Type Checking with Open Type Functions

Tom Schrijvers, Simon Peyton Jones,
Manuel Chakravarty, Martin Sulzmann

ICFP 2008

Our problem

```
class Collects c where
  insert::<element of c> -> c -> c
```

`<element of c>` follows from `c`

c	element of c
BitSet	Char
[a]	a
Tree a	a

What we want

```
class Collects c where
   type Elem c :: *
   insert :: Elem c -> c -> c

instance Collects BitSet
   where type Elem BitSet = Char

instance Collects [a]
   where type Elem [a] = a
```

Associated
type synonym
[Chakravarty 2005]

Elem is a Type Family

Stand-alone syntax

c **indexes** the type family

```
type family Elem c

type instance Elem BitSet = Char
type instance Elem [a]    = a
```

Type families are **open:** you can add instances anytime

Instances should be:
- **confluent**
- **terminating**

How it used to be

Functional dependency [Jones 2000]

```
class Collects c e | c -> e
  where insert :: e -> c -> c

instance Collects BitSet Char
  where ...
instance Collects [a] a
  where ...
```

Type-Level Functions

```
type family Add a b
```

No type class involved!

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

```
app :: List k -> List l
    -> List (Add k l)
```

Functional Programming at the level of types

Local Constraints

```
insx :: (Collects c,
         Elem c ~ Char)
      =>  c -> c
insx c = insert 'x' c
```

equality constraint

Should work for any collection c
whose elements are Chars

What we need

Type checking
for all this

Type checking is hard

Given

- E_t: top-level equations, e.g.

  `forall x. Elem [x] ~ x`
- E_g: local equations, e.g. `Elem a ~ Char`
- E_w: wanted equations, e.g.

  `Elem (Elem [a]) ~ Char`

Find a proof for $E_t, E_g \vdash E_w$

Simple Case

$E_t \vdash E_w$

- No local constraints
- Easy: for s ~ t
  - Use E_t as a left-to-right rewrite system
  - normalize s and t
  - Check for syntactic equality
- E.g.

1. `Elem BitSet ~ Elem [Char]`
2. `Char ~ Char`

Why is it hard with E_g?

- E_g not a **terminating** rewrite system
  - Not oriented
  - LHS not in constructor form
- May **diverge**: `F a ~ G (F a)`
- May **loop**:

```
F Int ~ F (G Int)
G Int ~ Int          ⊢ F Int ~ Int
```

Why is it hard?(2)

- Even if E_t and E_g are terminating, then $E_t + E_g$ rewrite may not be.

  e.g.

  E_t = { `F Int ~ F (G Int)` }

  E_g = { `G Int ~ Int` }

Our Solution

$E_t, E_g \vdash E_w$

- **Complete** E_g wrt E_t, giving E_g'

Hard

- Now $E_t + E_{g'}$ is an equivalent, terminating and confluent TRS.
- Decide $s \sim t$ as before:
  - normalize s and t wrt. $E_t + E_{g'}$
  - check for syntactic equality

Easy

Simple Example (1)

```
Eg = { G Int ~ F (G Int),
       F(G Int) ~ Int }
```

- **substitute** 2nd in 1st:

```
{ G Int ~ Int,
  F (G Int) ~ Int }
```

Completion

- **substitute** 1st in 2nd

```
Eg' = { G Int~Int, F Int~Int }
```

More than substitution, see paper

Simple Example (2)

- $E_g' = \{$ `G Int~Int,F Int~Int` $\}$
- To check $E_w = \{$ `G (F Int) ~ Int` $\}$
  - Rewrite `G (F Int)`
    $\Rightarrow$ `G Int`
    $\Rightarrow$ `Int`
  - See that reduced LHS and RHS are syntactically equal

Properties

- Our type checking algorithm is *sound, complete and terminating* given sufficiently strong restrictions on the top-level equations.
- These **restrictions** are pretty drastic.

The Restrictions

$$F\ t_1 \ldots t_n = r$$

- $t_1 \ldots t_n$ contain no type functions
- LHSs do not overlap
- r contains no type function, or
- r is G $s_1 \ldots s_m$:
  - $s_1 \ldots s_m$ contain no type functions
  - $\text{size}(s_1 \ldots s_m) < \text{size}(t_1 \ldots t_n)$
  - RHS has no more occurrences of schema variable than LHS

confluence

termination

based on Functional Dependency restrictions

Relaxed Restrictions

$$F\ t_1 \ldots t_n = r$$

- $t_1 \ldots t_n$ contain no type functions
- LHSs do not overlap
- for each $G\ s_1 \ldots s_m$ in r:
  - $s_1 \ldots s_m$ contain no type functions
  - $size(s_1 \ldots s_m) < size(t_1 \ldots t_n)$
- RHS has no more occurrences of schema variable than LHS

confluence

termination

more liberal than
Functional Dependencies

trade-off:
- either termination
- or completeness

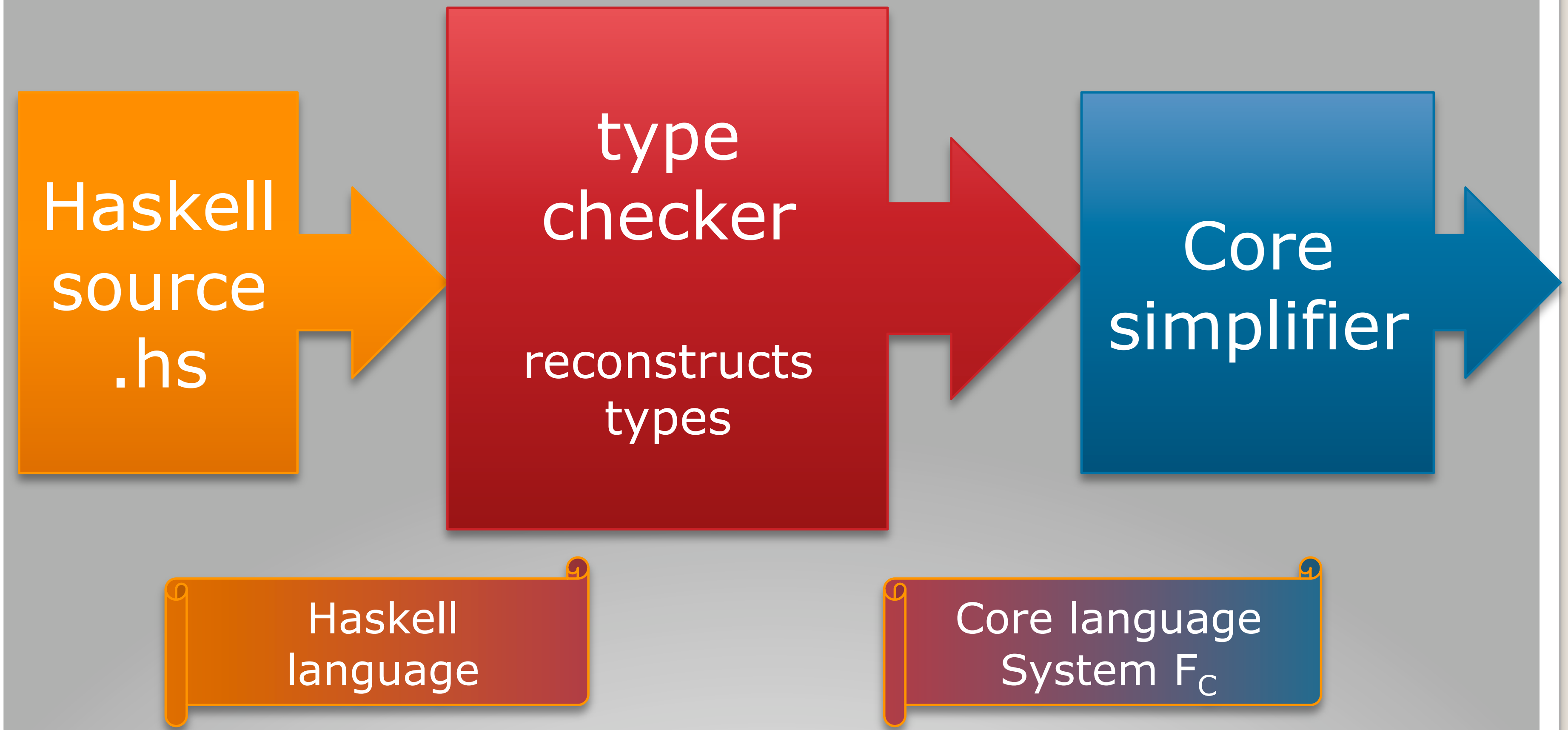
Type Famillies in Practice

Additional complications:

see paper

- Dealing with ***unification variables***
- **Inferring types** as well as checking
  e.g., function without signature
- Generation of **evidence**

Fit it into GHC



System F_C

[Sulzmann 2007]

System F with **equality coercions**

- coercion: evidence for non-syntactic type equality
- necessary for GADTs
- also for type families

see paper

our type checker generates coercions for all wanted equations

missing for Functional Dependencies

Summary

Type checking for type families

- **completion** of local equations
- **trade-off** between termination and completeness
- **evidence** and other complications

Hasn't this been done?

- **Vs. Congruence closure [Nelson'80]**
  - unification variables, also **[Tiwari'00]**
  - schema variables, also **[Beckert'94]**
  - evidence, also **[Nieuwenhuis'05]**
  - no completion of top-level equations
- **Vs.Functional Dependencies [Jones'00]**
  - liberated from type classes
  - evidence generation

Current Situation

Our algorithm

- has been ***simplified significantly*** since delivering the IFCP paper
  - same basic idea
  - but more aggressive *"flattening"*
  - many fewer rules
- is ***implemented***
- already has ***lots of applications***

GHC 6.10

Work in Progress

1. **Unified algorithm** for type classes and functions (see ICFP poster)
2. ***Invariants*** that must be satisfied by type function instances

```
forall x y. (Nat x, Nat y) =>
        Add x y = Add y x
```

Thank You!