Understanding Functional Dependencies via Constraint Handling Rules

Martin Sulzmann
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
`sulzmann@comp.nus.edu.sg`

Gregory J. Duck
NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne 3010, Australia
`gjd@cs.mu.oz.au`

Simon Peyton-Jones
Microsoft Research Ltd
7 JJ Thomson Avenue, Cambridge CB3 0FB, England
`simonpj@microsoft.com`

Peter J. Stuckey
NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne
3010, Australia
`pjs@cs.mu.oz.au`

Abstract

Functional dependencies are a popular and useful extension to Haskell style type classes. 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, complete and decidable type inference. The sufficient conditions imposed on functional dependencies can be very limiting. We show how to safely relax these conditions and suggest several sound extensions of functional dependencies. Our results allow for a better understanding of functional dependencies and open up the opportunity for new applications.

1 Introduction

Functional dependencies describe properties of relations, a functional dependency $a \rightarrow b$ for a relation $R(a, b, c)$ states that in the relation R for a given value of the first argument a there is a unique possible value for the second argument b. So for example the relation $\{(1,2,3),(1,2,1),(2,3,1)\}$ satisfies the functional dependency while $\{(1, 2, 3), (1, 1, 2)\}$ does not. Functional dependencies have a long history of use in the database community (?; ?) for query and index optimization. In this paper we are concerned with functional dependencies as they are applied to Haskell style type classes. Their principal use is for type improvement and the issues are quite distinct from their use in databases.

Functional dependencies, introduced by Mark Jones (Jones, 2000), 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 on the background in Section 2.

The goal of our work 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) (Frühwirth, 1995; ?). The current paper is an extended and revised version of (Duck *et al.*, 2004). In summary, our contributions are:

- 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 (Section 4).
- Based on this understanding, we provide the first proof that the restrictions imposed by Jones on functional dependencies (Jones, 2000) ensure sound, complete and decidable type inference (Section 5).
- Jones' restrictions can be very limiting. We propose several useful extensions (Section 6) such as *more liberal FDs* (Section 6.1), *sound non-full FDs* (Section 6.2) and *multi-range FDs* (Section 6.3).

Related work is discussed in Section 7. We conclude in Section 8. Proofs can be found in the Appendix.

2 Background: Functional Dependencies in Haskell

We begin by reviewing functional dependencies, as introduced by Jones (Jones, 2000), 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, and hence the expected type is

```
ins2 :: Coll c e => c -> e -> e -> c
```

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*" (Jones, 1995).

2.1 Examples of Functional Dependencies

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 | c -> 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, so that, for example, if `xs::[Int]` and `ys::[Bool]`, then we can write

```
rs1 :: [(Int,Bool)]
rs1 = zip xs ys

rs2 :: [((Int,Bool),Int)]
rs2 = zip xs ys xs
```

Without the functional dependencies, however, this function fails to type check:

```
z3 :: [a] -> [b] -> [c] -> [(a, (b,c))]
z3 xs ys zs = zip xs (zip ys zs)
```

The compiler emits a message something like

```
  No instance for (Zip a b1 [(a, (b, c))])
    arising from use of ‘zip’ at Zip.hs:26:15-17
```

Indeed, the first instance declaration contains a repeated use of `a`, and hence does not match the constraint, unless `b1` is instantiated to `(b,c)` — and that is precisely what the functional dependencies cause to happen.

2.2 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.

2.3 Summary

The bottom line of our informal overview is this. We want type inference to be *sound*, *complete* and *decidable*. Functional dependencies threaten this happy situation. The obvious solution is to place restrictions on how functional dependencies are used, as Jones indeed did, so that type inference remains well-behaved. But the situation is quite complicated, and we should *prove* that type inference is well behaved, something that no one has yet done.

3 Background: Constraint Handling Rules

In this section we introduce a useful notation called *Constraint Handling Rules* (CHRs), which has a rich theory (Frühwirth, 1998) with strong connections to type classes (**?**). Our plan is to translate a Haskell program into CHRs, and thereby give a more precise account of exactly what functional dependencies mean; and give formal proofs about sound and decidable type inference.

We begin with an informal account of CHRs and their relationship to functional dependencies.

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 following two constraint handling rules:

```
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 for Coll [a] a 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.

In the result of this Section we define CHRs more formally.

3.1 Definition of Constraint Handling Rules

For our purposes, CHRs are of the following two forms

Simplification	rule	c	<==>	$d_1, \ldots, d_m$
Propagation	rule	$c_1, \ldots, c_n$	==>	$d_1, \ldots, d_m$

In these rules $c, c_1, \ldots, c_n$ are type class constraints; and $d_1, \ldots, d_m$ are type class constraints or equations. The simplification rule states that given constraint c we can *replace* it by constraint $d_1, \ldots, d_m$. The propagation rule states that given constraint $c_1, \ldots, c_n$, we can *add* $d_1, \ldots, d_m$. We say a CHR is *single-headed* if the left hand side has exactly one user-defined constraint. A *CHR system* is a set of CHR rules.

CHR rules can also be interpreted as first-order formulae. We assume that the reader is familiar with substitutions, renamings, most general unifiers (m.g.u.) and the basics of first-order logic (Shoenfield, 1967; Lassez *et al.*, 1987). In the following, we assume that $\leftrightarrow$ denotes Boolean equivalence and $\supset$ denotes Boolean implication. We often write $\bar{o}$ to abbreviate a sequence of objects $o_1,...,o_n$. We write $fv(o)$ to denote the set of *free variables* in some object o. The translation function $[\![\cdot]\!]$ from CHR rules to first-order formulae is:

$$[\![\texttt{rule}\ c \quad \texttt{<==>} \quad d_1,\ldots,d_m]\!] \quad = \quad \forall\bar{a}'(c \leftrightarrow (\exists\bar{b}\ d_1 \wedge \cdots \wedge d_m))$$

$$[\![\texttt{rule}\ c_1,\ldots,c_n \quad \texttt{==>} \quad d_1,\ldots,d_m]\!] \quad = \quad \forall\bar{a}(c_1 \wedge \cdots \wedge c_n \supset (\exists\bar{b}\ d_1 \wedge \cdots \wedge d_m))$$

where $\bar{a}' = fv(c)$, $\bar{a} = fv(c_1 \wedge \cdots \wedge c_n)$ and $\bar{b} = fv(d_1 \wedge \cdots \wedge d_m) - \bar{a}$. We define the translation of a set of CHRs as the conjunction of the translation of each individual CHR rule.

The operational semantics of CHRs are straightforward. CHRs manipulate *constraint stores*, i.e. sets of primitive constraints. In contrast to the original CHR semantics which is based on multi-set rewriting, we use a set based operational semantics to match more closely the logical semantics. We can apply a rule R in program P to a constraint C if the left-hand side of a R is a subset of C (we assume that substitutions represented by equations have already been applied, see examples below). The resulting constraint C' replaces this subset by the right hand side of the rule (if it is a simplification rule), or adds the right hand side of the rule to C (if it is a propagation rule). This *derivation step* is denoted $C \rightarrowtail_R C'$, or sometimes $C \rightarrowtail_P C'$ when we do not want to specify which particular rule in CHR system P is being applied.

More precisely, we assume that each constraint C is split into a set of type class constraints C_u and a set of equations C_e, i.e. $C = C_u \cup C_e$. Variables in CHR rules R are renamed before rule application. We distinguish among the following derivation steps:

(Solve Step) $C_u \cup C_e \rightarrowtail_P \phi C_u \cup C_e$
ϕ mgu of C_e

(Simp) $C_u \cup C_e \rightarrowtail_P (C_u - c') \cup C_e \cup \theta(\bar{d})$
if `rule` c `<==>` $\bar{d} \in P$ and there exists a subset $c' \subseteq C_u$
and a substitution θ on variables in the rule
such that $\theta(c') \equiv c$

(Prop) $C_u \cup C_e \rightarrowtail_P C_u \cup C_e \cup \theta(\bar{d})$
if `rule` $\bar{c}$ `==>` $\bar{d} \in P$ and there exists a subset $\bar{c}' \subseteq C_u$
and a substitution θ on variables in the rule
such that $\theta(\bar{c}') \equiv \bar{c}$

In (Solve Step), we assume that we normalize stores by building most general unifiers. We often perform this step implicitly. In rules (Simp) and (Prop) we implicitly assume that we use new renamed copies of rules in P to avoid variable clashes. We write $\equiv$ to denote syntactic equality.

To prevent the infinite application of CHR propagation rules we assume that these rules are only applied once on the same set of constraints. We refer to (Abdennadher, 1997) for more details. Examples explaining CHR application steps follow shortly.

3.2 Properties of CHRs

A set of CHRs may or may not have three important properties, namely *confluence*, *termination*, and *range-restrictedness*. We define all three here for later reference.

A *derivation*, denoted $C \rightarrowtail_P^* C'$ is a sequence of derivation steps using rules in P where no derivation step is applicable to C'. A derivation $C \rightarrowtail_P^* C'$ is *successful* iff C'_e is satisfiable; that is, the set of equations in C' has a unifier.

Termination means that every derivation sequence terminates:

Definition 1 (Termination) A CHR system P is *terminating* iff for any constraint C there exists a constraint C' such that $C \rightarrowtail_P^* C'$.

Confluence means that different derivations starting from the same point can always be brought together again:

Definition 2 (Confluence) A CHR system P is *confluent* iff for each constraint C_0 for any two possible derivation steps applicable to C_0, say $C_0 \rightarrowtail_P C_1$ and $C_0 \rightarrowtail_P C_2$, then there exist derivations $C_1 \rightarrowtail_P^* C_3$ and $C_2 \rightarrowtail_P^* C_4$ such that C_3 is equivalent (modulo new variables introduced) to C_4, i.e. $\models (\bar{\exists}_{fv(C_0)} C_3) \leftrightarrow (\bar{\exists}_{fv(C_0)} C_4)$.

In this definition, we write $\models$ to denote the model-theoretic entailment relation; and we write $\bar{\exists}_{\bar{a}} C$ as a short-hand for $\exists \bar{b}.C$ where $\bar{b} = fv(C) - \bar{a}$.

Note that, when a set of CHRs are terminating, we can easily test for confluence by checking that all "critical pairs" are joinable (Abdennadher, 1997).

Finally, range-restrictedness for a CHR means that all variables on the right-hand side are bound by variables on the left-hand side. Silently, we assume that equations on the right-hand side of a CHR, say lhs `==>` rhs, have a unifier. Otherwise, we can effectively replace the right-hand side by $False$ which yields the equivalent (and range-restricted) CHR lhs `==>` $False$.

Definition 3 (Range-Restricted) A CHR lhs `<==>` rhs (or lhs `==>` rhs) is *range-restricted* iff $fv(\psi(rhs)) \subseteq fv(\psi(lhs))$ where ψ is an m.g.u. of all equations in rhs.

For the purposes of this paper, we often use a more specific definition.

Definition 4 (Variable-Restricted) A CHR lhs `<==>` rhs (or lhs `==>` rhs) is *variable-restricted* iff $fv(rhs) \subseteq fv(lhs)$.

Note that each variable-restricted CHR is immediately range-restricted. However, the other direction does not hold necessarily. As an example, consider $F\ a$ `<==>` $G\ a\ b, a = [b]$ which is range-restricted. Because, applying the unifier of the right-hand side yields $F\ [b]$ `<==>` $G\ [b]\ b$. However, $F\ a$ `<==>` $G\ a\ b, a = [b]$ is not variable-restricted because $b \notin fv(F\ a)$.

Type variables	a, b, c		
Type constructors	T		
Type classes	TC		
Types	t	$::=$	$a \mid t \rightarrow t \mid T\ \bar{t}$
Type Schemes	σ	$::=$	$t \mid \forall \bar{a}.C \Rightarrow t$
Class constraint	CC	$::=$	$TC\,\bar{t}$
Context	C, D	$::=$	$CC \ \mid\ C, C$
Functional dependencies	fd	$::=$	$a_1, \ldots a_n \rightarrow a$
Declarations	d	$::=$	`class` C `=>` $TC\ \bar{a} \mid fd_1, \ldots, fd_m$
		$\mid$	`instance` C `=>` $TC\,\bar{t}$

Fig. 1. Syntax of type and class declarations

4 Using CHRs to understand Haskell with Functional Dependencies

Now that we have said what CHRs are, we are ready to show how to use CHRs to understand Haskell extended with functional dependencies, a language we call "Haskell-FD". By "understand" we mean several things: we can give the meaning of a Haskell-FD program by translating it to CHRs (Section 4.2); we can use CHRs to reason about the equivalence of Haskell-FD programs (Section 4.3); we can use CHRs as a framework for type inference for Haskell-FD (Section 4.4). Lastly, in Section 5 we verify that Jones' restrictions allow us to establish some essential conditions in terms of CHRs which are the basis for sound, complete and decidable type inference.

We begin by fixing the language we consider. For the moment, we consider only the class and instance declarations, whose syntax is given in Figure 1. We include expressions later in Section 4.4 where we consider the formal type inference system.

To make the type-class system well behaved, Haskell imposes a number of conditions on the class and instance declarations, which we adopt:

Definition 5 (Basic Conditions) The Basic Conditions on class and instance declarations are these:

- The context C of a class and instance declaration can mention only type variables, not type constructors, and in each individual class constraint CC all the type variables are distinct.
- In an instance declaration `instance` C `=>` $TC\ t_1 \ldots t_n$, at least one of the types t_i must not be a type variable.
- The instance declarations must not overlap. That is, for any two instances declarations `instance` C `=>` $TC\ t_1 \ldots t_n$ and `instance` C' `=>` $TC\ t'_1 \ldots t'_n$ there is no substitution ϕ such that $\phi(t_1) = \phi(t'_1)$,....,$\phi(t_n) = \phi(t'_n)$.

Functional dependencies, written fd in Figure 1, appear only in class declarations:

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

Each functional dependency takes the form

$$a_{i_1}, \ldots, a_{i_k} \texttt{ -> } a_{i_0}$$

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* of the functional dependency.

Some Haskell systems (HUGS, n.d.; GHC, n.d.) that allow functional dependencies usually allow dependencies of the form `a -> b c`, with multiple type variables to the right of the arrow. We refer to such FDs as *multi-range* FDs, as opposed to *single-range* FDs which have a single variable to the right of the arrow. Initially we only consider the single-range form, later in Section 6.3 we will consider multi-range FDs.

4.1 Jones' Functional Dependency Restrictions

In Jones' original paper (Jones, 2000), the following two restrictions are imposed on functional dependencies:

Definition 6 (Consistency Condition) Consider a declaration for class TC and any pair of instance declarations for that class:

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

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

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

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

Definition 7 (Coverage Condition) Consider a declaration for class TC, and any instance declaration for that class:

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

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

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

The Consistency Condition is easy to motivate: it rules out inconsistent instance declarations (see Example 5). The motivation for the Coverage Condition[1] was that determining the types in the domain of the dependency should fully determine the type in the range. For example, consider the recursive `Vec` instance given earlier (Example 3):

```
class Mul a b c | a b -> c
instance Mul a b c => Mul a (Vec b) (Vec c) where ...
```

Fixing the first two arguments of `Mul` to (say) `Int` and `Vec Int` does not immediately fix the third argument `c`; formally `c` $\notin fv($`a`$,$ `Vec b`$)$. Instead, the programmer intends that it is fixed via the context `(Mul a b c)` of the instance declaration.

[1] Note that in the conference version (Duck *et al.*, 2004) we referred to the Coverage Condition as the "Termination Condition". We will shortly see why our new terminology is more appropriate.

The fact that the instance declaration does not satisfy the Coverage Condition leads directly to the non-termination we saw in Example 6.

As it stands the two conditions above are not sufficient to ensure sound, complete and decidable type inference. However, adding a third restriction, the "Bound Variable Condition", *does* guarantee sound, complete and decidable type inference:

Definition 8 (Bound Variable Condition) For each class declaration

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

we require that $fv(C) \subseteq \{a_1, \ldots, a_n\}$. Furthermore, for each instance declaration

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

we require that $fv(C) \subseteq fv(t_1, \ldots, t_n)$.

We will often refer to the three conditions (Coverage, Consistency, and Bound Variable) as the **FD-Conditions**. In Section 5 we discuss and motivate each of the FD-Conditions, and prove that they guarantee sound, complete and decidable type inference.

4.2 Translation to CHRs

We are now in the position to formalize the translation of `class` and `instance` declarations, including functional dependencies, into CHRs. The translation is given in Figure 2. The important point is that the translation makes explicit all improvement rules which are implied by the logical reading of the functional-dependency, super-class, and instance relations.

It is convenient to have names for particular sets of CHRs:

Definition 9 If p is a set of `class` and `instance` declarations, we define the following set of CHRs:

- $MPTC(p)$ denotes set of all class and instance CHRs generated from p. (The class and instance CHRs are defined in Figure 2.) These CHRs simply reflect the basic type-class system, ignoring functional dependencies.
- $FD(p)$ denotes set of all functional-dependency and instance-improvement CHRs generated from p (see Figure 2). These are the CHRs generated by the functional dependencies.
- $CHR(p) = MPTC(p) \cup FD(P)$ denotes set of all CHRs generated from p.

4.3 Using CHRs to reason about FDs: Implicit Improvement Rules

CHRs give us a way to reason formally about the effect of adding or removing functional dependencies.

Example 8 Consider the following class and instance declarations.

```
class D a b | a->b
class D a b => C a b
```

To translate a set of Haskell class and instance declarations into CHRs, proceed as follows. For each class declaration

`class` C `=>` `TC` a_1 ... a_n `|` $fd_1, ..., fd_m$

where each fd_i is of the form $a_{i_1}, ..., a_{i_k}$ `->` a_{i_0}, we generate the following CHRs:

- The *class CHR*:

  `rule` $TC\ a_1 \ldots a_n$ `==>` C

- The *functional-dependency CHRs*: for each functional dependency fd_i in the class declaration, we generate

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

  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:

- The *instance CHR*:

  `rule` $TC\ t_1 \ldots t_n$ `<==>` C

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

- The *instance-improvement CHRs*: for each functional dependency fd_i in the class declaration, we generate

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

  where $b_1 \ldots b_n$ are distinct type variables, $\theta(b_{i_j}) = t_{i_j}$, $j > 0$ and $\theta(b_l) = b_l$ if $\neg\exists j.l = i_j$.

Here, a *substitution* $\theta = [t_1/a_1, \ldots, t_n/a_n]$ simultaneously replaces each a_i by its corresponding t_i.

Fig. 2. Translation of class and instance declarations into CHRs

```
instance D [a] a
instance C [a] a
```

Would it make any difference if we added a functional dependency to the class declaration for `C`, thus?

```
class D a b => C a b | a->b
```

Intuitively, we might expect that it would have no effect, because `D` is superclass of `C`, and `D` has the functional dependency. And indeed, that turns out to be the case. Here are the CHRs arising from the original text:

```
rule D a b, D a b' ==>  b=b'  -- (FD-D)   Functional-dependency CHR for D
rule D [a] a       <==> True  -- (Inst-D) Instance CHR for D
rule D [a] b       ==>  a=b   -- (Imp-D)  Instance-improvement CHR for D
rule C a b         ==>  D a b -- (Cls-C)  Class CHR for C
rule C [a] a       <==> True  -- (Inst-C) Instance CHR for C
```

Adding the functional dependency to class `C` would add two new rules

$$\text{(Var)}\ \frac{(x:\sigma)\in\Gamma}{C,\Gamma\vdash x:\sigma} \qquad \text{(Let)}\ \frac{\begin{array}{c}C,\Gamma\vdash e:\sigma\\ C,\Gamma,x:\sigma\vdash e':t'\end{array}}{C,\Gamma\vdash \mathsf{let}\ x=e\,\mathsf{in}\ e':t'}$$

$$\text{(Abs)}\ \frac{C,\Gamma,x:t\vdash e:t'}{C,\Gamma\vdash \lambda x.e:t\rightarrow t'} \qquad \text{(App)}\ \frac{\begin{array}{c}C,\Gamma\vdash e_1:t_1\rightarrow t_2\\ C,\Gamma\vdash e_2:t_1\end{array}}{C,\Gamma\vdash e_1\ e_2:t_2}$$

$$\text{(}\forall\text{ Elim)}\ \frac{\begin{array}{c}C,\Gamma\vdash e:\forall\bar{a}.C'\Rightarrow t\\ [\![P]\!]\models C\supset[\bar{t}/\bar{a}]C'\end{array}}{C,\Gamma\vdash e:[\bar{t}/\bar{a}]t} \qquad \text{(}\forall\text{ Intro)}\ \frac{\begin{array}{c}C\wedge C',\Gamma\vdash e:t\\ \bar{a}\not\in fv(C,\Gamma)\end{array}}{C,\Gamma\vdash e:\forall\bar{a}.C'\Rightarrow t}$$

$$\text{(Class)}\ \frac{\begin{array}{c}\bar{a}=fv(\bar{t})-fv(\Gamma)\quad \Gamma'=\Gamma\cup\{x_i:\forall\bar{a},\bar{b}_i.TC\,\bar{t}\wedge C_i'\Rightarrow t_i\mid i\in I\}\\ C,\Gamma'\vdash p:\sigma'\end{array}}{C,\Gamma\vdash \mathsf{class}\ C'\Rightarrow TC\,\bar{a}\mid fd_1,\ldots,fd_m\,\mathsf{where}\ [x_i:\forall\bar{b}_i.C_i'\Rightarrow t_i]_{i\in I}\,\mathsf{in}\ p:\sigma'}$$

$$\text{(Inst)}\ \frac{\begin{array}{c}C,\Gamma\vdash p:\sigma''\\ C,\Gamma\vdash e_i:\forall\bar{a}_i'.C_i'\Rightarrow t_i'\quad (x_i:\forall\bar{a}_i.TC\,\bar{t}\wedge C_i''\Rightarrow t_i)\in\Gamma\quad \text{for } i\in I\\ [\![P]\!]\models(\forall\bar{a}_i'.(TC\,\bar{t}'\wedge C_i')\Rightarrow t_i')\preceq(\forall\bar{a}_i.(TC\,\bar{t}\wedge C_i''\wedge\bar{t}'=\bar{t})\Rightarrow t_i)\quad\text{for } i\in I\end{array}}{C,\Gamma\vdash \mathsf{instance}\ C'\Rightarrow TC\,\bar{t}'\,\mathsf{where}\ [x_i=e_i]_{i\in I}\,\mathsf{in}\ p:\sigma''}$$

Fig. 3. Typing rules

```
rule C a b, C a b' ==> b=b'    -- (FD-C)  Functional dependency CHR for C
rule C [a] b ==> a=b           -- (Imp-C) Instance-improvement CHR for C
```

But it is easy to see that these two new rules are derivable from the previous set. For example, starting with the left-hand-side of (FD-C), we can reason thus:

$$\begin{array}{ll} & C\ a\ b, C\ a\ b' \\ \rightarrowtail_{Cls-C} & C\ a\ b, C\ a\ b', D\ a\ b, D\ a\ b' \\ \rightarrowtail_{Imp-D} & C\ a\ b, C\ a\ b', D\ a\ b, D\ a\ b', b=b' \end{array}$$

Hence, rule (FD-C) adds nothing new: the CHR systems with and without the two new rules are equivalent

4.4 Type Inference for Type Classes

Next, we show how the process of type inference for type classes is connected to our CHR encoding. The following material is taken from (?). We introduce type class systems as an extension of the Hindley/Milner system. We assume the type language from Figure 1 but additionally introduce expressions. For brevity, we ignore recursive functions. We assume that a program consists of an expression, possibly preceded by a sequence of class and instance declarations.

$$\begin{array}{llll}
\text{Expressions} & e & ::= & x \mid \lambda x.e \mid e\,e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\
\text{Programs} & p & ::= & e \\
 & & \mid & \mathsf{class}\ C \Rightarrow TC\,\bar{a} \mid fd_1, \ldots, fd_m \\
 & & & \quad \mathsf{where}\ [x_i : \sigma_i]_{i \in I}\ \mathsf{in}\ p \\
 & & \mid & \mathsf{instance}\ C \Rightarrow TC\,\bar{t}\,\mathsf{where}\ [x_i = e_i]_{i \in I}\ \mathsf{in}\ p
\end{array}$$

4.4.1 Typing rules

The typing rules for programs are described in Figure 3. Typing judgments are of the form $C, \Gamma \vdash p : \sigma$ where C is a (global) constraint, Γ is a type environment associating free variables with their type and σ is a type scheme. Always implicit is the *type class theory* P describing the set of CHRs obtained from the translation of classes, instances and functional dependencies in p (see Figure 2). We say a program is well-typed if we can build a derivation tree according to the typing rules and the constraint in the final typing judgment is satisfiable.

Rules (Var) – ($\forall$ Intro) are the standard type class rules which can also be found in the theory of qualified types (Jones, 1992). We assume that let-bound and method identifiers are a-renamed to avoid name clashes. The set of free variables of an environment are computed by taking $fv(\{x_1 : \sigma_1, ..., x_n : \sigma_n\})$ to be equal to $fv(\sigma_1, ..., \sigma_n)$. In case of type schemes, we assume that $fv(\forall \bar{a}.C \Rightarrow t) = fv(\overline{[b/a]}C, \overline{[b/a]}t) - \bar{b}$ where $\bar{b}$ are fresh variables and $\overline{[b/a]}$ is a renaming.

In rule ($\forall$ Elim) the statement $[\![P]\!] \models C \supset [\bar{t}/\bar{a}]C'$ requires that constraint C implies the instantiated constraint $[\bar{t}/\bar{a}]C'$ (with $\bar{a}$ replaced by $\bar{t}$) in any model of $[\![P]\!]$ (we refer here to the logical reading of CHRs). In rule (Class) we process class declarations by adding the method declarations to the environment. Rule (Inst) defines an instance of class TC. We require that the type of each member function for a particular instance is subsumed by the type specified by the class declaration. We define $F \models (\forall \bar{a}_1.C_1 \Rightarrow t_1) \preceq (\forall \bar{a}_2.C_2 \Rightarrow t_2)$ iff $F \models C_2 \supset \exists \bar{a}_1.(C_1 \wedge t_1 = t_2)$ where we assume there are no name clashes between a_1 and a_2 (i.e. $\bar{a}_1 \cap \bar{a}_2 = \emptyset$) and F is a first-order formula. We say σ_1 *subsumes* σ_2 w.r.t. F if $F \models \sigma_1 \preceq \sigma_2$.

Example 9 Consider the program

```
class C a where f :: a->a
instance C Int where f x = x
instance C Bool where f x = 1
```

The class declaration states that x has type $\forall a.C\,a \Rightarrow a \rightarrow a$. The first instance declaration implements an instance of C at type *Int* where the member function x has type $\forall a.a \rightarrow a$. This is correct but more general than the $Int \rightarrow Int$ required. The second instance has a member function of type $\forall a.a \rightarrow Int$ which is incompatible with the type $Bool \rightarrow Bool$ required.

4.4.2 Ambiguity

An important restriction usually made on types appearing in typing derivations is that that they be *unambiguous.* This is an essential requirement to ensure a well-defined semantics for programs (Jones, 1993; **?**). An expression whose is ambiguous does not have a well-specified run-time behavior.

Example 10 Consider

```
class Show a where
  show :: a->String
  read :: String->a

f s = show (read s)
```

Note that the right-hand side expression `show (read s)`, and the lambda-bound variable `s`, are both of type `String`. Additionally we find the constraint `Show a`, so `f` would get the inferred type

```
f :: Show a => String -> String
```

This type is *ambiguous.* The type variable `a` is not mentioned in the part after the "`=>`", and hence a call of `f` cannot specify which `Show` instance to pick. Yet the chosen instance may make a difference at runtime: do we read a `Float` from `s` and then convert it back to a `String`? Or do we read a `Bool`? Or what? Clearly, such ambiguous programs must be rejected.

Example 11 Functional dependencies require us to revise the definition of ambiguity. Consider

```
class C a b | a -> b
g :: C a b => a -> a
g = ...
```

Here, `g`'s type should be unambiguous even though `b` is not mentioned in the part after the "`=>`", because fixing `a` fixes `b`, via the functional dependency.

We revise the definition of unambiguity to require simply that all variables appearing in the constraint component can be uniquely determined from the type component w.r.t. program theory P. An unambiguous type is then formally defined as follows:

Definition 10 (Unambiguity) Let P be the program theory and ρ be a variable renaming on $\bar{a}$. Then $\forall \bar{a}.C \Rightarrow t$ is *unambiguous* iff $[\![P]\!] \models (C \wedge \rho(C) \wedge (t = \rho(t))) \supset (a = \rho(a))$ for each $a \in \bar{a}$.
We can easily extend the definition of unambiguity from *types* to *judgments*. The judgement $C, \Gamma \vdash p : t$ is *unambiguous* iff $\forall \bar{a}.C \Rightarrow t$ is unambiguous where $\bar{a} = fv(C, t) - fv(\Gamma)$.

We demand that all typing judgments and type schemes appearing in a typing derivation are unambiguous. This requirement, together with the assumption that CHR derivations are unambiguous, leads to a well-defined semantics for programs (**?**).

$$\text{(Var)}\quad \frac{(x : \forall \bar{a}.C \Rightarrow t) \in \Gamma \quad \bar{b} \text{ new}}{\Gamma, x \vdash_{\mathit{inf}} ([\bar{b}/\bar{a}]C, [\bar{b}/\bar{a}]t)}
\qquad
\text{(Let)}\quad \frac{\begin{array}{c}\Gamma, e \vdash_{\mathit{inf}} (C_1, t) \\ \sigma = \mathit{gen}(C_1, \Gamma_x, t) \quad \mathit{unambig}(P, \sigma) \\ \Gamma, x : \sigma, e' \vdash_{\mathit{inf}} (C_3, t') \\ \mathit{sat}(P, C_3)\end{array}}{\Gamma, \textsf{let } x = e \textsf{ in } e' \vdash_{\mathit{inf}} (C_3, t')}$$

$$\text{(Abs)}\quad \frac{\Gamma, x : a, e \vdash_{\mathit{inf}} (C, t') \quad a \text{ new}}{\Gamma, \lambda x.e \vdash_{\mathit{inf}} (C, a \to t')}
\qquad
\text{(App)}\quad \frac{\begin{array}{c}\Gamma, e_1 \vdash_{\mathit{inf}} (C_1, t_1) \\ \Gamma, e_2 \vdash_{\mathit{inf}} (C_2, t_2) \\ C' = C_1 \wedge C_2 \wedge (t_1 = t_2 \to a) \\ a \text{ new} \quad \mathit{sat}(P, C')\end{array}}{\Gamma, e_1\ e_2 \vdash_{\mathit{inf}} (C', a)}$$

$$\text{(Class)}\quad \frac{\begin{array}{c}\bar{a} = \mathit{fv}(\bar{t}) \backslash \mathit{fv}(\Gamma) \quad \Gamma' = \Gamma \cup \{x_i : \forall \bar{a}, \bar{b}_i . TC\ \bar{t} \wedge C'_i \Rightarrow t_i \mid i \in I\} \\ \Gamma', p \vdash_{\mathit{inf}} (C, t')\end{array}}{\Gamma, \begin{array}{l}\textsf{class } C' \Rightarrow TC\ \bar{a} \mid fd_1, \ldots, fd_m \\ \textsf{where } [x_i : \forall \bar{b}_i . C'_i \Rightarrow t_i]_{i \in I}\end{array} \textsf{ in } p \vdash_{\mathit{inf}} (C, t')}$$

$$\text{(Inst)}\quad \frac{\begin{array}{c}\Gamma, e_i \vdash_{\mathit{inf}} (C'_i, t'_i) \quad \sigma'_i = \mathit{gen}(\Gamma, C'_i, t'_i) \\ (x_i : \forall \bar{a}_i . (TC \bar{t} \wedge C_i) \Rightarrow t_i) \in \Gamma \quad \bar{a}'_i = \mathit{fv}(t'_i, C'_i) \backslash \mathit{fv}(\Gamma) \\ \mathit{unambig}(P, \forall \bar{a}'_i . (TC \bar{t}' \wedge C'_i) \Rightarrow t'_i) \\ \mathit{subsumes}(P, (\forall \bar{a}'_i . (TC\ \bar{t}' \wedge C'_i) \Rightarrow t'_i), (\forall \bar{a}_i . (TC\ \bar{t} \wedge C''_i \wedge \bar{t}' = \bar{t}) \Rightarrow t_i)) \\ \text{for } i \in I \\ \Gamma, p \vdash_{\mathit{inf}} (C'', t'') \\ \mathit{sat}(P, C''')\end{array}}{\Gamma, \textsf{instance } c_1, \ldots, c_m \Rightarrow TC\ \bar{t}' \textsf{ where } [x_i = e_i]_{i \in I} \textsf{ in } p \vdash_{\mathit{inf}} (C'', t'')}$$

Fig. 4. Inference System

4.4.3 Type inference

To support type inference for type classes, we follow the standard approach, by generating constraints from the program text and checking whether constraints are *satisfiable* and types are *unambiguous.* In case of type annotations, we perform an additional *subsumption* check. The important point is that all three checks can be phrased in terms of the following CHR-based procedures. In Figure 4 we specify an inference system in terms of judgments of the form $\Gamma, p \vdash_{\mathit{inf}} (C, t)$ where environment Γ and program p are input parameters and constraint C and type t are output parameters. Note that in rule (Var) we assume as an invariant that types in the environment are unambiguous and their constraint component is satisfiable. In rule (Let) and (Inst) we make use of a generalization procedure to build type schemes. We define $\mathit{gen}(\Gamma, C, t) = \sigma$ where $\bar{a} = \mathit{fv}(C, t) \backslash \mathit{fv}(\Gamma)$, $\sigma = \forall \bar{a}.C \Rightarrow t$ and fresh variables $\bar{b}$.

The satisfiability check is defined as follow:

$$\begin{array}{lll} \multicolumn{3}{l}{sat(P, C)} \\ = & True & \text{if } C \rightarrowtail_P^* C' \text{ such that } \models \exists fv(C'_e).C'_e \\ = & False & \text{otherwise} \end{array}$$

We simply build a CHR derivation for C (assuming the CHRs are terminating) and check that the equations in the final store have a unifier. Note that C'_e refers to all equations in C'_e and the condition $\models \exists fv(C'_e).C'_e$ holds if a unifier exists.

Unambiguity is checked as follow:

$$\begin{array}{lll} \multicolumn{3}{l}{unambig(P, \forall \bar{a}.C \Rightarrow t)} \\ = & True & \text{if } C \wedge \rho(C) \wedge t = \rho(t) \rightarrowtail_P^* C' \\ & & \text{such that } \models C' \supset (a = \rho(a)) \text{ for each } a \in \bar{a} \\ & & \text{where } \rho \text{ is a variable renaming on } \bar{a} \\ = & False & \text{otherwise} \end{array}$$

Note that w.l.o.g. we assume that $\bar{a} \subseteq fv(C, t)$.

The subsumption checking procedure is defined as follows:

$$\begin{array}{lll} \multicolumn{3}{l}{subsumes(P, \forall \bar{a}.C \Rightarrow t, \forall \bar{a}'.C' \Rightarrow t')} \\ = & True & \text{if } C' \wedge t' = a' \wedge a = a' \rightarrowtail_P^* C_1, \text{ and} \\ & & C' \wedge t' = a' \wedge a = a' \wedge t = a \wedge C \rightarrowtail_P^* C_2 \\ & & \text{such that } \models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2) \\ & & \text{where } a,\ a' \text{ are new variables and} \\ & & V = fv(C' \wedge t' = a' \wedge a = a') \cup fv(\forall \bar{a}.C \Rightarrow t) \\ = & False & \text{otherwise} \end{array}$$

The idea here is to rewrite $\forall \bar{a}.C \Rightarrow t$ into the equivalent type scheme $\forall \bar{a}, a.C \wedge t = a \Rightarrow a$ where a is a new variable, and then use equivalence testing to test implication.

We know, from earlier work (?), that these procedures are *sound* for arbitrary CHRs. That is all answers are correct. We also know that if CHRs are confluent, terminating, and range-restricted (Section 3.2), then these procedures are:

- *decidable:* the algorithm always terminates; and
- *complete:* if there is an answer then the algorithm will find it.

In fact, the completeness result for the subsumption procedure additionally demands that the inferred type must be unambiguous. We refer the interested reader to (?) for more details.

Under what conditions does a source program involving functional dependencies generate a set of CHRs that enjoys these properties? That is the question to which we now turn.

5 Main Result

Our main result is that type inference for Haskell-FD programs that satisfy the three FD-Conditions of Section 4.1 is sound, complete, and decidable.

We prove this result in two steps.

Theorem 1 (Completeness and Decidability) Let p be a set of class and instance declarations (Definition 1). Then, if the CHR system $MPTC(p)$ is confluent, terminating, and range-restricted, and p satisfies the FD-Conditions (Section 4.3), then $CHR(p)$ is also confluent, terminating, and range-restricted. (Recall that $MPTC(p)$ and $CHR(p)$ were defined Section 4.2.)

The theorem says that, given any variant of Haskell (not necessarily even satisfying the Basic Conditions of Definition 5) that is well-behaved (by which we mean the CHR system is confluent, terminating, and range-restricted) without functional dependencies, adding functional dependencies to it maintains this good behavior, provided the FD-conditions hold. A proof is given in Appendix A.1.

The Basic Conditions were chosen by the designers of Haskell as a simple, syntactic way to guarantee that the type class system is well behaved. More precisely, if program p satisfies the Basic Conditions, then $MPTC(p)$ is confluent, terminating, and range-restricted. Our desired corollary follows immediately:

Corollary 1 (Basic FD-Conditions Completeness and Decidability) Type inference is sound, complete, and decidable for a Haskell program that satisfies both the Basic Conditions and the FD-Conditions.

Ross Paterson recently proposed the following "alternative" conditions for class and instance declarations.

Definition 11 (Paterson Conditions) The Paterson Conditions on class and instance declarations are these:

- The context C of a class declaration can mention only type variables, not type constructors, and in each individual class constraint CC all the type variables are distinct.
- For each declaration `instance` C `=>` $TC\ t_1 \ldots t_n$:
  - No variable has more occurrences in a type class constraint in the context C than the head $TC\ t_1 \ldots t_n$.
  - Each type class constraint in the context C has fewer constructors and variables (taken together counting repetitions) than the head.
- The instance declarations must not overlap. That is, for any two instances declarations `instance` C `=>` $TC\ t_1 \ldots t_n$ and `instance` C' `=>` $TC\ t'_1 \ldots t'_n$ there is no substitution ϕ such that $\phi(t_1) = \phi(t'_1)$,....,$\phi(t_n) = \phi(t'_n)$.

The first Basic Condition is only enforced on class declarations whereas the second Basic Condition is completely replaced by some alternative conditions. The third Basic Condition remains unchanged.

If we only focus on instances, the Paterson Conditions strictly subsume the Basic and Bound Variable Conditions. That is, any instance which satisfies the Basic and Bound Variable Conditions also satisfies the Paterson Conditions. For example, the first part of the second Paterson Condition implies that each variable in the context must appear in the head of a instance declaration. Otherwise, there would be more occurrences in the head than the context. Hence, the Bound Variable Condition for instances is satisfied.

However, there are instances which satisfy the Paterson Conditions but not the Basic Conditions. For example, the following is a legal Paterson program.

```
class C a b
instance (C b b, C b [a]) => C [a] (b,b)
```

But this this program does not satisfy the Basic Conditions. The constraint `C b [a]` mentions a type constructor and there are repeated occurrences of `b` in `C b b`.

In order to apply our Theorem 1 to Paterson programs p, we first need to verify that the Paterson Conditions indeed guarantee termination and confluence of $MPTC(p)$. Roughly, the proof goes as follows.

Following (**?**), we show that there exists a ranking function such that for each declaration `instance` $(CC_1, ..., CC_n)$ `=>` $Head$ we have that $rank(Head) > rank(CC_i)$ for each $i = 1, ..., n$. There is no interaction between derivations of context constraints, hence, it is sufficient to compare them individually against the head. As our ranking function we take $rank(TC\ t_1 \dots t_n) = rank(t_1) + ... + rank(t_n)$ and $rank(T\ t_1 \dots t_n) = 1 + rank(t_1) + ... + rank(t_n)$ where $rank(a) > 0$ for each variable a. Constant type class constraints are ignored from consideration. This "termination" order implies that instances are terminating. A similar argument applies to class rules. The third Paterson Condition guarantees confluence. Hence, we obtain the following result.

Lemma 1 Let p be a Haskell program satisfying the Paterson Conditions. Then, $MPTC(p)$ is confluent and terminating.

As an immediate consequence, we can restate Corollary 1 for Paterson programs.

Corollary 2 (Paterson FD-Conditions Completeness and Decidability) Type inference is sound, complete, and decidable for a Haskell program that satisfies both the Paterson Conditions and the FD-Conditions.

In the rest of this section we explain exactly why the three FD-Conditions ensure that the CHRs are well-behaved. Many useful programs do not satisfy the FD-Conditions, though, and we discuss ways to weaken them in Section 6.

5.1 Confluence

It is essential that the CHR system is confluent (Definition 2), or else all is lost. The improvement rules are essential to guarantee confluence for program satisfying the FD-Conditions.

Example 12 Consider again Example 7, which satisfies the three FD-Conditions. It translates to the following CHRs:

```
rule  Ord a                  ==> Eq a   -- (Super)
rule  Coll [a] a            <==> Ord a  -- (Inst)
rule  Coll [a] b             ==> a=b,   -- (Imp)
rule  Coll c e1, Coll c e2  ==> e1=e2  -- (FD)
```

This CHR system is indeed confluent. For any goal, all possible derivations will lead to the same result. We illustrate this on a single example, the critical pair between the (Imp) and (FD) rule.

$$\begin{array}{ll} & Coll\ [a]\ a, Coll\ [a]\ c \\ \rightarrowtail_{FD} & Coll\ [a]\ a, a = c \\ \rightarrowtail_{Inst} & Ord\ a, a = c \\ \rightarrowtail_{Super} & Ord\ a, Eq\ a, a = c \end{array}$$

In the first step, we apply rule (FD). Note that we assume set semantics. Hence, we remove the duplicate copy of $Coll\ [a]\ a$. Note that there is another derivation which yields the same result.

$$\begin{array}{ll} & Coll\ [a]\ a, Coll\ [a]\ c \\ \rightarrowtail_{Inst} & Ord\ a, Coll\ [a]\ c \\ \rightarrowtail_{Imp} & Ord\ a, Coll\ [a]\ a, a = c \\ \rightarrowtail_{Inst} & Ord\ a, a = c \\ \rightarrowtail_{Super} & Ord\ a, Eq\ a, a = c \end{array}$$

Here, we apply rule (Inst) first instead of rule (FD). Note that in the second step, we immediately normalize the store by applying the unifier to user-defined constraints. The important observation is that both derivations yield equivalent final stores. Indeed, the above CHRs are confluent. Note that if we remove rule (Imp), the CHRs become non-confluent.

Non-confluence can arise in several ways. Recall Example 5, which does not satisfy the Consistency Condition. Its translation to CHRs is as follows:

```
rule Mul a b c, Mul a b d ==> c=d  -- (M1)
rule Mul Int Float c <==> True     -- (M2)
rule Mul Int Float c  ==> c=Float  -- (M3)
rule Mul Int Float c <==> True     -- (M4)
rule Mul Int Float c  ==> c=Int    -- (M5)
```

We can easily find two contradicting CHR derivations. For example, consider

$$\begin{array}{lll} Mul\ Int\ Float\ c & \rightarrowtail_{M3,M2} & c = Float \\ Mul\ Int\ Float\ c & \rightarrowtail_{M5,M4} & c = Int \end{array}$$

Hence, the CHR system is not confluent.

It is rather intuitive that the Consistency Condition is *necessary* to guarantee confluence. It is much less obvious that the Consistency Condition alone is not *sufficient*. Here is a counter-example:

Example 13 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
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
```

The Consistency Condition is trivially fulfilled, because there is only one instance. 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)
```

These CHRs are terminating but non-confluent! Examining the critical pair for rules (E1) and (E2), we find that (applying (E2) twice)

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

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))

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

It turns out that adding the Coverage Condition (which is violated in this example) is sufficient to guarantee confluence.

5.2 Termination

If type inference is to be decidable, then the CHR system must be terminating (Definition 1). One very obvious way in which non-termination can arise, even without functional dependencies, is by allowing instance declarations such as:

```
instance Foo [Maybe a] => Foo [a]
```

The context of the declaration includes a constraint `Foo [Maybe a]` that is no smaller than the head of the declaration, `Foo [a]`. Although such instance declarations can sometimes be extremely useful, they are excluded by the Basic Conditions, and we do not consider them further here. Indeed, the CHRs generated from a program satisfying the Basic Conditions, and without functional dependencies, are always terminating.

Adding functional dependencies changes the picture.

Example 14 Recall these declarations from Example 3:

```
class Mul a b c | a b -> c
instance Mul a b c => Mul a (Vec b) (Vec c)
```

whose instance declaration does not satisfy the Coverage Condition. The corresponding instance-improvement CHR is this:

```
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`; notice the repeated type variable `b`. Applying the rules, we find that

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

Since the last line contains a copy of the first, the CHR derivation, and hence type inference, is non-terminating. The problem arises because fixing the first two arguments of `Mul` to t_1 and (`Vec` t_2) respectively fixes the third argument to (`Vec` t_3), but does not fix the type t_3.

The Coverage Condition does not, by itself, guarantee a terminating set of CHRs: we need the Bound Variable Condition as well.

Example 15 Consider the program.

```
class D a
class F a b | a->b
instance F [a] [[a]]
instance (D c, F a c) => D [a]
```

This program satisfies the Consistency and Coverage Conditions, but the instance declaration for `D [a]` violates the Bound Variable Condition — the variable `c` in the context is not mentioned in the instance head `D [a]`.

Translating these declarations to CHRs gives these rules, among others:

```
rule F [a] b   <==> b=[[a]]      -- (R1)
rule D [a]     <==> D c, F a c   -- (R2)
```

Note that breaking the Bound Variable Condition results in non-range restricted (therefore also non-variable-restricted) CHR (R2). Now we can get the following derivation:

$$\begin{array}{ll} & D\ [[a]] \\ \rightarrowtail_{R2} & D\ c,\ F\ [a]\ c \\ \rightarrowtail_{R1} & D\ [[a]],\ F\ [a]\ [[a]],\ c = [[a]] \end{array}$$

The last line contains a copy of the first, so we have found a non-terminating derivation. The intuition is that while the context of the instance, `(D c, F a c)`, looks innocuous enough, `F`'s dependency forces $c = [[a]]$ causing the loop.

5.3 Range Restriction

Example 15 from the previous section shows that non-range-restricted CHRs may lead to non-termination. Hence, range-restriction is an important condition which must be satisfied. Interestingly, even if we can establish termination in some other way, range-restriction in itself is important to guarantee completeness of type inference.

Example 16 Consider the following program.

```
class Q a where q :: a
class R a b
instance R a b => Q [a]  -- (Q)
instance R Bool Float    -- (R1)
instance R Bool Char     -- (R2)

exp1 = q :: [Bool]
```

Note that expression `exp1` demands a subsumption check.[2] The instance declaration `(Q)` violates the Bound Variable Condition, and hence the resulting set P of CHRs is not range-restricted. However, the CHRs are nevertheless confluent and terminating:

```
rule Q [a] <==> R a b        -- (Q)
rule R Bool Float <==> True  -- (R1)
rule R Bool Char <==> True   -- (R2)
```

The trouble is that (a) the program is well-typed (Section 4.4.1), (b) the typing derivation is unambiguous (Definition 10), but nevertheless (c) the meaning of the program depends on an arbitrary choice of which instance declaration to use. To see this, consider the subsumption test for `exp1`, namely $[\![P]\!] \models (\forall a.Q\ a \Rightarrow a) \preceq$ `[Bool]`; i.e., the inferred type subsumes the annotated type. This holds if $[\![P]\!] \models (\exists b.R\ \texttt{Bool}\ b)$ and, in the logical reading of CHRs, we can replace the existentially quantified variable b by either `Float` or `Char`.

This arbitrary choice would affect the meaning of the program, just as described in Section 4.4.2, and hence the inference procedure which tests for subsumption among $(\forall c.Q\ c \Rightarrow c)$ and *Bool* w.r.t. P fails. Concretely, following the subsumption procedure in Section 4.4.3 gives

$$Bool = a', a = a' \rightarrowtail^* Bool = a', a = a'$$

and

$$Bool = a', a = a', c = a, Q\ c \rightarrowtail^* Bool = a', a = a', c = a, R\ Bool\ b$$

Note that the two final stores are not logically equivalent.

The requirement that the CHRs be range-restricted, conservatively guarantees that this situation cannot arise, in much the same way as the restriction to unambiguous types and judgements (Section 4.4.2). Range restriction of the CHR system is in turn guaranteed by the Coverage and Bound Variable Conditions on the source program.

[2] We do not explicitly allow for type annotated expressions in our syntax. It should be clear that this extension is straightforward.

6 Variations of Functional Dependencies

Thus far we have used CHRs to prove that the three FD-conditions (Coverage, Consistency and Bound Variable) are sufficient to guarantee sound, complete, and decidable type inference. However, while they are *sufficient*, they are not always *necessary*. In particular, the Coverage and Bound Variable Conditions exclude reasonable and useful programs, as we saw in Section 4.1. In contrast, the Consistency Condition seems entirely reasonable, and is essential for confluence. Hence, our goal is to seek for more “liberal” FDs which safely break the Bound Variable and Coverage Condition.

First, in Section 6.1 we show how to safely weaken the Coverage Condition in a fairly natural way while maintaining confluence. We know from the earlier Section 5.2 that the Coverage Condition is essential for termination. Hence, we will assume termination from now on. It turns out, that the resulting CHRs can be non-confluent, unless the original FDs are “full”, which again surprised us. In Section 6.2, we establish some sufficient conditions under which can transform non-full FDs to full FD programs such that we achieve confluence. Abandoning the Coverage Condition allows us to make use of a richer class of FDs which we discuss in Section 6.3.

As said we must assume termination once we break the Coverage Condition. Though, this may be a too strong requirement because CHRs may only terminate for some goals. Fortunately, all of our results from Section 6.1 apply to “terminating” goals. Similarly, we can safely drop the Bound Variable Condition as long as we can guarantee that a specific inference goal is “range-restricted”. This is what we discuss in Section 6.4.

6.1 Weakening the Coverage Condition

Our next goal is to explore various ways to weaken the Coverage Condition that still ensure confluence.

Given an instance declaration

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

and a functional dependency $a_{i_1}, ..., a_{i_k}$ `->` a_{i_0} for class TC, our key intuition is this:

$$\text{fixing } t_{i_1}, ..., t_{i_k} \text{ should fix } t_{i_0}$$

But that might happen because of the functional dependencies expressed by the context C, rather than simply because $fv(t_{i_0}) \subseteq fv(t_{i_1}, ..., t_{i_k})$, and that is what the Weak Coverage Condition says.

Definition 12 (Weak Coverage Condition) For each functional dependency $a_{i_1}, ..., a_{i_k}$ `->` a_{i_0} for class TC and instance declaration

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

we must have that $fv(t_{i0}) \subseteq closure(C, fv(t_{i1}, \ldots, t_{ik}))$ where

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

The Weak Coverage and Consistency Condition are essential though not sufficient to guarantee confluence.

Example 17 Consider the following program.

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

Here is the translation to CHRs.

```
rule F a b1 c, F a b2 d  ==> b1=b2       -- (FD)
rule F Int Bool Char    <==> True        -- (Inst1)
rule F Int a b           ==> a=Bool      -- (Imp1)
rule F [a] [b] Bool     <==> F a b Bool -- (Inst2)
rule F [a] c d           ==> c=[b]       -- (Imp2)
```

Now consider two CHR reduction sequences, both starting with $F\ [a]\ [b]\ Bool, F\ [a]\ b_2\ d$:

$$\begin{array}{ll} & F\ [a]\ [b]\ Bool, F\ [a]\ b_2\ d \\ \rightarrowtail_{FD} & F\ [a]\ [b]\ Bool, F\ [a]\ [b]\ d, b_2 = [b] \\ \rightarrowtail_{Inst2} & F\ a\ b\ Bool, F\ [a]\ [b]\ d, b_2 = [b] \end{array}$$

and

$$\begin{array}{ll} & F\ [a]\ [b]\ Bool, F\ [a]\ b_2\ d \\ \rightarrowtail_{Inst2} & F\ a\ b\ Bool, F\ [a]\ b_2\ d \\ \rightarrowtail_{Imp2} & F\ a\ b\ Bool, F\ [a]\ [c]\ d, b_2 = [c] \end{array}$$

The final stores of the two sequences are not logically equivalent, and hence, the above CHRs are non-confluent. In essence, the above derivations show that the critical pair between the FD rule and the second instance is not joinable.

The gist of the (non-confluence) problem here is that the FD is not "full". That is, in the declaration `class F a b c | a->b` not all class parameters are involved in the FD.

Definition 13 (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 $k = n - 1$.

For full FDs we can "shorten" the translation to CHRs by combining the instance improvement and instance rules into one rule.

Lemma 2 (Full FD Translation Equivalence) For full functional dependencies we can equivalently express the instance and instance improvement CHR in terms of one single CHR. Specifically, 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 (instead of the instance and instance improvement CHR):

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

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

The real benefit of full FDs is that together with the Weak Coverage Condition they guarantee confluence

Let's consider a variant of the example in which the functional dependency is "full"; we simply drop the third parameter. Using Lemma 2, we can combine the instance CHR and instance improvement CHR into one rule. For simplicity, we leave out the rule corresponding to the first instance.

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

Now, we find that

$$\begin{array}{ll} & F\ [a]\ [b], F\ [a]\ b_2 \\ \rightarrowtail_{Full-FD} & F\ [a]\ [b], b_2 = [b] \\ \rightarrowtail_{Full-Inst-Imp} & F\ a\ b, b_2 = [b] \end{array}$$

and

$$\begin{array}{ll} & F\ [a]\ [b], F\ [a]\ b_2 \\ \rightarrowtail_{Full-Inst-Imp} & F\ a\ b, F\ [a]\ b_2 \\ \rightarrowtail_{Full-Inst-Imp} & F\ a\ b, F\ a\ c, b_2 = [c] \\ \rightarrowtail_{Full-FD} & F\ a\ b, b_2 = [b], c = b \end{array}$$

Final stores are logically equivalent. Hence, the critical pair between the full FD rule and the second instance is now joinable. Note that the Weak Coverage Condition is crucial (see last step in the second derivation).

The above example shows that we need to assume Consistency and Weak Coverage *and* FDs are full. We say that a program p satisfies the *Liberal-FD Conditions* iff the Consistency and Weak Coverage Conditions are satisfied.

Theorem 2 (Liberal-FD Confluence) Let p be a set of Haskell class and instance declarations such that all FDs are full, p satisfies the Liberal-FD Conditions and the CHR system $MPTC(p)$ is confluent and terminating. If the CHR system $CHR(p)$ is terminating the $CHR(p)$ is also confluent.

A proof is given in Appendix A.2. In our (confluence) proof we only check that critical pairs are joinable. From that we can conclude confluence if we have termination. Hence, termination is a necessary assumption because Weak Coverage does not guarantee termination in general.

Though, we can identify a class of programs for which we can guarantee termination.

Example 18 Here is an excerpt of the Control.Monad library in GHC.

```
class (Monad m) => MonadReader r m | m -> r
instance (Monoid w, MonadReader r m) => MonadReader r (WriterT w m)
```

The instance breaks the Coverage Condition because $r \notin fv(WriterT\ w\ m)$. But variable `r` is fixed by `MonadReader r m`. Hence, the Weak Coverage Condition is clearly satisfied. What is interesting about this example is that the resulting CHRs are terminating.

```
rule MonadReader r1 m, MonadReader r2 m ==> r1=r2
rule MonadReader r (WriterT w m) <==> Monoid w, MonadReader r m
rule MonadReader r' (WriterT w m) ==> r'=r
```

The last rule is the instance improvement rule. In case we break the Coverage Condition such rules introduce fresh variables and therefore CHRs may become non-terminating (see the earlier Example 14). Though, this rule is trivial here. Effectively, we can replace this rule by

```
rule MonadReader r (WriterT w m) ==> True
```

Hence, CHRs are terminating. Then, Theorem 2 applies and thus we obtain confluence. Our program also satisfies the Bound Variable Condition. Hence, type inference is complete and decidable here.

The conclusion is that if the range FD parameter in the instance head is a variable, the resulting instance improvement rule is trivial and therefore does not endanger termination of CHRs.

Definition 14 (Terminating Weak Coverage Condition) For each functional dependency $a_{i_1}, ..., a_{i_k}$ `->` a_{i_0} for class TC and instance declaration

`instance` C `=> TC` $t_1 ... t_n$

where the Coverage Condition is broken, we must have that the Weak Coverage Condition is satisfied and t_{i_0} is a variable.

It follows immediately that such programs maintain the good CHR properties.

Corollary 3 Let p be a set of class and instance declarations such that the CHR system $MPTC(p)$ is confluent, terminating, and range-restricted, all FDs are full, the Terminating Weak Coverage and Consistency Condition is satisfied. Then, $CHR(p)$ is also confluent, terminating, and range-restricted.

In the event that we cannot guarantee termination and range-restriction in general not all is lost as long as we restrict ourselves to "terminating" and "range-restricted" goals.

Corollary 4 Let p be a set of Haskell class and instance declarations such that all FDs are full, p satisfies the Liberal-FD Conditions and $MPTC(p)$ is confluent. Let C be constraint for which CHRs are terminating and range-restricted. Then, the CHR solver is confluent for goal C.

The above follows from results which we discuss in the upcoming Section 6.4.

The following example shows that we can generalise the Weak Coverage Condition still further.

Example 19 Consider

```
class G a b | a->b
class H a b | a->b
class F a b | a->b
instance (G a c, H c b) => F [a] [b]
```

The above instance does not satisfy the Weak Coverage condition. However, the range variable `b` is captured by `H c b` where `c` is captured by `G a c` where `a` is in the domain of the FD. A more realistic example is given in Appendix B (see instance (S3)). The difference to our previous definition is that as long there is a "sequence" of FDs which captures the range variable, we are fine.

Definition 15 (Refined Weak Coverage Condition) For each functional dependency $a_{i_1}, ..., a_{i_k}$ `->` a_{i_0} for class TC and

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

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.

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

It is not difficult to restate Theorem 2 (and Corollaries 3 and 4) using the Refined Weak Coverage Condition, where the functional dependencies are full.

6.2 Sound Non-Full Functional Dependencies

We investigate the issue of non-confluence raised by non-full FDs in Example 17. Our goal is to transform non-full FD programs into full FD programs such that we can apply Theorem 2 to the resulting program. The idea is to remove non-full FD relations and introduce appropriate subclasses with full FD relations instead.

Example 20 Recall Example 17.

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

As observed the CHR arising out of the above program are non-confluent. We perform a simple transformation where we drop `F`'s FD `a->b` but introduce a subclass `G a b | a->b`. Additionally, we project all of `F`'s instances onto `G`.

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

The new class G is somewhat artificial. Its sole purpose is to encode the improvement condition specified by F's non-full FD. Consider the CHR arising.

```
rule F a b c             ==> G a b      -- (Super)
rule F Int Bool Char    <==> True       -- (F-Inst1)
rule F [a] [b] Bool     <==> F a b Bool -- (F-Inst2)
rule G a b, G a c        ==> b=c        -- (G-FD)
rule G Int Bool         <==> True       -- (G-Inst1)
rule G Int a             ==> a=Bool     -- (G-Imp1)
rule G [a] [b]          <==> G a b      -- (G-Inst2)
rule G [a] c             ==> c=[b]      -- (G-Imp2)
```

For example, rule (FD) in Example 17 is encoded by rules (Super) and (G-FD). Similarly, we find that the all other improvement rules in Example 17 are represented. Note that the above CHRs are "stronger" than the ones in Example 17 due to the presence of the new type class G. The benefit of the encoding is that the above CHRs are confluent. Note that FDs are now full and the introduction of G does not introduce any non-confluence which would not have been there before. Hence, Theorem 2 applies. Consider the previously problematic goal which is now joinable.

$$\begin{array}{ll}
 & F\ [a]\ [b]\ Bool, F\ [a]\ b_2\ d \\
\rightarrowtail^2_{Super} & F\ [a]\ [b]\ Bool, F\ [a]\ b_2\ d, G\ [a]\ [b], G\ [a]\ b_2 \\
\rightarrowtail_{G-FD} & F\ [a]\ [b]\ Bool, F\ [a]\ [b]\ d, G\ [a]\ [b], G\ [a]\ [b], b_2 = [b] \\
\rightarrowtail_{F-Inst2} & F\ a\ b\ Bool, F\ [a]\ [b]\ d, G\ [a]\ [b], G\ [a]\ [b], b_2 = [b] \\
\rightarrowtail_{G-Inst2} & F\ a\ b\ Bool, F\ [a]\ [b]\ d, G\ a\ b, G\ [a]\ [b], b_2 = [b]
\end{array}$$

and

$$\begin{array}{ll}
 & F\ [a]\ [b]\ Bool, F\ [a]\ b_2\ d \\
\rightarrowtail_{F-Inst2} & F\ a\ b\ Bool, F\ [a]\ b_2\ d \\
\rightarrowtail_{Super} & F\ a\ b\ Bool, F\ [a]\ b_2\ d, G\ [a]\ b_2 \\
\rightarrowtail_{G-Imp2} & F\ a\ b\ Bool, F\ [a]\ [c]\ d, G\ [a]\ [c], b_2 = [c] \\
\rightarrowtail_{Super} & F\ a\ b\ Bool, F\ [a]\ [c]\ d, G\ a\ b, G\ [a]\ [c], b_2 = [c] \\
\rightarrowtail_{G-Inst2} & F\ a\ b\ Bool, F\ [a]\ [c]\ d, G\ a\ b, G\ a\ c, b_2 = [c] \\
\rightarrowtail_{G-FD} & F\ a\ b\ Bool, F\ [a]\ [c]\ d, G\ a\ b, G\ a\ c, b_2 = [b], c = b
\end{array}$$

The crucial difference is that even after using `F-Inst2` in the second derivation we are still able, eventually, to use the `G-FD` rule to equate second arguments.

The above transformation trick only works if we can guarantee that the projection onto the "full" part of a non-full FD instance is unique.

Example 21 Consider the following variant of our running example

```
class F a b c | a->b
class H a b c | a->b
instance F a b Bool => F [a] [b] Bool  -- (F1)
instance H a b => F [a] [b] Char       -- (F2)
```

We cannot apply our transformation trick from above. Otherwise, we obtain two overlapping instances (G1) and (G2) which results in a non-confluent program.

```
class G a b | a->b
class G a b => F a b c
class I a b | a->b
class I a b => H a b c
instance F a b Bool => F [a] [b] Bool  -- (F1)
instance H a b => F [a] [b] Char       -- (F2)
instance G a b => G [a] [b]   -- (G1)
instance I a b => G [a] [b]   -- (G2)
```

Our solution is to apply the transformation from non-full to full FDs to only those programs which satisfy the following condition.

Definition 16 (Transformable Non-Full FDs) Consider a class declaration

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

We say that TC is *transformable* to a full FD iff for each FD $fd_i \equiv a_{i_1}, ..., a_{i_k}$ `->` a_{i_0} there are no two declarations `instance` C `=> TC` $t_1 ... t_n$ and `instance` C' `=> TC` $t'_1 ... t'_n$ such that $\phi(t_{i_1}) = \phi(t'_{i_1}), ..., \phi(t_{i_k}) = \phi(t'_{i_k})$ for some substitution ϕ.

Definition 17 (Non-Full to Full FD Translation) Given a type class program p where each type class is transformable to a full FD, we apply the following transformations exhaustively yielding a type class program p' where p' only contains full FDs.
For each class declaration

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

in p where for some I we have that fd_i is not full if $i \in I$, otherwise fd_i is full. We replace the above class declaration by

$$\begin{array}{ll} \texttt{class } (C, \bigcup_{i \in I} \texttt{TC}_{fd_i}) \texttt{ => TC } a_1 \ ... \ a_n \texttt{ | } \bigcup_{j \in \{1,...,m\} - I} fd_j & \\ \texttt{class TC}_{fd_i} \ a_{i_1} \ ... \ a_{i_k} \texttt{ | } fd_i \quad \text{for } i \in I & \end{array}$$

where $\texttt{TC}_{fd_i}$ is a new subclass of `TC`.
For each instance declaration

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

in p with a non-full FD fd_i we additionally introduce

$$\texttt{instance } \mathit{proj}(C) \texttt{ => TC}_{fd_i} \ t_{i_1} \ ... \ t_{i_k}$$

where

$$\begin{array}{l} \mathit{proj}(CC_1, ..., CC_n) \ = \ \mathit{proj}(CC_1) \cup .... \cup \mathit{proj}(CC_n) \\ \mathit{proj}(\texttt{TC } t_1 ... t_n) \ = \\ \qquad \left\{ \begin{array}{l|l} \texttt{TC}_{fd_i} \ t_{i_1} \ ... \ t_{i_k} & \texttt{class } C \texttt{ => TC } a_1 \ ... \ a_n \texttt{ | } fd_1, ..., fd_m \in p, \\ & fd_i \text{ is not full} \end{array} \right\} \\ \cup \ \left\{ \begin{array}{l|l} \texttt{TC } t_1 ... t_n & \texttt{class } C \texttt{ => TC } a_1 \ ... \ a_n \texttt{ | } fd_1, ..., fd_m \in p, \\ & fd_i \text{ is full} \end{array} \right\} \end{array}$$

All other instances remain unchanged.

For each non-full FD we introduce an additional superclass with a full FD but drop the non-full FD from the class declaration. Thus, all resulting FDs are full.

For convenience, we abuse notation and use set notation to denote a sequence of constraints and FDs. For each instance declaration with a non-full FD we build the projection onto its full part.

We find that all improvement conditions imposed by non-full FDs in the original program are equivalently represented by super-classes with full FDs in the transformed program. Therefore, the above transformation is sound in the sense that the logical reading of CHRs resulting from the transformed program logically entails the CHRs from the original program.

Lemma 3 (Sound Transformation) Let p be a set of Haskell class and instance declarations. where Let p' be p's transformation to full FDs as defined by Definition 17. Then $[\![MPTC(p')]\!] \models [\![MPTC(p)]\!]$.

We can also verify that confluence and range-restriction of instances/super-classes and the Liberal-FD Conditions are preserved. By assumption type classes are transformable. Hence, none of the new instances will break confluence. The same applies to range-restriction. By construction, the newly generated instances inherit the Liberal-FD Conditions from the classes and instances they were derived from.

Lemma 4 (Liberal-FD and $MPTC(p)$ Conf/RR Preservation) Let p be a set Haskell class and instance declarations such that each type class is transformable to a full FD, p satisfies the Liberal-FD Conditions and $MPTC(p)$ is confluent and range-restricted. Let p' be p's transformation to full FDs as defined by Definition 17. Then, p' satisfies the Liberal-FD Conditions and $MPTC(p')$ is confluent and range-restricted.

We conjecture that also termination is preserved but leave detailed investigations to future work. Note that termination is of less concern here because instances involved in the full FD transformation do not satisfy the Coverage Condition. Hence, termination must be enforced by additional checks anyway.

In summary, based on the above results we can avoid non-confluence in case of transformable non-full FDs. We simply generate a stronger set of CHRs (Definition 17). Hence, Theorem 2 is applicable.

6.3 Multi-Range Functional Dependencies

In case of more liberal FDs we may make use of a richer class of FDs of the form $a_{i_1}, ..., a_{i_k}$ `->` $a_{j_1}, ..., a_{j_{i_0}}$ where $\{i_1, ..., i_k\}$ and $\{j_1, ..., j_{i_0}\}$ are two disjoint subsets of $\{1...n\}$. We refer to such FDs as *multi-range* FDs. Compare this to the FDs described before which are of *single-range*. It is straightforward to translate multi-range FDs to CHRs.

Definition 18 (Multi-Range FD Translation) Consider a class declaration

`class` C `=>` `TC` $a_1 \ ... \ a_n$ `|` $fd_1, ..., fd_m$

where fd_i may be multi-range FDs.

Multi-Range FD CHR: For each multi-range FD $a_{i_1}, ..., a_{i_k}$ `->` $a_{j_1}, ..., a_{j_{i_0}}$ we generate rule TC $a_1 \dots a_n$, TC $\theta(b_1) \dots \theta(b_n)$ ==> $a_{j_1} = b_{j_1}, \dots, a_{j_{i_0}} = b_{j_{i_0}}$

where $\theta(b_{i_j}) = a_{i_j}$, $j \in \{1, ..., k\}$ and $\theta(b_l) = b_l$ otherwise.

Multi-Range Instance Improvement CHR: For each multi-range FD $a_{i_1}, ..., a_{i_k}$ -> $a_{j_1}, ..., a_{j_{i_0}}$ and instance declaration instance C => TC $t_1 \dots t_n$ we generate

rule TC $\theta(b_1) \dots \theta(b_n)$ ==> $t_{j_1} = b_{j_1}, \dots, t_{j_{i_0}} = b_{j_{i_0}}$

where $\theta(b_{i_j}) = a_{i_j}$, $j \in \{1, ..., k\}$ and $\theta(b_l) = b_l$ otherwise.

The translation of instances and super-classes remains unchanged.

We note that multi-range FDs impose strictly stronger improvement when the Coverage Condition is violated. Hence, they represent a more expressive class of FDs.

Example 22 Consider the following program which breaks the Coverage Condition.

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

The translation to CHRs is as follows.

```
rule C a b c, C a d e  ==> b=d, c=e       -- (FD)
rule C [a] [b] [b]    <==> C a b b        -- (Inst)
rule C [a] c d         ==> c=[b], d=[b]   -- (Imp)
```

Let us compare this against the following translation resulting from the program where we have replaced `C a b c | a->b c` by `C a b c | a->b, a->c`

```
rule C a b c, C a d e  ==> b=d            -- (FD1)
rule C a b c, C a d e  ==> c=e            -- (FD2)
rule C [a] [b] [b]    <==> C a b b        -- (Inst)
rule C [a] c d         ==> c=[b]          -- (Imp1)
rule C [a] c d         ==> d=[b]          -- (Imp2)
```

Let P_1 be the set of CHRs consisting of rules (FD), (Inst) and (Imp) and let P_2 be the set of CHRs consisting of rules (FD1-2), (Inst) and (Imp1-2). Then, $C\ [a]\ c\ d \rightarrowtail^*_{P_1} C\ a\ b\ b, c = [b], d = [b]$ and $C\ [a]\ c\ d \rightarrowtail^*_{P_2} C\ [a]\ [b_1]\ [b_2], c = [b_1], d = [b_2]$. Clearly, rules (Imp1-2) are weaker than rule (Imp). We are unable to establish a connection between variables b_1 and b_2. Hence, P_1 enforces "stronger" improvement than P_2.

The next obvious question is under which conditions single-range FDs are equivalent to multi-range FDs. The above example suggests that we can in general safely break multi-range FDs into a sequence of single-range FDs if for each instance the variables in the range component are distinct.

Definition 19 (Transformable Multi-Range FDs) Consider a class declaration

class C => TC a_1 ... a_n | $fd_1, ..., fd_m$

where fd_i may be multi-range FDs. We say that TC is *transformable* to single-range FDs iff for each multi-range FD $a_{i_1}, ..., a_{i_k}$ -> $a_{j_1}, ..., a_{j_{i_0}}$ and instance declaration `instance` C => TC $t_1 \dots t_n$ we have that $fv(t_{j_l}) \cap fv(t_{j_m}) = \emptyset$ for $j_l, j_m \in \{j_1, ..., j_{i_0}\}$ and $j_l \neq j_m$.

Under the above condition, we can safely break instance improvement rules with multiple equations on the right-hand side into a sequence of rules with a single equation on the right-hand side. Both sets of rules impose equivalent improvement conditions. Thus, we can conclude the following.

Lemma 5 In case the Transformable Condition is satisfied we can express multi-range FDs $a_{i_1}, ..., a_{i_k}$ -> $a_{j_1}, ..., a_{j_{i_0}}$ equivalently in terms of single-range FDs $a_{i_1}, ..., a_{i_k}$ -> a_{j_1}, ..., $a_{i_1}, ..., a_{i_k}$ -> $a_{j_{i_0}}$.

Note that the Transformable Condition is always satisfied if instances satisfy the Coverage Condition. Hence, we can conclude the following.

Lemma 6 In case the Coverage Condition is satisfied we can express multi-range FDs equivalently in terms of single-range FDs.

Analogously to the previous development we define consistency and weak coverage for multi-range FDs.

Definition 20 (Multi-Range Consistency Condition) Consider a declaration for class TC and any pair of instance declarations for that class:

```
class C => TC a1 ... an | fd1, ..., fdm
instance D1 => TC t1...tn
instance D2 => TC s1...sn
```

Then, for each functional dependency fd_i, of form $a_{i_1}, ..., a_{i_k}$ -> $a_{j_1}, ..., a_{j_{i_0}}$, the following condition must hold: for any substitution ϕ such that

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

we must have that $\phi(t_{j_1}) = \phi(s_{j_1})$,..., $\phi(t_{j_{i_0}}) = \phi(s_{j_{i_0}})$.

Definition 21 (Multi-Range Weak Coverage Condition) For each multi-range functional dependency $a_{i_1}, ..., a_{i_k}$ -> $a_{j_1}, ..., a_{j_{i_0}}$ for class TC and instance declaration

```
instance C => TC t1...tn
```

we must have that $fv(t_{j_1}, ..., t_{j_{i_0}}) \subseteq closure(C, fv(t_{i1}, \dots, t_{ik}))$ where

$$closure(C, vs) \quad = \quad \bigcup_{\substack{\texttt{TC}\ t_1 \dots t_n \in C \\ \texttt{TC}\ a_1 \dots a_n \mid a_{i_1}, ..., a_{i_k} \texttt{ -> } a_{j_1}, ..., a_{j_{i_0}}}} \{fv(t_{j_1}, ..., t_{j_{i_0}}) \mid fv(t_{i_1}, \dots, t_{i_k}) \subseteq vs\}$$

All the results of Sections 6.1 and 6.2 carry over to multi-range FDs. Note that we retain the consistency and weak coverage condition when translating transformable multi-range FDs into single-range FDs. There are in fact also cases where we can transform single-range FDs into equivalent multi-range FDs. This has the advantage that non-full single-range FDs become full multi-range FDs and then we can apply the results from Section 6.1.

Example 23 Here are parts of Example 4.

```
  class Zip a b c | c -> b, c -> a
  instance Zip a b [(a,b)]
  instance Zip (a,b) c e => Zip a b ([c]->e)
```

The program satisfies the Liberal-FD Conditions but the FD is not full. As we will see shortly, the above program is equivalent to the following full multi-range FD program.

```
  class Zip a b c | c -> a b
  instance Zip a b [(a,b)]
  instance Zip (a,b) c e => Zip a b ([c]->e)
```

Here are the CHRs for the multi-range version (see Definition 18).

```
rule Zip a b c, Zip d e c ==> b=e, a=d                  -- (Z1)
rule Zip d e [(a,b)]     <==> e=b, d=a                  -- (Z2)
rule Zip d f ([c]->e)    <==> d=a, f=b, Zip (a,b) c e  -- (Z3)
```

Rule (Z1) represents the multi-range FD. The multi-range FD is full and therefore we can shorten the translation to CHRs by combining the instance and instance improvement CHR, see rules (Z2) and (Z3). In case of the single-range version we would instead find the following CHRs (see Figure 2).

```
rule Zip a b c, Zip d e c ==> b=e                  -- (Z1-1)
rule Zip a b c, Zip d e c ==> a=d                  -- (Z1-2)
rule Zip d e [(a,b)]     <==> True                 -- (Z2-1)
rule Zip d e [(a,b)]      ==> e=b                  -- (Z2-2)
rule Zip d e [(a,b)]      ==> d=a                  -- (Z2-3)
rule Zip d f ([c]->e)    <==> d=a, Zip (a,f) c e  -- (Z3-1)
rule Zip d f ([c]->e)    <==> f=b, Zip (d,b) c e  -- (Z3-2)
```

These rules are equivalent to (Z1), (Z2) and (Z3) and therefore the single-range and multi-range version are equivalent. The above programs violate the Coverage Condition (see the second instance declaration) but satisfy the (Multi-Range) Consistency and Weak Coverage Condition. As we know from earlier sections, breaking the Coverage Condition has the danger that we introduce new variables (see rule (Z3) where `a` and `b` are new variables on the right-hand side) which in turn may lead to non-termination of CHRs. However, the introduction of new variables is harmless here. Effectively, we can replace rule (Z3) by

```
rule Zip a b ([c]->e)    <==> Zip (a,b) c e  -- (Z3')
```

which is equivalent to the instance CHR. In the earlier Example 18 we have made a similar observation. Hence, generated CHRs are always terminating. Hence, the multi-range version of Theorem 2 applies.

6.4 Local Termination, Range-Restriction and Confluence Check

The results of the previous sections are derived from results stated in (**?**). There, the proofs to establish completeness of the satisfiability, subsumption and unambiguity checks, rely only on the property that all derivations for a particular goal are confluent, range-restricted and terminating. Hence, we can give up the "global" CHRs properties of termination, range-restriction and confluence as long as we can verify that these properties hold "locally". Concretely, inference is complete for a specific program program if we can show that for the specific inference goal arising out of this program all derivations are confluent, range-restricted and terminating.

In case we have termination and confluence, there is an easy test to check for local range-restriction, without having to explore all derivations.

Lemma 7 Let D be a constraint for which all derivations are confluent. Let $D \rightarrowtail_P^* C$ for some C such that $fv(C) \subseteq fv(D)$. Then, we have that if $D \rightarrowtail_P^* C'$ for any C' then $fv(C') \subseteq fv(D)$.

Proof
We have that $D \rightarrowtail_P^* C$ for some C such that $fv(C) \subseteq fv(D)$ and $D \rightarrowtail_P^* C'$. Confluence implies that $\models (\bar{\exists}_{fv(D)}.C') \leftrightarrow C$ (1). Assume we find $a \in fv(C')$ such that $a \notin fv(D)$ (2). W.l.o.g., variable a appears in a type class constraint. Condition (1) implies that type class constraints in C' and C must be renamings of each other modulo variables in D (3). However, all variables in C are mentioned in D. Hence, (2) and (3) is a clear contradiction. $\square$

The lemma says that if one derivation is range-restricted then all other possible derivations must be range-restricted as well, provided we have confluence.

7 Related Work

The idea of improving types in the context of Haskell type classes is not new. For example, Chen, Hudak and Odersky (Chen *et al.*, 1992) 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' Consistency and Coverage 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 as discussed in Section 6.

In (Jones, 1995), Jones introduces a general theory of simplifying and improving types as a refinement of his theory of qualified types (Jones, 1992). His description relies on the logical interpretation of type classes whereas we actually show how to solve them (which is the central task of type inference). Hence, he does not answer the question which improvement strategies lead to sound, complete and decidable type inference. Subsequently, Jones extended multi-parameter type classes with functional dependencies (Jones, 2000). He imposes on them the Consistency and

Coverage Conditions. In this paper we finally could verify that in combination with the Basic and Bound Variable Condition, these conditions are sufficient to ensure sound, complete and decidable type inference. Surprisingly, he introduces Example 3 (which breaks the Coverage Condition) as a motivation for functional dependencies.

Duggan and Ophel (Duggan & Ophel, 2002) 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 liberal FDs nor do they consider how to cope with the termination problem.

Stuckey and Sulzmann (?) 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.

In (Chakravarty *et al.*, 2005b), Chakravarty, Keller, Peyton Jones and Marlow introduce a system where each class comes with a set of methods and a set of associated data types (ADTs). Chakravarty, Keller and S. Peyton Jones (Chakravarty *et al.*, 2005a) later extend ADTs to associated type synonyms (ATs). In essence, ATs allow one to establish a mapping between types. Here is a simple example

```
class C a where
 type T a
 op :: a->T a

instance C Int where
  type T Int = Int
  op x = x+1
```

For each instance we need to define the particular type mapping. The advantage of such an approach is that we can avoid "redundant" parameters. Though, we can mimic such style of programming rather straightforwardly via FDs. Here is the FD "equivalent" of the above AT program.

```
class C a where
  op :: T a b => a->b
instance C Int where
 op x = x+1
class T a b | a->b
instance T Int Int
```

The type function `type T a` is simply represented by the the functional dependency `a->b` imposed on class `T a b`. Each occurrence of `T a` is replaced by some fresh variable `b` under the constraint `T a b`. We consider it an important task to study the precise connection between FDs, ADTs and ATs, by translating both to CHRs, so that we can evaluate their relative benefits. We believe that the methods developed in this paper will be highly useful for this task.

8 Conclusions and Future Work

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 Basic, Bound Variable, Consistency and Coverage Conditions (see Definitions 5, 8, 7 and 6) guarantee that resulting CHRs are range-restricted, terminating and confluent (see Theorem 1) and thus we obtain sound, complete and decidable type inference (see Corollary 1). The Basic Conditions could be replaced by alternatives such as the Paterson Conditions (see Corollary 2) as long as we can guarantee that CHRs resulting from instances and super-classes are terminating, confluent and range-restricted. A maybe surprising observation was that Consistency, Coverage and the Bound Variable Conditions are essential for completeness and decidability (see the discussion in Sections 5.1, 5.2). Specifically, the Consistency and Coverage Condition are essential for confluence (complete type inference) whereas the Coverage and Bound Variable Condition are essential for termination (decidable type inference). The Bound Variable Condition on its own is also essential for complete type inference because this condition prevents us from guessing types.

There are many examples which demand dropping the Coverage Condition. Hence, our focus was to find weaker conditions under which we can maintain confluence assuming that we can guarantee termination via some other means. For this purpose, we have introduced more liberal FDs in Section 6.1. We have identified conditions (Consistency, Weak Coverage and fullness of FDs) which guarantee confluence (see Theorem 2). For a certain class of programs we could even show that breaking the Coverage Condition will not break termination (see Corollary 3). We could also show that in some situations non-full FDs can be transformed into full FD (thus we can apply Theorem 2) by generating a slightly stronger set of CHRs (Section 6.2). For more liberal FDs, we could show that a new class of multi-range FDs provides for additional expressiveness (Section 6.3).

In Section 6.4, we have shown that we can drop all three essential CHR conditions (range-restriction, termination and confluence) if we can guarantee that the derivations for a particular goal are range-restricted, terminating and confluent. Effectively, inference becomes semi-decidable but we retain completeness (see Corollary 4).

In turn, we briefly elaborate on some further future research topics. For our proposed extensions of FDs (Section 6) it becomes much harder to guarantee decidability unless the instance improvement rules generated are *trivial*. By trivial we mean that 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 (see Examples 18 and 23). To ensure termination for larger classes of programs we need to devise static and/or dynamic termination checks or alternative termination conditions.

So far, we were mostly concerned with relaxing the Coverage Condition. In another line of future work we plan to investigate how to safely relax 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.

An interesting extension not discussed so far is to allow the programmer to impose stronger improvement rules by providing user-provided CHRs. This is useful because there are situations where FDs do not enforce sufficient improvement. Recall Example 4. Assume we write the following definitions.

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

The inferred types of `e1` and `e2` 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 (Z2) (see Example 23) states that only if we see `Zip d e [(a,b)]` we can improve `e` by `b` and `d` by `a`. However, in case of `e2` 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 rule (Z2) by

```
rule Zip a d [c] ==> c=(a,d)   -- (Z2')
```

which imposes stronger improvement to achieve the desired typing of `e2` and `e3`. Note that rule (Z2') respects the Consistency and Coverage Conditions (assuming we enforce these conditions for user-provided improvement rules). Hence, we retain confluence and termination of CHRs. An extended example making use of user-provided improvement rules plus some of the other variants of functional dependencies can be found in Appendix B.

Acknowledgments

We thank the reviewers and Ross Paterson for their comments.

References

Abdennadher, S. (1997). Operational semantics and confluence of constraint propagation rules. *Pages 252–266 of: Proc. of CP'97*. LNCS, vol. 1330. Springer-Verlag.

Chakravarty, M., Keller, G., & Jones, S. Peyton. (2005a). *Associated types synonyms*. To appear in ICFP'05.

Chakravarty, M., Keller, G., Jones, S. Peyton, & Marlow, S. (2005b). Associated types with class. *Pages 1–13 of: Proc. of POPL'05*. ACM Press.

Chen, K., Hudak, P., & Odersky, M. (1992). Parametric type classes. *Pages 170–191 of: Proc. of ACM Conference on Lisp and Functional Programming*. ACM Press.

Duck, G. J., Peyton-Jones, S., Stuckey, P. J., & Sulzmann, M. (2004). Sound and decidable type inference for functional dependencies. *Pages 49–63 of: Proc. of ESOP'04*. LNCS, vol. 2986. Springer-Verlag.

Duggan, D., & Ophel, J. (2002). Type-checking multi-parameter type classes. *Journal of functional programming*, **12**(2), 133–158.

Frühwirth, T. (1995). Constraint handling rules. *Constraint programming: Basics and trends*. LNCS, vol. 910. Springer-Verlag.

Frühwirth, T. (1998). Theory and practice of constraint handling rules. *Journal of logic programming*, **37**(1–3), 95–138.

GHC. *Glasgow haskell compiler home page*. http://www.haskell.org/ghc/.

Hall, C. V., Hammond, K., Jones, S. Peyton, & Wadler, P. (1996). Type classes in Haskell. *ACM TOPLAS*, **18**(2), 109–138.

HUGS. *Hugs home page*. `http://www.haskell.org/hugs/`.

Jones, M. P. 1992 (Sept.). *Qualified types: Theory and practice*. Ph.D. thesis, Oxford University.

Jones, M. P. 1993 (September). *Coherence for qualified types*. Research Report YALEU/DCS/RR-989. Yale University, Department of Computer Science.

Jones, M. P. (1995). Simplifying and improving qualified types. *FPCA '95: Conference on functional programming languages and computer architecture*. ACM Press.

Jones, M. P. (2000). Type classes with functional dependencies. *Pages 230–244 of: Proc. ESOP'00*. LNCS, vol. 1782. Springer-Verlag.

Jones, S. Peyton, Jones, M. P., & Meijer, E. 1997 (June). Type classes: an exploration of the design space. *Haskell workshop*.

Kiselyov, O., Lämmel, R., & Schupke, K. (2004). Strongly typed heterogeneous collections. *Pages 96–107 of: Proc. of Haskell'04*. ACM Press.

Lassez, J., Maher, M., & Marriott, K. (1987). Unification revisited. *Foundations of deductive databases and logic programming*. Morgan Kauffman.

Shoenfield, J.R. (1967). *Mathematical logic*. Addison-Wesley.

Stuckey, P. J., & Sulzmann, M. (2002). A theory of overloading. *Pages 167–178 of: Proc. of ICFP'02*. ACM Press.

A Proofs

A.1 Proof of Theorem 1

Our assumptions are: Let p be a set of Haskell class and instance declarations which satisfies the Haskell-FD restrictions. Let $Simp(p)$ and $Prop(p)$ be the sets of CHRs defined by Figure 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.

Proof
The class and instance declarations satisfy the Haskell-FD restrictions. Hence, we immediately find that $Simp(p) \cup Prop(p)$ is range-restricted.

For the (sub)proof that $Simp(p) \cup Prop(p)$ are terminating, we need an even stronger propery though. None of the CHRs in $Simp(p) \cup Prop(p)$ will ever introduce any new variables during CHR solving. This holds if $Simp(p) \cup Prop(p)$ is variable-restricted. We call a CHR *variable-restricted* iff all variables on the right-hand side already appear on the left-hand side.

Each simplification c `<==>` $\bar{d}$ rule is range-restricted. Note that $\bar{d}$ are user-defined constraints. Hence, $fv(\bar{d}) \subseteq fv(c)$, hence each simplification rule is variable-restricted.

Similarly, we conclude that each range-restricted class CHR is variable-restricted.

We consider the propagation rules generated by our FD via CHR encoding. Consider the CHR describing the FD.

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

We have that $\{a_{i_0}, b_{i_0}\} \subseteq fv(a_1, \ldots, a_n, \theta(b_1), \ldots, \theta(b_n))$, since $\theta(b_{i_0}) = b_{i_0}$ hence the rule is variable-restricted.

Consider the propagation rules arising from instance declarations, that is of the form

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

First since $\theta(b_{i_0}) = b_{i_0}$ we have that $fv(b_{i_0}) \subseteq fv(\theta(b_1), \ldots, \theta(b_n))$, while the Coverage Condition ensures that $fv(t_{i_0}) \subseteq fv(\theta(b_1), \ldots, \theta(b_n))$, hence the rule is variable-restricted.

1. $Simp(p) \cup Prop(p)$ are terminating: Assume the contrary. Hence, there must be a non-terminating derivation $D \rightarrowtail \ldots$ for some constraint D. Note that new variables are never introduced since the rules are variable-restricted. Hence, there is a limit on the number of equations we can add without adding trivial equations, i.e. equations already logically contained in the store. Hence, there exists D' such that $D \rightarrowtail^* D' \rightarrowtail \ldots$ and the derivation from D' on is not terminating. Let θ be a substitution satisfying all equations in D' and grounding all variables. We say ϕ is a grounding substitution for a type t iff for each $a \in fv(t)$ we have that $fv(\phi(a)) = \emptyset$. This naturally extends to constraints. The grounded version of the (still non-terminating) derivation $\theta(D') \rightarrowtail \ldots$ consists only of simplification rule applications (we can safely skip propagation rule applications because such rules add only trivial information). This is a contradiction to our assumption that $Simp(p)$ is terminating.
2. $Simp(p) \cup Prop(p)$ are confluent: Note that by assumption the set of instance and class CHRs is confluent. Hence, it is sufficient to consider all critical pairs among instances rules and the functional dependency and instance improvement rules.
   First, we consider the critical pair between instance rule `TC` $t_1 \ldots t_n$`<==>`C

and the improvement rule generated from it `TC` $\theta(b_1)\ldots\theta(b_n)$`==>`$t_{i_0} = b_{i_0}$. This leads to derivations

$$\begin{array}{ll} & TC\ t_1 \ldots t_n \\ \rightarrowtail & C \end{array}$$

$$\begin{array}{ll} & TC\ t_1 \ldots t_n \\ \rightarrowtail & TC\ t_1 \ldots t_n, t_{i_0} = t_{i_0} \\ \rightarrowtail & C, t_{i_0} = t_{i_0} \\ \leftrightarrow & C \end{array}$$

Note that the Consistency Condition ensures that any other applicable instance improvement rule will lead to the same result.
Second the critical pair between the instance rule `TC` $t_1 \ldots t_n$`<==>`C and the functional dependency rule

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

is $TC\ t_1 \ldots t_n, TC\ \theta(b_1)\ldots\theta(b_n)$ where $\theta(b_{i_j}) = t_{i_j}, 1 \leq j \leq k_i$ and $\theta(b_j) = b_j$ otherwise. Note that CHRs are variable-restricted. Hence, we do not need to rename rules involved in rule application in the following derivations. Hence,

$$\begin{array}{ll} & TC\ t_1 \ldots t_n, TC\ \theta(b_1)\ldots\theta(b_n) \\ \rightarrowtail & C, TC\ \theta(b_1)\ldots\theta(b_n) \\ \rightarrowtail & C, TC\ \theta(b_1)\ldots\theta(b_n), b_{i_0} = t_{i_0} \end{array}$$

$$\begin{array}{ll} & TC\ t_1 \ldots t_n, TC\ \theta(b_1)\ldots\theta(b_n) \\ \rightarrowtail & TC\ t_1 \ldots t_n, TC\ \theta(b_1)\ldots\theta(b_n), t_{i_0} = b_{i_0} \\ \rightarrowtail & C, TC\ \theta(b_1)\ldots\theta(b_n), t_{i_0} = b_{i_0} \end{array}$$

We have verified that all critical pairs are joinable. CHRs are terminating. Hence, the result from (Abdennadher, 1997) allows us to conclude that $Simp(p) \cup Prop(p)$ are confluent.

□

A.2 Proof of Theorem 2

Our assumptions are: Let p be a set of Haskell class and instance declarations which satisfies the Liberal-FD restrictions and all FDs are full. Let $Simp(p)$ and $Prop(p)$ be defined by Figure 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.

Proof
We immediately find that $Simp(p) \cup Prop(p)$ is range-restricted. As in the previous proof, we prove confluence by verifying that critical pairs are joinable. W.l.o.g., we assume that instances are strictly more liberal (i.e. the Coverage Condition is violated). We consider the functional dependency $a_{i_1}, ..., a_{i_k}$ `->` a_{i_0} and its associated CHRs. Note that FDs are full.

```
rule TC $a_1 \ldots a_n$, TC $\theta(b_1) \ldots \theta(b_n)$ ==> $a_{i_0} = b_{i_0}$ -- (FD)
rule TC $\theta(b_1) \ldots \theta(b_n)$ <==> $t_{i_0} = b_{i_0}, C$ -- (Inst-Imp)
```

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

As before, there are two kind of critical pairs. For the critical pair among instance improvement and instance rule, the Consistency Condition prevents any critical pairs arising except between an instance and its instance improvement rule. Hence, confluence follows immediately.

For the remaining critical pair (among FD and instance) the Weak Coverage Condition becomes important. We make the following observation. Consider `rule TC` $\theta(b_1) \ldots \theta(b_n)$ `<==>` $t_{i_0} = b_{i_0}, C$. Let $\bar{a} = \mathit{fv}(C) \backslash \mathit{fv}(t_1, \ldots, t_n)$ and π be a renaming substitution with domain $\bar{a}$. Let $\mathit{PropFDs}(C)$ be the set of functional dependency rules of all type classes in C. Then, $C, \pi(C) \rightarrowtail^n_{\mathit{PropFDs}(C)} C, a_1 = \pi(a_1), \ldots, a_l = \pi(a_l)$ after some n number of derivation steps.

Let $TC\ t_1 \ldots t_n, TC\ \theta(b_1) \ldots \theta(b_n)$ be the critical pair between the instance rule `TC` $t_1 \ldots t_n$`<==>`C and the functional dependency rule

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

where $\theta(b_{i_j}) = t_{i_j}, 1 \leq j \leq k_i$ and $\theta(b_j) = b_j$ otherwise.

This leads to derivations

$$\begin{array}{ll}
 & TC\ t_1 \ldots t_n, TC\ \theta(b_1) \ldots \theta(b_n) \\
\rightarrowtail_{Inst-Imp} & \pi_1(C), TC\ \theta(b_1) \ldots \theta(b_n) \\
 & \pi_1 \text{ renames local variables involved in rule application} \\
\rightarrowtail_{Inst-Imp} & \pi_1(C), \pi_2(C), \pi_2(t_{i_0}) = b_{i_0} \\
 & \text{FD is full, hence we combine (I-FD), (Inst)} \\
 & \pi_2 \text{ renames local variables involved in rule application} \\
\rightarrowtail^*_{PropFDs(C)} & C, \pi_2(t_{i_0}) = b_{i_0}, \pi_1(a_1) = \pi_2(a_1), \ldots, \pi_1(a_l) = \pi_2(a_l) \\
 & \text{follows from the above observation}
\end{array}$$

and

$$\begin{array}{ll}
 & TC\ t_1 \ldots t_n, TC\ \theta(b_1) \ldots \theta(b_n) \\
\rightarrowtail_{FD} & TC\ t_1 \ldots t_n, t_{i_0} = b_{i_0} \\
\rightarrowtail_{Inst} & \pi'_1(C), t_{i_0} = b_{i_0} \\
 & \pi'_1 \text{ renames local variables involved in rule application}
\end{array}$$

We conclude that $\models \bar{\exists}_V(C \wedge \pi_2(t_{i_0}) = b_{i_0} \wedge \pi_1(a_1) = \pi_2(a_1) \wedge \ldots \wedge \pi_1(a_l) = \pi_2(a_l)) \leftrightarrow \bar{\exists}_V(\pi'_1(C) \wedge t_{i_0} = b_{i_0})$ where $V = \mathit{fv}(t_1, \ldots, t_n, \theta(t_1), \ldots, \theta(t_n))$ which shows that critical pairs are joinable. □

B Stronger Improvement Example

We consider an encoding of extensible records. Such an encoding has been independently considered in (Kiselyov *et al.*, 2004). In our encoding, we drop the Bound Variable Condition see upcoming instance (S3). Though, we can guarantee that unbound variables are functionally defined by bound variables. Hence, our results from Section 6.4 apply. We also drop the Coverage Condition. Instead we can guarantee the Refined Weak Coverage Condition from Section 6.1. It turns out that the improvement conditions resulting from liberal FDs are too weak. Therefore, we impose additional CHRs to achieve the desired improvement.

```
-- Encoding records with type classes
-- we use singleton list to model records
data Nil = Nil
data Cons x xs = Cons x xs

-- we use singleton numbers to model record labels
data Zero   = Zero
data Succ n = Succ n

-- result of type level computations
data T = T
data F = F

class EqR a b c | a b -> c where eq :: a->b->c

instance EqR Zero Zero T
instance EqR a b c => EqR (Succ a) (Succ b) c
-- we rely on trivial termination, hence CHRs are terminating
-- CHRs are also confluent
instance EqR Zero (Succ a) F
instance EqR (Succ a) Zero F

class Select r l a | r l -> a where select :: r->l->a
class Select' r l t a | r l t -> a where select' :: r->l->t->a

instance (Select' (Cons (x1,v) e) x2 b v', EqR x1 x2 b)
          => Select (Cons (x1,v) e) x2 v'                     -- (S3)
-- the above instance satisfies the
-- Refined Weak Coverage Condition
-- note that we also break the Bound Variable Condition,
-- though b is 'functionally' defined by x1 and x2
instance Select' (Cons (x,v) e) x T v
instance Select e x2 v' =>  Select' (Cons (x1,v) e) x2 F v'   -- (S'3)
-- the improvement rules linked to the two recursive instances
-- are trivial, hence CHRs are terminating (Trivial Termination),
-- and also confluent

-- things we couldn't do with FDs
-- stronger improvement rules
rule Select Nil l a ==> False
rule Select' Nil l t a ==> False
rule Select' (Cons (x1,v1) e) x2 T v2 ==> x1=x2, v1=v2
```

```
-- we cannot enforce the following rule,
-- otherwise non-confluent with (S'3)
--rule Select' (Cons (x,v1) e) x t v2 ==> v1=v2, t=T

-- sample code
l0 = Zero
l1 = Succ Zero

f r = select r l0

r1 = Cons (l0, True) (Cons (l1,'a') Nil)
r2 = Cons (l1, False) (Cons (l0, 'b') Nil)

e1 :: Bool
e1 = f r1
e2 :: Char
e2 = f r2
```