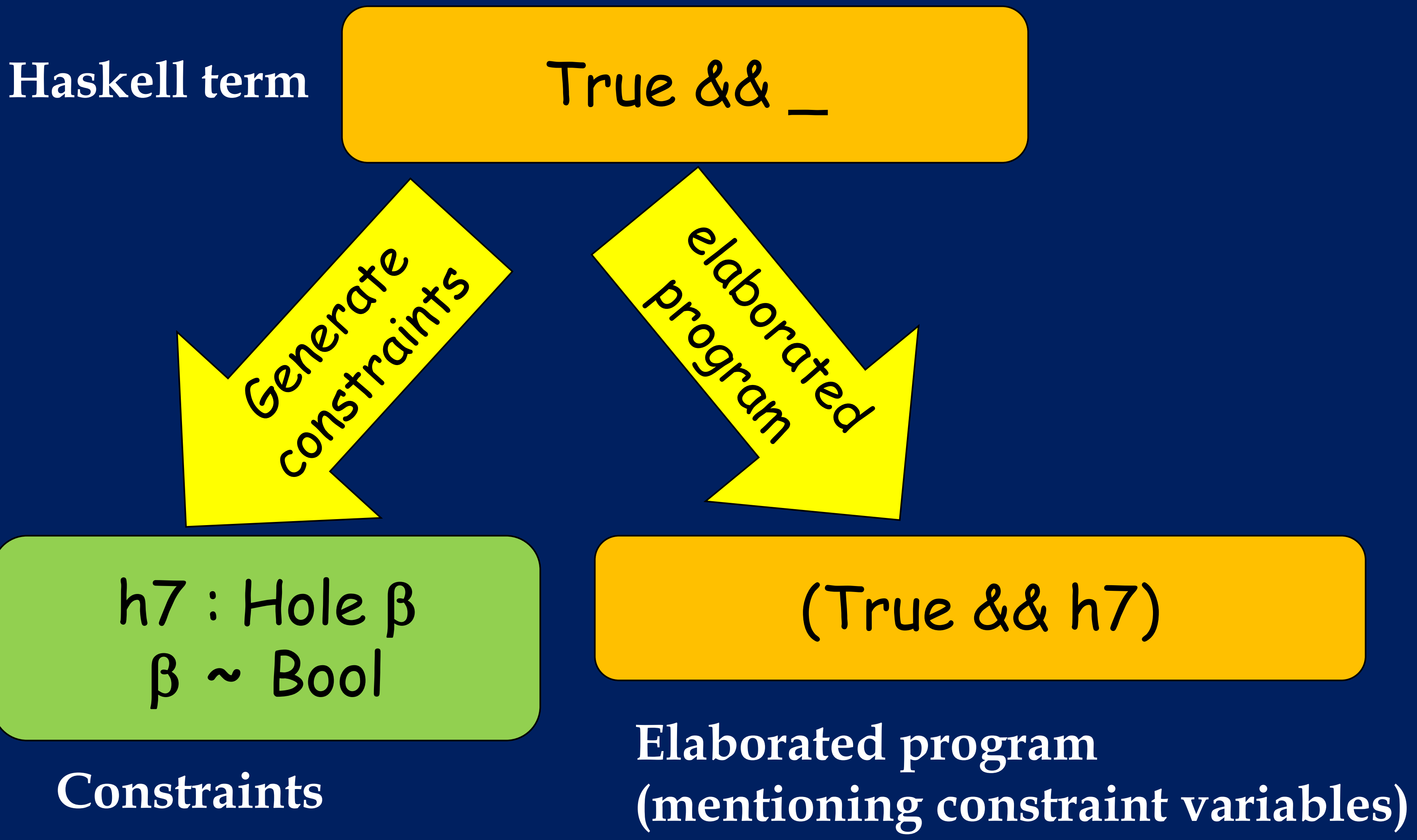
Constraints

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)

Generalisation

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 match on C

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
```

Here we abstract over the a~Bool constraint

But this surely should!

A possible path [Pottier et al]

Abstract over **all** unsolved constraints from RHS

- Big types, unexpected to programmer
- Errors postponed to usage sites
- (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
```

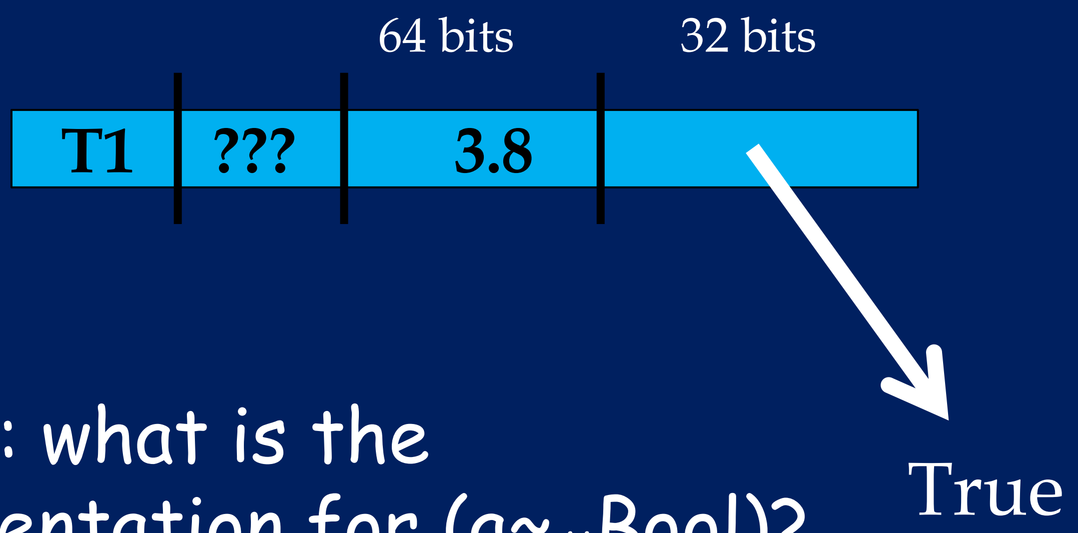
Inline + optimise

- 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

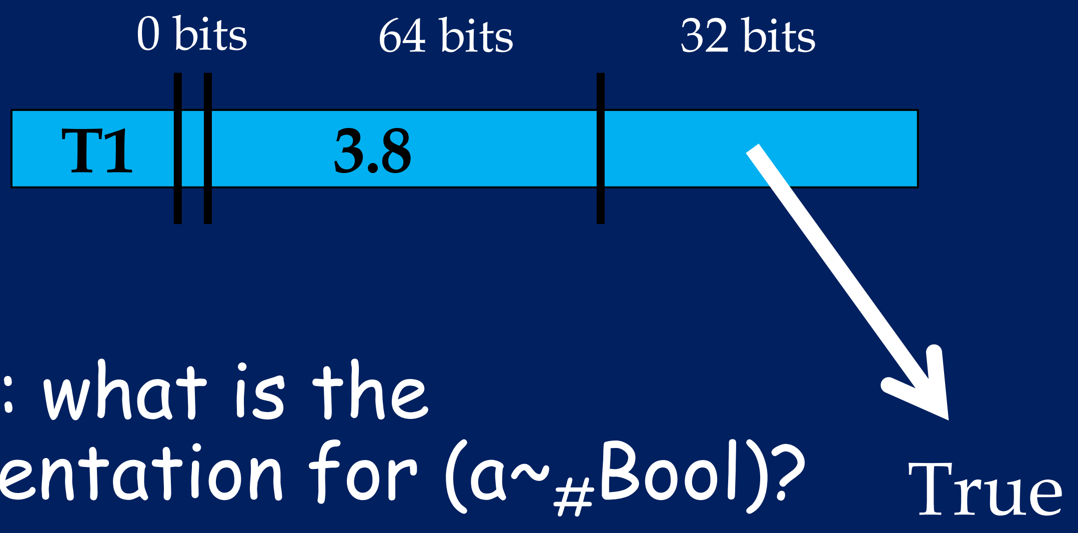


Question: what is the representation for $(a \sim_{\#} Bool)$?

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 ~# 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