Chapter 2

Confluence for Non-Full Functional Dependencies

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

Abstract: Previous work on type inference for functional dependencies demands that the dependency must fully cover all parameters of a type class to guarantee that the constraint solver is confluent. However, several interesting programs rely on non-full functional dependencies, For these, the underlying constraint is non-confluent, and hence type inference for these programs is possibly ill-behaved.

We investigate two approaches to restore confluence for non-full FDs. In the first approach, we characterize a class of transformable non-full to full FD programs where the resulting full FD program is confluent. This approach has some inherent limitations due to the use of constraint simplification during type inference. In the second approach, we show how achieve confluence in general by applying a radically different type-inference approach which favors constraint propagation over simplification.

Our results provide new insights in type-inference issues behind functional dependencies and help to clarify some of the on-going discussions about the possible adoption of functional dependencies in a future Haskell standard.

2.1 INTRODUCTION

Haskell-style functional dependencies [4] provide a relational specification of user-programmable type improvement [3] connected to type-class instances. Functional dependencies are supported by both GHC and Hugs, and they are employed in numerous Haskell programs. The current state of the art on type inference for functional dependencies employs Constraint Handling Rules (CHRs) [2]. CHRs serve as a meta-language to describe the constraint solver underlying the type in-

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

ferencer. Confluence of CHRs is important to ensure that the type inferencer is well-behaved. Unfortunately, the CHR encoding of FDs proposed in [9] cannot guarantee confluence for non-full FDs.

In this paper, we attack the challenging problem of restoring confluence for non-full functional dependencies. In summary, we make the following contributions:

- We revisit the non-confluence issue of Constraint Handling Rules (CHRs) resulting from non-full FDs and provide realistic examples which show that non-full FD programs represent an interesting and useful class of FD programs (Section 2.3).
- We establish a confluence result for non-full FDs by transformation to full FDs. Full FD programs are generally confluent. Hence, by transformation to full FDs we obtain confluence for non-full FDs. However, only a restricted class of non-full FD programs are transformable. (Section 2.4).
- We achieve a less restrictive confluence result for non-full FD programs by applying a radically different type-inference approach which exclusively uses constraint propagation (Section 2.5).

The idea of the propagation encoding is due to Claus Reinke but to the best of our knowledge we are the first to formally investigate the implications of his encoding scheme.

The upcoming section gives an overview of type inference with functional dependencies based on CHRs. Related work is discussed in Section 2.6. We conclude in Section 2.7.

2.2 TYPE INFERENCE WITH FUNCTIONAL DEPENDENCIES

This section gives an overview of the necessary concepts and previous results that are needed for a good understanding of the technical results of this paper.

Functional Dependencies Functional dependencies [4] allow the programmer to influence the type-inference process.They improve types [3], and help to resolve ambiguities when translating type classes. We illustrate both points using the parameterized collections example from [4].

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

Type classes are compiled 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] Int` but an arbitrary choice leads to ambiguities. By imposing the functional dependency `| ce -> e` such situations do not happen because the functional dependency guarantees that fixing the collection type `ce` uniquely determines the element type `e`. So `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 type-class constraints `Collects ce e1` and `Collects ce e2`. Each constraints results from one of the two occurrences of `insert`. 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 x xs = insert x (tail xs)
```

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 is often hard to understand. Previous work makes use of Constraint Handling Rules (CHRs) [2] to give a precise and systematic description of context reduction and type improvement. According to [9], the above class and instance declarations translate to the following CHRs.

- For each functional dependency fd_i of the form $a_{i_1},...,a_{i_k}$ `->` a_{i_0}, in a class declaration `class` $TC\ a_1\ ...\ a_n$ `|` $fd_1,...,fd_m$, we generate

$$TC\ a_1 \ldots a_n,\ TC\ \theta(b_1) \ldots \theta(b_n)\ \texttt{==>}\ a_{i_0} = b_{i_0}$$

  where $a_1 \ldots a_n, b_1 \ldots b_n$ are distinct type variables,
  and $\theta(b_j) = \begin{cases} a_j & \text{if } j \in \{i_1, \ldots, i_k\} \\ b_j & \text{otherwise} \end{cases}$

- Each instance declaration `instance` C `=>` $TC\ t_1 \ldots t_n$ generates

$$TC\ t_1 \ldots t_n\ \texttt{<==>}\ C$$

  If the context C is empty, we introduce the always-satisfiable constraint *True* on the right-hand side of generated CHRs.

- In addition, for each functional dependency fd_i in the class declaration, of the form $a_{i_1},...,a_{i_k}$ `->` a_{i_0}, we generate

$$TC\ \theta(b_1) \ldots \theta(b_n)\ \texttt{==>}\ t_{i_0} = b_{i_0}$$

  where $b_1 \ldots b_n$ are distinct type variables,
  and $\theta(b_j) = \begin{cases} t_j & \text{if } j \in \{i_1, \ldots, i_k\} \\ b_j & \text{otherwise} \end{cases}$

FIGURE 2.1. Translation of FD programs to 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 from 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.

Figure 2.1 summarizes the translation to CHRs. We omit the treatment of superclasses for brevity. We write $FDToCHR(p)$ to denote the set of all CHRs generated from a FD program p.

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 constraints out of the program text and then solving 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 $C_g \supset C_w$ is logically equivalent to the equivalence check $C_g \Longleftrightarrow C_g \wedge C_w$. We can test for equivalence by normalizing (exhaustive CHR solving) the left-hand and right-side of $\Longleftrightarrow$ and checking the resulting constraints for syntactic equivalence. In case of the above type-annotated function, we effectively need to check for `Eq e` $\supset$ `Collects [e] e`. Normalization yields

$$\begin{array}{ll} & \texttt{Eq e, Collects [e] e} \\ \rightarrowtail_{Inst} & \texttt{Eq e} \end{array}$$

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 [7]. Termination of CHRs ensures that we can build normal forms in a finite number of steps. Confluence guarantees canonical normal forms: different derivations starting from the same point can always be brought together again.

Termination is notoriously hard to achieve [5] unless we impose rather harsh conditions on the set of allowable programs [9]. We therefore assume that guaranteeing termination is the user's responsibility. 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.

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 [9] to guarantee confluence of CHRs resulting from full functional-dependency programs are:

Definition 2.1 (Full Functional Dependencies). *We say the functional dependency* `class TC` $a_1 \ldots a_n | a_{i_1}, \ldots, a_{i_k}$ `->` a_{i_0} *for a type class* `TC` *is* full *iff* $k = n - 1$.

A functional dependency is full if all type parameters of a type class are covered by the functional-dependency relation.

Definition 2.2 (Weak Consistency Condition). *Consider a declaration for class TC and any pair of instance declarations for that class:*

$$\texttt{class}\, C \texttt{ => } TC\, a_1 \ldots a_n \texttt{ | } fd_1, \ldots, fd_m$$
$$\texttt{instance}\, D_1 \texttt{ => } TC\, t_1 \ldots t_n$$
$$\texttt{instance}\, D_2 \texttt{ => } TC\, s_1 \ldots s_n$$

Then, for each functional dependency fd_i, of form $a_{i_1}, \ldots, a_{i_k}$ -> a_{i_0}, the following condition must hold: for any substitution ϕ such that

$$\phi(t_{i_1}, \ldots, t_{i_k}) = \phi(s_{i_1}, \ldots, s_{i_k})$$

we must have that $\phi(t_{i_0})$ and $\phi(s_{i_0})$ are unifiable.

The Weak Consistency Condition says that the improvement conditions derived from instances must be non-conflicting. For example, the above program is inconsistent

```
class F a b | a -> b
instance F Int Float
instance F Int Bool
```

Consider the resulting CHR rules, in particular, the two last rules which conflict.

```
F a b, F a c ==> b = c
F Int Float  <=> True
F Int Bool   <=> True
F Int a      ==> a = Float
F Int a      ==> a = Bool
```

Inconsistency immediately implies non-confluence. Therefore, consistency is an essential condition.

The above condition is weaker than the (stronger) consistency condition in [9] which instead of "$\phi(t_{i_0})$ and $\phi(s_{i_0})$ are unifiable" requires that "$\phi(t_{i_0}) = \phi(s_{i_0})$". Both conditions are equivalent in case we apply the Coverage Condition [9].

Definition 2.3 (Coverage Condition). *Consider a declaration for class TC, and any instance declaration for that class:*

$$\texttt{class}\, TC\, a_1 \ldots a_n \texttt{ | } fd_1, \ldots, fd_m$$
$$\texttt{instance}\, C \texttt{ =>} TC\, t_1 \ldots t_n$$

Then, for each functional dependency $fd_i = a_{i_1}, \ldots, a_{i_k}$ -> a_{i_0}, we require that

$$fv(t_{i_0}) \subseteq fv(t_{i_1}, \ldots, t_{i_k})$$

In the above, we assume that function *fv* computes the set of free variables.

For many FD programs the Coverage Condition is overly restrictive. See [9] for more details. However, we can weaken the Coverage Condition as follows.

Definition 2.4 (Weak Coverage Condition). [3] *For each functional dependency* $a_{i_1},...,a_{i_k}$ `->` a_{i_0} *of class TC and*

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

we must have that $fv(t_{i_0}) \subseteq closure(C, fv(t_{i_1},\ldots,t_{i_k}))$*, where* $closure(C, vs)$ *is the least fix-point of the following equation:*

$$F(X) \quad = \quad vs \cup \left\{ fv(t_{i_0}) \;\middle|\; \begin{array}{l} TC\ t_1 \ldots t_n \in C, \\ TC \text{ has functional dependency } a_{i_1},...,a_{i_k} \texttt{ -> } a_{i_0}, \\ fv(t_{i_1},\ldots,t_{i_k}) \subseteq X \end{array} \right\}$$

In the above, we treat the instance context C as a set of (type class) constraints.

The Weak Coverage Condition says that all variables in the domain of the FD are uniquely determined by FDs in the instance context (plus building the transitive closure of those uniquely determined variables). In essence, the Weak Coverage Condition says that functional dependencies must behave "functionally".

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

Both instances satisfy the Weak Consistency Condition. But they do not satisfy the (stronger) Consistency Condition from [9]. This situation is possible, because the Coverage Condition is violated. For example, in `H [a] [b] Int` variable `b` is not covered by the first parameter `[a]`. However, the Weak Coverage Condition holds because `b` is covered by `H a b Int` from the instance context. The second instance violates the Weak Coverage Condition because the parameters of the `H` type class have been swapped. Breaking the Weak Coverage Condition immediately results in non-confluence of the resulting CHRs. See [9] for details.

The up-coming section examines what role (non)fullness plays for (non)confluence. Later Sections 2.4 and 2.5 investigate methods how to achieve confluence for non-full FDs.

2.3 THE CONFLUENCE PROBLEM OF NON-FULL FDS

Non-Full FDs The `Stream` type class from the `parsec` package [4] provides a generic interface for streams:

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

Here `s` is the stream, `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:

[3] Referred to as Refined Weak Coverage in [9].

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

```
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 (c,Cursor h p')
```

A recursive example, is the pairing of two streams (without methods for brevity):

```
instance (Stream s1 m t1, Stream s2 m t2) =>
        Stream (s1,s2) m (t1,t2)
```

An advanced example is the `BlockStream`, which chops the output of a stream in blocks of a certain size. If the underlying stream runs out of tokens in the middle of a block, the block is padded with tokens from a default source, which depends on the monad. In the case of the `IO` monad, the padding is taken from standard input. In the case of the state transformer monad `StateT`, the padding is taken from the state. The code looks looks like:

```
 data BlockStream s = BS { block_stream ::  s,
                           block_size ::  Int }
instance (Stream s IO t, Read t) =>
         Stream (BlockStream s) IO [t]
instance Stream s m t =>
         Stream (BlockStream s) (StateT [t] m) [t]
```

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 [9]:

```
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]
↣Inst     F a b Bool, F [a] [b] d, b2 = [b]         (2)

↣Inst     F a b Bool, F [a] b2 d
↣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 FULL FD ENCODING OF NON-FULL FDS

2.4.1 Transformable 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 [9]. In [9], already a class of *transformable* non-full FD is identified. A transformation is given to transform these non-full FDs in full FDs.

Rather than to copy the formal definition of the transformation, we illustrate the transformation by applying it to our 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
```

The transformation factors out the (non-full) FD `a -> b` into a separate type class `FD a b` with a full FD `a -> b`.

Unfortunately, the transformation of [9] is restricted to instances that do not overlap in the domain of the functional dependency. In the following, we will see a more general transformation.

2.4.2 Extended Transformation of Non-Full FDs

In general, the above transformation 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
```

After the transformation, we get:

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

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

The problem in this encoding is that the two instances of the type class `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 class `Sel` as follows.[5]

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

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

While the `F` type class's functional dependency has both `a` and `c` in its domain, it only matches on `a` in the instance. The latter instantiates the `b` parameter with the top-level type constructor `[]`, and delegates to the selector type class `Sel`. Now `Sel` is allowed to match on `c` to choose the appropriate instance.

Formally, the class of programs our transformation addresses is:

Definition 2.5 (Extended Transformable Non-Full FDs). *Consider a class declaration*

$$\texttt{class}\, C \texttt{ => } TC\, a_1 \ \ldots\ a_n \mid a_1, \ldots, a_k \texttt{ -> } a_{k+1}$$

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

where $k+1 < n$. We say that `TC` *is extended transformable to full FDs iff for every two declarations* `instance C => TC` $t_1 \ldots t_n$ *and* `instance C' => TC` $t'_1 \ldots t'_n$ *and such that $\phi(t_1) = \phi(t'_1), \ldots, \phi(t_k) = \phi(t'_k)$ for some substitution ϕ, there exist two alpha-renamings α_1 and α_2 such that $\alpha_1(t_1) = t'_1, \ldots, \alpha_1(t_k) = t'_k$ and $t_1 = \alpha_2(t'_1), \ldots, t_k = \alpha_2(t'_k)$.*

The extended transformation is then:

Definition 2.6 (Extended Non-Full to Full FD Transformation). *Consider an FD program p with a class declaration*

$$\texttt{class}\, C \texttt{ => } \texttt{TC}\, a_1 \ \ldots\ a_n \mid a_1, \ldots, a_k \texttt{ -> } a_{k+1}$$

where $k+1 < n$. If `TC` *is extended transformable, then the transformation, denoted xNFToF(p), consists of:*

- *One class declaration:*

$$\texttt{class}\, C \texttt{ => } \texttt{TC}\, a_1 \ \ldots\ a_n \mid a_1, \ldots, a_k, a_{k+2}, \ldots, a_n \texttt{ -> } a_{k+1}$$

- *For each sequence $t_1, \ldots, t_k, t_{k+1}$ of a* `TC` *instance head that is unique modulo variable renaming, we get one new instance of* `TC`*:*

$$\texttt{instance Sel}_j\, t_1 \ \ldots\ t_{k+1}\, a_{k+2} \ \ldots\ a_n \texttt{ => TC } t_1 \ \ldots\ t_{k+1}\, a_{k+2} \ \ldots\ a_n$$

  and a class declaration for `Sel`$_j$*, which is a fresh type class name:*

$$\texttt{class Sel}_j\, a_1 \ \ldots\ a_n \mid a_1, \ldots, a_k, a_{k+2}, \ldots, a_n \texttt{ -> } a_{k+1}$$

- *For each original* `instance C => TC` $t_1 \ \ldots\ t_n$ *a new instance of the corresponding* `Sel`$_j$ *type class:*

$$\texttt{instance}\, C \texttt{ => Sel}_j\, t_1 \ \ldots\ t_n$$

The transformation establishes confluence, because it yields only full FDs and the original program's instance heads in their entirety were non-overlapping.

Theorem 2.7 (Confluence). *Let p be an extended transformable non-full FD program. Then, FDToCHR(xNFToF(p)) is confluent iff p satisfies the Weak Consistency Condition.*

It is interesting to compare the relative strengths of the original and transformed program with the standard translation *FDToCHR*$(\cdot)$. We measure the strength in terms of the logical (first-order) meaning of CHRs [2].

Definition 2.8 (CHR Logical Meaning). *The translation function $[\![\cdot]\!]$ from CHR rules to first-order formulas is:*

$$\begin{array}{rcl} [\![c \texttt{ <==> } d_1, \ldots, d_m]\!] & = & \forall \bar{a}'(c \leftrightarrow (\exists \bar{\beta}\ d_1 \wedge \cdots \wedge d_m)) \\ [\![c_1, \ldots, c_n \texttt{ ==> } d_1, \ldots, d_m]\!] & = & \forall \bar{a}(c_1 \wedge \cdots \wedge c_n \supset (\exists \bar{\beta}\ d_1 \wedge \cdots \wedge d_m)) \end{array}$$

where $\bar{a}' = fv(c)$, $\bar{a} = fv(c_1 \wedge \cdots \wedge c_n)$ and $\bar{\beta} = fv(d_1 \wedge \cdots \wedge d_m) - \bar{a}$. In the above, we assume that $\leftrightarrow$ and $\supset$ denote Boolean equivalence and implication.

It turns out that the transformed program's theory is weaker than that of the original program.

Theorem 2.9 (Soundness). *Let p be an extended transformable non-full FD program. Then, $[\![FDToCHR(p)]\!], \mathscr{T} \models [\![FDToCHR(xNFToF(p))]\!]$, where $\mathscr{T}$ relates the intermediate type-class symbols $\texttt{Sel}_j$ to the main type-class symbols $\texttt{TC}$. For each*

$$\texttt{instance Sel}_j\ t_1\ \ldots\ t_{k+1}\ a_{k+2}\ \ldots\ a_n\ \texttt{=>}\ \texttt{TC}\ t_1\ \ldots\ t_{k+1}\ a_{k+2}\ \ldots\ a_n$$

in $xNFToF(p)$, $\mathscr{T}$ contains an axiom:

$$\forall \overline{a}.\texttt{Sel}_j\ t_1\ \ldots\ t_{k+1}\ a_{k+2}\ \ldots\ a_n \leftrightarrow \texttt{TC}\ t_1\ \ldots\ t_{k+1}\ a_{k+2}\ \ldots\ a_n$$

where $\overline{a} = fv(t_1, \ldots, t_{k+1}, a_{k+2}, \ldots, a_n)$

Of course, we want the transformed program's theory to be weaker, because the original program's theory may be inconsistent, equating too many types. For instance, if we extend our running example with some additional instances:

```
instance F Int Foo Bool
instance H Int Bar
```

Then from `F [Int] b1 Bool` and `F [Int] b2 Char`, we could conclude `b1 = b2 = Foo = Bar` for the original program, which is clearly inconsistent. As it should be, this is not the case for the transformed program.

Our attempts to further relax the class of transformable non-full FDs based on the $FDToCHR(\cdot)$ translation have failed. Even with increasingly complex non-full to full transformations, we could not establish confluence.

2.5 PROPAGATION TRANSLATION OF NON-FULL FDS

A different encoding of FDs, in terms of propagation CHRs only, was suggested by Claus Reinke on the Haskell-Prime mailing list. Below we formalize his idea and state important results.

The basic idea of Claus Reinke's translation scheme is to replace <=> (simplification) in the (Inst) rule with ==> (propagation). Under this propagation-only translation scheme, the earlier "problematic" non-full FD example

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

The propagation translation of FDs is short and elegant. Moreover, it directly supports improper FDs. Recall the earlier improper FD programs

```
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[6]

```
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)
```

Definition 2.10 (Propagation Translation). *The propagation translation of an FD program p, denoted FDToPropCHR(p), is FDToCHR(p) where all <==> arrows are substituted by ==>.*

We easily obtain the desired confluence result for non-full FDs. A CHR program with only propagation rules is trivially confluent; there are no critical pairs.

Theorem 2.11 (Confluence of Propagation Translation). *Let p be an FD program. Then, the set FDToPropCHR(p) of CHRs is confluent.*

We also compare the relative strengths of *FDToPropCHR*($\cdot$), the propagation translation, with *FDToCHR*($\cdot$), the standard translation. We refer to the latter as the *standard theory* and to the former as the *propagation theory*. It straightforwardly follows that the propagation theory is weaker than the standard theory.

Theorem 2.12 (Standard Entails Propagation Translation). *Let p be a FD program. Then,* $[\![FDToCHR(p)]\!] \models [\![FDToPropCHR(p)]\!]$.

An immediate consequence is that any improvement in the propagation theory also holds in the standard theory.

Corollary 2.13 (Soundness of Propagation Improvement). *Let p be a FD program, C be a set of type-class constraints and t_1 and t_2 two types.*
If $[\![FDToPropCHR(p)]\!] \models C \supset t_1 = t_2$ *then* $[\![FDToCHR(p)]\!] \models C \supset t_1 = t_2$.

However, the other direction does not hold in general as shown by the program

```
class F a b | a -> b
instance G a b => F a b
```

In the standard theory, the constraint set `G a b, F a c` implies `b = c` which is not the case in the propagation theory because from `G a b` we cannot conclude `F a b`.

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

This is not surprising because while the set $FDToPropCHR(p)$ is confluent in general, hence, the propagation theory is consistent in the logical sense, the set $FDToCHR(p)$ is non-confluent unless we impose fullness of FDs and the Weak Consistency and Coverage Conditions. The above program violates the Weak Coverage Condition and the standard theory becomes even inconsistent if we include

```
instance G Int Int
instance G Int Bool
```

From `G Int Int, G Int Bool` we derive `G Int Int, F Int Bool` which implies `Int=Bool`.

However, we can establish that the propagation theory entails the same amount of improvement as the standard theory if the standard translation yields a terminating and confluent set of CHRs.

Theorem 2.14 (Completeness of Propagation Improvement). *Let p be a FD program, C be a set of type-class constraints and t_1 and t_2 two types such that $FDToCHR(p)$ is terminating and confluent. If $[\![FDToCHR(p)]\!] \models C \supset t_1 = t_2$ then $[\![FDToPropCHR(p)]\!] \models C \supset t_1 = t_2$.*

Proof. (Sketch) Under assumptions $[\![FDToCHR(p)]\!] \models C \supset t_1 = t_2$ and termination and confluence we can apply the completeness of entailment result stated in [7] which yields $C \rightarrowtail^*_{FDToCHR(p)} C', t_1 = t_2$ for some appropriate C'. Any derivation step in $FDToCHR(p)$ is also possible in $FDToPropCHR(p)$, the only difference being that derivations in $FDToPropCHR(p)$ never delete constraints. Hence, we can conclude that $C \rightarrowtail^*_{FDToPropCHR(p)} C'', t_1 = t_2$ for some appropriate C''. The logical soundness of CHR results in [2] then immediately yields $[\![FDToPropCHR(p)]\!] \models C \supset t_1 = t_2$ which concludes the proof.

The above results show that the propagation encoding is the superior translation scheme. If the standard translation is well-behaved, i.e. terminating and confluent, then the propagation translation behaves similarly. If the standard translation is ill-behaved, then the propagation translation is still well-behaved. Below, we discuss further the practical consequences of using the propagation encoding.

Time vs. Space Trade-Off The propagation encoding potentially uses more space than the standard 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, i.e. the refined operational semantics [1], exhibits $\mathcal{O}(2^n)$ 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.

2.6 RELATED WORK

The original paper on functional dependencies [4] introduced two conditions (Consistency and Coverage) 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 [9]. However, 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 [9] 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). In this paper, we have shown how to restore confluence for a large class of non-full FDs based on a transformation scheme to confluent full FDs.

The propagation encoding (suggested by Claus Reinke) even guarantees confluence for FD programs without imposing any restrictions. The current implementation of checking type classes in GHC is already fairly close to the propagation encoding: it implements a similar memoing technique.

2.7 CONCLUSION AND FUTURE WORK

We have shown that the non-confluence of non-full FDs is not inherent. Confluence for non-full FDs can be restored based on alternative encoding schemes.

The precise results we have stated relate the alternative encoding schemes to the original FD to CHR scheme introduced in [9].

The evidence translation of functional dependencies is still an open issue. We believe that the non-full to full FD transformation scheme in combination with results reported in [6] make it possible to translate a significant set of FD programs to type families (also known as type functions) [6, 5]. Type families are the "functional" equivalent of FDs. Their evidence translation is well-studied [8] and already implemented in GHC.

Acknowledgments

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, 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.

[2] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, 1998.

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

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

[5] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type Checking with Open Type Functions. In Peter Thiemann, editor, *International Conference on Functional Programming*, pages 51–62. ACM Press, 2008.

[6] T. Schrijvers, M. Sulzmann, M. M. T. Chakravarty, and S. Peyton Jones. Towards open type functions for Haskell. In Olaf Chitil, editor, *Implementation and Application of Functional Languages*, pages 233–251, 2007. Published as Tech. Report No. 12-97, Computing Laboratory, University of Kent.

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

[8] 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.

[9] 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.