Sound and Decidable Type Inference for Functional Dependencies

Gregory J. Duck[1], Simon Peyton-Jones[2], Peter J. Stuckey[1] and Martin Sulzmann[3]

[1] Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
`{gjd,pjs}@cs.mu.oz.au`
[2] Microsoft Research Ltd
7 JJ Thomson Avenue, Cambridge CB3 0FB, England
`simonpj@microsoft.com`
[3] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
`sulzmann@comp.nus.edu.sg`

Abstract. Functional dependencies are a popular and useful extension to Haskell style type classes. In this paper, we give a reformulation of functional dependencies in terms of Constraint Handling Rules (CHRs). In previous work, CHRs have been employed for describing user-programmable type extensions in the context of Haskell style type classes. Here, we make use of CHRs to provide for the first time a concise result that under some sufficient conditions, functional dependencies allow for sound and decidable type inference. The sufficient conditions imposed on functional dependencies can be very limiting. We show how to safely relax these conditions.

1 Introduction

Functional dependencies, introduced by Mark Jones [Jon00], have proved to be a very attractive extension to multi-parameter type classes in Haskell. For example, consider a class intended to describe a collection of type `c` containing values of type `e`:

```
class Coll c e | c -> e where
  empty :: c
  insert :: c -> e -> c
  member :: c -> e -> Bool
```

The part "`| c->e`" is a functional dependency, and indicates that fixing the collection type `c` should fix the element type `e`. These functional dependencies have proved very useful, because they allow the programmer to control the type inference process more precisely. We elaborate in Section 2.

The purpose of this paper is to explore and consolidate the design space of functional dependencies (FDs). The main tool we use in this exploration is the reformulation of FDs in terms of *Constraint Handling Rules* (CHRs) [Frü95,SS02], an idea that we review in Section 3. This reformulation allows us to make several new contributions:

- Despite their popularity, functional dependencies have never been formalised, so far as we know. CHRs give us a language in which to explain more precisely what functional dependencies *are*. In particular, we are able to make the so-called "improvement rules" implied by FDs explicit in terms of CHRs.
- Based on this understanding, we provide the first concise proof that the restrictions imposed by Jones on functional dependencies [Jon00] ensure sound and decidable type inference (Section 3).
- Jones's restrictions can be very limiting. We propose several useful extensions (Section 4) such as *more liberal FDs* (Section 4.1). We establish some concise conditions under which liberal FDs are sound.

Throughout, we provide various examples to support the usefulness of our improvement strategies. Related work is discussed in Section 5. We conclude in Section 6.

We refer the interested reader to [DPJSS03] for proofs and additional material.

2 Background: Functional Dependencies in Haskell

We begin by reviewing functional dependencies, as introduced by Jones [Jon00], assuming some basic familiarity with Haskell-style type classes.

Example 1. Recall the collection class

```
class Coll c e | c -> e where
  empty :: c
  insert :: c -> e -> c
  member :: c -> e -> Bool
```

plus the following

```
instance Eq a => Coll [a] a where ...
ins2 xs a b = insert (insert xs a) b
```

Consider the function `ins2`. In the absence of functional dependencies, type inference would give

```
ins2 :: (Coll c e1, Coll c e2) => c -> e1 -> e2 -> c
```

which is of course not what we want: we expect `a` and `b` to have the same type. The functional dependency `c->e` expresses the idea that the collection type `c` fixes the element type `e`, and hence that `e1` and `e2` must be the same type. In such a situation, we commonly say that types are "*improved*" [Jon95].

Functional dependencies are useful in many different contexts. Here are some representative examples.

Example 2. Consider the following class for representing state monads and two instances

```
class SM m r | m->r, r->m where
  new   :: a -> m (r a)
  read  :: r a -> m a
  write :: r a -> a -> m ()

instance SM IO IORef where
  new   = newIORef
  read  = readIORef
  write = writeIORef

instance SM (ST s) (STRef s) where
  new   = newSTRef
  read  = readSTRef
  write = writeSTRef
```

The part "`| m->r, r->m`" gives two functional dependencies, and indicates that fixing the monad type `m` should fix the reference type `r` as well, and vice versa. Now consider the code

```
f x = do { r <- new x; print "Hello"; return r }
```

The call to `print`, whose type is `String -> IO ()`, makes it clear that `f` is in the `IO` monad, and hence, by the functional dependency, that `r` must be an `IORef`. So we infer the type

```
f :: a -> IO (IORef a)
```

From this example we can see the main purpose of functional dependencies: they allow the programmer to place stronger conditions on the set of constraints generated during type inference, and thus allow more accurate types to be inferred. In their absence, we would infer the type

```
f :: (SM IO r) => IO (r a)
```

which is needlessly general. In other situations, ambiguity would be reported. For example:

```
g :: a -> IO a
g x = do { r <- new x ; read r }
```

Without functional dependencies, the type system cannot work out which reference type to use, and so reports an ambiguous use of `new` and `read`.

Example 3. Consider the following application allowing for (overloaded) multiplication among base types such as `Int` and `Float` and user-definable types such as vectors. For simplicity, we omit the obvious function bodies.

```
class Mul a b c | a b -> c where
    (*)::a->b->c
instance Mul Int Int Int where ...
instance Mul Int Float Float where ...
type Vec b = [b]
instance Mul a b c => Mul a (Vec b) (Vec c) where ...
```

The point here is that the argument types of (*) determine its result type. In the absence of this knowledge an expression such as (a*b)*c cannot be typed, because the type of the intermediate result, (a*b), is not determined. The type checker would report type ambiguity, just as it does when faced with the classic example of ambiguity, (read (show x)).

Example 4. Here is an another useful application of FDs to encode a family of zip functions.

```
zip2 :: [a]->[b]->[(a,b)]
zip2 (a:as) (b:bs) = (a,b) : (zip2 as bs)
zip2 _      _      = []

class Zip a b c | a c -> b, b c -> a where
    zip :: [a] -> [b] -> c
instance Zip a b [(a,b)] where
    zip = zip2
instance Zip (a,b) c e => Zip a b ([c]->e) where
    zip as bs cs = zip (zip2 as bs) cs
```

These definitions make zip into an n-ary function. For example, we may write

```
e1 :: (Bool,Char)
e1 = head (zip [True,False] ['a','b','c'])
e2 :: ((Bool,Char),Int)
e2 = head (zip [True,False] ['a','b','c'] [1::Int,2])
```

2.1 Functional Dependencies are Tricky

As we have seen, functional dependencies allow the programmer to exert control over the type inference process. However, used uncritically, this additional control can have unexpected consequences. Specifically: they may lead to *inconsistency*, whereby the type inference engine deduces nonsense such as Int = Bool; and they may lead to *non-termination*, whereby the type inference engine goes into an infinite loop. We illustrate each of these difficulties with an example.

Example 5. Suppose we add instance Mul Int Float Int to Example 3. That is, we have the following declarations:

```
class Mul a b c | a b -> c
instance Mul Int Float Float  -- (I1)
instance Mul Int Float Int    -- (I2)
```

Note that the first two parameters are meant to uniquely determine the third parameter. In case type inference encounters Mul Int Float a we can either argue that a=Int because of instance declaration (I2). However, declaration (I1) would imply a=Float. These two answers are inconsistent, so allowing both (I1) and (I2) makes the whole program inconsistent, which endangers soundness of type inference.

Example 6. Assume we add the following function to the classes and instances in Example 3.

```
f b x y = if b then (*) x [y] else y
```

The program text gives rise to the constraint `Mul a (Vec b) b`. The improvement rules connected to `instance Mul a b c => Mul a (Vec b) (Vec c)` imply that `b=Vec c` for some `c`; applying this substitution gives the constraint `Mul a (Vec (Vec c)) (Vec c)`. But this constraint can be simplified using the instance declaration, giving rise to the simpler constraint `Mul a (Vec c) c`. Unfortunately, now the entire chain of reasoning simply repeats! We find that type inference becomes suddenly non-terminating. Note that the instances (without the functional dependency) are terminating.

The bottom line is this. We want type inference to be *sound* and *decidable*. Functional dependencies threaten this happy situation. The obvious solution is to place restrictions on how functional dependencies are used, so that type inference remains well-behaved, and that is what we discuss next.

2.2 Jones's Functional Dependency Restrictions

We assume that $fv(t)$ takes a syntactic term t and returns the set of *free variables* in t. A *substitution* $\theta = [t_1/a_1, \ldots, t_n/a_n]$ simultaneously replaces each a_i by its corresponding t_i.

In Jones's original paper [Jon00], the following restrictions are imposed on functional dependencies.

Definition 1 (Haskell-FD Restrictions). *Consider a class declaration*

$$\texttt{class}\ C\ \texttt{=>}\ \texttt{TC}\ a_1\ ...\ a_n\ \texttt{|}\ fd_1, ..., fd_m$$

where the a_i are type variables and C is the class context consisting of a (possibly empty) set of type class constraints. Each fd_i is a functional dependency *of the form*[4] $a_{i_1}, ..., a_{i_k}$ `->` a_{i_0} *where* $\{i_0, i_1, ..., i_k\} \subseteq \{1...n\}$. *We commonly refer to* $a_{i_1}, ..., a_{i_k}$ *as the* domain *and* a_{i_0} *as the* range.

The following conditions must hold for functional dependency fd_i:

Consistency. *Consider every pair of instance declarations*

$$\texttt{instance}\ ...\ \texttt{=>}\ \texttt{TC}\ t_1\ ...\ t_n$$
$$\texttt{instance}\ ...\ \texttt{=>}\ \texttt{TC}\ s_1\ ...\ s_n$$

for a particular type class TC. Then, for any substitution ϕ such that

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

we must have that $\phi(t_{i_0}) = \phi(s_{i_0})$.

Termination. *For each* `instance` ... `=>` `TC` t_1 ... t_n *we must have that*

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

[4] Haskell systems that allow functional dependencies usually allow dependencies of the form `a -> b c`, with multiple type variables to the right of the arrow. But this is equivalent to the form `a -> b, a -> c`, so in the rest of the paper we only deal with the case where there is a single type variable to the right of the arrow.

The first of these conditions rules out inconsistent instance declarations (see Example 5); and it turns out that the second ensures termination, although the informal argument in Jones's original paper does not mention termination as an issue. In particular, the second restriction makes illegal the recursive `Vec` instance in Example 3 (since $fv(c) \not\subseteq fv(a, b)$), and hence prevents the divergence of Example 6.

To the best of our knowledge, no one has proved that the restrictions given above ensure sound and decidable type inference. We do so, for the first time, in Section 3.

While these two restrictions make the system well-behaved, it is natural to ask whether either condition could be weakened. The consistency condition seems entirely reasonable, but we have seen many examples in which the termination restriction excludes entirely reasonable and useful programs. Besides Examples 3 (which appears in Jones's original paper) and 4, there are a number of other examples in the literature which break the termination condition [Kar03,WW03,CK03]. In Section 4.1, we propose a more liberal form of FDs which allows for breaking the termination condition under some additional conditions.

3 Functional Dependencies expressed using CHRs

In this section we explain how to translate functional dependencies into a lower-level notation, called *Constraint Handling Rules* (CHRs) [Frü98]. This translation has two benefits: it allows us to give a more precise account of exactly what functional dependencies mean; and it allows us to formally verify that Jones's conditions are sufficient to ensure sound and decidable type inference.

Example 7. Let us return to the collection example:

```
class Coll c e | c -> e where
  empty :: c
  insert :: c -> e -> c
  member :: c -> e -> Bool

class Eq a => Ord a where
  (>=) :: a -> a -> Bool

instance Ord a  => Coll [a] a where ...
```

From the functional dependency `c->e` we generate the two improvement rules which we shall express using the following CHRs:

```
rule  Coll c e1, Coll c e2 ==> e1=e2
rule  Coll [a] b           ==> a=b
```

Informally, the first rule says that if the two constraints `(Coll c e1)` and `(Coll c e2)` both hold, then it must be that `e1` and `e2` are the same type. This rule is generated from the `class` declaration alone, and expresses the idea that `c` uniquely determines `e`. The second rule is generated from the `instance` declaration, together with the functional dependency, and states that if `(Coll [a] b)` holds, then it follows that `a = b`. During type inference, the inference engine is required to solve sets of constraints, and it can apply these improvement rules to narrow its choices.

These CHRs have one or more type-class constraints on the left hand side, and one or more equality constraints on the right. The *logical* interpretation of `==>` is implication. Its *operational* interpretation — that is, its effect on the type inference process — is this: when the type inference engine sees constraints matching the left hand side, it adds the constraints found on the right-hand side.

Superclass relations also generate CHR rules. The superclass relationship `class Eq a => Ord a where...` generates the CHR

```
rule Ord a ==> Eq a
```

Informally, the rule states that if the constraint `Ord a` holds then also the constraint `Eq a` holds. During typing this rule is used to check that all superclass constraints are also satisfied.

The instance declaration above also generates the following CHR rule, which allows us to simplify sets of constraints to remove class constraints which are known to hold.

```
rule Coll [a] a <==> Ord a
```

Informally, the rule states that the constraint `Coll [a] a` holds if and only if `Ord a` holds. The logical interpretation of the `<==>` is bi-implication, while the operational interpretation is to replace the constraints on the left hand side by those on the right hand side.

Although not relevant to the content of this paper, the rule generated from the instance is also intimately connected to the evidence translation for the program above, we refer readers to [SS02] for more details.

3.1 Translation to CHRs

Formalising the translation given above, `class` and `instance` declarations are translated into CHRs as follows:

Definition 2 (CHR Translation). *Consider a class declaration*

$$\texttt{class}\ C\ \texttt{=>}\ \texttt{TC}\ a_1\ ...\ a_n\ \texttt{|}\ fd_1, ..., fd_m$$

where the a_i are type variables and each functional dependency fd_i is of the form $a_{i_1}, ..., a_{i_k}$ `->` a_{i_0}, where $\{i_0, i_1, ..., i_k\} \subseteq \{1...n\}$. From the class declaration we generate the following CHRs:

Class CHR: `rule TC` $a_1 \ldots a_n$ `==>` C

Functional dependency CHR: *for each functional dependency fd_i in the class declaration, we generate*

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

where $\theta(b_{i_j}) = a_{i_j}$, $j > 0$ and $\theta(b_l) = b_l$ if $\neg\exists j.l = i_j$.

In addition, for each instance declaration of the form `instance` C `=>` `TC` $t_1 \ldots t_n$

we generate the following CHRs:

Instance CHR: `rule TC` $t_1 \ldots t_n$ `<==>` C . *In case the context C is empty, we introduce the always-satisfiable constraint $True$ on the right-hand side of generated CHRs.*

Instance improvement CHR: *for each functional dependency fd_i in the class declaration,*

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

where $\theta(b_{i_j}) = t_{i_j}$, $j > 0$ and $\theta(b_l) = b_l$ if $\neg\exists j.l = i_j$.

If p is a set of `class` *and* `instance` *declarations, we define* Simp(p) *to be the set of all instance CHRs generated from p; and* Prop(p) *to be the set of all class, functional-dependency and instance-improvement CHRs generated from p*[5]*. We define* Prop_{class}(p) *to be the set of all class CHRs in* Prop(p)*, and similarly* Prop_{inst}(p) *to be the set of all instance improvement CHRs in* Prop(p)*,*

The class and instance CHRs, $Prop_{class}(p) \cup Simp(p)$, are standard Haskell, while the functional-dependency and instance-improvement CHRs arise from the functional-dependency extension to Haskell.

For convenience, in the case where the functional dependency $a_{i_1}, ..., a_{i_k}$ `->` a_{i_0} imposed on `TC` is *full*, that is, when $k = n - 1$, we are able to combine the instance improvement and instance rule into one rule. In such a situation, for each `instance` C `=>` `TC` $t_1 \ldots t_n$ and full functional dependency $a_{i_1}, ..., a_{i_k}$ `->` a_{i_0} we generate the following CHR: `rule TC` $\theta(b_1) \ldots \theta(b_n)$ `<==>` $t_{i_0} = b_{i_0}$ `, C` where $\theta(b_{i_j}) = t_{i_j}$, $j > 0$ and $\theta(b_l) = b_l$ if $\neg\exists j.l = i_j$.

By having a uniform description of (super) class and instance relations and FDs in terms of CHRs, we can establish some important criteria (in terms of CHRs) under which type inference is sound and decidable.

3.2 Main Result

The translation to CHRs allows us to phrase the entire type inference process as CHR solving. We know from earlier work that if a set of CHRs is (a) *confluent*, (b) *terminating*, and (c) *range-restricted* (all terms that we explain shortly) we achieve type inference that is *sound* (all answers are correct), *complete* (if there is an answer then type inference will provide us with an answer), and *decidable* (the type inference engine always terminates) [SS02].

Then our main result is as follows:

Theorem 1 (Soundness and Decidability). *Let p be a set of Haskell class and instance declarations which satisfies the Haskell-FD restrictions (see Definition 1). Let Simp(p) and Prop(p) be the sets of CHRs defined by Definition 2. If the set $Prop_{class}(p) \cup Simp(p)$ of CHRs is confluent, terminating and range-restricted then $Simp(p) \cup Prop(p)$ is confluent, terminating and range-restricted.*

[5] "Simp" is short for "simplification rule" and "Prop" for "propagation rule", terminology that comes from the CHR literature.

The design of Haskell 98 ensures that the CHRs $Prop_{class}(p) \cup Simp(p)$, which represent the Haskell type system with no FD extension, are indeed confluent, terminating and range-restricted. Hence, our theorem says that provide the FDs satisfy the Jones restrictions, then type inference is sound and decideable.

To explain this result we need to say what we mean for a set of CHRs to be confluent, terminating, and range restricted.

Confluence Recall Example 5 whose translation to CHRs is as follows (note that the functional dependency is fully imposed).

```
rule Mul a b c, Mul a b d ==> c=d  -- (M1)
rule Mul Int Float c <==> c=Float  -- (M2)
rule Mul Int Float c <==> c=Int    -- (M3)
```

We find two contradicting CHR derivations. We write $C \rightarrowtail_R D$ to denote the CHR derivation which applies rule (R) to constraint store C yielding store D. E.g. consider $Mul\ Int\ Float\ c \rightarrowtail_{M2} c = Float$ and $Mul\ Int\ Float\ c \rightarrowtail_{M3} c = Int$. The problem with the code of Example 5 manifests itself in the CHR rules as *non-confluence*. That is there are two possible sequences of applying rules, that lead to different results. Just considering the rules as logical statements, the entire system is unsatisfiable; that is, there are no models which satisfy the above set of rules.

Non-confluence also arises in case of "overlapping" instances. Assume we add the following declaration to the code of Example 7.

```
instance Eq a ==> Coll [a] a where
```

In case type inference encounters `Coll [t] t` we can either reduce this constraint to `Ord t` (by making use of the original instance) or `Eq t` (by making use of the above instance). However, both derivations are non-joinable. In fact, a common assumption is that instances must be non-overlapping, in which case non-confluence only occurs due to "invalid" FDs.

We note that the consistency condition *alone* is not sufficient to guarantee confluence (assuming that instances and super classes are already confluent of course).

Example 8. The following code fragment forms part of a type-directed evaluator.

```
data Nil        = Nil
data Cons a b   = Cons a b
data ExpAbs x a = ExpAbs x a
-- env represents environment, exp expression
-- and t is the type of the resulting value
class Eval env exp t | env exp -> t where
    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
```

The translation to CHRs yields

```
rule Eval env exp t1, Eval env exp t2 ==> t1=t2  -- (E1)
rule Eval env (ExpAbs x exp) v <==>
     v=(v1->v2), Eval (Cons (x,v1) env) exp v2   -- (E2)
```

Note that the termination condition is violated but the consistency condition is trivially fulfilled (there is only one instance). However, we find that CHRs are terminating but non-confluent. E.g. we find that (applying (E2) twice)

$$\begin{array}{ll} & Eval\ env\ (ExpAbs\ x\ exp)\ t_1, Eval\ env\ (ExpAbs\ x\ exp)\ t_2 \\ \rightarrowtail^* & t_1 = v_1 \rightarrow v_2, Eval\ (Cons\ (x, v_1)\ env)\ exp\ v_2, \\ & t_2 = v_3 \rightarrow v_4, Eval\ (Cons\ (x, v_3)\ env)\ exp\ v_4 \end{array}$$

Note that rule (E1) cannot be applied on constraints in the final store. But there is also another non-joinable derivation (applying rule (E1) then (E2))

$$\begin{array}{ll} & Eval\ env\ (ExpAbs\ x\ exp)\ t_1, Eval\ env\ (ExpAbs\ x\ exp)\ t_2 \\ \rightarrowtail^* & t_1 = t_2, t_1 = v_5 \rightarrow v_6, Eval\ (Cons\ (x, v_5)\ env)\ exp\ v_6 \end{array}$$

So the "termination condition" is perhaps mis-named; in this example, its violation leads to non-confluence rather than non-termination.

Termination Recall Example 3. The translation to CHRs yields (among others) the following.

```
rule Mul a (Vec b) d <==> d=Vec c, Mul a b c -- (M4)
```

The program text in Example 6 gives rise to `Mul a (Vec b) b`. We find that

$$\begin{array}{ll} & Mul\ a\ (Vec\ b)\ b \\ \rightarrowtail_{M4} & Mul\ a\ (Vec\ c)\ c, c = Vec\ b \\ \rightarrowtail_{M4} & Mul\ a\ (Vec\ d)\ d, d = Vec\ c, c = Vec\ b \\ \ldots & \end{array}$$

That is, the CHR derivation, and hence type inference, is non-terminating. The important point here is that non-termination was introduced through the FD.

For the purpose of this paper, we generally assume that instance CHRs are terminating. There exists some sufficient criteria to ensure that instance CHRs are terminating, e.g. consider [Pey99]. Clearly, we can possibly identify further classes of terminating instance CHRs which we plan to pursue in future work. Note that, when a set of CHRs are terminating, we can easily test for confluence by checking that all "critical pairs" are joinable [Abd97].

Range restriction Range-restrictedness is the third condition we impose on CHRs. We say a CHR is *range-restricted* iff grounding all variables on the left-hand side of a CHR, grounds all variables on the right-hand side.

Example 9. Consider

```
class C a b c
class D a b
instance C a b c => D [a] [b]
```

Our translation to CHRs yields

```
rule D [a] [b] <==> C a b c  -- (D1)
```

Note that rule (D1) is not range-restricted. After grounding the left-hand side, we still find non-ground variable `c` on the right-hand side. Range-restrictedness ensures that no unconstrained variables are introduced during a derivation and is a necessary condition for complete type inference. We refer readers to [SS02] for more details.

4 Extensions

In turn we discuss several extensions and variations of functional dependencies.

4.1 More Liberal Functional Dependencies

Earlier in the paper we argued that, while Jones's consistency condition is reasonable, the termination condition is more onerous than necessary, because it excludes reasonable and useful programs (Section 2.2). In this section we suggest replacing the termination restriction with the following weaker one, with the goal of making these useful programs legal.

Definition 3 (Liberal-FD). *Consider a class declaration*

$$\texttt{class}\ C\ \texttt{=>}\ \texttt{TC}\ a_1\ ...\ a_n\ \texttt{|}\ fd_1, ..., fd_m$$

where the a_i are type variables and C is the class context consisting of a (possibly empty) set of type class constraints. Each fd_i is a functional dependency *of the form $a_{i_1}, ..., a_{i_k}$ `->` a_{i_0} where $\{i_0, i_1, ..., i_k\} \subseteq \{1...n\}$.*

In addition to the consistency condition (see Definition 1), the following condition must hold for more liberal functional dependency fd_i:

Context Consistency. *For each* `instance` C `=>` `TC` $t_1\ ...\ t_n$ *we must have that* $fv(t_{i0}) \subseteq closure(C, fv(t_{i1}, \ldots, t_{ik}))$ *where*

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

The basic idea of the context consistency condition is that the variables in the range are captured by some FDs imposed on type classes present in the context. Note that although the context consistency condition resembles a more "liberal" version of the termination condition, context consistency does not prevent non-termination. Example 3 satisfies both of the above conditions, however, resulting CHRs are non-terminating. More precisely, adding the improvement rules $Prop(p)$ to a terminating set $Simp(p)$ of instance CHRs yields a non-terminating set $Simp(p) \cup Prop(p)$. Hence, for the following result to hold we need to assume that CHRs are terminating.

Theorem 2 (More Liberal FDs Soundness). *Let p be a set of Haskell class and instance declarations which satisfies the Liberal-FD restrictions. Let $Simp(p)$ and $Prop(p)$ be defined by Definition 2. If the set $Simp(p) \cup Prop(p)_{class}$ is confluent and range-restricted and $Simp(p) \cup Prop(p)$ is terminating, then $Simp(p) \cup Prop(p)$ is confluent and range-restricted.*

Note that Example 4 also satisfies the more liberal FD conditions. According to Definition 2 we generate the following improvement rules. Note that the functional dependency imposed is full. For simplicity, we only focus on improvement rules.

```
rule Zip a b c, Zip a d c ==> b=d       -- (Z1)
rule Zip a b c, Zip d b c ==> a=d       -- (Z2)
rule Zip a d [(a,b)]      ==> d=b       -- (Z3)
rule Zip d b [(a,b)]      ==> d=a       -- (Z4)
rule Zip a d ([c]->e)     ==> d=b       -- (Z5)
rule Zip d b ([c]->e)     ==> d=a       -- (Z6)
```

Rules (Z5) and (Z6) are generated from the second instance. Note that both rules introduce some new variables since we violate the termination condition. However, both rules are harmless. Effectively, we can replace them by

```
rule Zip a d ([c]->e)     ==> True      -- (Z5')
rule Zip d b ([c]->e)     ==> True      -- (Z6')
```

which makes them trivial. Hence, we can omit them altogether. We observe that we can "safely" violate the termination condition (without breaking termination) in case the improvement rules generated are trivial, i.e. the right-hand side of CHRs can be replaced by the always true constraint. This is always the case if the range component of an instance is a variable.

4.2 Stronger Improvement

There are situations where FDs do not enforce sufficient improvement. Note that the inferred types of `e1` and `e2` in Example 4 are

```
  e1 :: Zip Bool Char [a] => a
  e2 :: Zip (Bool,Char) Int [a] => a
```

rather than

```
  e1 :: (Bool,Char)
  e2 :: ((Bool,Char),Int)
```

For example rule (Z3) states that only if we see `Zip a d [(a,b)]` we can improve `d` by `b`. However, in case of `e1` we see `Zip Bool Char [a]`, and we would like to improve `a` to `(Bool,Char)`. Indeed, in this context it is "safe" to replace rules (Z3) and (Z4) by

```
rule Zip a b [c] ==> c=(a,b)   -- (Z34)
```

which imposes stronger improvement to achieve the desired typing of `e1` and `e2`. Note that rule (Z34) respects the consistency and termination conditions (assuming we enforce these conditions for user-provided improvement rules). Hence, we retain confluence and termination of CHRs.

Of course, if a user-provided improvement violates any of the sufficient conditions, it is the user's responsibility to ensure that resulting CHRs are confluent and terminating.

4.3 Instance Improvement Only

Instead of *stronger* improvement it might sometimes be desirable to *omit* certain improvement rules. For example, in case the context consistency condition is violated, we can recover confluence by dropping the functional dependency rule.

Theorem 3 (Instance Improvement Soundness). *Let p be a set of Haskell class and instance declarations which satisfies the Haskell-FD consistency restriction. If the set $Simp(p) \cup Prop_{class}(p)$ is confluent and range-restricted and $Simp \cup Prop_{class}(p) \cup Prop_{inst}(p)$ is terminating, then $Simp(p) \cup Prop_{class}(p) \cup Prop_{inst}(p)$ is confluent and range-restricted.*

Here is a (confluent) variation of Example 8 where we only impose the instance improvement rule.

```
data Nil        = Nil
data Cons a b   = Cons a b
data ExpAbs x a = ExpAbs x a
-- env represents environment, exp expression
-- and t is the type of the resulting value
class Eval env exp t where  eval :: env->exp->t
-- we only impose the instance improvement rule but NOT
-- the class FD
rule Eval env (ExpAbs x exp) v ==> v=v1->v2
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
```

5 Related Work

The idea of improving types in the context of Haskell type classes is not new. For example, Chen, Hudak and Odersky [CHO92] introduce type classes which can be parameterized by a specific parameter. For example, the declaration `class SM m r | m->r` from Example 2 can be expressed as the parametric declaration `class m::SM r`. Interestingly, they impose conditions similar to Jones's consistency and termination condition to achieve sound and decidable type inference. However, their approach is more limited than ours. Functional dependencies must be always of the form `a->b` where `b` is not allow to appear in the domain of any other functional dependency. Furthermore, they do not consider any extensions such as more liberal FDs.

In [Jon95], Jones introduces a general theory of simplifying and improving types as a refinement of his theory of qualified types [Jon92]. However, he does not provide any formal results which improvement strategies lead to sound and decidable type inference.

Subsequently, Jones extends multi-parameter type classes with functional dependencies [Jon00]. He states some conditions (consistency and termination) which in this paper we finally verify as sufficient to ensure sound and decidable type inference. Surprisingly, he introduces Example 3 (which breaks the termination condition) as a motivation for functional dependencies.

Duggan and Ophel [DO02] describe an improvement strategy, domain-driven unifying overload resolution, which is very similar to functional dependencies. Indeed, they were the first to point out the potential problem of non-termination of type inference. However, they do not discuss any extensions such as more liberal FDs nor do they consider how to cope with the termination problem.

Stuckey and Sulzmann [SS02] introduce a general CHR-based formulation for type classes. They establish some general conditions, e.g. termination and confluence, in terms of CHRs under which type inference is sound and decidable. Here, we rephrase functional dependencies as a particular instance of their framework.

6 Conclusion

We have given a new perspective on functional dependencies by expressing the improvement rules implied by FDs in terms of CHRs. We have verified, for the first time, that the conditions (termination and consistency, see Definition 1) stated by Jones are sufficient to guarantee sound and decidable type inference (see Theorem 1).

There are many examples which demand dropping the termination condition. For this purpose, we have introduced more liberal FDs in Section 4.1. We have identified an additional condition (context consistency) which guarantees confluence (see Theorem 2). We have also discussed further useful extensions such as stronger improvement rules (Section 4.2) and instance improvement rules only (Section 4.3).

For such extensions it becomes much harder to guarantee decidability (unless the generated improvement rules are trivial). For example, the more liberal FD conditions only ensure soundness but not decidability. We are already working on identifying further decidable classes of CHRs. We expect to report results on this topic in the near future.

In another line of future work we plan to investigate how to safely drop the consistency condition. Consider

```
class Insert ce e | ce -> e where insert :: e->ce->ce
instance Ord a => Insert [a] a
instance Insert [Float] Int
```

Our intention is to insert elements into a collection. The class declaration states that the collection type uniquely determines the element type. The first instance states that we can insert elements into a list if the list elements enjoy an ordering relation. The second instance states that we have a special treatment in case we insert `Int`s into a list of `Float`s (for example, we assume that `Int`s are internally represented by `Float`s). This sounds reasonable, however, the above program is rejected because the consistency condition is violated. To establish confluence we seem to require a more complicated set of improvement rules. We plan to pursue this topic in future work.

Acknowledgements

We thank Jeremy Wazny and the reviewers for their comments.

References

[Abd97] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, volume 1330 of *LNCS*, pages 252–266. Springer-Verlag, 1997.

[CHO92] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Proc. of ACM Conference on Lisp and Functional Programming*, pages 170–191. ACM Press, June 1992.

[CK03] M. Chakravarty and S. Keller. Classy type analysis, 2003.

[DO02] D. Duggan and J. Ophel. Type-checking multi-parameter type classes. *Journal of Functional Programming*, 12(2):133–158, 2002.

[DPJSS03] G. J. Duck, S. Peyton-Jones, P. J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. Technical report, National University of Singapore, 2003. `http://www.comp.nus.edu.sg/~sulzmann/chr/download/fd-chr.ps.gz`.

[Frü95] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer-Verlag, 1995.

[Frü98] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.

[Jon92] M. P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, Oxford University, September 1992.

[Jon95] M. P. Jones. Simplifying and improving qualified types. In *FPCA '95: Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 1995.

[Jon00] M. P. Jones. Type classes with functional dependencies. In *Proc. of the 9th European Symposium on Programming Languages and Systems, ESOP 2000*, volume 1782 of *LNCS*. Springer-Verlag, March 2000.

[Kar03] J. Karczmarczuk. Structure and interpretation of quantum mechanics – a functional framework. In *Proc. of Haskell Workshop'03*, pages 50–61. ACM Press, 2003.

[Pey99] S. Peyton Jones et al. Report on the programming language Haskell 98, February 1999. http://haskell.org.

[SS02] P. J. Stuckey and M. Sulzmann. A theory of overloading. In *Proc. of ICFP'02*, pages 167–178. ACM Press, 2002.

[WW03] G. Washburn and S. Weirich. Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism. In *Proc. of ICFP'03*, pages 249 – 262. ACM Press, 2003.