Functional dependencies versus type families and beyond

Martin Sulzmann   National University of Singapore

Joint work with Tom Schrijvers

What's this talk about

- Functional dependencies: A relational specification for user-programmable type improvement connected to type class instances.
- Type families (aka functions): A functional specification for user-programmable type improvement decoupled from type class instances.
- Enthusiastic and lively debate which feature shall make it into the next Haskell standard (see Haskell').
- In this talk, we clarify some of the type inference issues behind functional dependencies (FDs) and type functions (TFs).

Type inference results

FDs (explained in terms of CHRs [SDPS07]):

- Coverage + Consistency $\Rightarrow$ Termination + Confluence
- No Coverage $\Rightarrow$ Non-Termination (in general)
- Weak Coverage + (Non-)Fullness $\Rightarrow$ (Non-)Confluence

Our new results: FDs explained in terms of TFs

- Weak Coverage + Non-Fullness $\Rightarrow$ Confluence
- Violating Weak Coverage is beyond FDs and TFs (type improvement is not functional anymore)

We contrast our results against Claus Reinke's alternative CHR encoding of FDs (informally discussed on Haskell')

- No Coverage + Non-Fullness $\Rightarrow$ Confluence

Outline

- Informal introduction to FDs and TFs.
- ...

Functional dependencies

Warm-up: Multi-parameter type class `Collects ce e` with FD `ce -> e`.

```
class Collects ce e | ce -> e where
  insert :: e->ce->ce
  delete :: e->ce->ce
  member :: e->ce->Bool
  empty  :: ce
```

The FD guarantees that fixing the collection type `ce` will fix the element type e.

Important to ensure that method `empty` is unambiguous.

FDs enforce type improvement

```
class Collects ce e | ce -> e where ...
```

```
inserttwo x y ce = insert x (insert y ce)
```

We infer the type

```
inserttwo :: Collects ce e => e -> e -> ce -> ce
```

Without the FD imposed on `Collects` we'd infer the type

```
inserttwo :: (Collects ce e1, Collects ce e2) =>
             e1 -> e2 -> ce -> ce
```

FD instance improvement

Let's consider some instance.

```
instance Eq e => Collects [e] e where
  insert e [] = [e]
  insert e (x:xs) = e:x:xs
  delete e (x:xs)
    | x == e    = xs
    | otherwise = x:(delete e xs)
  member e [] = False
  member e  (x:xs)
   | x == e     = True
   | otherwise = member e xs
  empty = []
```

FDs instance improvement

```
class Collects ce e | ce -> e where ...
instance Eq e => Collects [e] e where ...
```

```
insert3 xs x = insert (tail xs) x
```

We infer the type

```
insert3 :: Eq e => [e] -> e -> [e]
```

Without the FD and instance, we'd infer

```
insert3 :: Collects [e1] e2 => [e1] -> e2 -> [e1]
```

FDs intermediate summary

- FDs allow us to assign better (more precise) types to programs.
- This can be essential in case of ambiguities.
- FDs are implemented by most major Haskell implementations (Hugs and GHC).
- Formal descriptions exist [Jon00, SDPS07].
- Many sophisticated uses of FDs.

FDs on steroids

```
-- numerals on the level of types
data Z = Z
data S n = S n

-- addition on the level of types
class Add l m n | l m -> n where
  add :: l -> m -> n

instance Add Z n n where
  add Z x = x

instance Add l m n => Add (S l) m (S n) where
  add (S x) y = S (add x y)
```

FDs on steroids

```
two = S (S Z)
four = add two two
```

Type inference yields

```
four :: S (S (S (S Z)))
```

How?

```
class Add l m n | l m -> n
instance Add Z n n                          -- (1)
instance Add l m n => Add (S l) m (S n)   -- (2)

    Add (S (S Z)) (S (S Z)) n
--> Add (S (S Z)) (S (S Z)) (S n1), n = S n1   instance improvement (2)
--> Add (S Z) (S (S Z)) n1, n = S n1           instance reduction (2)
--> Add (S Z) (S (S Z)) (S n2),                instance improvement (2)
    n1 = S n2, n = S (S n2)
--> Add Z (S (S Z)) n2,                        instance reduction (2)
    n1 = S n2, n = S (S n2)
--> Add Z (S (S Z)) S (S Z),                   instance improvement (1)
    n2 = S (S Z), n1 = S (S (S Z)),
    n = S (S (S (S Z)))
--> n2 = S (S Z), n1 = S (S (S Z)),            instance reduction (1)
    n = S (S (S (S Z)))
```

FDs on steroids (with syntactic sugar)

```
two = 1 + (1 + 0)
four = add two two
```

Type inference yields

```
four :: 1 + (1 + (1 + (1 + 0)))
```

How?

```
class Add l m n | l m -> n
instance Add 0 n n                           -- (1)
instance Add l m n => Add (1+l) m (1+n)      -- (2)

    Add 2 2 n
--> Add 2 2 (1+n1), n = 1+n1                       instance improvement (2)
--> Add 1 2 n1, n = 1+n1                           instance reduction (2)
--> Add 1 2 (1+n2),                                instance improvement (2)
    n1 = 1+n2, n = 2+n2
--> Add 0 2 n2,                                    instance reduction (2)
    n1 = 1+n2, n = 2+n2
--> Add 0 2 2,                                     instance improvement (1)
    n2 = 2, n1 = 3, n = 4
--> n2 = 2, n1 = 3, n = 4                          instance reduction (1)
```

FDs on steroids summary

- We can write programs on the level of types.
- Relational style (powerful):
  Instance reduction and FD/instance improvement.
- But hard to read/see the type-level program:
  - From right to left

    ```
    instance Add l m n => Add (S l) m (S n)
    ```
  - Improvement is implicit. Derived from FD and instances.
- Up-next we introduce a concept to write type-level programs in functional style.

Type families (functions)

```
type family Add l m
type instance Add Z m = m
type instance Add (S l) m = S (Add l m)

class Plus l m where
  plus :: l -> m -> Add l m
instance Plus Z m where
  plus Z x = x
instance Plus l m => Plus (S l) m where
  plus (S x) y = S (plus x y)
```

- Clear functional notation for type improvement.
- Decoupled from type class instances.

Outline

- *Informal introduction to FDs and TFs.*
- Type inference issues:
  - Termination
  - Confluence
- ...

Type inference

- Reduced to constraint solving
- Plain Hindley/Milner $\Rightarrow$ unification

$$f\ e \quad \Rightarrow \quad t_f = t_e \rightarrow t_r$$

- Haskell-FDs $\Rightarrow$ type class solving + unification + ...

$$\texttt{four = add two two} \quad \Rightarrow$$

$$\begin{array}{ll} & t_{four} = t_1 \rightarrow t_2 \rightarrow t_3, \\ & t_1 = S\ (S\ Z), t_2 = S\ (S\ Z), \\ & Add\ (S\ (S\ Z))\ (S\ (S\ Z))\ t_3 \\ \rightarrowtail^* & t_{four} = S\ (S\ Z) \rightarrow S\ (S\ Z) \rightarrow S\ (\ S\ (S\ (S\ Z))) \\ & t_1 = S\ (S\ Z), t_2 = S\ (S\ Z), \\ & t_3 = S\ (\ S\ (S\ (S\ Z))) \end{array}$$

- Haskell-TFs $\Rightarrow$ term rewriting plus unification

Type inference properties

- Decidable: Solver terminates.

  Any constraint set of a fixed size can be reduced in a finite number of solving steps.

- Complete: Solver is confluent.

  If several solving steps are applicable, the specific choice won't affect the final result.

FDs No Coverage ⇒ Non-Termination

- Classic FD example (simplified “add”):

```
class F a b | a -> b
instance F a b => F [a] [b]
```

- Coverage violated: For instance head `F [a] [b]`, the variable b in the FD range is not covered by the FD domain a.
- Solving of `F [c] c` won’t terminate:
  - Instance improvement:
    `F [c] c, c=[d]` iff `F [[d]] [d], c=[d]`
  - Instance reduction:
    `F [d] d, c=[d]`   cycle!

TFs No Coverage $\Rightarrow$ Non-Termination

- ```
  type family F a
  type instance F [a] = [F a]
  instance F a b => F [a] [b]
  ```
- TFs coverage violated: We find a value constructor ("list") on the rhs.
- Solving of `F [c] = c` won't terminate:
  - TFs improvement: `[F c] = c`
  - Substitution: `[F [F c]] = c`
  - TFs improvement: `[[[F c]]] = c` cycle!
  - We must substitute if further improvements are possible (otherwise incomplete).
  - Suppose, we need to deduce that `[[[[[ ... F c ...]]]]] = c`.
  ```

Outline

- *Informal introduction to FDs and TFs.*
- *Type inference issues:*
  - *Termination*
  - *Confluence*
- *No Coverage* $\Rightarrow$ *Non-Termination (for FDs and TFs).*
- Termination is hard, so we focus on confluence.
- Current FD-encoding of CHRs (and its problem).
- ...

CHR Encoding of FDs

The type class program

```
class Add l m n | l m -> n
instance Add Z n n
instance Add l m n => Add (S l) m (S n)
```

translates to the Constraint Handling Rules (CHR) program

```
Add Z n n <==> True                        -- (Inst1)
Add (S l) m (S n) <==> Add l m n           -- (Inst2)

Add l m n1, Add l m n2 ==> n1 = n2         -- (FD-improvement)
Add Z n m ==> m = n                        -- (Inst1-improvement
Add (S l) m n ==> n = S n'                 -- (Inst2-improvement
```

See [SDPS07] for details.

Non-Full Problem

```
class F a b c | a -> b
  -- non-full cause c plays no part
instance F a b Bool => F [a] [b] Bool
```

translates to

```
F a b1 c, F a b2 d ==> b1 = b2            (FD)
F [a] [b] Bool <=> F a b Bool           (Inst)
F [a] b c ==> b = [b1]                 (Imp)
```

but the above CHRs are non-confluent.

```
             F [a] [b] Bool F [a] b2 d

>--> FD     F [a] [b] Bool, F [a] [b] d, b2 = [b]
>--> Inst   F a b Bool, F [a] [b] d, b2 = [b]

>--> Inst   F a b Bool, F [a] b2 d
>--> Imp    F a b Bool, F [a] [c] d, b2 = [c]
```

The two derivations have a different outcome.

Outline

- *Informal introduction to FDs and TFs.*
- *Type inference issues:*
  - *Termination*
  - *Confluence*
- *No Coverage* $\Rightarrow$ *Non-Termination (for FDs and TFs).*
- *Termination is hard, so we focus on confluence.*
- Current FD-encoding of CHRs (and its problem):
  Non-confluent for non-full FDs
- Our solution: Project non-full FDs onto full FDs.
- We will use TFs notation for full FDs.

TFs Encoding of (Non-Full) FDs

```
class F a b c | a -> b
instance F a b Bool => F [a] [b] Bool
```

translates to

```
type family FD a
type instance FD [a] = [FD a]

class FD a ~ b => F a b c
instance F a b Bool => F [a] [b] Bool
```

We project non-full FD

```
class F a b c | a -> b
```

onto a “full” TF

```
type family FD a
class FD a ~ b => F a b c
```

Resulting TF program is confluent.

TFs Encoding of (Non-Full) FDs

```
class H a b | a -> b
class F a b c | a -> b
instance F a b Bool => F [a] [b] Bool
instance H a b => F [a] [b] Char
```

translates to

```
type family FD a
type family HD a
type instance FD [a] = [FD a]   -- overlap !!!
type instance FD [a] = [HD a]

class FD a ~ b => F a b c
instance F a b Bool => F [a] [b] Bool
```

Projecting a non-full FD onto a “full” TF leads to overlapping TF definitions ($\Rightarrow$ non-confluence).

Modular TFs Encoding of FDs

```
class H a b | a -> b
class F a b c | a -> b
instance F a b Bool => F [a] [b] Bool
instance H a b => F [a] [b] Char
```

translates to

```
type family FD a
type family Sel a b
type family HD a
type instance FD [a] c = [Sel a c]
type instance Sel a Bool = [FD a Bool]
type instance Sel a Char = [HD a]

class FD a c ~ b => F a b c
instance F a b Bool => F [a] [b] Bool
instance H a b => F [a] [b] Char
```

Trick: We keep track of the context! There's no overlap anymore.

Outline

- *Informal introduction to FDs and TFs.*
- *Type inference issues:*
  - *Termination*
  - *Confluence*
- *No Coverage* $\Rightarrow$ *Non-Termination (for FDs and TFs).*
- *Termination is hard, so we focus on confluence.*
- Current FD-encoding of CHRs (and its problem): Non-confluent for non-full FDs
- Our solution: Project non-full FDs onto full FDs, use context to avoid overlaps.
- We use TFs notation for full FDs (because Haskell-GHC has native support for TFs, see [SCPD07]).
- We yet need to relate our solution against Claus Reinke's.

Claus Reinke's FDs Encoding

Encoding based on CHRs:

```
class F a b c | a -> b
  -- non-full cause c plays no part
instance F a b Bool => F [a] [b] Bool
```

translates to

```
F a b c <=> Finst a b c, Fimp a b c          (Split)
Fimp a b1 c, Fimp a b2 d ==> b1 = b2         (FD)
Finst [a] [b] Bool <=> F a b Bool            (Inst)
Fimp [a] b c ==> b = [d]                     (Imp)
```

`Fimp a b c`
plays a similar role as
`FD a c`
in our TFs-based encoding.

Claus Reinke’s FDs Encoding

“Problematic” example:

```
class H a b | a -> b
class F a b c | a -> b
instance F a b Bool => F [a] [b] Bool
instance H a b => F [a] [b] Char
```

translates to

```
F a b c <=> Finst a b c, Fimp a b c       (Split)
Fimp a b1 c, Fimp a b2 d ==> b1 = b2      (FD)
Finst [a] [b] Bool <=> F a b Bool         (Inst)
Fimp [a] b c ==> b = [d]                  (Imp)

Finst [a] [b] Char <==> H a b             (Inst2)
Fimp [a] b c ==> b = [d]                  (Imp2)
```

There’s a harmless overlap among (Imp) and (Imp2).

Claus Reinke's FDs Encoding

Improvement (propagation) steps are mixed with instance (simplification) steps.

```
    F [[a]] b Bool
--> Finst [[a]] b Bool, Fimp [[a]] b Bool                  (Split)
--> Finst [[a]] [b1] Bool, Fimp [[a]] [b1] Bool, b = [b1]  (Imp)
--> F [a] b1 Bool, Fimp [[a]] [b1] Bool, b = [b1]          (Inst)
--> Finst [a] b1 Bool, Fimp [a] b1 Bool,                   (Split)
    Fimp [[a]] [b1] Bool, b = [b1]
--> Finst [a] [b2] Bool, Fimp [a] [b2] Bool,               (Imp)
    Fimp [[a]] [[b2]] Bool, b = [[b2]], b1 = [b2]
--> F a b2 Bool, Fimp [a] [b2] Bool,                       (Inst)
    Fimp [[a]] [[b2]] Bool, b = [[b2]], b1 = [b2]
--> Finst a b2 Bool, Fimp a b2 Bool,                       (Split)
    Fimp [a] [b2] Bool, Fimp [[a]] [[b2]] Bool,
    b = [[b2]], b1 = [b2]
```

A Comparison

- We require Weak Coverage:

  ```
  class F a b | a -> b
  instance F a b => F [a] [b]
  ```

  Coverage violated: For instance head `F [a] [b]`, the variable b in the FD range is not covered by the FD domain a.
  But b is weakly covered by the instance context `F a b`.

- Weak Coverage means that the FD behaves functionally.
- Violating Weak Coverage $\Rightarrow$ Beyond FDs

  ```
  class F a b | a -> b
  instance F b a => F [a] [b]
  ```

Beyond FDs

Claus Reinke's encoding works for the below (i.e. is confluent):

- ```
  class F a b | a -> b
  instance F b a => F [a] [b]
  ```
  translates to
  ```
  F a b <=> Finst a b, Fimp a b              (Split)
  Finst [a] [b] <=> F b a                    (Inst)
  Fimp a b1, Fimp a b2 ==> b1 = b2           (FD)
  Fimp [a] c ==> c = [b]                     (Imp)
  ```
- ```
  class F a b | a -> b
  instance F [a] [b]
  ```

Beyond FDs

```
data Nil        = Nil
data Cons a b   = Cons a b
data ExpAbs x a = ExpAbs x a
class Eval env exp t | env exp -> t where
    -- env represents environment, exp expression
    -- and t is the type of the resulting value
    eval :: env->exp->t
instance Eval (Cons (x,v1) env) exp v2
      => Eval env (ExpAbs x exp) (v1->v2) where
    eval env (ExpAbs x exp) =
           \v -> eval (Cons (x,v) env) exp
```

Instance head: Variable v1 appears in the FD range.
Instance context: Variable v1 appears in the FD domain.

Beyond Functional Dependencies = Relational Dependencies

Conclusion

- FDs and TFs are two forms of user-specifiable type improvement.
- Choose for yourself which one you prefer.
- We can encode FDs via TFs.
- We can go beyond FDs with Claus Reinke's encoding of FDs.

Future work:

- Design a typed-intermediate language for CHRs (like System FC [SCPD07] for TFs).

References

[Jon00] M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[SCPD07] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Proc. of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 53–66. ACM Press, 2007.

[SDPS07] M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129, 2007.