Type directed compilation in the wild GHC and System FC

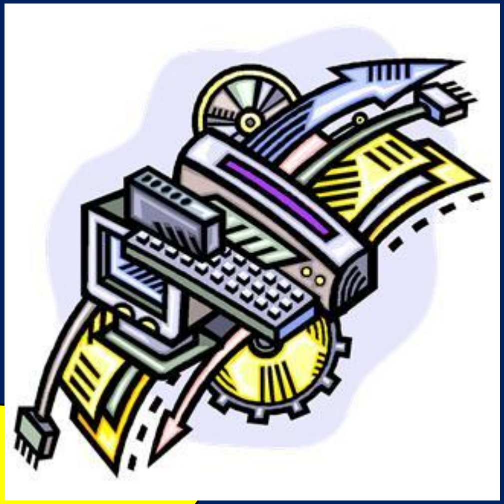
Simon Peyton Jones

Microsoft Research

June 2013

GHC Haskell

A rich language



GHC Haskell

A very complicated and ill-defined language, with a long user manual, that almost no one understands completely

GHC is big and old

Module	Lines (1992)	Lines (2011)	Increase
Compiler			
Main	997	11,150	11.2
Parser	1,055	4,098	3.9
Renamer	2,828	4,630	1.6
Type checking	3,352	24,097	7.2
Desugaring	1,381	7,091	5.1
Core tranformations	1,631	9,480	5.8
STG transformations	814	840	1
Data-Parallel Haskell	—	3,718	—
Code generation	2913	11,003	3.8
Native code generation	—	14,138	—
LLVM code generation	—	2,266	—
GHCi	—	7,474	—
Haskell abstract syntax	2,546	3,700	1.5
Core language	1,075	4,798	4.5
STG language	517	693	1.3
C-- (was Abstract C)	1,416	7,591	5.4
Identifier representations	1,831	3,120	1.7
Type representations	1,628	3,808	2.3
Prelude definitions	3,111	2,692	0.9
Utilities	1,989	7,878	3.96
Profiling	191	367	1.92
Compiler Total	28,275	139,955	4.9
Runtime System			
All C and C-- code	43,865	48,450	1.10

Figure 1: Lines of code in GHC, past and present

GHC is big and old

Module	Lines (1992)	Lines (2011)	Increase
Compiler			
Main	997	11,150	11.2
Parser	1,055	4,098	3.9
Renamer	2,828	4,630	1.6
Type checking	3,352	24,097	7.2
Desugaring	1,381	7,091	5.1
Core tranformations	[illegible]	[illegible]	[illegible]
STG [illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	2.3
[illegible]	3,111	2,692	0.9
[illegible]	1,989	7,878	3.96
Profiling	191	367	1.92
Compiler Total	28,275	139,955	4.9
Runtime System			
All C and C-- code	43,865	48,450	1.10

Figure 1: Lines of code in GHC, past and present

Question

how to stay sane?

How GHC works

Source language

Haskell

Massive language
Hundreds of pages of user manual
Syntax has dozens of data types
100+ constructors

Typecheck

Desugar

Typed intermediate language

Core
3 types,
15 constructors

Rest of GHC

A typed intermediate language

Haskell	Core (the typed IL)
Big	Small
Implicitly typed	Explicitly typed
Binders typically un-annotated \x. x && y	Every binder is type-annotated \(x:Bool). x && y
Type inference (complex, slow)	Type checking (simple, fast)
Complicated to specify just which programs will type-check	Very simple to specify just which programs are type-correct
Ad-hoc restrictions to make inference feasible	Very expressive indeed; simple, uniform

A typed intermediate language: why?

1. Small IL means that analysis, optimisation, and code generation, handle only a small language.

2. Type checker ("Lint") for Core is a very powerful internal consistency check on most of the compiler
   - Desugarer must produce well-typed Core
   - Optimisation passes must transform well-typed Core to well-typed Core

3. Design of Core is a powerful sanity check on crazy type-system extensions to source language. If you can desugar it into Core, it must be sound; if not, think again.

A typed intermediate language

- Small IL means that analysis ... tion, and code generati...
- Type... powerful inter... compiler
  - Des...
  - Opt... Core to well-...
- Desig... on crazy type-sy... source language. If you can desugar it into Core, it must be sound; if not, think again.

GHC is the only production compiler that remorselessly pursues this idea of a strongly-typed intermediate language

The design of Core is probably GHC's single most substantial technical achievement

WHAT SHOULD CORE BE LIKE?

What should Core be like?

- Start with lambda calculus. From "Lambda the Ultimate X" papers we know that lambda is super-powerful.
- But we need a TYPED lambda calculus
- Idea:
  - start with lambda calculus
  - sprinkle type annotations
- But:
  - Don't want to be buried in type annotations
  - Types change as you optimise

Example

```
compose :: (b->c) -> (a->b) -> a -> c
compose = λf:b->c. λg:a->b. λx:a.
            let tmp:b = g x
            in f tmp
```

- Idea: put type annotations on each binder (lambda, let), but nowhere else
- But: where is 'a' bound?
- And: unstable under transformation...

Example

```
compose :: (b->c) -> (a->b) -> a -> c
compose = λf:b->c. λg:a->b. λx:a.
            let tmp:b = g x
            in f tmp
```

```
neg :: Int -> Int
isPos :: Int -> Bool
```

```
compose isPos neg
=     (inline compose:
        f=isPos, g=neg)
   λx:a. let tmp:b = neg x
         in isPos tmp
```

- Now the type annotations are wrong
- Solution: learn from Girard and Reynolds!

System F

```
compose :: ∀abc. (b->c) -> (a->b) -> a -> c
compose = Λabc. λf:b->c. λg:a->b. λx:a.
            let tmp:b = g x
            in f tmp
```

- Idea: an explicit (big) lambda binds type variables

System F

```
compose :: ∀abc. (b->c) -> (a->b) -> a -> c
compose = Λabc. λf:b->c. λg:a->b. λx:a.
                  let tmp:b = g x
                  in f tmp
```

```
compose Int Int Bool isPos neg
=    (inline compose:
     a=Int, b=Int, c=Bool, f=isPos, g=neg)
   λx:Int. let tmp:Int = neg x
           in isPos tmp
```

- Big lambdas are applied to types,
  just as little lambdas are applied to values
- Now the types stay correct!

The real "System F"

- In GHC, the IL is like what we've seen but we add:
  - Algebraic data type declarations

    ```
    data Maybe a = Nothing | Just a
    ```

  - Data constructors in terms

    ```
    λx:Int. Just (Just x)
    ```

  - Case expressions

    ```
    case x of { Nothing -> 0; Just x -> x+1 }
    ```

  - Let expressions

    ```
    let x:Int = 4 in x+x
    ```

Existentials

```
data T a where
  T1 :: ∀a. ∀b. b -> (b -> a) -> T a

f :: T a -> a
f = Λa. \(x:T a).
    case x of
      T1 (b:*) (y:b) (g:b->a) -> g y
```

'b' is not mentioned in T1's result type

Pattern-matching on T1 binds the type variable 'b' as well as the term variables 'y' and 'g'

- We say that 'b' is an existential variable of T1

```
T1 :: ∀ab. b -> (b -> a) -> T a
    ≡ ∀a. (∃b.(b, b->a)) -> T a
```

System F is GHC's intermediate language

$$
\begin{aligned}
e ::= &\ x \mid k \\
&\mid e_1\ e_2 \quad \mid \lambda(x{:}\tau).e \\
&\mid e\ \tau \quad\ \ \mid \Lambda(a{:}\kappa).e \\
&\mid \text{let bind in } e \\
&\mid \text{case } e \text{ of } \{\ alt_1 \ldots alt_n\ \}
\end{aligned}
$$

$$
\begin{aligned}
bind ::= &\ x{:}\tau{=}e \\
&\mid \ \text{rec}\ \{\ x_1{:}\tau_1{=}e_1 \ldots x_n{:}\tau_n{=}e_n\ \}
\end{aligned}
$$

$$alt := C\ (x_1{:}\tau_1) \ldots (x_n{:}\tau_n) \text{ -> } e \ \mid \ \text{DEFAULT -> } e$$

Core: GHC's intermediate language

```
data Expr
  = Var       Var
  | Lit       Literal
  | App       Expr Expr
  | Lam       Var Expr   -- Both term and type lambda
  | Let       Bind Expr
  | Case      Expr Var Type [(AltCon, [Var], Expr)]
  | Type      Type       -- Used for type application

data Var = Id    Name Type  -- Term variable
         | TyVar Name Kind  -- Type variable

data Type = TyVarTy  Var
          | LitTy    TyLit
          | AppTy    Type Type
          | TyConApp TyCon [Type]
          | FunTy    Type Type   -- Not really necy
          | ForAllTy Var Type
```

Core: GHC's intermediate language

```
data Expr
  = Var       Var
  | Lit       Literal
  | App       Expr Expr
  | Lam       Var Expr   -- Both term and type lambda
  | Let       Bind Expr
  | Case      Expr Var Type [(AltCon, [V          r)]
  | Type      Type       -- Used f             tion

data Var = Id    Nam                     iable
         | TyV                         ype variable

data                Var
          ty        TyLit
          | AppTy    Type Type
          | TyConApp TyCon [Type]
          | FunTy    Type Type   -- Not really necy
          | ForAllTy Var Type
```

22 years old and still only 15 constructors.
Bravo Girard & Reynolds!

What's good about System F

- In our presentation of System F, each variable occurrence is annotated with its type.
- Hence every term has a unique type

```
exprType :: Expr -> Type
exprType (Var v)   = varType v
exprType (Lam v a) = Arrow (varType v) (exprType a)
...more equations...
```

- `exprType` is pure; needs no "Gamma" argument
- Sharing of the `Var` means that the apparent duplication is not real

What's good about System F?

- Type checking (Lint) is fast and easy, because the rules are syntax-directed

The syntax of a term encodes its typing derivation

(fvar) r:Int->Bool |- r : Int -> Bool

(fvar) r:Int->Bool |- 4 : Int

(fapp) r:Int->Bool |- r 4 : Bool

(fabs) |- λr:(Int->Bool). r 4 : (Int -> Bool) -> Bool

Story so far

- Robust to transformations (ie if the term is well typed, then the transformed term is well typed):
  - beta reduction
  - inlining
  - floating lets outward or inward
  - case simplification
- Simple, pure

```
exprType :: Expr -> Type
```

- Type checking (Lint) is easy and fast

Adding GADTs

GADTs in Core?

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

f :: T a -> a -> Bool
f = Λa. λ(x:T a) (y:a).
    case x of
      T1 (z:Bool) -> let (v:Bool) = not y
                     in v && z
      T2 -> False
```

Problem 1
not :: Bool -> Bool
but
y::a

```
f :: T a -> a -> Bool
f = Λa. λ(x:T a) (y:a).
    let (v:Bool) = not y
    in case x of
      T1 (z:Bool) -> v && z
      T2 -> False
```

Problem 2
Floating the let seems well-scoped, but gives a bogus program

Solution to both problems: EVIDENCE

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

f :: T a -> a -> Bool
f = Λa. λ(x:T a) (y:a).
    case x of
      T1 (c:a~Bool) (z:Bool)
         -> let (v:Bool) = not (y ▷ c)
            in v && z
       -> False
```

Pattern matching on T1 brings into scope some EVIDENCE that (a=Bool)

We can USE the evidence to convert (y::a) to type Bool

c is an EVIDENCE VARIABLE

If e:τ and c: τ~σ, then (e ▷ c) : σ

```
T1 :: ∀a. (a~Bool) -> Bool -> T a
```

Evidence

```
T1 :: ∀a. (a~Bool) -> Bool -> T a
```

- Any application of T1 must supply evidence
  T1 σ e1 e2
  where e1 : (σ~Bool), e2 : Bool
- Here e1 is a value that denotes evidence that **σ=Bool**
- And any pattern match on T1 gives access to evidence
  case s of { T1 (c:σ~Bool) (y:Bool) -> ... }
  where s : T σ

System FC

$$
\begin{aligned}
e ::= {} & x \mid k \\
& \mid e_1\ e_2 \mid \lambda(x{:}\tau).e \\
& \mid e\ \tau \mid \Lambda(a{:}\kappa).e \\
& \mid \text{let } bind \text{ in } e \\
& \mid \text{case } e \text{ of } \{ alt_1 \ldots alt_n \} \\
& \mid e\ \gamma \mid \lambda(c{:}\tau_1{\sim}\tau_2).e \\
& \mid e \triangleright \gamma
\end{aligned}
$$

A coercion $\gamma{:}\tau_1{\sim}\tau_2$ is evidence that t1 and t2 are equivalent

Coercion abstraction and application

Type-safe cast

If $e{:}\tau$ and $\gamma{:}\ \tau{\sim}\sigma$, then $(e \triangleright \gamma) : \sigma$

The syntax of a term (again) encodes its typing derivation

Modifications to Core

```
data Expr
  = Var      Var
  | Lit      Literal
  | App      Expr Expr
  | Lam      Var Expr
  | Let      Bind Expr
  | Case     Expr Var Type [(AltCon, [Var], Expr)]
  | Type     Type
  | Coercion Coercion  -- Used for coercion apps
  | Cast     Expr Coercion    -- Type-safe cast

data Var = Id    Name Type   -- Term variable
         | TyVar Name Kind   -- Type variable
         | CoVar Name Type Type  -- Coercion var
```

Evidence terms

```
T1 :: ∀a. (a~Bool) -> Bool -> T a
```

- Consider the call:
  T1 Bool <Bool> True : T Bool
- Here <Bool> : Bool ~ Bool
  $\gamma ::= \langle\tau\rangle \mid \ldots$
- Can I call T1 Char γ True : T Char?
- No: that would need (γ : Char ~ Bool) and there are no such terms γ

Composing evidence terms

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

g :: T a -> Maybe a
g = Λa. λ(x:T a).
    case x of
      T1 (c:a~Bool) (z:Bool)
           -> Just a (z ▷ sym c)
      T2 -> Nothing
```

Have evidence c:a~Bool
Need evidence
sym c : Bool~a

γ ::= <τ> | sym γ | ...

- If $\gamma : \tau \sim \sigma$ then sym $\gamma : \sigma \sim \tau$

Composing evidence terms

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

g :: T a -> Maybe a
g = Λa. λ(x:T a).
    case x of
      T1 (c:a~Bool) (z:Bool)
          -> (Just Bool z) ▷ Maybe (sym c)
      T2 -> Nothing
```

Have evidence c:a~Bool
Need evidence
Maybe (sym c) : Maybe Bool ~ Maybe a

$$\gamma ::= \langle\tau\rangle \mid \mathrm{sym}\,\gamma \mid T\,\gamma_1 \ldots \gamma_n \mid \ldots$$

- If $\gamma_i : \tau_i \sim \sigma_i$
  then $T\,\gamma_1 \ldots \gamma_n : T\,\tau_1 \ldots \tau_n \sim T\,\sigma_1 \ldots \sigma_n$

Evidence terms

Coercion values

γ, δ	$::=$	x	Variables
	$\mid$	$C\ \overline{\gamma}$	Axiom application
	$\mid$	$\gamma_1\ \gamma_2$	Application
	$\mid$	$\langle\varphi\rangle$	Reflexivity
	$\mid$	$\gamma_1\ ;\ \gamma_2$	Transitivity
	$\mid$	$sym\ \gamma$	Symmetry
	$\mid$	$nth\ k\ \gamma$	Injectivity
	$\mid$	$\forall a{:}\eta\,.\,\gamma$	Polymorphic coercion
	$\mid$	$\gamma @ \varphi$	Instantiation

Cost model

- Coercions are **computationally irrelevant**
- Coercion abstractions, applications, and casts are erased at runtime

Bottom line

- Just like type abstraction/application, **evidence abstraction/application** provides a simple, elegant, consistent way to
  - express programs that use local type equalities
  - in a way that is fully robust to program transformation
  - and can be typechecked in an absolutely straightforward way
- Cost model: coercion abstractions, applications, and casts are erased at runtime

AXIOMS

newtypes

- Haskell

```
newtype Age = MkAge Int

bumpAge :: Age -> Int -> Age
bumpAge (MkAge a) n = MkAge (a+n)
```

- No danger of confusing Age with Int
- Type abstraction by limiting visibility of MkAge
- Cost model: Age and Int are represented the same way

In Core?

```
newtype Age = MkAge Int

bumpAge :: Age -> Int -> Age
bumpAge (MkAge a) n = MkAge (a+n)
```

```
axiom ageInt :: Age ~ Int

bumpAge :: Age -> Int -> Age
bumpAge = \(a:Age) (n:Int).
             (a ▷ ageInt + n) ▷ sym ageInt
```

- Newtype constructor/pattern matching turn into casts
- (New) Top-level axiom for equivalence between Age and Int
- Everything else as before

Parameterised newtypes

type GenericQ r = GQ (forall a. Data a => a -> r)

```
axiom axGQ r :: GenericQ r ~ ∀a. Data a => a -> r
```

- Axioms can be parameterised, of course
- No problem with having a polytype in s~t

Type functions

```
type family Add (a::Nat) (b::Nat) :: Nat

type instance Add Z     b = b
type instance Add (S a) b = S (Add a b)
```

```
axiom axAdd1 b   :: Add Z b ~ b
axiom axAdd2 a b :: Add (S a) b ~ S (Add a b)
```

- More about this on Saturday

OPTIMISING EVIDENCE

```
axiom ageInt :: Age ~ Int
```

Another worry

```
(λ(x:Int).x) 3  ==>  3  -- Beta reduction

(λ(x:Int).x) ▷ g) (3 ▷ sym ageInt)  ==>  ???
   where g :: (Int->Int) ~ (Age->Int)
```

- We do not want casts to interfere with optimisation
- And the very same issue comes up when proving the progress lemma

Decomposing evidence

```
g :: (σ1 -> σ2) ~ (τ1 -> τ2)
nth[1] g :: σ1 ~ τ1
nth[2] g :: σ2 ~ τ2
```

$$\frac{\Gamma \vdash^{co} \gamma : H\,\overline{\sigma} \sim_{\#} H\,\overline{\tau}}{\Gamma \vdash^{co} nth\; k\; \gamma : \sigma_k \sim_{\#} \tau_k}$$

$$(e_1 \triangleright g)\ e_2$$
$$\Longrightarrow$$
$$(e_1\ (e_2 \triangleright \mathtt{sym}\ (\mathtt{nth}[1]\ g))) \triangleright \mathtt{nth}[2]\ g$$

- Push the cast out of the way
- Something similar for (case (K e) ▷ g of ...)
- NB: consistency needed for progress lemma

```
axiom ageInt :: Age ~ Int
```

A worry

```
Assume g :: (Int->Int)~(Age->Int) = sym ageInt -> <Int>

(λ(x:Int).x) ▷ g) (3 ▷ sym ageInt)
==>
(λ(x:Int).x) ((3 ▷ sym ageInt) ▷ sym (nth[1] g))
   ▷ nth[2] g
```

A coercion built by composition

- All this pushing around just makes the coercions bigger! Compiler gets slower, debugging the compiler gets harder.
- Solution: rewrite the coercions to simpler form

```
nth[1] g
= nth[1] (sym ageInt -> <Int>)
= sym ageInt
```

Decomposition

```
nth[2] g
= <Int>
```

```
axiom ageInt :: Age ~ Int
```

A worry

```
Assume g :: (Int->Int)~(Age->Int) = sym ageInt -> <Int>

(λ(x:Int).x) ((3 ▷ sym ageInt) ▷ sym (nth[1] g))
   ▷ nth[2] g
==>
(λ(x:Int).x) ((3 ▷ sym ageInt) ▷ sym (sym ageInt))
   ▷ <Int>
```

- More simplifications

```
sym (sym g) = g

e ▷ g1 ▷ g2 = e ▷ (g1;g2)
e ▷ <t> = e
```

```
axiom ageInt :: Age ~ Int
```

A worry

```
Assume g :: (Int->Int)~(Age->Int) = sym ageInt -> <Int>

(λ(x:Int).x) ((3 ▷ sym ageInt) ▷ sym (sym ageInt))
   ▷ <Int>
==>
(λ(x:Int).x) (3 ▷ (sym ageInt ; ageInt))
```

- More simplifications

```
sym g ; g = <t>   -- g :: s ~ t
```

A worry (fixed)

```
Assume g :: (Int->Int)~(Age->Int) = sym ageInt -> <Int>

(λ(x:Int).x) (3 ▷ (sym ageInt ; ageInt))
==>
(λ(x:Int).x) 3    -- Hurrah
```

- See paper in proceedings for a terminating (albeit not confluent) rewrite system to optimise coercions
- Lack of confluence doesn't matter; it's just to keep the compiler from running out of space/time

CONSISTENCY

A worry

- What if you have stupid top-level axioms?

```
axiom bogus :: Int ~ Bool
```

- Then "well typed programs don't *go wrong*" would be out of the window
- Standard solution: insist that the axioms are **consistent**:

Consistency

If $g : T_1\ \tau_1 \sim T_2\ \tau_2$,
where T_1, T_2 are data types,
then $T_1 = T_2$

- But how to guarantee consistency of axioms? Hard to check, so instead guarantee by construction.

Guaranteeing consistency

Consistency

If $g : T_1\ \tau_1 \sim T_2\ \tau_2$,
where T_1, T_2 are data types,
then $T_1 = T_2$

- Axioms in Core are not freely written by user; they are generated from Haskell source code
- e.g. Newtypes: the axioms are never inconsistent

```
newtype Age = MkAge Int
➔    axiom ageInt :: Age ~ Int
     -- Age is not a data type
```

Guaranteeing consistency

Consistency

If $g : T_1\ \tau_1 \sim T_2\ \tau_2$,
where T_1, T_2 are data types,
then $T_1 = T_2$

- What about type functions?

```
type instance F Int y = Bool
type instance F x Int = Char
```

- These generate axioms that would allow us to prove
  Bool ~ F Int Int ~ Char
- Obvious solution: prohibit overlap.
- Two equations overlap if their LHSs unify.

Guaranteeing consistency

Consistency

If $g : T_1\ \tau_1 \sim T_2\ \tau_2$,
where T_1, T_2 are data types,
then $T_1 = T_2$

- What about type functions?

```
type instance F Int y = Bool
type instance F x Int = Char
```

- These generate axioms that would allow us to prove
  Bool ~ F Int In… …ar
- Obvious solu… …verlap.
- Two equati… …s overlap if th… …LHSs unify.

Wrong

Overlap is tricky

```
type instance Loop = [Loop]        -- (A)

type instance F a a   = Bool       -- (B)
type instance F b [b] = Char       -- (C)
```

- The LHSs of the F equations don't unify
- But

```
F Loop Loop ~ Bool      -- By (B)

F Loop Loop
~ F Loop [Loop]        -- By (A)
~ Char                  -- By (C)
```

- Eeek! The combination of non-left-linear LHSs and non-termination type families is tricky. Very tricky. Actually very tricky indeed.

Conjecture

- All is well if replace "unify" by "unify$_\infty$". Roughly, unify allowing infinite types in the solving substitution.
- Then unify$_\infty$((a,a),(b,[b])) succeeds, and hence these two equations overlap, and are rejected

```
type instance F a a   = Bool      -- (B)
type instance F b [b] = Char      -- (C)
```

Conjecture
If all the LHSs of axioms don't overlap using unify_∞, then the axioms are consistent.

- We think it's true
- GHC uses this criterion
- But we have not been able to prove it
- Obvious approach: treat axioms as left-to-right rewrite rules, and prove confluence
- Alas: if rules are (a) non-left-linear and (b) non-terminating, confluence doesn't hold!

Confluence does not hold [Klopp]

```
type instance A = C A
type instance C x = D x (C x)
type instance D x x = Int

(1) A →  C A → D A (C A) → D (C A) (C A) → Int
(2) A →  C A → C Int

But C Int does not reduce to Int!
```

- Notice that this counter-example depends on
  - non-linear left-hand sides
  - non-terminating rewrite rules

ROLES

Coercing newtypes

```
data Maybe a = Nothing | Just a

newtype Age = Int    -- axAge :: Age ~ Int
```

```
f :: Maybe Age -> Maybe Int
f Nothing  = Nothing
f (Just x) = Just (x ▷ axAge)
```

or

```
f :: Maybe Age -> Maybe Int
f xs = xs ▷ Maybe axAge
```

Confluence does not hold [Klopp]

```
newtype Age = Int                  -- axAge :: Age ~ Int

type family F a :: *
type instance F Age = Bool         -- axF1 :: F Age ~ Bool
type instance F Int = Char         -- asF2 :: F Int ~ Char

data T a = MkT (F a)
```

```
f :: T Age -> T Int
f xs = xs ▷ T axAge
```

Confluence does not hold [Klopp]

```
newtype Age = Int                    -- axAge :: Age ~ Int

type family F a :: *
type instance F Age = Bool   -- axF1 :: F Age ~ Bool
type instance F Int = Char   -- asF2 :: F Int ~ Char

data T a = MkT (F a)
```

```
f :: T Age -> T Int
f xs = xs ▷ T axAge
```

```
bad :: Bool -> Char
bad b = case y of { MkT fi -> fi ▷ axF2 }
  where
    x :: T Age = MkT (b ▷ sym axF1))
    y :: T Int = f x
```

Key ideas [POPL11]

```
newtype Age = Int                -- axAge :: Age ~R Int

type instance F Age = Bool       -- axF1 :: F Age ~N Bool
type instance F Int = Char       -- asF2 :: F Int ~N Char
```

- Two different equalities:
  - **representational** equality (R)
  - **nominal** equality (N)
- Nominal implies representational, but vice versa; nominal makes more distinctions
- Cast ($e \triangleright g$) takes a representational equality

Key ideas [POPL11]

```
data Maybe a = Nothing | Just a
data W a = MkT (F a)
data K a = MkP Int
```

- Three different argument "roles" for type constructors:
  - Maybe uses its argument parametrically (role R)
  - W dispatches on its argument (role N)
  - K ignores its argument (role P)
- To get $(T\ s \sim_N T\ t)$, we need $(s \sim_N t)$
- To get $(T\ s \sim_R T\ t)$, we need
  - $s \sim_R t$ for T=Maybe
  - $s \sim_N t$ for T=W
  - nothing for T=K

WRAP UP

Wrap up

- Many more aspects not covered in this talk
  - "Closed" type families with non-linear patterns, and proving consistency thereof

```
type family Eq a b where
  Eq a a = True
  Eq a b = False
```

POPL submission

  - Heterogeneous equalities; coercions at the type level
- A more complicated and interesting design space than we had at first imagined

Wrap up

- Main "new" idea: programs manipulate **evidence** along with types and values
- This single idea in Core explains multiple source-language concepts:
  - GADTs
  - Newtypes
  - Type and data families (both open and closed)
- Typed evidence-manipulating calculi perhaps worthy of more study
  - E.g. McBride/Gundry: lambda-cube-like idea applied to types/terms/evidence
  - Open problems of establishing consistent axiom sets (e.g. non-linear patterns + non-terminating functions… help!)