On Termination, Confluence and Consistent CHR-based Type Inference

GREGORY J. DUCK
Department of Computer Science, National University of Singapore
(*e-mail:* `gregory@comp.nus.edu.sg`)

RÉMY HAEMMERLÉ*
Universidad Politécnica de Madrid & IMDEA Software Institute
(*e-mail:* `Remy.Haemmerle@imdea.org`)

MARTIN SULZMANN
Hochschule Karlsruhe - Technik und Wirtschaft
(*e-mail:* `Martin.Sulzmann@hs-karlsruhe.de`)

Abstract

We consider the application of Constraint Handling Rules (CHR) for the specification of type inference systems, such as that used by Haskell. Confluence of CHR guarantees that the answer provided by type inference is correct and consistent. The standard method for establishing confluence relies on an assumption that the CHR program is terminating. However, many examples in practice give rise to non-terminating CHR programs, rendering this method inapplicable. Despite no guarantee of termination or confluence, the Glasgow Haskell Compiler (GHC) supports options that allow the user to proceed with type inference anyway, e.g. via the use of the `UndecidableInstances` flag. In this paper we formally identify and verify a set of relaxed criteria, namely *range-restrictedness*, *local confluence*, and *ground termination*, that ensure the consistency of CHR-based type inference that maps to potentially non-terminating CHR programs.

KEYWORDS: Constraint Handling Rules, confluence, termination, type classes

1 Introduction

Constraint Handling Rules (CHR) (Frühwirth 2009) are a powerful rule-based programming language for specification and implementation of constraint solvers. CHR has many application domains, including constraint solving (Sneyers et al. 2010), type inference systems (Sulzmann et al. 2007), coinductive reasoning (Haemmerlé 2011), theorem proving (Duck 2012) and program verification (Duck et al. 2013). This paper concerns the application of CHR to *type inference systems* for high-level declarative programming languages such as Haskell (Jones 2003) and Mercury (Somogyi et al. 1996). In particular,

* The research by Rémy Haemmerlé has received funding from the Programme for Attracting Talent/young PHDs of the Montegancedo Campus of International Excellence *PICD*, the European Union 7th Framework Programme under grant agreement 318337 *ENTRA*, Spanish MINECO TIN'12-39391 *StrongSoft* and TIN'08-05624 *DOVES* projects, and Madrid TIC-1465 *PROMETIDOS-CM* project.

type constraints imposed by *type classes* (Wadler and Blott 1989) can be straightforwardly mapped into a set of CHR rules. Type inference with type classes is then reduced to CHR solving.

For example, consider the following Haskell type class declarations

```
class Eq a where (==) :: a->a->Bool     instance Eq a => Eq [a] where ...
```

The class declaration introduces some (overloaded) equality operator `==` whose type is constrained by the type class (`Eq a`). The instance declaration states that we obtain equality among lists assuming we supply equality on the element type. Here the notation $\texttt{[}a\texttt{]}$ represents a list type with element type a. Following (Sulzmann et al. 2007), the above maps to the following CHR simplification rule

$$\texttt{Eq [a]} \iff \texttt{Eq a}$$

Type inference via CHR solving is performed by (1) generating the appropriate constraints out of the program text, (2) solving these constraints w.r.t. the set of CHR rules derived from class and instance declarations. For example, consider the function `f` that tests if a list `xs` is equal to a singleton list containing `y`.

```
f xs y = xs == [y]
```

To infer the type of `f` we (roughly) generate $\texttt{Eq}\ t_{xs}, t_{xs} = \texttt{[}t_y\texttt{]}, t_f = (t_{xs} \to t_y \to \texttt{Bool})$ which reduces, via application of the above CHR rule and substitution, to $\texttt{Eq}\ t_y, t_{xs} = \texttt{[}t_y\texttt{]}, t_f = (\texttt{[}t_y\texttt{]} \to t_y \to \mathit{Bool})$. Hence, function `f` has type $\forall a.\texttt{Eq}\ a \Rightarrow \texttt{[}a\texttt{]} \to a \to \mathit{Bool}$.

This approach extends to richer sets of type class programs such as multi-parameter type classes and functional dependencies (Jones 2000). The advantage is that important type inference properties such as decidability and consistency can be verified by establishing the respective properties for the resulting CHR rules.

The answer of type inference is guaranteed to be *consistent* if the set of CHR rules is confluent. A terminating set of CHR rules is *confluent* if it reduces any given goal to the same answer regardless of rule application ordering. Earlier work (Stuckey and Sulzmann 2005; Sulzmann et al. 2007) identifies sufficient conditions on type class programs to guarantee that the resulting CHR rules are confluent. A critical assumption is that CHR rules are terminating. In doing so, the proof of confluence can be reduced to establishing a weaker condition, namely *local confluence*, via the application of Newman's Lemma (Newman 1942).

The problem is that the kinds of CHR programs that arise in practice often violate the termination assumption. For example, consider the following set of multi-parameter class and instance declarations that incorporates a *functional dependency* $\texttt{a} \to \texttt{b}$

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

The functional dependency roughly states that: given a type-class constraint $\texttt{F}\ a\ b$, then the type b is a functionally determined by a. These class and instance declarations can be mapped to the following CHR rules

$$\texttt{F Int}\ b \iff b = \texttt{Bool} \qquad \texttt{F [}a\texttt{]}\ b \iff b = \texttt{[}c\texttt{]},\ \texttt{F}\ a\ c$$
$$\texttt{F}\ a\ b,\ \texttt{F}\ a\ c \Longrightarrow b = c$$

The first two rules capture the two instances and also enforce the functional dependency for the respective instance.

At first glance, the above CHR program may appear to be terminating. Indeed, any *ground* constraint ($\texttt{F}\ \texttt{t}_1\ \texttt{t}_2$) will be reduced in a finite number of steps. However, consider the *non-ground* goal ($\texttt{F [}a\texttt{]}\ a$), where a is some variable, for which we find the following non-terminating derivation

$$\texttt{F [}a\texttt{]}\ a \ \rightarrowtail\ (\texttt{F [}b\texttt{]}\ b,\ a = \texttt{[}b\texttt{]})\ \rightarrowtail\ (\texttt{F [}c\texttt{]}\ c,\ a = \texttt{[}b\texttt{]},\ b = \texttt{[}c\texttt{]})\ \rightarrowtail\ \ldots$$

The above example represents a typical (albeit much simplified) example that is found when applying type classes for expressive type reasoning (Hallgren 2001). For example, consider the following more realistic example, which encodes addition at the level of types

```
data Zero              data Succ n              class Add a b c | a b -> c
instance Add Zero b b        instance Add a b c => Add (Succ a) b (Succ c)
```

Here the non-ground goal $\texttt{Add (Succ}\ a\texttt{)}\ b\ a$ also exhibits the same non-terminating behavior.

Fortunately, realistic programs will usually *not* yield devious non-terminating constraints such as ($\texttt{F [}a\texttt{]}\ a$). Hence, practical implementations of type inference systems, such as the Glasgow Haskell Compiler (GHC 2014), typically enable the user[1] to proceed with type inference even though the corresponding CHR program is potentially non-terminating. If the flag is enabled, the type inference engine must compute the answer within a fixed number of reduction steps, otherwise an error is reported.

This paper is concerned with the correctness of the above "practical implementations". Current CHR theory concerning confluence and consistency of CHR programs explicitly assumes the program is *terminating for all goals*, which is simply not applicable under our setting. Our main contributions are:

- We establish that range-restricted, ground confluent CHR programs are *consistent* (Section 3). This extends the classical CHR consistency result from (Abdennadher et al. 1999).
- We establish that terminating goals are confluent for range-restricted, ground-terminating and locally confluent programs (Section 4).
- We discuss how these results apply to the GHC/type class setting (Section 5). In particular, we show that if type inference finitely terminates with an answer, then that answer is *unique*.

Section 2 reviews background material on CHR. Section 6 summarizes related work and concludes.

2 Constraint Handling Rules

Throughout this paper we use *Haskell type notation* to represent terms, constraints and predicates. Under this scheme:

- functors (a.k.a. atoms) begin with an upper case letter (the opposite to Prolog);
- variables begin with a lower-case letter (the opposite to Prolog);
- term arguments are separated by whitespace; and
- the special functor `[a]` is shorthand for (`List a`) and `a=b` for equality.

[1] Via GHC's `UndecidableInstances` flag.

For example, the term `p(X, q(Y), list(Z))` under Prolog syntax would be represented as (`P x (Q y) [z]`) under Haskell type syntax.

Constraint Handling Rules (CHR) is a rule-based constraint rewriting programming language designed for implementing constraint solvers.

We assume, as given, the following disjoint infinite sets of variables: $\mathsf{ProgVars}$, $\mathsf{GlobalVars}$ and $\mathsf{LocalVars}$. We define:

$$
\begin{array}{rl}
\text{Terms} & t ::= v \mid F\ t\ \ldots\ t \\
\text{Built-in Constraints} & b ::= \texttt{True} \mid \texttt{False} \mid t = t \\
\text{User Constraints} & u ::= C\ t\ \ldots\ t \\
\text{Constraints} & c ::= b \mid u
\end{array}
$$

where v is a variable from some variable set. A *substitution* is a mapping from variables to terms. We use the notation $\theta.X$ to represent a substitution θ applied to a term or constraint X. We respectively define $\mathsf{Cons}(V)$ and $\mathsf{Usr}(V)$ as the set of all constraints and the set of all user-constraints over the set of variables V. There are two main types of CHR rules:

$$H \iff B \quad \text{(Simplification)} \qquad \text{and} \qquad H \implies B \quad \text{(Propagation)}$$

where $H \in \mathcal{M}(\mathsf{Usr}(\mathsf{ProgVars}))$ and $B \in \mathcal{M}(\mathsf{Cons}(\mathsf{ProgVars}))^2$. The local variables of a rule are those variables that appear in the body but not in the head. Logically simplification rules (resp. propagation rules) are understood as equivalence (implication) between the head and the body where local variables are implicitly existentially quantified.

Let $\mathsf{StateVars} = \mathsf{GlobalVars} \cup \mathsf{LocalVars}$ and let $S_1, S_2 \in \mathcal{M}(\mathsf{Cons}(\mathsf{StateVars}))$ then we define $S_1 \equiv S_2$ as the least relation satisfying

$$\mathcal{CT} \models (\exists_{\mathsf{LocalVars}} : usr(S_1) = usr(S_2) \wedge S_1) \leftrightarrow (\exists_{\mathsf{LocalVars}} : usr(S_1) = usr(S_2) \wedge S_2)$$

where $\mathcal{CT}$ is the theory of term equality and $usr(S)$ the user constraints of the state S. The set of all *CHR states* Σ is defined as the quotient set $\Sigma = (\mathcal{M}(\mathsf{Cons}(\mathsf{StateVars}))/\equiv)$. Note that we usually represent a state $[S] \in \Sigma$ by an S, and we often drop the braces $\{\ldots\}$ around sets of constraints, i.e. we write $\texttt{P}\ a, a = \texttt{[}b\texttt{]}$ instead of $\{\texttt{P}\ a, a = \texttt{[}b\texttt{]}\}$. We will say that a rule is *purely built-in* if its body does not contain any user constraints.

Operationally, CHR is the *abstract rewriting system* $\langle \Sigma, \rightarrowtail \rangle$, where binary relation $(\rightarrowtail) \in \Sigma \times \Sigma$ is the CHR *derivation step* defined as the least relation satisfying:

$$\frac{(H \iff B) \qquad \mathcal{CT} \models S \rightarrow (\theta.H = C) \qquad \mathcal{CT} \models \exists : C \wedge S}{C \uplus S \rightarrowtail \theta.B \uplus S}$$

$$\frac{(H \implies B) \qquad \mathcal{CT} \models S \rightarrow (\theta.H = C) \qquad \mathcal{CT} \models \exists : C \wedge S \qquad C \uplus S \neq \theta.B \uplus C \uplus S}{C \uplus S \rightarrowtail \theta.B \uplus C \uplus S}$$

where $(\uplus)$ is multi-set union, $\theta : \mathsf{ProgVars} \rightarrow \mathsf{StateVars}$ is a substitution mapping $vars(H)$ to $vars(C)$ and $vars(B) - vars(H)$ to fresh variables from $\mathsf{LocalVars} - vars(C, S)$.

Note that there are many different definitions for the operational semantics of CHR. Our version does not keep propagation histories, and as such will only terminate for propagation rules with built-in only bodies. This is sufficient for our purposes.

[2] $\mathcal{M}(X)$ is the set of all multisets built form the set X.

Let $(\rightarrowtail^=)$ the reflexive closure of $(\rightarrowtail)$, and let $(\rightarrowtail^*)$ be the transitive closure of $(\rightarrowtail^=)$. A pair of states $S_1, S_2 \in \Sigma$ is *join-able* if there exists an $S' \in \Sigma$ and derivations $S_1 \rightarrowtail^* S'$, $S_2 \rightarrowtail^* S'$. An abstract rewriting system $\langle \Sigma, \rightarrowtail \rangle$ is:

- *terminating* if there is no infinite derivation $(S \rightarrowtail \ldots)$ for all $S \in \Sigma$.
- *locally confluent* if for all $S, S_1, S_2 \in \Sigma : S_1 \leftarrowtail S \rightarrowtail S_2$ then S_1 and S_2 are join-able.
- *confluent* if for all $S, S_1, S_2 \in \Sigma : S_1 \leftarrowtail^* S \rightarrowtail^* S_2$ then S_1 and S_2 are join-able.

If a program P is both locally confluent and terminating, then P is confluent (Newman 1942). Confluence implies *logical consistency* of P (Abdennadher et al. 1999; Haemmerlé et al. 2011). That is, the logical reading of P does not imply false.

Let $\mathcal{I}$ be any property over states such that: for all $S, S' \in \Sigma$ where $S \rightarrowtail S'$, if $\mathcal{I}(S)$ holds then $\mathcal{I}(S')$ also holds. Then $\mathcal{I}$ is an *observable invariant* (Duck et al. 2007). If we define $\Sigma_{\mathcal{I}} = \{S \mid S \in \Sigma \wedge \mathcal{I}(S)\}$ then program P is respectively $\mathcal{I}$*-terminating*, $\mathcal{I}$*-locally-confluent*, and $\mathcal{I}$*-confluent* if the abstract rewriting system $\langle \Sigma_{\mathcal{I}}, \rightarrowtail \rangle$ is terminating, locally-confluent and confluent.

We define the set of all *ground states* Σ_g as the canonical surjection of $\mathcal{M}(\mathsf{Cons}(\emptyset))$ onto Σ. A CHR program P is *range restricted* iff groundness is an observable invariant.[3]

Before continuing we state the monotonicity of CHR transitions as a number of proofs of the present paper rely on it. This property means that if a transition step is possible in a state, then it is possible in any state that contains additional constraints.

Proposition 1 (Monotonicity) *Let S, T, and U be three states such that $vars(S,T) \cap vars(U) \subset \mathsf{GlobalVars}$. If $S \rightarrowtail T$ holds, then so does $(S \uplus U) \rightarrowtail^= (T \uplus U)$.*

3 Consistency of Ground Confluent CHR

In our first result we show that ground-confluence with range-restrictedness guarantees the logical consistency of programs.

Theorem 1 (Consistency) *If P is range-restricted and ground confluent program, then it is consistent.*

Proof
Define $\mathcal{H} = \{c \mid (\{c\} \uplus S) \in \Sigma_g \text{ and } S \rightarrowtail^* \texttt{True}\}$. To establish consistency of P, it is sufficient to show $\mathcal{H}$ is an Herbrand model for both the constraint theory and the logical reading of the program:

For the constraint theory, clearly $\texttt{True} \in \mathcal{H}$ whilst $\texttt{False} \notin \mathcal{H}$. Now consider an equality constraint $t = s$ between two ground terms. If t is syntactically equal to s, then the derivation $(t = s) \rightarrowtail^* \texttt{True}$ trivially holds, i.e $\mathcal{H} \models t = s$. Otherwise if t syntactically differs from s, for any $S \in \Sigma$ we have that $(\{t = s\} \uplus S) = \texttt{False} \not\rightarrowtail^* \texttt{True}$, i.e. $\mathcal{H} \not\models t = s$.

For the logic reading of a simplification rule $(H \iff B)$, we are required to show that $\mathcal{H} \models \forall (H \leftrightarrow \exists_{vars(H)} B)$, or equivalently $\theta.H \subseteq I$ iff there exists ρ that coincides with θ on $vars(H)$ such that $\rho.B \subseteq \mathcal{H}$. If $\theta.H \subseteq \mathcal{H}$ then for any $h \in H$ there exists $(\{\theta.h\} \uplus S_h) \rightarrowtail^* \texttt{True}$ for some $S_h \in \Sigma_g$. Therefore by monotonicity for $S = \biguplus_{h \in H} \{S_h\}$,

[3] Note that our definition of *range-restricted*-ness is more general than the standard definition, i.e. that $vars(B) \subseteq vars(H)$ for all rules $(H \iff B)$ or $(H \implies B)$.

$(\theta.H \uplus S) \rightarrowtail^* \texttt{True}$. Let $S' \in \Sigma_g$ be the state obtained by applying the simplification rule on $(\theta.H \uplus S)$. By definition of $\rightarrowtail$, $\rho.B \subseteq S'$ for some ρ that coincides with θ on $\mathit{vars}(H)$. Then $S' \rightarrowtail^* \texttt{True}$ by ground-confluence, and therefore $\rho.B \subseteq \mathcal{H}$, i.e. $\mathcal{H} \models \theta.B$. Conversely, if $\mathcal{H} \models \rho.B$ then for any $b \in B$ there exists some $S_b \in \Sigma_g$ such that $(\{\rho.b\} \uplus S_b) \rightarrowtail^* \texttt{True}$. Define $S = \biguplus_{b \in B}\{S_b\}$, then by monotonicity $(\rho.H \uplus S) \rightarrowtail^* \texttt{True}$, therefore $\rho.H \subseteq S \subseteq \mathcal{H}$, i.e. $\mathcal{H} \models \theta.H$.

For the logical reading of a propagation rule $(H \Longrightarrow B)$ one may consider the simplification $(H \Longleftrightarrow H, B)$ and apply previous case. $\square$

Our consistency result is similar to the original result from (Abdennadher et al. 1999), except that it (1) uses a more general definition of range restrictedness, and (2) assumes ground confluence versus confluence. Also note that ground confluence follows from ground termination and local confluence, using Newman's Lemma (Newman 1942).

4 Confluence for Terminating Goals

A non-confluent, non-terminating CHR program P may still be confluent and terminating for *specific goals*. For example, the CHR program from the introduction has both non-terminating and terminating goals:

$\texttt{F [}a\texttt{]}\ a \rightarrowtail \texttt{F [}b\texttt{]}\ b, a = \texttt{[}b\texttt{]} \rightarrowtail \ldots$ (non-termination) $\quad \texttt{F [}a\texttt{] [}a\texttt{]} \rightarrowtail \texttt{F}\ a\ a$ (termination)

Since type inference is the same as CHR solving, the first goal is clearly problematic. On the other hand, the second goal always terminates, and thus is acceptable.

In the following, we identify sufficient conditions which guarantee that if for goal S we find *some* derivation $S \rightarrowtail^* S'$ where S' is a non-`False` final state then (a) *all* derivations starting from S will terminate and (b) these derivations lead to the same state S'. Part (b) follows rather easily once we have (a). Hence, we first consider part (a).

4.1 Universal Termination follows from Existential Termination

We distinguish between different types of termination. *Universal termination* means that all derivations from a state S will terminate, i.e. there does not exist an infinite derivation $(S \rightarrowtail^* \ldots)$. In contrast, *existential termination* means that there exists a terminating derivation from S, i.e. there exists at least one derivation of the form $S \rightarrowtail^* T \not\rightarrowtail$. For example, the goal $(\texttt{F [}a\texttt{] [}a\texttt{]})$ is both existentially and universally terminating.

Our main result is as follows: Given a range-restricted, ground-terminating and locally-confluent program P, then a given state S is universally terminating if it is existentially terminating to a non-`False` final state.

Theorem 2 (Universal Termination) *Let P be a range-restricted, ground-terminating, and locally-confluent program. If S is existentially terminating to a non-*`False` *final state, i.e. $S \rightarrowtail^* T \neq$* `False` *and $T \not\rightarrowtail$, then S is universally terminating.*

The proof of the Theorem 2 relies on the following lemmas.

Lemma 1 *If $S \rightarrowtail^* T$, $T \neq$* `False` *and $T \not\rightarrowtail$, then there exists a ground substitution θ such that $\theta.S \rightarrowtail^* \theta.T$, $\theta.T \neq$* `False` *and $\theta.T \not\rightarrowtail$.*

Proof
Let ψ be the m.g.u. of the equations in T. Let $\rho = \{x \mapsto c_x \mid x \in \mathsf{Vars}\}$ be a ground substitution mapping variables to fresh constants c_x. Then define $\theta = \{x \mapsto \rho.\psi(x) \mid x \in \mathsf{Vars}\}$ and we see that:

- $\theta.S \rightarrowtail^* \theta.T$ by monotonicity;
- $\theta.T \neq \texttt{False}$ since θ is a unifier of the equations in T; and
- $\theta.T \not\rightarrowtail$ otherwise $T \rightarrowtail$ since c_x were fresh constants. $\square$

Lemma 2 *A set P of purely built-in propagation rules is terminating.*

Proof
Let S be a state. Now consider all pairs $((H \Longrightarrow B), \psi)$ such that $(H \Longrightarrow B)$ is a rule of P and ψ is the m.g.u. between H and some subset S' of S (i.e. $S' \subseteq S$ and $\psi.H = \psi.S'$). There exists at most finitely many such pairs which we may enumerate thusly:

$$((H_1 \Longrightarrow B_1), \psi_1), \ldots, ((H_n \Longrightarrow B_n), \psi_n)$$

where the local variables of the $(H_i \Longrightarrow B_i)$ have been previously renamed apart. Now let define the ranking of S as $\mathit{rank}(S) = n - |\{i \mid \mathcal{CT} \models S \rightarrow \psi_i.B_i\}|$ where $|X|$ is the cardinality of the set X. One verifies that if $S \rightarrowtail T$ then $\mathit{rank}(S) > \mathit{rank}(T)$. It follows that $\rightarrowtail$ is terminating. $\square$

Lemma 3 *Let P be a range-restricted and ground-terminating program. Suppose there exists an infinite derivation $(S \rightarrowtail^* \ldots)$ with $vars(S) \subseteq \mathsf{GlobalVars}$. Then for all ground substitution θ with domain $\mathsf{GlobalVars}$, we have that $\theta.S \rightarrowtail^* \texttt{False}$.*

Proof
Assume there exists an infinite derivation of the form:

$$S \rightarrowtail S_1 \rightarrowtail \cdots \rightarrowtail S_n \rightarrowtail \cdots$$

Let ρ be an arbitrary ground substitution with domain $\mathsf{GlobalVars}$. By monotonicity:

$$\rho.S \rightarrowtail^= \rho.S_1 \rightarrowtail^= \cdots \rightarrowtail^= \rho.S_n \rightarrowtail^= \cdots$$

Since P is ground terminating, we get that there exists some $i \in \mathbb{N}$ such that for any $j \geq i$, $\rho.S_i = \rho.S_j \not\rightarrowtail$. By Lemma 2, there is a $k \geq i$ such that the transition step $S_k \rightarrowtail S_{k+1}$ is induced by a simplification rule or a propagation rule with user-defined constraints in the body. In such a case, we observe if $S_k \rightarrowtail S_{k+1}$ and $\rho.S_k \not\rightarrowtail \rho.S_{k+1}$ then $\rho.S_k = \texttt{False}$. $\square$

Proof of Theorem 2
By contradiction:

1. Assume there exists an infinite derivation $S \rightarrowtail^* \ldots$
2. Since $S \rightarrowtail^* T$, there exists a ground substitution θ satisfying Lemma 1.
3. Then $\texttt{False} \leftarrowtail^* \theta.S \rightarrowtail^* \theta.T$ by Lemmas 3 and 1.
4. Since $\theta.S$ is ground and $\theta.T \neq \texttt{False}$, then P is not ground-confluent.
5. Since P is locally-confluent, P is ground-locally-confluent.
6. Since P is ground-locally-confluent and ground-terminating, P is ground-confluent.
7. Contradiction between 4 and 6. $\square$

Note that Theorem 2 cannot be extended to the case where $S \rightarrowtail^* \texttt{False}$. For example, consider the CHR program

$$\texttt{P}\ x \iff \texttt{False} \qquad\qquad \texttt{P}\ x \iff x = \texttt{[}y\texttt{]},\ \texttt{P}\ y$$

This program is range restricted, ground-terminating, and locally confluent since the only critical pair ($\texttt{False} \leftarrowtail \texttt{P}\ x \rightarrowtail x = \texttt{[}y\texttt{]},\ \texttt{P}\ x$) is join-able. Although ($\texttt{P}\ x$) is existentially terminating, i.e. ($\texttt{P}\ x \rightarrowtail \texttt{False}$), it is not universally terminating because of the infinite derivation ($\texttt{P}\ x \rightarrowtail x = \texttt{[}y\texttt{]}, \texttt{P}\ y \rightarrowtail x = \texttt{[}y\texttt{]}, y = \texttt{[}z\texttt{]}, \texttt{P}\ z \rightarrowtail \ldots$). If we change the first rule $\texttt{P}\ x \iff \texttt{True}$ (or any other non-$\texttt{False}$ body), the program becomes non-locally-confluent.

4.2 Observable Confluence w.r.t. Existential Termination

What remains is to establish confluence for terminating goals. For notational convenience, we define $\mathsf{T}_\forall(S)$ and $\mathsf{T}_\exists(S)$ to respectively hold if state S is universally or existentially terminating. Clearly $\mathsf{T}_\forall$ is an observable invariant.

Lemma 4 *If P is locally-confluent, then P is $\mathsf{T}_\forall$-confluent.*

Proof
Define $\Sigma_\forall = \{S \mid S \in \Sigma \wedge \mathsf{T}_\forall(S)\}$, then the abstract rewriting system $\langle \Sigma_\forall, \rightarrowtail\rangle$ is locally-confluent and terminating (by construction), and is therefore confluent by a straightforward application of Newman's Lemma (Newman 1942). □

Alternatively, one can use the method from (Duck et al. 2007) to prove $\mathsf{T}_\forall$-confluence. However, this is overkill, as P is already assumed to be locally confluent.

The condition $\mathsf{T}_\exists$ by itself is *not* an observable invariant, since an existentially terminating state can be rewritten into a universally non-terminating state. However, if we define $\mathsf{T}'_\exists(S)$ to mean existential termination to a non-false state, i.e. $\mathsf{T}'_\exists(S)$ holds iff there exists a derivation $S \rightarrowtail^* T \neq \texttt{False}$ and $T \not\rightarrowtail$, then we can state the following:

Corollary 1 *Let P be a range-restricted, ground-terminating and locally-confluent program, then P is (1) $\mathsf{T}'_\exists$ is an observable invariant, and (2) $\mathsf{T}'_\exists$-confluent.*

Proof
By Theorem 2, $\mathsf{T}'_\exists = \mathsf{T}_\forall$, and therefore (1) holds. By Lemma 4, P is $\mathsf{T}_\forall$-confluent, and therefore (2) holds. □

To elaborate further: given a state S, suppose we execute S and discover a finite derivation $S \rightarrowtail^* T$, then T is the only possible answer for S.

Corollary 2 (Uniqueness of Answers) *Let P be a range-restricted, ground-terminating and locally-confluent program, if $S \rightarrowtail^* T \not\rightarrowtail$, then for all $S \rightarrowtail^* U \not\rightarrowtail$ we have that $T = U$.*

Proof
If $\mathsf{T}'_{\exists}(S)$ then $T = U$ follows from Corollary 1. If $\neg\mathsf{T}'_{\exists}(S)$ then $T = U =$ `False` since $\mathsf{T}_{\exists}(S)$ holds by assumption. $\Box$

Note that uniqueness of answers is not equivalent to confluence for non-terminating programs. For example, if $(\ldots \leftarrowtail^* T \leftarrowtail^* S \rightarrowtail^* U \rightarrowtail^* \ldots)$ and if T, U are non-joinable and universally non-terminating, then P is not confluent. But P may still produce unique answers for terminating goals.

5 Practical Implications for Type Classes

Type inference with type class constraints is an important application of CHR. Previously, strong conditions must be imposed in order to guarantee the consistency, confluence and termination of type inference. As will be explained in this section, the results from Sections 3 and 4 allow for the relaxation of some of these conditions, which in turn, allows for more programs to be safely accepted.

First, we summarize the standard translation scheme from type classes to CHR, as well as the strong conditions required for termination and confluence. The remainder of this section discusses the relaxed conditions based on our earlier results.

5.1 From Type Classes to CHR

The basic syntax for a `class`-declaration is:

$$\texttt{class}\ D\ \texttt{=>}\ C\ a_1\ \ldots\ a_n\ \texttt{|}\ \mathit{fd}_1, \ldots, \mathit{fd}_n \qquad (\textsc{Class})$$

The declaration defines a new type-class $(C\ a_1\ \ldots\ a_n)$ where a_i is a (type-variable) argument type. Here, D is a set of (super) type-class constraints for which C depends, and fd_i is a *functional dependency* of the form $a_{i_1}, \ldots a_{i_k}\texttt{->}a_{i_0}$ where $\{i_0, .., i_k\} \subseteq 1..n$. Both D and the *fd* set may be empty and omitted. The basic syntax for `instance`-declarations[4] is:

$$\texttt{instance}\ D\ \texttt{=>}\ C\ t_1\ \ldots\ t_n \qquad (\textsc{Instance})$$

Here D is a set of type-class constraints for which the instance depends, and t_i are bound types.

For example, the following class declaration defines a (`Coll` c e) type-class constraint representing an abstract *collection-type* c with *element-type* e:

```
class Coll c e | c -> e      instance Eq a => Coll [a] a
```

Here the class declaration states that the element type e is *functionally dependent* on the collection type c, for more formally: for all a, b, c, if (`Coll` a b) and (`Coll` a c) then $b = c$. The instance declaration states that (`Coll` $[a]$ a) holds for any type satisfying (`Eq` a).

Both class and instance declarations can be understood as syntactic sugar for collections of CHR rules (Sulzmann et al. 2007). The basic translation schema is as follows:

[4] Both class and instance declarations also provide function interfaces and implementations respectively. However, these are not relevant to type inference, so we shall ignore them here.

For `class`-declarations of the form (Class) we generate the following rules:

$$C\ a_1\ \dots\ a_n \implies D \qquad \text{(Class-Rule)}$$
$$C\ a_1\ \dots\ a_n, C\ b_1\ \dots\ b_n \implies a_{i_0} = b_{i_0} \qquad \text{(FD-Rule)}$$

One (FD-Rule) is generated for each fd_i (of the form $a_{i_1}, \dots a_{i_k}$`->`a_{i_0}). Here b_j is a_j if $j \in \{i_1, \dots i_k\}$, otherwise b_j is a fresh variable. Instance-declarations of the form (Instance) generate the following rules:

$$C\ t_1\ \dots\ t_n \iff D \qquad \text{(Instance-Rule)}$$
$$C\ b_1\ \dots\ b_n \implies b_{i_0} = t_{i_0} \qquad \text{(Improvement-Rule)}$$

One (Improvement-Rule) is generated per fd_i provided t_j is not a variable. Here b_j is t_j if $j \in \{i_1, \dots i_k\}$, otherwise b_j is a fresh variable. For example, the CHRs generated by the declarations for `Coll` are:

$$\texttt{Coll}\ c\ e,\ \texttt{Coll}\ c\ d \implies e = d \qquad \texttt{Coll [}c\texttt{]}\ e \implies e = c \qquad \texttt{Coll [}a\texttt{]}\ a \iff \texttt{Eq}\ a$$

It is possible to combine the last two rules into a single rule $\texttt{Coll [}a\texttt{]}\ e \iff e = a, \texttt{Eq}\ a$ as we have done in the introduction.

5.2 Strong Conditions to guarantee Sound and Decidable Type Classes

In order for type inference to be both *sound* and *decidable*, the resulting CHR rules must be consistent, confluent and terminating. If we allow for arbitrary class and instance declarations, this will not always be the case.

Earlier work (Sulzmann et al. 2007) identifies a set of conditions that guarantee that the resulting CHR rules are both terminating and confluent. The CHR resulting from instance declarations must be terminating and class declarations must satisfy the following two conditions:

- (*Consistency Condition*) Consider a pair of instance declarations for a class `TC`:

  $$\texttt{instance}\ D_1\ \texttt{=> TC}\ t_1\ \dots\ t_n \qquad \texttt{instance}\ D_2\ \texttt{=> TC}\ s_1\ \dots\ s_n$$

  Then, for each functional dependency $fd_i = (a_{i_1}, ..., a_{i_k}$ `->` $a_{i_0})$ for `TC`, the following condition must hold: for any substitution θ such that $\theta(t_{i_1}, ..., t_{i_k}) = \theta(s_{i_1}, ..., s_{i_k})$ we must have that $\theta(t_{i_0}) = \theta(s_{i_0})$.
- (*Coverage Condition*) Consider an instance declaration for class `TC`:

  $$\texttt{instance}\ D\ \texttt{=> TC}\ t_1\ \dots\ t_n \qquad (1)$$

  Then, for each functional dependency $fd_i = (a_{i_1}, ..., a_{i_k}$ `->` $a_{i_0})$ for `TC`, we require that $vars(t_{i_0}) \subseteq vars(t_{i_1}, \dots, t_{i_k})$.

5.3 Relaxed Conditions to guarantee Soundness for Terminating Goals

Many practical programs violate the Coverage Condition. Recall the program

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

which violates the Coverage Condition because $vars(\texttt{[b]}) \not\subseteq vars(\texttt{[a]})$.

We cannot naively drop the Coverage Condition; but we may impose the following *Weak Coverage Condition*.

- (*Weak Coverage Condition*) For the instance declaration (1) and each functional dependency $fd_i = (a_{i_1}, \ldots, a_{i_k}$ `->` $a_{i_0})$, then $vars(t_{i_0}) \subseteq closure(D, vs)$ where

$$
\begin{array}{lcl}
closure(D, vs) & = & \bigcup_{i=1}^{\infty} covered^i(D, vs) \\
covered^1(D, vs) & = & \bigcup_{\substack{\mathtt{TC}\ t_1 \ldots t_n \in D \\ \mathtt{TC}\ a_1 \ldots a_n \mid a_{i_1}, \ldots, a_{i_k}\ \texttt{->}\ a_{i_0}}} \{vars(t_{i_0}) \mid vars(t_{i_1}, \ldots, t_{i_k}) \subseteq vs\} \\
covered^{i+1}(D, vs) & = & covered^1(D, covered^i(D, vs))
\end{array}
$$

Like the Coverage Condition, the Weak coverage is sufficient to establish *local confluence* of the resulting CHR rules in combination with the Consistency Condition (Sulzmann et al. 2007). However, unlike the Coverage Condition, Weak Coverage is not sufficient to establish *termination*. Recall the infinite derivation from the introduction

$$\mathtt{F}\ \mathtt{[}a\mathtt{]}\ a \rightarrowtail (\mathtt{F}\ \mathtt{[}b\mathtt{]}\ b,\ a = \mathtt{[}b\mathtt{]}) \rightarrowtail (\mathtt{F}\ \mathtt{[}c\mathtt{]}\ c,\ a = \mathtt{[}b\mathtt{]},\ b = \mathtt{[}c\mathtt{]}) \rightarrowtail \ldots$$

Fortunately, such devious goals usually do not show up for realistic programs.

We can summarize the *relaxed conditions* as follows. Given a set C of class and instance declarations, we derive the corresponding CHR program P from C using the translation from Section 5.1. The relaxed conditions are essentially the same as that used by our CHR theoretical results, namely

- (*Range restrictedness*): P must be range restricted;
- (*Local Confluence*): P must be locally-confluent; and
- (*Ground Termination*): P must be ground-terminating.

Range restrictedness of P can be established via simple syntactic checks. For example, if all given instance declarations of the form (`instance` $Ctx \Rightarrow H$) satisfy the constraint $vars(Cxt) \subseteq vars(H)$, then the resulting P will be range-restricted (Sulzmann et al. 2007). Local confluence follows directly from the Weak Coverage Condition and the Consistency Condition (Sulzmann et al. 2007).

To prove Ground Termination we can rely on the existing state-of-the-art work on termination for CHR programs, such as (Frühwirth 2000) and (Pilozzi and Schreye 2008). For example, we can prove that the rule ($\mathtt{F}\ \mathtt{[}a\mathtt{]}\ b \iff b = \mathtt{[}c\mathtt{]},\ \mathtt{F}\ a\ c$) is ground terminating by defining $rank(\mathtt{[}x\mathtt{]}) = 1 + rank(x)$. Each rule application to a ground state decreases the rank, so any corresponding derivation must eventually terminate.

An alternative method for proving ground termination in our context is the notion of CLP projection as described in (Haemmerlé et al. 2011). Formally, the *projection* of a simplification rule $(h_1, \ldots, h_n \iff B)$ is the set of Horn clauses $\{h_i \leftarrow B \mid i \in 1, \ldots n\}$. The projection of a CHR program is the union of the projections of its simplifications. If the projection of a set P of mono-headed simplifications is terminating then so is P (Haemmerlé et al. 2011). Since purely built-in propagation rules either do not apply or fail on ground states, there exists a direct correspondence between the ground termination if P and its projection. We can therefore use state-of-the-art CLP termination analysis tools to verify ground-termination of the CHR type inference programs. For instance, we used the AProVE analyzer (Giesl et al. 2006) to automatically prove ground-termination of all the programs given as examples in the present paper.

5.4 Correctness of the `UndecidableInstances` Flag

Assuming the relaxed conditions are satisfied, we can verify the correctness of type inference in GHC under the `UndecidableInstances` flag. We can formalize the behavior of this flag as follows: given a depth bound B and a goal S, we choose a bounded derivation $S \rightarrowtail S_1 \rightarrowtail \cdots \rightarrowtail S_b$ for S such that either:

- (*Final State*) $S_b \not\rightarrowtail$, $b \leq B$, then the answer is S_b; or
- (*Unknown*) $S_b \rightarrowtail \ldots$, $b = B$, then the answer is *unknown*.

An answer of *unknown* is reported to the user in the form of a compiler error. Otherwise, by Corollary 2 we know that S_b is the one unique answer for any finite derivation of S.

6 Conclusion and Related Work

The idea that confluent programs are consistent can be traced back to early CHR confluence results (Abdennadher et al. 1999), but the general proof is more recent (Haemmerlé et al. 2011). In comparison with these earlier works, the main result of Section 3 requires a weaker form of confluence (i.e. ground confluence) in combination with the additional condition that CHR are range-restricted. In the context of types, consistency is an important condition to guarantee type safety ("well-typed programs will not go wrong"). Hence, the result of Section 3 provides some general consistency criteria to ensure that type class programs are safe.

Establishing confluence in the presence of non-termination is a notoriously difficult problem (Haemmerlé 2012). Our results in Section 4 advance the state of the art in this area by showing that existentially-terminating goals (to non-`False` states) are confluent for range-restricted, ground-terminating and locally confluent programs. These results have an important practical applications in the type inference setting for type classes.

In our current formulation, the ground termination assumption trivially rules out super classes, i.e. CHR rules which propagate user constraints. Range-restrictedness rules out instance declarations such as (`instance (F a c, F c b) => F [a] [b]`) since variable `c` does not appear in `F [a] [b]`. We believe that it is possible to relax both restrictions. This is something we plan to investigate in future work.

In another direction, we intend to investigate to what extent our results are transferable to type functions (Schrijvers et al. 2008), a concept related to type classes with functional dependencies.

From the point of view of general CHR confluence state of art, we plan generalizing consistency of ground-confluent but non range-restricted program by using CLP projection (Haemmerlé et al. 2011). It seems also worthwhile to prove ground-confluence of non ground-terminating programs using diagrammatic techniques (Haemmerlé 2012).

Acknowledgments

We thank the reviewers for their comments.

References

Abdennadher, S., Frühwirth, T., and Meuss, H. 1999. Confluence and semantics of Constraint Simplification Rules. *Constraints 4,* 2, 133–165.

Duck, G. 2012. SMCHR: Satisfiability modulo Constraint Handling Rules. *Theory and Practice of Logic Programming, Proceedings of the 28th International Conference on Logic Programming Special Issue 12,* 4-5, 601–618.

Duck, G., Jaffar, J., and Koh, N. 2013. Constraint-based program reasoning with heaps and separation. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming.* LNCS, vol. 8124. Springer, 282–298.

Duck, G., Stuckey, P., and Sulzmann, M. 2007. Observable confluence for Constraint Handling Rules. In *Proceedings of the 23rd International Conference on Logic Programming.* LNCS, vol. 4670. Springer, 224–239.

Frühwirth, T. 2000. Proving termination of constraint solver programs. In *New Trends in Constraints.* LNCS, vol. 1865. Springer, 298–317.

Frühwirth, T. 2009. *Constraint Handling Rules*, 1st ed. Cambridge University Press.

GHC 2014. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

Giesl, J., Schneider-Kamp, P., and Thiemann, R. 2006. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning.* LNCS, vol. 4130. Springer, 281–286.

Haemmerlé, R. 2011. (Co)-inductive semantics for Constraint Handling Rules. *Theory and Practice of Logic Programming, Proceedings of the 27th International Conference on Logic Programming Special Issue 11,* 4–5, 593–609.

Haemmerlé, R. 2012. Diagrammatic confluence for Constraint Handling Rules. *Theory and Practice of Logic Programming, Proceedings of the 28th International Conference on Logic Programming Special Issue 12,* 4-5, 737–753.

Haemmerlé, R., Lopez-Garcia, P., and Hermenegildo, M. 2011. CLP projection for Constraint Handling Rules. In *Proceedings of the 13th International Symposium on Principles and Practices of Declarative Programming.* ACM, 137–148.

Hallgren, T. 2001. Fun with functional dependencies. In *Proceedings of the Joint CS/CE Winter Meeting.* Department of Computing Science, Chalmers, Sweden, 135–145.

Jones, M. P. 2000. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming.* LNCS, vol. 1782. Springer, 230–244.

Jones, S. P., Ed. 2003. *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press.

Newman, M. H. A. 1942. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics 43,* 2, 223–243.

Pilozzi, P. and Schreye, D. D. 2008. Termination analysis of CHR revisited. In *Proceedings of the 24th International Conference on Logic Programming.* LNCS, vol. 5366. Springer, 501–515.

Schrijvers, T., Jones, S. P., Chakravarty, M., and Sulzmann, M. 2008. Type checking with open type functions. In *Proceedings of the 13th International Conference on Functional Programming.* ACM, 51–62.

Sneyers, J., Weert, P. V., Schrijvers, T., and Koninck, L. D. 2010. As time goes by: Constraint Handling Rules. *Theory and Practice of Logic Programming 10,* 1, 1–47.

Somogyi, Z., Henderson, F., and Conway, T. 1996. The execution algorithm of mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 29,* 1-3, 17–64.

Stuckey, P. and Sulzmann, M. 2005. A theory of overloading. *Transactions on Programming Languages and Systems 27,* 6, 1216–1269.

Sulzmann, M., Duck, G., Jones, S. P., and Stuckey, P. 2007. Understanding functional dependencies via Constraint Handling Rules. *Journal of Functional Programming 17,* 1, 83–129.

Wadler, P. and Blott, S. 1989. How to make *ad-hoc* polymorphism less *ad-hoc.* In *Proceedings of the 17th Symposium on Principles of Programming Languages.* ACM, 60–76.