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.

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

Figure 1: Lines of code in GHC, past and present.
GHC is big and old

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main</td>
<td>997</td>
<td>11,150</td>
<td>11.2</td>
</tr>
<tr>
<td>Parser</td>
<td>1,055</td>
<td>4,098</td>
<td>3.9</td>
</tr>
<tr>
<td>Renamer</td>
<td>2,828</td>
<td>4,630</td>
<td>1.6</td>
</tr>
<tr>
<td>Type checking</td>
<td>3,352</td>
<td>24,097</td>
<td>7.2</td>
</tr>
<tr>
<td>Desugaring</td>
<td>1,381</td>
<td>7,091</td>
<td>5.1</td>
</tr>
<tr>
<td>Core transformations</td>
<td>1,631</td>
<td>7,296</td>
<td>3.9</td>
</tr>
<tr>
<td>STG transformations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Utilities</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3,111</td>
<td>2,692</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>1,989</td>
<td>7,878</td>
<td>3.96</td>
</tr>
<tr>
<td>Profiling</td>
<td>191</td>
<td>367</td>
<td>1.92</td>
</tr>
<tr>
<td>Compiler Total</td>
<td>28,275</td>
<td>139,955</td>
<td>4.9</td>
</tr>
<tr>
<td>Runtime System</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All C and C++ code</td>
<td>43,865</td>
<td>48,450</td>
<td>1.10</td>
</tr>
</tbody>
</table>

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

Question: how to stay sane?
Haskell
Massive language
Hundreds of pages of user manual
Syntax has dozens of data types
100+ constructors

How GHC works

Source language

Typed intermediate language

Core
3 types, 15 constructors

Rest of GHC

Haskell

Typecheck

Desugar
<table>
<thead>
<tr>
<th>Haskell</th>
<th>Core (the typed IL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big</td>
<td>Small</td>
</tr>
<tr>
<td>Implicitly typed</td>
<td>Explicitly typed</td>
</tr>
<tr>
<td>Binders typically un-annotated</td>
<td>Every binder is type-annotated</td>
</tr>
<tr>
<td>(x. x &amp;&amp; y)</td>
<td>((x:\text{Bool}). x &amp;&amp; y)</td>
</tr>
<tr>
<td>Type inference (complex, slow)</td>
<td>Type checking (simple, fast)</td>
</tr>
<tr>
<td>Complicated to specify just which programs will type-check</td>
<td>Very simple to specify just which programs are type-correct</td>
</tr>
<tr>
<td>Ad-hoc restrictions to make inference feasible</td>
<td>Very expressive indeed; simple, uniform</td>
</tr>
</tbody>
</table>
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, optimisation, and code generation is more straightforward.

- 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.

- Design of Core is a powerful sanity check on crazy type-systems on the 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
compose :: (b -> c) -> (a -> b) -> a -> c
compose = \f : b -> c. \g : a -> b. \x : 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...
compose :: (b -> c) -> (a -> b) -> a -> c
compose = \f:b->c. \g:a->b. \x:a. 
    let tmp:b = g x
    in f tmp

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

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

- Now the type annotations are wrong
- Solution: learn from Girard and Reynolds!
compose :: \forall abc. (b -> c) -> (a -> b) -> a -> c
compose = \Lambda abc. \lambda f: b -> c. \lambda g: a -> b. \lambda x: a.
    let tmp: b = g x
    in f tmp

- Idea: an explicit (big) lambda binds type variables
compose :: ∀abc. (b->c) -> (a->b) -> a -> c
  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
    ```haskell
data Maybe a = Nothing | Just a
```
  - Data constructors in terms
    ```haskell
λx:Int. Just (Just x)
```
  - Case expressions
    ```haskell
case x of { Nothing -> 0; Just x -> x+1 }
```
  - Let expressions
    ```haskell
let x:Int = 4 in x+x
```
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

- 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

\[
\begin{align*}
\text{e} & ::= \text{x} \mid \text{k} \\
& \mid \text{e}_1 \text{e}_2 \mid \lambda (\text{x}:\tau).\text{e} \\
& \mid \text{e} \tau \mid \Lambda (\text{a}:\kappa).\text{e} \\
& \mid \text{let bind in e} \\
& \mid \text{case e of \{} \text{alt}_1 .. \text{alt}_n \} \\
\end{align*}
\]

\[
\begin{align*}
\text{bind} & ::= \text{x}:\tau=e \\
& \mid \text{rec \{} \text{x}_1:\tau_1=e_1 .. \text{x}_n:\tau_n=e_n \} \\
\end{align*}
\]

\[
\begin{align*}
\text{alt} & ::= \text{C} (\text{x}_1:\tau_1) .. (\text{x}_n:\tau_n) \rightarrow \text{e} \mid \text{DEFAULT} \rightarrow \text{e}
\end{align*}
\]
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

```haskell
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 function declaration
```

```haskell
data Var = Id Name Type
    | TyVar Name Kind
    | LitTy 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

\[
\begin{align*}
\text{exprType} & : \text{Expr} \rightarrow \text{Type} \\
\text{exprType} \ (\text{Var} \ v) & = \text{varType} \ v \\
\text{exprType} \ (\text{Lam} \ v \ a) & = \text{Arrow} \ (\text{varType} \ v) \ (\text{exprType} \ a)
\end{align*}
\]

...more equations...

- \text{exprType} is pure; needs no “Gamma” argument
- Sharing of the \text{Var} means that the apparent duplication is not real
Type checking (Lint) is fast and easy, because the rules are syntax-directed.

The syntax of a term encodes its typing derivation.

- \( r: \text{Int} \to \text{Bool} \vdash r : \text{Int} \to \text{Bool} \)
- \( r: \text{Int} \to \text{Bool} \vdash 4 : \text{Int} \)
- \( r: \text{Int} \to \text{Bool} \vdash r\ 4 : \text{Bool} \)
- \( \vdash \lambda r: (\text{Int} \to \text{Bool}). \ r\ 4 : (\text{Int} \to \text{Bool}) \to \text{Bool} \)
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

Type checking (Lint) is easy and fast

exprType :: Expr -> Type
ADDING GADTS
data T a where
   T1 :: Bool -> T Bool
   T2 :: T a

f :: T a -> a -> Bool
f = \a. \(x:T a) (y:a).
         let (v:Bool) = \a. not y
           in v && z
   T2 -> False

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

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

Problem 2
Floating the let seems well-scoped, but gives a bogus program
Solution to both problems: EVIDENCE

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

f :: T a -> a -> Bool
f = \a. \(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**

Thus:

```
T1 :: ∀a. (a~Bool) -> Bool -> T a
```
Any application of $T1$ must supply evidence $T1 \sigma e1 e2$ where $e1 : (\sigma \sim \text{Bool})$, $e2 : \text{Bool}$

Here $e1$ is a value that denotes evidence that $\sigma = \text{Bool}$

And any pattern match on $T1$ gives access to evidence

```
  case s of { T1 (c:\sigma \sim \text{Bool}) (y:\text{Bool}) \rightarrow ... } 
  where s : T \sigma
```
System FC

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

A coercion $\gamma : \tau_1 \sim \tau_2$ is evidence that $\tau_1$ and $\tau_2$ are equivalent

Coercion abstraction and application

Type-safe cast
If $e : \tau$ and $\gamma : \tau \sim \sigma$, then $(e \triangleright \gamma) : \sigma$
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
Consider the call:

\[ T1 \text{ Bool } <\text{Bool}> \text{ True : } T \text{ Bool} \]

Here \(<\text{Bool}> : \text{Bool} \sim \text{Bool}\)

\[ \gamma ::= <\tau> | ... \]

Can I call \( T1 \text{ Char } \gamma \text{ True : } T \text{ Char} \)?

No: that would need \((\gamma : \text{Char} \sim \text{Bool})\) and there are no such terms \(\gamma\)
Composing evidence terms

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

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

\(\gamma ::= \langle \tau \rangle \ | \ sym \ \gamma \ | \ ...\)

- 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 = \a. \(x:T a) .
      \ case x of
        T1 (c:a~Bool) (z:Bool)
        -> (Just Bool z) \\> Maybe (sym c)
        T2 -> Nothing

\( \gamma ::= \langle \tau \rangle \mid \text{sym} \gamma \mid T \gamma_1 \ldots \gamma_n \mid \ldots \)

- If \( \gamma_i : \tau_i \sim \sigma_i \)
  then \( T \gamma_1 \ldots \gamma_n : T \tau_1 \ldots \tau_n \sim T \sigma_1 \ldots \sigma_n \)
### Coercion values

<table>
<thead>
<tr>
<th>$\gamma, \delta$</th>
<th>$x$</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C \overline{\gamma}$</td>
<td>Application</td>
<td></td>
</tr>
<tr>
<td>$\gamma_1 \gamma_2$</td>
<td>Reflexivity</td>
<td></td>
</tr>
<tr>
<td>$\langle \varphi \rangle$</td>
<td>Transitivity</td>
<td></td>
</tr>
<tr>
<td>$\text{sym} \gamma$</td>
<td>Symmetry</td>
<td></td>
</tr>
<tr>
<td>$\mathit{nth} k \gamma$</td>
<td>Injectivity</td>
<td></td>
</tr>
<tr>
<td>$\forall a: \eta . \gamma$</td>
<td>Polymorphic coercion</td>
<td></td>
</tr>
<tr>
<td>$\gamma@\varphi$</td>
<td>Instantiation</td>
<td></td>
</tr>
</tbody>
</table>
Cost model

- Coercions are *computationally irrelevant*
- Coercion abstractions, applications, and casts are erased at runtime
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
  
  ```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
newtype Age = MkAge Int

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

axiom ageInt :: Age ~ Int

bumpAge :: Age -> Int -> Age
bumpAge = \(a:\text{Age}) \ (n:\text{Int}).\)

\(a \triangleright \text{ageInt} + n) \triangleright \text{sym ageInt}

- Newtype constructor/pattern matching turn into casts
- (New) Top-level axiom for equivalence between Age and Int
- Everything else as before
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
We do not want casts to interfere with optimisation

And the very same issue comes up when proving the progress lemma
Decomposing evidence

Push the cast out of the way

Something similar for (case (K e) ▷ g of ... )

NB: consistency needed for progress lemma
A worry

All this pushing around just makes the coercions bigger! Compiler gets slower, debugging the compiler gets harder.

Solution: rewrite the coercions to simpler form

```
axiom ageInt :: Age ~ Int

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

(\(x:\text{Int})\cdot x) \circ g) (3 \circ \text{sym ageInt})
==>
(\(x:\text{Int})\cdot ((3 \circ \text{sym ageInt}) \circ \text{sym (nth[1] g)})
\circ \text{nth[2] g}
```

- A coercion built by composition

- Decomposition

\[
\text{nth[1] g} \\
= \text{nth[1]} (\text{sym ageInt} \rightarrow <\text{Int}>) \\
= \text{sym ageInt}
\]

\[
\text{nth[2] g} \\
= <\text{Int}>
\]
axiom ageInt :: Age ~ Int

A worry

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

(\(x:\text{Int})\cdot x) ((3 \triangleright \text{sym ageInt}) \triangleright \text{sym (nth[1] g)})
\triangleright \text{nth[2] g}

==>
(\(x:\text{Int})\cdot x) ((3 \triangleright \text{sym ageInt}) \triangleright \text{sym (sym ageInt)})
\triangleright <\text{Int}>

- More simplifications

\text{sym (sym g) = g}

e \triangleright g1 \triangleright g2 = e \triangleright (g1;g2)
e \triangleright <t> = e
A worry

Assume $g :: (\text{Int} \rightarrow \text{Int}) \sim (\text{Age} \rightarrow \text{Int}) = \text{sym} \ age\text{Int} \rightarrow <\text{Int}>

$(\lambda(x: \text{Int}). x) ((3 \triangleright \text{sym} \ age\text{Int}) \triangleright \text{sym} \ (\text{sym} \ age\text{Int}))$

$\triangleright <\text{Int}>

==>

$(\lambda(x: \text{Int}). x) (3 \triangleright (\text{sym} \ age\text{Int}; \ age\text{Int}))$

- More simplifications

$sym \ g ; g = <t> -- g :: s \sim t$
A worry (fixed)

Assume \( g : (\text{Int} \to \text{Int}) \sim (\text{Age} \to \text{Int}) = \text{sym ageInt} \to <\text{Int}> \)

\[(\lambda(x: \text{Int}).x) \ (3 \triangleright (\text{sym ageInt} ; \text{ageInt}))\]

\(\Rightarrow\)

\[(\lambda(x: \text{Int}).x) \ 3 \quad -- \text{Hurrah}\]

- 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?
  
  ```haskell
  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:

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

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
```
What about type functions?

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

These generate axioms that would allow us to prove:

\[ \text{Bool} \sim F \text{Int Int} \sim \text{Char} \]

Obvious solution: prohibit overlap.

Two equations overlap if their LHSs unify.
What about type functions?

These generate axioms that would allow us to prove

\[ \text{Bool} \sim F \text{Int Int Int} \]

Obvious solution: prohibit overlap.

Wrong
The LHSs of the F equations don’t unify

But

Eek! The combination of non-left-linear LHSs and non-termination type families is tricky. Very tricky. Actually very 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\(_\infty\), 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]

- Notice that this counter-example depends on
  - non-linear left-hand sides
  - non-terminating rewrite rules
ROLES
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 \(\bowtie\) axAge)

or

f :: Maybe Age -> Maybe Int
f xs = xs \(\bowtie\) Maybe axAge
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]

- Two different equalities:
  - *representational* equality \((R)\)
  - *nominal* equality \((N)\)

- Nominal implies representational, but vice versa; nominal makes more distinctions

- \(\text{Cast } (e \triangleright g)\) takes a representational equality

```haskell
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
```
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 \(T=\text{Maybe}\)
- \(s \sim_N t\) 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
    
    ```haskell
    type family Eq a b where
    Eq a a = True
    Eq a b = False
    ```
    
  - Heterogeneous equalities; coercions at the type level

- A more complicated and interesting design space than we had at first imagined
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!)