

Towards Open Type Functions for Haskell

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

K.U.Leuven, National University of Singapore,
Microsoft Research Cambridge, University of New South Wales

presented at IFL 2007, Freiburg, Germany



- 1 Haskell in 5 (+1) slides
- 2 Informal Overview
- 3 Type Checking
 - Type Checking
 - Problems
 - The Algorithm
 - Type-Directed Compilation
- 4 Type Functions vs. Functional Dependencies
 - Problem
 - Expressivity
 - Secondary Criteria
- 5 Conclusion



- 1 Haskell in 5 (+1) slides
- 2 Informal Overview
- 3 Type Checking
 - Type Checking
 - Problems
 - The Algorithm
 - Type-Directed Compilation
- 4 Type Functions vs. Functional Dependencies
 - Problem
 - Expressivity
 - Secondary Criteria
- 5 Conclusion

Haskell λ

A Purely Functional Language

- ▶ Functional Programming (FP)
- ▶ pure: no side effects / no destructive update
- ▶ call-by-need evaluation strategy, *lazy* evaluation
- ▶ designed by committee, vibrant community, kicking off **now**

```
qsort []      = []
qsort (x:xs) = qsort (filter (< x) xs)
               ++ [x] ++
               qsort (filter (>= x) xs)
```

Haskell's Type System

5/39

- ▶ strong and static typing
- ▶ type inference: type signatures are optional
- ▶ algebraic data types (ADTs)

```
Integer Functor Ord Char
Either          Monad
Bool           Enum
Int            [...]
->             Eq
Num           Read
Bounded       (,-)
Integral ()   IO Show
Maybe String Ratio Float
```

I prefer the strong, static type.

```
-- polymorphic ADT
data List e = Nil | Cons e (List e)

mylist :: List Char
mylist = Cons 't' (Cons 'o' (Cons 'm' Nil))
```

Type Classes

6/39

- ▶ extensible overloading
- ▶ not OO-style subtyping!
- ▶ Mercury, C++0x concepts

HASKELL



A pure functional language
with Class!

```

class Num a where
  (+)      :: a -> a -> a

instance Num Int where ...
instance Num Float where ...
instance Num a => Num (List a) where ...

double :: Num a => a -> a
double x = x + x
  
```

Advanced ADTs – Part 1: Phantom Types

7/39

```
-- phantom type
data MoneyAmount currency = Amt Int
instance Num (MoneyAmount c) where
  (Amt x) + (Amt y) = Amt (x + y)
-- uninhabited types
data Euro ; data Dollar

mybalance :: MoneyAmount Euro
mybalance = Amt 0
yourbalance :: MoneyAmount Dollar
yourbalance = Amt 1000

total = mybalance + yourbalance -- TYPE ERROR!
```

```
-- empty types
data Z ; data S nat

-- GADT: length-indexed list
data List e len where
  Nil    :: List e Z
  Cons   :: e -> List e len -> List e (S len)

mylist :: List Char (S (S (S Z)))
mylist = Cons 't' (Cons 'o' (Cons 'm' Nil))

zip :: List a l -> List b l -> List (a,b) l
> zip mylist (Cons 7 Nil) -- TYPE ERROR
```

Haskell Type System: Summary

9/39

General Tendency

- ▶ increasingly expressive type system:
programming with types
- ▶ encode more and more properties of values in their types:
(value-)dependent typing
- ▶ design space: any two of
 - ▶ expressivity vs. restricted type language
 - ▶ decidability vs. non-termination
 - ▶ automatic type checking vs. hand-written proofs

Haskell: conservative, but pushing the expressivity boundary!



- 1 Haskell in 5 (+1) slides
- 2 Informal Overview**
- 3 Type Checking
 - Type Checking
 - Problems
 - The Algorithm
 - Type-Directed Compilation
- 4 Type Functions vs. Functional Dependencies
 - Problem
 - Expressivity
 - Secondary Criteria
- 5 Conclusion

Functional Dependencies:

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

```
instance Eq e => Collects [e] e where  
    insert = ...
```

```
instance Eq e => Collects (Trie e) [e] where  
    insert = ...
```

```
instance Collects BitSet Char where  
    insert = ...
```

Functional Dependencies:

```
class Add a b c | a b -> c  
  -- No Operations
```

```
instance Add a b c =>  
  Add (S a) b (S c)
```

```
instance Add Z b b
```

```
append :: Add m n o => List e m -> List e n ->  
  List e o
```

Associated Type Synonyms:

```
class Collects c where  
    type Elem c  
    insert :: c -> Elem c -> c  
  
instance Eq e => Collects [e] where  
    type Elem [e] = e  
    insert = ...  
  
instance Collects BitSet where  
    type Elem BitSet = Char  
    insert = ...
```

Elem is a **type family**:

```
type family Elem c

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

- ▶ type families are *open*: add new instances any time
- ▶ instances should be *confluent* and *terminating*

Add is a **type family**:

```
type family Add a b

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

append :: List e m -> List e n ->
        List e (Add m n)
```

Intermezzo : Type Families are open

16/39

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

Can I show

$$\text{Add } x \ y \sim Z \Rightarrow x \sim Z \wedge y \sim Z$$

?

Intermezzo : Type Families are open

16/39

```
type family Add a b
```

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

```
type instance Add Z      b = b
```

Can I show

$$\text{Add } x \ y \sim Z \Rightarrow x \sim Z \wedge y \sim Z$$

? **No**, not modular/incremental, e.g. `Add Foo b = Z`



- 1 Haskell in 5 (+1) slides
- 2 Informal Overview
- 3 Type Checking**
 - Type Checking
 - Problems
 - The Algorithm
 - Type-Directed Compilation
- 4 Type Functions vs. Functional Dependencies
 - Problem
 - Expressivity
 - Secondary Criteria
- 5 Conclusion

Type Checking Easy?

18/39

Type checking is straightforward:

- ▶ unification performs *rewriting* using the top-level equations

$$\begin{array}{l} \text{Elem BitSet} \sim \text{Char} \\ \rightsquigarrow \quad \text{Char} \sim \text{Char} \end{array}$$

using **type family** Elem BitSet = Char

Type Checking Easy?

18/39

Type checking is straightforward:

- ▶ unification performs *rewriting* using the top-level equations

$$\begin{array}{l} \text{Elem BitSet} \sim \text{Char} \\ \rightsquigarrow \quad \text{Char} \sim \text{Char} \end{array}$$

using **type family** `Elem BitSet = Char`

- ▶ rewrite may *suspend* on unification variable x , and continue when a is known

$$\begin{array}{l} \underline{\text{Elem } x \sim \text{Char}, x \sim \text{BitSet}} \\ \rightsquigarrow \quad \text{Elem } x \sim \text{Char}, \underline{x \sim \text{BitSet}} \\ \rightsquigarrow \quad \underline{\text{Elem } \text{BitSet} \sim \text{Char}} \\ \rightsquigarrow \quad \underline{\text{Char} \sim \text{Char}} \end{array}$$

Problem: local constraints

19/39

Should work for any collection whose elements are **Chars**:

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

- ▶ locally **given** equation: $\text{Elem } c \sim \text{Char}$

Problem: local constraints

20/39

Local equations also arise from GADT pattern matches:

```
data Eq a b where  
  EQ :: forall a . EQ a a  
  
f :: Collects c => Eq (Elem c) Char -> c -> c  
f e c = case e of  
      EQ -> insert c 'x'
```

Problem: local constraints

20/39

Local equations also arise from GADT pattern matches:

```
data Eq a b where  
  EQ :: forall a b. a ~ b => Eq a b  
  
f :: Collects c => Eq (Elem c) Char -> c -> c  
f e c = case e of  
      EQ -> insert c 'x'
```

- ▶ pattern match brings `Elem c ~ Char` into scope

Type Checking is **Hard**!

21/39

Given

- ▶ E_t , the (quantified) top-level equations (e.g. $\forall a.\text{Elem } [e] \sim e$)
- ▶ E_g , a set of local equations (e.g. $\text{Elem } c \sim \text{Char}$)
- ▶ E_w , a set of wanted equations (e.g. $[\text{Char}] \sim [\text{Elem } c]$)

find a proof for

$$E_t, E_g \models E_w$$

Why it is hard

22/39

E_g is not a strongly-normalising rewrite system:

- ▶ non-confluent (dead code), e.g.

$$\text{Elem [Int]} \sim [\text{Int}] \text{ wrt. } \forall e. \text{Elem [e]} \sim e$$

- ▶ non-terminating (badly oriented), e.g.

$$F \ a \sim G \ (F \ a)$$

- ▶ even if E_g and E_t terminating, $E_g \cup E_t$ may not be, e.g.

$$\begin{aligned} E_t &= \{F \ \text{Int} \sim F \ (G \ \text{Int})\} \\ E_g &= \{G \ \text{Int} \sim \text{Int}\} \end{aligned}$$

What we want

23/39

- ▶ Conditions on the top-level equations
- ▶ ... that are modular
- ▶ ... with arbitrary local equations
- ▶ ... to make type checking decidable
- ▶ and (of course) an algorithm to do so

Algorithm Overview

24/39

- ▶ Compute the completion of E_g , yielding E'_g
- ▶ Now $E'_g \cup E_t$ is confluent and terminating, and equivalent to $E_g \cup E_t$
- ▶ Decide $s \sim t$ by
 - ▶ rewriting s, t to normal forms wrt. $E'_g \cup E_t$

$$s \rightarrow^* u$$

$$t \rightarrow^* v$$

- ▶ compare for syntactic equality

$$u \equiv v$$

Completion of Local Equations

25/39

$$E_g = \{G \text{ Int} \sim F (G \text{ Int}), F (G \text{ Int}) \sim \text{Int}\}$$

- ▶ Give every function application a name (skolem)

$$\begin{aligned} &\rightsquigarrow a \sim F \ a, F \ a \sim \text{Int} \\ &\text{where } a = G \ \text{Int} \end{aligned}$$

- ▶ Put function on the LHS

$$\rightsquigarrow F \ a \sim a, F \ a \sim \text{Int}$$

- ▶ Substitute equation in the others

$$\begin{aligned} &\rightsquigarrow F \ a \sim a, a \sim \text{Int} \\ &\rightsquigarrow F \ \text{Int} \sim \text{Int}, a \sim \text{Int} \end{aligned}$$

$$E'_g = \{G \ \text{Int} \sim \text{Int}, F \ \text{Int} \sim \text{Int}\}$$

Type Checking with Completed Equations

26/39

$$E'_g = \{G \text{ Int} \sim \text{Int}, F \text{ Int} \sim \text{Int}\}$$

$$E_w = \{F (G \text{ Int}) \sim \text{Int}\}$$

- ▶ Normalise wrt. E'_g

$$\begin{array}{l} F (G \text{ Int}) \mapsto F \text{ Int} \mapsto \text{Int} \not\mapsto \\ \text{Int} \not\mapsto \end{array}$$

- ▶ Check for syntactic equality

$$\text{Int} \stackrel{?}{\equiv} \text{Int}$$

The Conditions on Top-Level Equations

27/39

Harsh Conditions

the Functional Dependency conditions:

- ▶ confluence: no overlapping heads
NO: $F \text{ Int } a = \dots \quad F \text{ a Bool} = \dots$
- ▶ termination (1): fewer symbols/variables in calls on the RHS
NO: $F \text{ a} = F \text{ [a]}$
- ▶ termination (2): at most a function call at the top of the RHS
NO: $F \text{ [a]} = [F \text{ a}]$

The Conditions on Top-Level Equations

28/39

Relaxed Confluence Condition

- ▶ confluence: **overlapping heads have overlapping bodies**

YES: $F \text{ Int } a = [\text{Int}] \quad F \text{ a Bool} = [a]$

- ▶ we don't have to consider overlapping methods!
- ▶ for modular checking: no general confluence check

The Conditions on Top-Level Equations

29/39

Work in Progress

- ▶ add loop checking to completion algorithm
- ▶ termination (2):
no nested function calls
YES: $F [a] = [F a]$
NO: $F [a] = F (G a)$

Type Checking with Evidence

30/39

Given

- ▶ E_t , the (quantified) top-level equations (e.g. $g : \forall a. \text{Elem } [e] \sim e$)
- ▶ E_g , a set of local equations (e.g. $h : \text{Elem } c \sim \text{Char}$)
- ▶ E_w , a set of wanted equations (e.g. $[\text{Char}] \sim [\text{Elem } c]$)

find a proof for

$$E_t, E_g \models k : E_w$$

- ▶ k is a term that justifies E_w
- ▶ a *trace* of the proof
- ▶ used for translation to System F_C

Type-directed Compilation

31/39

- ▶ System F: strongly-typed lambda calculus
- ▶ System F_C : System F extended with *coercions*: explicit type casts using evidence for equality
- ▶ GHC core language: sanity checks of compiler

```
id :: forall a b . b ~ a => a -> b

-- Haskell
id x = x

-- System FC
id =  $\Lambda(a:*) . \Lambda(b:*) . \Lambda(co:b \sim a) . \lambda(x:a) .$ 
      (x ▶ sym co)
```



- 1 Haskell in 5 (+1) slides
- 2 Informal Overview
- 3 Type Checking
 - Type Checking
 - Problems
 - The Algorithm
 - Type-Directed Compilation
- 4 Type Functions vs. Functional Dependencies**
 - Problem
 - Expressivity
 - Secondary Criteria
- 5 Conclusion

Type Functions vs. Functional Dependencies

33/39

Lively debate on Haskell-Prime:

Type Functions or Functional Dependencies?

- ▶ Last technical issue holding back the standard

Type Functions vs. Functional Dependencies

34/39

Primary Criterion: **Expressivity**

Functional Dependencies \rightarrow Type Functions

```
class C a b | a -> b, b -> a
```

\Rightarrow

```
class (F1 a ~ b, F2 b ~ a) => C a b where  
  type F1 a  
  type F2 b
```

Type Functions vs. Functional Dependencies

34/39

Primary Criterion: **Expressivity**Functional Dependencies \rightarrow Type Functions

```
class C a b | a -> b, b -> a
```

 \Rightarrow

```
class (F1 a ~ b, F2 b ~ a) => C a b where
  type F1 a
  type F2 b
```

Type Functions \rightarrow Functional Dependencies

```
type family F a b c
```

 \Rightarrow

```
class C a b c d | a b c -> d
```

Type Functions vs. Functional Dependencies

34/39

Primary Criterion: **Expressivity** same!

Functional Dependencies \rightarrow Type Functions

```
class C a b | a -> b, b -> a
```

\Rightarrow

```
class (F1 a ~ b, F2 b ~ a) => C a b where  
  type F1 a  
  type F2 b
```

Type Functions \rightarrow Functional Dependencies

```
type family F a b c
```

```
 $\Rightarrow$  class C a b c d | a b c -> d
```

Type Functions vs. Functional Dependencies

35/39

Secondary Criteria:

	Fun. Dependencies	Type Functions
paradigm	Logic Programming	Functional Programming
class-less	no	yes
status	second class*	first class
core language	none**	System F_C
rt. overhead	dictionary passing	none (erased)

* first-class syntax for type classes [Neubauer et al., Diatchki]

** System F_{CHR} ?



- 1 Haskell in 5 (+1) slides
- 2 Informal Overview
- 3 Type Checking
 - Type Checking
 - Problems
 - The Algorithm
 - Type-Directed Compilation
- 4 Type Functions vs. Functional Dependencies
 - Problem
 - Expressivity
 - Secondary Criteria
- 5 Conclusion

We have:

- ▶ an algorithm for type checking and inference
- ▶ ...that generates evidence...
- ▶ ...that allows translation into System F_C
- ▶ an implementation in the GHC HEAD: **try it yourself!**
- ▶ correspondence with Functional Dependencies

Theory

- ▶ relaxation of conditions
- ▶ *closed* type functions
- ▶ interaction with GADTs (rigidity conditions)
- ▶ uniform CHR-based theory for type classes & functions

Practice

- ▶ stable implementation
- ▶ meaningful error messages
- ▶ scalability

Thank you!

39/39

Questions?