Chapter 2

Restoring Confluence for Functional Dependencies

Tom Schrijvers[1], Martin Sulzmann[2]
Category: Research Paper

Abstract: Haskell-style functional dependencies allow the programmer to influence the type inference process by automatically deriving type improvement rules from class and instance declarations.

Previous work demands that the dependency must fully cover all type parameters of a type class to guarantee confluence. Confluence is an important property to ensure that type inference is well-behaved.

We restore confluence for non-full functional dependencies by giving a novel encoding of functional dependencies in terms of type families as supported by the Glasgow Haskell Compiler. We compare our method against another alternative constraint-propagation encoding due to Claus Reinke. Our work provides new insights in the connection between functional dependencies and type families, and in the relative trade-offs of rewriting based or constraint-propagation based type inference approaches.

2.1 INTRODUCTION

Haskell-style functional dependencies [6] provide a relational specification of user-programmable type improvement [5] connected to type class instances. The more recent type families (also known as type functions) [8, 7] equip the programmer with similar functionality, but in a functional form and decoupled from type classes. Functional dependencies are supported by both GHC and Hugs, while the most recent version of GHC also supports type functions.

[1]K.U.Leuven, Celestijnenlaan 200A, Belgium;
`tom.schrijvers@cs.kuleuven.be`
[2]IT University of Copenhagen, Denmark; `martin.sulzmann@gmail.com`

There was an enthusiastic and lively debate about which feature should be adopted by the next Haskell standard, Haskell-Prime.[3] Currently, further progress in the standardization appears to be stalled on this issue.

In this work, we attempt to rekindle the debate with new insights in type inference issues behind functional dependencies (FDs) and type functions (TFs), without taking sides. Specifically, we make the following contributions:

- We revisit the non-confluence issue of Constraint Handling Rules (CHRs) resulting from non-full FDs (Section 2.3). CHRs serve as a meta-language to describe the constraint solver underlying the type inferencer. Confluence of CHRs is important to ensure that the type inferencer is well-behaved. Unfortunately, the CHR encoding of FDs proposed in [11] cannot guarantee confluence for non-full FDs.
- We propose an alternative encoding which achieves confluence even for non-full FDs. Our result is based on an encoding of FDs via TFs (Section 2.4).
- Based on an alternative CHR *propagation* encoding (proposed by Claus Reinke) we can even guarantee confluence for FDs which break the Weak Coverage Condition (Section 2.5). Such FDs cannot be expressed in terms of confluent TFs anymore.
- We establish the respective merits of our two new encodings (Section 2.6). The TF encoding on the one hand achieves a type-preserving translation of FDs to a System F style typed-intermediate language. On the other hand, we get memoing for free in the propagation encoding.

2.2 TYPE INFERENCE WITH FUNCTIONAL DEPENDENCIES

Functional Dependencies Functional dependencies [6] allow the programmer to influence the type inference process, we say they improve types [5], and help to resolve ambiguities when translating type classes. We illustrate both points using the parameterized collections example from [6].

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

Type classes are translated by turning them into dictionaries (records holding member functions). Method `empty` yields an empty collection but the element type is not mentioned. Hence, it's not immediately clear which dictionary to choose. For example, we could encounter the situation of having to choose between `Collects [Integer] Integer` and `Collects [Integer]`

[3] For example, see discussions on the Haskell-Prime and Haskell mailing lists.

`Int` but an arbitrary choice leads to ambiguities. By imposing the functional dependency `| ce -> e` such situations cannot happen because the functional dependency guarantees that fixing the collection type `ce` uniquely determines the element type `e`. Hence, `Collects [Integer] Integer` and `Collects [Integer] Int` cannot exist at the same time. Hence, there is no ambiguity.

Type Inference In terms of type inference, functional dependencies help the programmer to *improve* types.

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

The program text of `inserttwo` yields `Collects ce e1` and `Collects ce e2`. Thanks to the functional dependency we can infer that `e1` and `e2` must be equal. Hence, type inference yields the improved type

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

Instance Improvement Type improvement provided by functional dependencies depends also on the set of available instances.

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

For brevity, we omit the instance body. Thanks to the functional dependency and the above instance, type inference for

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

yields

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

Here is why. The program text of `insert3` gives rise to `Collects [e1] e2`. The functional dependency says that the collection type `[e1]` uniquely determines the element type `e2`. The programmer has narrowed down the possible choices for `e2` by specifying `instance ... => Collects [e] e`. Hence, `e2` and `e1` must be equivalent. But then we can reduce `Collects [e1] e1` to `Eq e1`.

The CHR Encoding In general, the type improvement conditions implied by functional dependencies can be fairly complex and often hard to understand. Previous work makes use of Constraint Handling Rules (CHRs) [3] to give a precise and systematic description of context reduction and type improvement. According to [11], the above class and instance declarations translate to the following CHRs.

```
Collects ce e1, Collects ce e2 ==> e1 = e2   (FD)
Collects [e1] e2 ==> e1 = e2                (Imp)
Collects [e] e <==> Eq e                   (Inst)
```

CHR solving is an extension of Herbrand unification over a user-programmable constraint domain. This is exactly what we need to describe type class constraint solving. The first CHR rule `(FD)` captures the functional dependency. Each time the solver sees the two constraints (or instantiations of it) `Collects ce e1` and `Collects ce e2`, the solver will add (propagate) the information that `e1` and `e2` are equivalent. In CHR syntax, constraint propagation is represented via `==>`. Rule `(Imp)` therefore captures the improvement resulting form the type instance. The last rule `(Inst)` expresses context reduction. That is, whenever we see the constraint `Collects [e] e` (or instantiations of it), we reduce (simplify) this constraint to `Eq e`. Constraint simplification is represented via `<==>` in CHR syntax.

CHR-Based Type Inference The benefit of translating functional dependency programs to CHRs is that we obtain a systematic type inference method based on CHR solving. Earlier we have seen that type inference (without type annotations) simply boils down to generating appropriate constraint out of the program text and then solve them with respect to the set of available CHR rules.

Type inference in the presence of type annotations is a bit more involved. Consider the earlier example where we have added some type annotation.

```
insert3 :: Eq e => [e] -> e -> [e]
insert3 xs x = insert (tail xs) x
```

To verify the correctness of the annotation, the type inferencer needs to check that the *given* constraints C_g from the annotation entail the *wanted* constraints C_w from the program text. The entailment check $C_g \implies C_w$ is logically equivalent to the equivalence check $C_g \iff C_g \wedge C_w$. We can test for equivalence by normalizing (exhaustive CHR solving) the left-hand and right-side of $\iff$ and checking the resulting constrains for syntactic equivalence. In case of the above type annotated function, we effectively need to check for `Eq e` $\implies$ `Collects [e] e`. Normalization yields

```
                Eq e, Collects [e] e
↣Inst           Eq e
```

We assume set semantics and the constraint `Eq e` is in normal form. We find that the equivalence holds and therefore the function is type correct.

The Role of Confluence For the equivalence testing method (and therefore type inference) to be decidable and complete we critically rely on termination and confluence of CHRs [9]. Termination of CHRs ensures that we can build normal forms in a finite number of steps. Confluence guarantees canonical normal forms because different derivations starting from the same point can always be brought together again.

Termination is notoriously hard to achieve [7] unless we impose rather harsh conditions on the set of allowable programs [11]. We therefore assume that guar-

anteeing termination is the users responsibility. [4] But we wish that confluence is guaranteed by some simple conditions which are satisfied by a reasonably large class of programs. The conditions identified in [11] to guarantee confluence of CHRs resulting from functional dependency programs are:

Definition 2.1 (Weak Coverage Condition). [5] *For each functional dependency* $a_{i_1},...,a_{i_k}$ `->` a_{i_0} *for class TC and*

$$\texttt{instance}\ C \texttt{ => } \texttt{TC}\ t_1 ... t_n$$

we must have that $fv(t_{i0}) \subseteq closure(C, fv(t_{i1},\ldots,t_{ik}))$. *where* $closure(C, vs)$ *is the least fix-point of the following equation.*

$$F(X) \quad = \quad \bigcup_{\substack{TC\ t_1 \ldots t_n \in C \\ \texttt{TC}\ a_1 \ldots a_n \mid a_{i_1},...,a_{i_k}\ \texttt{->}\ a_{i_0}}} \{fv(t_{i_0}) \mid fv(t_{i_1},\ldots,t_{i_k}) \subseteq X\}$$

We treat the instance context C as a set of (type class) constraints and assume that function *fv* computes the set of free variables.

Definition 2.2 (Full Functional Dependencies). *We say the functional dependency* `class` *TC* $a_1 \ldots\ a_n | a_{i_1},...,a_{i_k}$ `->` a_{i_0} *for a type class* `TC` *is* full *iff* $k = n-1$.

The Weak Coverage Condition says that all variables in the domain of the FD are "functionally" covered by FDs in the instance context (plus building the transitive closure of those functionally covered variables). In essence, the Weak Coverage Condition says that that functional dependencies must behave "functional". The Weak Coverage Condition will become much clearer once we express functional dependencies in terms of type families in the upcoming Section 2.4. For the moment, we omit further details and argue that the Weak Coverage Condition is a natural condition which ought to be satisfied by all functional dependencies. The second condition is less obvious and can be limiting. However, as we discuss in the next section, the CHR encoding of FDs suggested in [11] requires fullness to guarantee confluence and this seems like a fairly limiting assumption.

2.3 THE CONFLUENCE PROBLEM OF NON-FULL FDS

Non-Full FDs The `Stream` type class from the `parsec` package [6] provides a generic interface for streams:

```
class (Monad m) => Stream s m t | s -> t where
    uncons :: s -> m (Maybe (t,s))
```

[4] Possibly using some heuristics by for example limiting the number of solving steps to some number *k*. Any solving beyond *k* leads to the "don't know" answer.

[5] Referred to as Refined Weak Coverage in [11].

[6] Available at http://hackage.haskell.org

Here `s` is the streams, `t` is the type of tokens and `m` is the monad in which the stream can be read. The functional dependency `s -> t` expresses that the token type can be derived from the stream type. The above FD is *non-full* because the type class parameter `m` plays no part in the FD `s -> t`.

Lists are a simple instance of streams:

```
instance (Monad m) => Stream [t] m t where
    uncons []     = return Nothing
    uncons (x:xs) = return $ Just (x,xs)
```

The token type is the element type of the list. The monad type is independent of the two others: we can read from a list in any monad and the element type does not depend on the monad type.

A file cursor (the combination of a file handle and position) would be another kind of stream, one that is only usable inside the IO monad.

```
data Cursor = Cursor Handle Int
instance Stream Cursor IO Char where
    uncons (Cursor h p)
      =  do hSeek AbsoluteSeek p h
            eof <- hIsEOF h
            if eof then return Nothing
            else do c  <- hGetChar h
                    p' <- hGetPosn h
                    return $ Just $ Cursor h p'
```

Other examples of non-full FDs can be found in the `ArrayRef` and `StrategyLib` packages on HackageDB.

Confluence Problem Consider this type class program with a non-full FD:

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

The consequence of non-fullness is that the CHRs resulting from the above program are non-confluent. We show non-confluence by giving two non-joinable derivations. Here are the CHRs resulting from the above program according to [11]:

```
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)
```

These two distinct derivations from the same set of constraints illustrate the non-confluence:

```
  F [a] [b] Bool, F [a] b2 d                (1)

↣FD   F [a] [b] Bool, F [a] [b] d, b2 = [b]
```

$\rightarrowtail_{Inst}$ `F a b Bool, F [a] [b] d, b2 = [b]` (2)

$\rightarrowtail_{Inst}$ `F a b Bool, F [a] b2 d`
$\rightarrowtail_{Imp}$ `F a b Bool, F [a] [c] d, b2 = [c]` (3)

In the first derivation, we first apply the `(FD)` rule and propagate `b2 = [b]`, followed by an application of rule `(Inst)`. The other derivation immediately applies rule `(Inst)` and then rule `(Imp)`. No further rule applications are possible and we can see that both derivations yield a different result. We say that the CHRs are non-confluent which means that our CHR-based type inference is potentially incomplete.

For example, suppose that type inference needs to verify that constraint (1) entails constraint (2) (which clearly holds see the first derivation). CHRs are non-confluent and therefore we possibly reduce (1) to (3) using the second derivation. Constraints (2) and (3) differ and therefore we falsely report that the entailment does not hold.

2.4 THE TYPE FUNCTIONS ENCODING OF FDS

2.4.1 Proper Non-Full FDs

The above non-confluence is not inherent in the non-full FDs themselves; it is an artefact of the CHR encoding of [11]. We investigate an alternative encoding of FDs in terms of type functions that does not suffer from this confluence issue. This new encoding factors out the (non-full) FD `a -> b` into a separate type function `FD a`.

The type function is defined as:

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

The type class program is similar to the one before:[7]

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

but now the functional dependency annotation `| a -> b` is replaced by the equality constraint `FD a ~ b =>` in the class context. Unlike functional dependencies, type functions are always *full* by construction.

From this new encoding we get the two CHRs:

```
F a b c ==> FD a ~ b                (CC)
F [a] [b] Bool <=> F a b Bool    (Inst)
```

The latter is the instance rule we've seen before. The former is also standard in [9]; it adds the class context.

[7] The symbol ~ is GHC's notation for type equality.

These CHRs for the type class program are complemented by the rewrite rules for type functions. These rules are explained in great detail in [7]. Here we explain them by example.

This new TF-based encoding of the program is confluent. Let us revisit the earlier example.

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

↣CC       F [a] [b] Bool, F [a] b2 d, FD [a] ~ [b]
↣CC       F [a] [b] Bool, F [a] b2 d, FD [a] ~ [b], FD [a] ~ b2
↣Subst  F [a] [b] Bool, F [a] b2 d, b2 ~ [b], FD [a] ~ b2
↣Inst     F a b Bool, F [a] b2 d, b2 ~ [b], FD [a] ~ b2
↣TF       F a b Bool, F [a] b2 d, b2 ~ [b], [FD a] ~ b2
↣Subst  F a b Bool, F [a] b2 d, b2 ~ [b], [FD a] ~ [b]
↣Decomp F a b Bool, F [a] b2 d, b2 ~ [b], FD a ~ b

↣Inst     F a b Bool, F [a] b2 d
↣CC       F a b Bool, F [a] b2 d, FD [a] ~ b2
↣TF       F a b Bool, F [a] b2 d, [FD a] ~ b2
↣CC       F a b Bool, F [a] b2 d, [FD a] ~ b2, FD a ~ b
↣Subst  F a b Bool, F [a] [FD a] d, [FD a] ~ b2, FD a ~ b
↣Subst  F a b Bool, F [a] [b] d, [b] ~ b2, FD a ~ b
```

Here `Subst` and `Decomp` are substitutions and decompositions of equations. The above are but two of the possible derivations, but one can verify that all derivations yield the same result.

Back to full FDs While the choice of type functions in the above encoding is enlightening, it is not essential. A transformation between type functions and type classes with functional dependencies was proposed in [8]. We can readily apply to our example program.

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

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

Instead of the type function `FD a` we now have a type class `FD a b` with a functional dependency `a -> b`. The functional dependency of this type class is full, in contrast with that of the original type class `F a b c`. In the new type class `F a b c` the functional dependency is captured by the `FD a b` superclass. Overall, we have a type class program without non-full FDs. So we know from [11] that its CHR encoding is confluent.

2.4.2 Improper Non-Full FDs

In general, the above naive TF encoding leads to problems. Consider the program:

```
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
```

with TF encoding

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

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

The problem in this encoding is that the two instances of the type function `FD` overlap, and are non-confluent. We say that the FD is an *improper* non-full FD. Superficially, it seems that the `b` parameter is functionally determined by the `a` parameter alone. However, the above overlap bears out that this is incorrect. Only the top-level type constructor `[]` is determined by `a` alone. The remainder of `b` also depends on the particular instance that is matched. In other words, it also depends on `c`. Hence, an *improper* non-full FD is situated somewhere in-between a full and a proper non-full FD.

The improper non-full FD can be captured with an additional *instance selector* type function `Sel` as follows.[8]

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

While the `FD` type function takes both `a` and `c` as parameters, it only matches on `a` in the function instance. The latter procures the top-level type constructor `[]` and delegates the rest of the result type to the selector function `Sel`. Now `Sel` is allowed to match on `c`, and choose the appropriate recursive call corresponding to the instance.

[8] In general we have one selector function for each set of overlapping instances.

Back to full FDs Again, we can translate the TF encoding to a full FD encoding:

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

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

2.5 BEYOND FUNCTIONAL DEPENDENCIES

Yet another encoding of FDs, in terms of propagation CHRs only, was suggested by Claus Reinke on the Haskell-Prime mailing list. Although Reinke's original proposal was more involved, we reduce it to its essence as a minimal change of the CHR encoding of [11]: replace `<=>` in the (Inst) rule with `==>`. The earlier "problematic" non-full FD example

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

translates then 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)
```

A CHR program with only propagation rules is trivially confluent; there are no critical pairs.

The propagation encoding of FDs is short and elegant and directly supports improper FDs. Recall the earlier improper FD program

```
class H a b | a -> b
class F a b c | a -> b
instance F a b Bool => F [a] [b] Bool  -- (1)
instance H a b => F [a] [b] Char
```

and its propagation encoding[9]

```
H a b, H a c ==> b = c          (FDH)
F a b c, F a d e ==> c = e      (FDF)
F [a] [b] Bool ==> F a b Bool (Inst1)
F [a] [b] Char ==> H a b      (Inst2)
F [a] c d ==> c = [b1]         (Imp1)
F [a] c d ==> c = [b2]         (Imp2)
```

[9] One of the two improvement rules is redundant.

For the TF encoding, we had to introduce an auxiliary selector function to resolve the overlap among the type functions derived from the above instances. Such cases are unproblematic for the propagation encoding because we strictly use propagation instead of simplification.

Relational Dependencies The propagation encoding goes beyond functional relations between type class parameters. Consider for example the type class program:

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

There is no functional relationship between the type class parameters in the above instance. We cannot implement it with the type function encoding. However, the propagation encoding yields:

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

The point is that via propagation we can additionally capture relational dependencies whereas type functions can only capture functional dependencies. This can be useful for examples such as the following type-directed evaluator.

```
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
```

In the instance head, variable `v1` appears in the range of the FD whereas in the Instance context variable `v1` appears in the FD domain. Hence, the FD is clearly *not* functional and cannot be expressed via type functions. Such relational dependencies can be easily expressed via CHR propagation rules.

```
Eval env exp t, Eval env exp t2 ==> t = t2     (FD)
Eval env (ExpAbs x exp) (v1->v2) ==>
Eval (Cons (x,v1) env) exp v2                  (Inst)
Eval env (ExpAbs x exp) v ==> v = (v1->v2)     (Imp)
```

2.6 COMPARISON OF APPROACHES

The CHR encoding of FDs in [11] requires fullness in order to guarantee confluence. We have shown that our new type function encoding restores confluence for non-full FDs. The propagation encoding of FDs goes beyond functional dependencies and can even express relational dependencies.

The propagation encoding itself is short and elegant but unfortunately suffers from two problems: (1) it is not clear yet how to compute evidence based on the propagation encoding, and (2) the propagation encoding has a higher space complexity but possibly a lower time complexity compared to the other approaches. We elaborate on both issues below.

Type-Preserving Translation Type classes are translated based on the dictionary (also known as evidence) translation [4]. In the context of a type-preserving compiler such as GHC, we additionally need to compute evidence for type improvement. For example, a naive translation of

```
class F a b | a -> b
instance F Int Int
f :: F Int a => a -> Int
f = \ x -> x + 1
```

yields the following *ill-typed* System F term

```
Λa.λd:FDict Int a.λx:a.x + 1
```

The subexpression `x + 1` fails to type check because we assume that `x` is of the universally quantified type `a`. However, in the source program we know that the constraint `F Int a` implies the improvement `a = Int`. This information is completely lost in our naive translation.

How to build evidence for type improvement specified via type functions has been studied before, namely in System F_C [10] which supports type coercions justified by equational evidence. By applying the type function encoding for FDs from Section 2.4 we can thus achieve a correct type-preserving translation of FDs. For instance, the type instance `FD Int = Int` yields an evidence constant `coInt`, and the dictionary data type `FDict` captures the equational evidence as follows:

```
coInt :: (FD Int ~ Int)
data FDict x y = FDict (FD x ~ y)
```

This evidence information can be used to produce a well-typed System F_C term:

```
Λa.λd:FDict Int a.λx:a.case d of
      { FDict co:(FD Int ~ a) ->
          (x ▶ sym co ∘ coInt) + 1 }
```

Here the case expression matches on the dictionary `t` to bring the evidence `co` in scope. The combinator `sym` returns the symmetric evidence `a ~ FD Int`.

Composed with `coInt`, we get the desired evidence `a ~ Int` which allows us to coerce `x` from `a` to the desired type `Int`.

Evidence generation for CHRs arising in the propagation encoding is a topic which yet needs to be studied.

Time vs. Space Trade-Off The propagation encoding potentially uses more space than the type function encoding. However, it can put this space to good use in the form of memoing, which comes naturally.

Consider the type-level Fibonacci relation `Fib n f` which denotes that `f` is the `n`th Fibonacci number:

```
data Z     -- type-level
data S n   -- natural numbers

class Fib n f | n -> f
instance Fib Z     (S Z)
instance Fib (S Z) (S Z)
instance (Fib (S n) f1, Fib n f2, Add f1 f2 f) => Fib (S (S n)) f

class Add a b c | a b -> c
instance Add Z b b
instance Add a b c => Add (S a) b (S c)
```

Using the standard evaluation strategy for CHRs, the refined operational semantics [2], the above program corresponds faithfully to the value-level Haskell function:

```
data Nat = Z | S Nat

fib :: Nat -> Nat
fib Z     = S Z
fib (S Z) = S Z
fib (S (S n)) = fib (S n) `add` fib n

add :: Nat -> Nat -> Nat
add Z     b = b
add (S a) b = S (add a b)
```

which exhibits an $\mathcal{O}(2^n)$ time complexity and $\mathcal{O}(1)$ space complexity.

The propagation encoding, however, stores all intermediate calls and avoids repeated calls with the (FD) rule:

```
Fib n f1, Fib n f2 ==> f1 = f2        (FD)
```

This results in a linear time complexity ($\mathcal{O}(n)$) at the cost of storing the n intermediate calls ($\mathcal{O}(n)$ space complexity).

The memoing effect comes naturally in the propagation encoding, also for other evaluation strategies, as long as we give precedence to the FD rule over

instance reductions. The memoing effect can also be achieved with the other encoding, but requires a specific evaluation strategy that gives precedence to (1) the FD rule and (2) instance reduction of *bigger* constraints.

2.7 RELATED WORK AND DISCUSSION

The original paper on functional dependencies [6] introduced two conditions (Termination and Coverage) [10] and conjectured that these conditions are sufficient to guarantee termination and confluence of the constraint solver underlying the type inferencer. This conjecture was formally verified in [1]. In practice, the Termination and Coverage Conditions are too limiting and rule out a large class of interesting programs.

The Weak Coverage Condition developed in [11] covers a much wider class of programs while guaranteeing confluence. However, the CHR-based encoding scheme could not deal properly with non-full FDs (and therefore a large class of interesting programs are ruled out again). We showed how to restore confluence for non-full FDs based on a type function encoding. As a side effect, we obtain a type-preserving translation of functional dependencies.

The propagation encoding (due to Claus Reinke) guarantees confluence for programs with relational dependencies which are beyond functional dependencies and type functions. The current implementation of checking type classes in GHC is actually quite close to the propagation encoding: it implements the same memoing technique.

However, evidence generation for propagation generation is a topic which has not been studied yet. Hence, the current GHC implementation of functional dependencies is incomplete, because improvement cannot be expressed in GHC's type-preserving core language. On the other hand, evidence generation for type functions is well understood by now [10]. Based on our type function encoding of functional dependencies we can therefore hope for a "more complete" implementation of functional dependencies in GHC (either by automatic or user-provided transformation from functional dependencies to type functions).

2.8 CONCLUSION AND FUTURE WORK

In summary, we have shown that the non-confluence of non-full FDs is not inherent. With an alternative encoding in terms of TFs, confluence can be established. More generally, we believe that the design space for FD inference algorithms is all but exhausted, and that the TF and FD language features should not be judged by the adequateness of current inference algorithms.

[10] We follow the notation used in [11].

Acknowledgements

We are grateful to Claus Reinke for explaining us his idea of using propagation rules for not forgetting improvement opportunities. Thanks to Ross Paterson and Bulat Ziganshin for kindly pointing out several practical examples of non-full FDs. The feedback of Claus Reinke and Mark Jones on this paper is greatly appreciated.

Tom Schrijvers is a post-doctoral researcher of the Fund for Scientific Research - Flanders (Belgium).

REFERENCES

[1] G. J. Duck, S. Peyton-Jones, P. J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *Proc. of ESOP'04*, volume 2986 of *LNCS*, pages 49–63. Springer-Verlag, 2004.

[2] G. J. Duck, P. J. Stuckey, M. J. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *Proc of ICLP'04*, volume 3132 of *LNCS*, pages 90–104. Springer-Verlag, 2004.

[3] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.

[4] C. V. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. In *ESOP'94*, volume 788 of *LNCS*, pages 241–256. Springer-Verlag, April 1994.

[5] M. P. Jones. Simplifying and improving qualified types. In *FPCA '95: Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 1995.

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

[7] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type Checking with Open Type Functions, 2008. Submitted to ICFP'08.

[8] T. Schrijvers, M. Sulzmann, M. M. T. Chakravarty, and S. Peyton Jones. Towards open type functions for Haskell, 2007. Presented at IFL'07.

[9] P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.

[10] 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.

[11] 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.