On Termination, Confluence and Consistent CHR-based Type Inference

GREGORY J. DUCK
National University of Singapore
(e-mail: gregory@comp.nus.edu.sg)

RÉMY HAEMMERLÉ
Universidad Politécnica de Madrid
(e-mail: remy@clip.dia.fi.upm.es)

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

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

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 are 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-restricted-ness, local confluence, and ground termination, that ensure the consistency of CHR-based type inference that maps to potentially non-terminating CHR programs.

1 Introduction

Constraint Handling Rules (CHR) 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, 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 \((\text{Eq } a)\). The instance declaration states that we obtain equality among lists assuming we supply equality on the element type. Here the notation \([a]\) represents a list type with element type \(a\). Following (Sulzmann et al. 2007), the above maps to the following CHR simplification rule

\[
\text{Eq } [a] \iff \text{Eq } a
\]

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 \(\text{Eq } t_{xs}, t_{xs} = [t_y], t_f = (t_{xs} \rightarrow t_y \rightarrow \text{Bool})\) which reduces, via application of the CHR rules and substitution, to \(t_{xs} = [t_y], t_f = ((t_y) \rightarrow t_y \rightarrow \text{Bool})\). Hence, function \(f\) has type \(\forall a. [a] \rightarrow a \rightarrow \text{Bool}\).

This approach extends to richer set 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 types 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 \(a \rightarrow b\)

\[
\begin{align*}
\text{class } & F \; a \; b \mid a \rightarrow b \\
\text{instance } & F \; \text{Int} \; \text{Bool} \\
\text{instance } & F \; a \; b \Rightarrow F \; [a] \; [b]
\end{align*}
\]

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

\[
\begin{align*}
F \; a \; b, F \; a \; c & \implies b = c \\
F \; \text{Int} \; b & \iff b = \text{Bool} \\
F \; [a] \; b & \iff b = [c], F \; a \; c
\end{align*}
\]

The last 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 \((F \; t_1 \; t_2)\) will be reduced in a finite number of steps. However, consider the non-ground goal \((F \; [a] \; a)\), where \(a\) is some variable, for which we find the following non-terminating derivation

\[
F \; [a] \; a \Rightarrow (F \; [b] \; b, a = [b]) \Rightarrow (F \; [c] \; c, a = [b], b = [c]) \Rightarrow \ldots
\]
The above example represents a typical (albeit much simplified) example that is found when applying type classes for expressive type reasoning (Hallgren 2000).

Fortunately, the program text of realistic programs will usually not yield devious constraints such as \((F[a]a)\). 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 \((\text{List } a)\) and \(a=b\) for equality.

For example, the term \(p(X, q(Y), \text{list}(Z))\) 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: \(\text{ProgVars}\), \(\text{GlobalVars}\) and \(\text{LocalVars}\). We define:

\[
\begin{align*}
\text{Terms} & \quad t ::= v \mid F t \ldots t \\
\text{Built-in Constraints} & \quad b ::= \text{True} \mid \text{False} \mid t = t \\
\text{User Constraints} & \quad u ::= C t \ldots t \\
\text{Constraints} & \quad c ::= b \mid u
\end{align*}
\]

\(^1\) Via GHC’s \texttt{UndecidableInstances} flag.
there exists a state $S'$ such that $S \rightarrow^* S'$.

An abstract rewriting system $(\Sigma, \rightarrow^*)$ is:

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

If a program $P$ is both locally confluent and terminating, then $P$ is confluent (Frühwirth 1998). Confluence implies **logical consistency** of $P$ (Abdennadher et al. 1999; Haemmerlé et al. 2011).

Let $I$ be any property over states such that: for all $S, S' \in \Sigma$ where $S \rightarrow S'$, if $I(S)$ holds then $I(S')$ also holds. Then $I$ is an observable invariant (Duck et al. 2007). If $\mathcal{M}(X)$ is the set of multisets built from the set $X$. 

---

*Duck and Haemmerlé and Sulzmann*

where $v$ is a variable from some variable set. A *substitution* $\theta$ is a mapping from variables to terms. We use the notation $\theta.X$ to represent a substitution applied to a term or constraint $X$. We respectively define $\text{Cons}(V)$ and $\text{Usr}(V)$ 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:

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

where $H \in \mathcal{M}(\text{Usr}(\text{ProgVars}))$ and $B \in \mathcal{M}(\text{Cons}(\text{ProgVars}))$. 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 \( \text{StateVars} = \text{GlobalVars} \cup \text{LocalVars} \) and let \( S_1, S_2 \in \mathcal{M}(\text{Cons}(\text{StateVars})) \) then we define \( S_1 \equiv S_2 \) as the least relation satisfying

\[\mathcal{C}T \models (3_{\text{LocalVars}} : \text{usr}(S_1) = \text{usr}(S_2) \land S_1) \iff (3_{\text{LocalVars}} : \text{usr}(S_1) = \text{usr}(S_2) \land S_2)\]

where \( \mathcal{C}T \) is the theory of term equality and \( \text{usr}(S) \) the user constraints of the state \( S \). The set of all CHR states \( \Sigma \) is defined as the quotient set \( \Sigma = (\mathcal{M}(\text{Cons}(\text{StateVars}))/\equiv) \).

Note that we usually represent a state \([S] \in \Sigma\) by an \( S \), and we often drop the parenthesis \{\ldots\} around sets of constraints, i.e. we write \( \text{P} a, a = [b] \) instead of \( \{\text{P} a, a = [b]\} \). 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 \( (\Sigma, \rightarrow) \), where binary relation \( \rightarrow \in \Sigma \times \Sigma \) is the CHR derivation step defined as the least relation satisfying:

\[
( H \iff B ) \quad \mathcal{C}T \models S \rightarrow (\theta.H = C) \quad \mathcal{C}T \models \exists : C \land S \\
\quad \quad C \cup S \rightarrow \theta.B \cup S \\
( H \implies B ) \quad \mathcal{C}T \models S \rightarrow (\theta.H = C) \quad \mathcal{C}T \models \exists : C \land S \quad C \cup S \neq \theta.B \cup C \cup S \\
\quad \quad C \cup S \rightarrow \theta.B \cup C \cup S
\]

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

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.

Let \( (\rightarrow^*) \) the reflexive closure of \( (\rightarrow) \), and let \( (\rightarrow^*) \) be the transitive closure of \( (\rightarrow^*) \).

---

---
we define $\Sigma_I = \{S \mid S \in \Sigma \wedge I(S)\}$ then program $P$ is respectively $I$-terminating, $I$-locally-confluent, and $I$-confluent if the abstract rewriting system $(\Sigma_I, \rightarrow_I)$ is terminating, locally-confluent and confluent.

We define the set of all ground states $\Sigma_g$ as the canonical surjection of $\mathcal{M}($Constraints$(\emptyset))$ onto $\Sigma$. A CHR program $P$ is range restricted iff groundness is an observable invariant.\footnote{Note that our definition of range-restricted-ness is more general than the standard definition, i.e. that $\text{vars}(B) \subseteq \text{vars}(H)$ for all rules $(H \leftrightarrow B)$ or $(H \Rightarrow B)$.}

Before continuing we state the monotonicity of CHR transitions as number of proofs of the present paper relied 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 $\text{vars}(S,T) \cap \text{vars}(U) \subseteq \text{GlobalVars}$. If $S \rightarrow T$ holds, then so does $(S \uplus U) \rightarrow^* (T \uplus U)$.

### 3 Consistency of Ground Confluent CHR

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

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

**Proof**

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

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

For the logic reading of a simplification rule $(H \leftrightarrow B)$, we are required to show that $I \models \forall(H \leftrightarrow \exists_{\text{vars}(H)} B)$, or equivalently $\theta.H \subseteq I$ iff there exists $\rho$ that coincides with $\theta$ on $\text{vars}(H)$ such that $\rho.B \subseteq I$. If $\theta.H \subseteq I$ then for any $S \in \Sigma$ we have that $((\{\theta.h\} \uplus S_b) \rightarrow^* \text{True}$ for some $S_b \in \Sigma_g$. Therefore by monotonicity for $S = \bigcup_{h \in H} \{S_h\}$, $(\theta.H \uplus S) \rightarrow^* \text{True}$. Let $S' \in \Sigma_g$ be the state obtained by applying the simplification rule on $(\theta.H \uplus S)$. By definition of $\rightarrow$, $\rho.B \subseteq S'$ for some $\rho$ that coincides with $\theta$ on $\text{vars}(H)$. Then $S' \rightarrow^* \text{True}$ by ground-confluence, and therefore $\rho.B \subseteq I$, i.e. $I \models \theta.B$. Conversely, if $I \models \theta.B$ then for any $B$ there exists some $S_b \in \Sigma_g$ such that $((\{\rho.b\} \uplus S_b) \rightarrow^* \text{True}$. Define $S = \bigcup_{b \in B} \{S_b\}$, then by monotonicity $(\rho.H \uplus S) \rightarrow^* \text{True}$, therefore $\rho.H \subseteq S \subseteq I$, i.e. $I \models \theta.H$.

For the logical reading of a propagation rule $(H \Rightarrow B)$ one may consider the simplification $(H \leftrightarrow H, B)$ and apply previous case. \qed
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 restricted-ness, 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:

\[
\begin{align*}
F \ [a] & \rightarrow F \ [b] \ b, a = \ [b] \rightarrow \ldots \ \text{(non-termination)} \\
F \ [a] & \rightarrow F \ a \ a \ (termination)
\end{align*}
\]

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 \( S \rightarrow^* S' \) 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 \( (S \rightarrow^* \ldots) \). In contrast, existential termination means that there exists a terminating derivation from \( S \), i.e. there exists at one derivation of the form \( S \rightarrow^* T \not\rightarrow \). For example, the goal \( (F \ [a] \ [a]) \) from above 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 non-False final state, i.e. \( S \rightarrow^* T \not= \text{False} \) and \( T \not\rightarrow \), then \( S \) is universally terminating.

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

**Lemma 1** If \( S \rightarrow^* T, T \not= \text{False} \) and \( T \not\rightarrow \), then there exists a ground substitution \( \theta \) such that \( \theta.S \rightarrow^* \theta.T, \theta.T \not= \text{False} \) and \( \theta.T \not\rightarrow \).

**Proof**

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

- \( \theta.S \rightarrow^* \theta.T \) by monotonicity;
- \( \theta.T \not= \text{False} \) since \( \theta \) is a unifier of the equations in \( T \); and
- \( \theta.T \not\rightarrow \) otherwise \( T \not\rightarrow \) since \( c_x \) were fresh constants.
Lemma 2 A set $P$ of purely built-in propagation rules is terminating.

Proof

Let $S$ be a state. Now consider all pair $((H \Rightarrow B), \psi)$ such that $(H \Rightarrow B)$ is a rule of $P$ and $\psi$ 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 thus:

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

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

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

Proof

Let assume there exists an infinite derivation of the form:

$S \rightarrow S_1 \rightarrow \cdots \rightarrow S_n \rightarrow \cdots$

Let $\rho$ be an arbitrary ground substitution with domain $\text{GlobalVars}$. By monotonicity we have

$\rho.S \rightarrow^* \rho.S_1 \rightarrow^* \cdots \rightarrow^* \rho.S_n \rightarrow^* \cdots$

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

Proof of Theorem 2

By contradiction:

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

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

$P \ x \leftrightarrow \text{False} \quad P \ x \leftrightarrow x = [y], P \ y$

This program is range restricted, ground-terminating, and locally confluent since the only critical pair $(\text{False} \leftarrow P \ x \rightarrow x = [y], P \ x)$ is join-able. Although $(P \ x)$ is existentially
terminating, i.e. \((P \ x \rightarrow \ \text{False})\), it is not universally terminating because of the infinite derivation \((P \ x \rightarrow \ x = \ [y], \ P \ y \rightarrow \ x = \ [y], \ y = \ [z], \ P \ z \rightarrow \ldots)\). If we change the first rule \(P \ x \leftarrow \leftarrow \ \text{True} \) (or any other non-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 \(T_\forall(S)\) and \(T_\exists(S)\) to respectively hold if state \(S\) is universally or existentially terminating. Clearly \(T_\forall\) is an observable invariant.

**Lemma 4** If \(P\) is locally-confluent, then \(P\) is \(T_\forall\)-confluent.

**Proof**
Define \(\Sigma_\forall = \{ S \mid S \in \Sigma \land T_\forall(S) \}\), then the abstract rewriting system \((\Sigma_\forall, \rightarrow)\) 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 \(T_\forall\)-confluence. However, this is overkill, as \(P\) is already assumed to be locally confluent.

The condition \(T_\exists\) 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 \(T'_\exists(S)\) to mean existential termination to a non-false state, i.e. \(T'_\exists(S)\) holds iff there exists a derivation \(S \rightarrow^* T \neq \text{False}\) and \(T \not\rightarrow\), then we can state the following:

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

**Proof**
By Theorem 2, \(T'_\exists = T_\forall\), and therefore (1) holds. By Lemma 4, \(P\) is \(T_\forall\)-confluent, and therefore (2) holds.

To elaborate further: given a state \(S\), suppose we execute \(S\) and discover a finite derivation \(S \rightarrow^* T\), 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 \(S \rightarrow^* T \not\rightarrow\), then for all \(S \rightarrow^* U \not\rightarrow\) we have that \(T = U\).

**Proof**
If \(T'_\exists(S)\) then \(T = U\) follows from Corollary 1. If \(\neg T'_\exists(S)\) then \(T = U = \text{False}\) since \(T_\exists(S)\) holds by assumption.

Note that uniqueness of answers is not equivalent to confluence for non-terminating programs. For example, if \((\ldots \leftarrow T \leftarrow S \rightarrow^* U \rightarrow^* \ldots)\) 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 some of these conditions, which in turn, allows for more expressive types.

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:

\[ \text{class } D \Rightarrow C \ a_1 \ldots a_n | \ fd_1,\ldots,fd_n \]  

(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 \(a_{i_1},\ldots,a_{i_k} \Rightarrow a_{i_0}\) where \(\{i_0,\ldots,i_k\} \subseteq 1..n\).

Both \(D\) and the \(fd\) set may be empty and omitted. The basic syntax for instance-declarations\(^4\) is:

\[ \text{instance } D \Rightarrow C \ t_1 \ldots t_n \]  

(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 \((\text{Coll } c\ e)\) type-class constraint representing an abstract collection-type \(c\) with element-type \(e\):

\[ \text{class Coll } c\ e | \ c \Rightarrow e \]  

\[ \text{instance Eq } a \Rightarrow \text{Coll } [a] a \]

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 \((\text{Coll } a\ b)\) and \((\text{Coll } a\ c)\) then \(b = c\). The instance declaration states that \((\text{Coll } [a] \ a)\) holds for any type satisfying \((\text{Eq } a)\).

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: For class-declarations of the form \(\text{(CLASS)}\) we generate the following rules:

\[ C \ a_1 \ldots a_n \Rightarrow D \]  

(CLASS-Rule)

\[ C \ a_1 \ldots a_n, C \ b_1 \ldots b_n \Rightarrow a_{i_0} = b_{i_0} \]  

(FD-Rule)

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

\[ C \ t_1 \ldots t_n \Leftrightarrow D \]  

(INSTANCE-Rule)

\[ C \ b_1 \ldots b_n \Rightarrow b_{i_0} = t_{i_0} \]  

(IMPROVEMENT-Rule)

\(^4\) 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.
One (Improvement-Rule) is generated per \(fd_i\) provided \(t_j\) is not a variable. Here \(b_j\) is \(t_j\) if \(j \in \{i_1, \ldots, i_k\}\), otherwise \(b_j\) is a fresh variable. For example, the CHRs generated by the declarations for Coll are:

\[
\text{Coll } c \ e, \quad \text{Coll } c \ d \Rightarrow e = d \quad \text{Coll } [c] \ e \Rightarrow e = c \quad \text{Coll } [a] \ a \iff \text{Eq } a
\]

It is possible to combine the last two rules into a single rule Coll \([a] \ e \iff e = a, \text{Eq } a\) 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:

\[
\text{instance } D_1 \Rightarrow TC t_1 \ldots t_n \quad \text{instance } D_2 \Rightarrow TC s_1 \ldots s_n
\]

Then, for each functional dependency \(fd_i = (a_{i_1}, \ldots, a_{i_k} \rightarrow a_{i_0})\) for TC, the following condition must hold: for any substitution \(\theta\) such that \(\theta(t_{i_1}, \ldots, t_{i_k}) = \theta(s_{i_1}, \ldots, s_{i_k})\) we must have that \(\theta(t_{i_0}) = \theta(s_{i_0})\).

- (Coverage Condition) Consider an instance declaration for class TC:

\[
\text{instance } D \Rightarrow TC t_1 \ldots t_n
\]

Then, for each functional dependency \(fd_i = (a_{i_1}, \ldots, a_{i_k} \rightarrow a_{i_0})\) for TC, we require that \(\text{vars}(t_{i_0}) \subseteq \text{vars}(t_{i_1}, \ldots, t_{i_k})\).

5.3 Relaxed Conditions to guarantee Soundness for Terminating Goals

Many practical programs violate the Coverage Condition. Recall the program

\[
\text{class } F \ a \ b \mid a \to b \quad \text{instance } F \ 	ext{Int} \ 	ext{Bool} \quad \text{instance } F \ a \ b \Rightarrow F \ [a] \ [b]
\]

which violates the Coverage Condition because \(\text{vars}([b]) \nsubseteq \text{vars}([a])\).

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 \(fd_i = (a_{i_1}, \ldots, a_{i_k} \rightarrow a_{i_0})\), then we must have that \(\text{vars}(t_{i_0}) \subseteq \text{closure}(D, vs)\) where

\[
\begin{align*}
\text{closure}(D, vs) &= \bigcup_{i=0}^{\infty} \text{covered}^i(D, vs) \\
\text{covered}^1(D, vs) &= \bigcup_{TC \ t_1 \ldots t_n \in D} \{\text{vars}(t_{i_0}) \mid \text{vars}(t_{i_1}, \ldots, t_{i_k}) \subseteq vs\} \\
\text{TC} \ a_1 \ldots a_n \mid a_{i_1}, \ldots, a_{i_k} \rightarrow a_{i_0} \\
\text{covered}^{i+1}(D, vs) &= \text{covered}^{i}(D, \text{covered}^{i}(D, vs))
\end{align*}
\]
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

\[
F \left[ a \right] \rightarrow (F \left[ b \right], a = [b]) \rightarrow (F \left[ c \right], a = [b], b = [c]) \rightarrow \ldots
\]

Fortunately, such devious goals will 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 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 restricted-ness):** \( P \) must be range restricted;
- **(Local Confluence):** \( P \) must be locally-confluent; and
- **(Ground Termination):** \( P \) must be ground-terminating.

Range restricted-ness of \( P \) can be established via simple syntactic checks. For example, if all given instance declarations of the form \((\text{instance } Ctx \Rightarrow H)\) satisfy the constraint \( \text{vars}(Ctx) \subseteq \text{vars}(H) \), 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).

Finally, 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 \((F \left[ a \right] b \iff b = [c], F a c)\) is ground terminating by defining \( \text{rank}([x]) = 1 + \text{rank}(x) \). 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 \( S \rightarrow S_1 \rightarrow \cdots \rightarrow S_b \) for \( S \) such that either:

- **(Final State)** \( S_b \nvdash \), \( b \leq B \), then the answer is \( S_b \); or
- **(Unknown)** \( S_b \rightarrow \ldots, b = B \), 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 provide 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-restricted-ness rules out instance declarations such as

\[
\text{instance } (F \ a \ c, F \ c \ b) \Rightarrow F \ [a] \ [b]
\]

because 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).

References


