Constraint Abduction and Constraint Handling Rules

Martin Sulzmann[1], Jeremy Wazny[2] and Peter J. Stuckey[2]

[1] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg
[2] NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
{pjs,jeremyrw}@cs.mu.oz.au

Abstract. Type inference for advanced type extensions demands constraints consisting of implications, universal quantifiers etc. Hence, the current CHR solving which rewrites sets of primitive constraints is not applicable. We define an extension of the CHR solving mechanism which incorporates abductive reasoning to solve implication constraints. We state some fundamental results such as soundness and under some sufficient conditions completeness and termination. We have implemented a prototype solver as part of the Chameleon system. Our results allows us to advance the state of art in constraint solving and achieve inference for expressive type extensions.

1 Introduction

In our previous work [SS02,GJDS04], we have used Constraint Handling Rules[Frü95] (CHRs) as a powerful formalism to specify type inference systems. The benefits of CHRs should be clear. The standard approach towards type inference [Wan87,Mil78] generates a set of constraints out of the program text which are then solved by some domain-specific solver. Such solvers can often naturally be expressed in terms of CHRs [CF02,GJDS04]. In particular, we could establish some general criteria in terms of CHRs under which type inference via CHR solving is sound, complete and decidable [SS02].

However, there are a number of typing features whose type inference demands implication constraints rather than sets of constraints to capture precisely the typing problem. Typical examples are type annotations and pattern matching over constrained constructors of data types. By implication constraints we refer to constraints which are composed of Boolean conjunction, implication and a mix of universal and existential quantifiers. We give a brief overview in Section 3.

To actually obtain a type inference result, we yet need to find a solution to these implication constraints. Obviously, we could take the implication constraint itself as the solution. But then the problem of consistency checking becomes theorem proving. Something we definitely want to avoid. After all, type inference should remain (fairly) simple. Another problem is that types become unreadable. Users of typed programming languages are accustomed to type descriptions in the form of sets of constraints but will be overwhelmed by more complex forms such as implication constraints.

In its simplest form, the problem we are facing is as follows. Given $C_1 \supset C_2$ find C_0 such that $C_0 \wedge C_1 \supset C_2$. This is known as abduction [Pei65,Mah05]. We concentrate on the framework of *constraint abduction* [Mah05], where we have no notion of abducible predicates that would limit the answers to C_0 we will allow. In particular we are interested in abducing conjunctions of Herbrand equations and predicates (or in our terminology user-defined constraints) that force the implication, rather than simply abducible predicates, and we have to contend with nested quantifiers in the formula C_2.

In our setting, the meaning of user-defined constraints is defined in terms of CHRs. Hence, we need to find C_0 such that $P \models C_0 \wedge C_1 \supset C_2$ where P refers to the logical reading of CHRs.

In this paper, we consider an extension of the CHR solving mechanism to allow for abductive reasoning. Other work on abduction and CHRs e.g. [SAC00] concentrates on using CHRs to specify and solve the standard abduction problem, C_1 is a logic program with integrity constraints, and C_0 are limited to be a set of abducible predicates. Here we extend CHRs to allow implication constraints to be part of the language, and extend the CHR solving approach to tackle these constraints. This requires us to solve constraint abduction problems. To the best of our knowledge, such an extension has not been considered before.

Specifically, our contributions are:

- We extend the CHR solving mechanism with abductive reasoning to solve implication constraints (Section 4.1).
- We establish some soundness, completeness and termination results (Section 4.3).
- We report our experiences with an implementation of an instance of our implication solving scheme (Section 5).

In Section 2, we provide some background material about CHRs and give a brief overview of our previous work on CHR type applications. We conclude in Section 6. The curious reader may take a look at Section 4 for a quick overview of our solving method for implications constraints.

Proofs and background material on the precise connection between type inference and implication constraints can be found here [SWS05].

2 Background

We first review the semantics of CHRs (Section 2.1). Then, we take a look at our previous work where we have used CHRs to model type relations (Section 2.2).

2.1 Constraint Handling Rules

The syntax of CHRs is as follows.

Terms	t	$::= v \mid a \mid t \rightarrow t \mid (t,t) \mid ...$
Constraint	C	$::= t = t \mid U\ t \mid C \wedge C$
Simplification	s	$::= c_1, ..., c_n \Longleftrightarrow d_1, ..., d_m$
Propagation	p	$::= c_1, ..., c_n \Longrightarrow d_1, ..., d_m$
CHR	P	$::= s \mid p \mid P, P$

We assume a standard Herbrand term language consisting of variables v, constants a, function constructors $(\cdot \rightarrow \cdot)$, pairs $((\cdot,\cdot))$, and other constructors etc. Constraints consist of equations among Herbrand terms $(t = t)$ and user-defined constraints $(U\ t)$ to which we sometimes refer to as *primitive* constraints. Note that we treat constraints as sets. Hence, we often abbreviate $\wedge$ by "," and use set notation for a sequence of primitive constraints. Note that we make use of this Haskell like notation for CHRs, rather than the usual Prolog like notation (although we show correspondences for a few examples).

We make use of CHR *simplification* and *propagation* rules. We assume that c_i refer to user-defined constraints and d_i refer to user-defined constraints and equations. Sometimes, we refer to a set P of CHRs as a CHR program.

We use $fv(F)$ to denote the free variables in formula F. We use $\bar{\exists}_V F$ to denote the existential quantification of variables in F other than varialbes in the set V, i.e. $\exists x_1 \cdots \exists x_n F$ where $\{x_1, \ldots x_n\} = fv(F) - V$.

The logical reading of CHRs is straightforward. Details can be found in [Frü95].

The operational reading of CHRs is as follow. In case of propagation rules we add the right hand side if we find a matching copy of the left hand side in the constraint store. In case of simplifications rules we remove the matching copy and replace it by the right hand side. The formal definition is as follows.

Definition 1 (CHR Semantics). *Let P be a set of CHRs. Let (R) $c_1, ..., c_n \Longrightarrow d_1, ..., d_m \in P_p$ and C be a constraint. Let ϕ be an m.g.u. of all equations in C. Let $c'_1, ..., c'_n \in C$ such that there exists a substitution θ on variables in rule (R) such that $\theta(c_i) = \phi(c'_i)$ for $i = 1...n$, that is user-defined constraints c'_1,...,c'_n* match *the left-hand side of rule (R). Then, $C \rightarrowtail_R C, \theta(d_1), ..., \theta(d_m)$.*

Let (R) $c_1, ..., c_n \Longleftrightarrow d_1, ..., d_m \in P_p$ and C be a constraint. Let ϕ be the m.g.u. of all equations in C. Let $c'_1, ..., c'_n \in C$ such that there exists a substitution θ on variables in rule (R) such that $\theta(c_i) = \phi(c'_i)$ for $i = 1, ..., n$. , Then, $C \rightarrowtail_R C - \{c'_1, ..., c'_n\}, \theta(d_1), ..., \theta(d_m)$.

Often, we perform some equivalence transformations (e.g. normalize equations by building the m.g.u. etc) which are either implicit or explicitly denoted by $\longleftrightarrow$. A *derivation*, denoted $C \rightarrowtail^*_P C'$ is a sequence of derivation steps using rules in P such that no further derivation step is applicable to C'. CHRs are applied exhaustively, being careful not to apply propagation rules twice on the same constraints (to avoid infinite propagation). For more details on avoiding re-propagation see e.g. [Abd97]. We say a set of CHRs is *terminating* if for each C there exists C' such that $C \rightarrowtail^*_P C'$. The CHR Soundness Result [Frü95] guarantees that if $C \rightarrowtail^*_P C'$ then $P \models C \leftrightarrow \bar{\exists}_{fv(C)}.C'$. This shows that CHR rule applications perform equivalence transformations.

Note that in our description of the CHR semantics we assume set semantics. This is in contrast to the standard description which assumes a multi-set semantics [Frü95]. Set semantics is vital for our purposes. Thus, we are able to build first-order models which allows us to state some strong results. E.g., under some sufficient conditions we can guarantee canonical normal forms [SS02].

2.2 Type Inference via CHRs

Our main interest in CHRs lies in modeling type relations. A typical example are type classes which allow to specify a user-definable constraint domain.

Example 1. Here is a Haskell type class program making use of multi-parameter type classes [JJM97] with functional dependencies [Jon00].

```
data Zero
data Succ x
class Add l m n | l m -> n
instance Add Zero m m            -- (A1)
instance Add l m n =>
         Add (Succ l) m (Succ n) -- (A2)
```

We introduce a constant type constructor `Zero` and a unary type constructor `Succ` to represent natural numbers. The functional dependency `| l m -> n` imposed on the ternary type class `Add` states that the first two parameters uniquely determine the third one. The instances provide for an inductive definition of addition among our representation of natural numbers. Their meaning will become clear shortly when translating type classes to CHRs. Note that type classes usually come with a set of methods and instance bodies. We omit these details here for simplicity.

Type classes follow the open world assumption, meaning a new instance could be available at any time. Hence, it is very natural to encode them in terms of CHRs.

Example 2. Here are the CHRs resulting from the above example. Note that our CHR syntax only allows for unary user-defined constraints. Hence, we silently consider $Add\ a\ b\ c$ as a short-hand for $Add\ (a, b, c)$.

$$\begin{array}{lrl}
\text{(A1)} & Add\ Zero\ m\ m & \Longleftrightarrow True \\
\text{(A2)} & Add\ (Succ\ l)\ m\ (Succ\ n) & \Longleftrightarrow Add\ l\ m\ n \\
\text{(A3)} & Add\ l\ m\ n_1, Add\ l\ m\ n_2 & \Longrightarrow n_1 = n_2 \\
\text{(A4)} & Add\ Zero\ m\ n & \Longrightarrow m = n \\
\text{(A5)} & Add\ (Succ\ l)\ m\ n' & \Longrightarrow n' = Succ\ n
\end{array}$$

Note that we use "type class" syntax to write CHRs. In a more usual CHR syntax we have

```
A1 @ add(0,M,N)              <=> true.
A2 @ add(s(L), M, s(N))      <=> add(L,M,N).
```

```
A3 @ add(L,M,N1), add(L,M,N2) ==> N1 = N2.
A4 @ add(0,M,N)               ==> N = M.
A5 @ add(s(L),M,N)            ==> N = s(_).
```

The important point is that each instance declaration is turned into a simplification rule, see rules (A1-2). The functional dependency gives rise to rule (A3). Rules (A4-5) enforce the functional dependency for each specific instance. We refer the interested reader to [SS02,GJDS04] for more details how to translate type classes to CHRs.

Thus, we can explain the type class semantics in terms of CHR solving. For example, assume some program text generates the constraint

$$\mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ m\ n, \mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ m\ n', m = \mathit{Zero}$$

Type inference becomes CHR solving. We have that

$$\begin{array}{ll}
& \mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ m\ n, \mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ m\ n', m = \mathit{Zero} \\
\rightarrowtail_{A3} & \mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ m\ n, m = \mathit{Zero}, n = n' \\
\longleftrightarrow & \mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ \mathit{Zero}\ n, m = \mathit{Zero}, n = n' \\
\rightarrowtail_{A5} & \mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ \mathit{Zero}\ (\mathit{Succ}\ n''), n = \mathit{Succ}\ n'', m = \mathit{Zero}, n = n' \\
\rightarrowtail_{A1} & \mathit{Add}\ \mathit{Zero}\ \mathit{Zero}\ n'', n = \mathit{Succ}\ n'', m = \mathit{Zero}, n = n' \\
\rightarrowtail_{A4} & \mathit{Add}\ \mathit{Zero}\ \mathit{Zero}\ \mathit{Zero}, n'' = \mathit{Zero}, n = \mathit{Succ}\ \mathit{Zero}, m = \mathit{Zero}, n = n' \\
\rightarrowtail_{A2} & n'' = \mathit{Zero}, n = \mathit{Succ}\ \mathit{Zero}, m = \mathit{Zero}, n = n'
\end{array}$$

Note that we apply set semantics (see the first CHR derivation step). In the third and fifth step, we perform some implicit equivalence transformation steps. The result is a type constraint that gives the types for the variable n, n', m which correspond to program fragments.

Here is the above derivation written in the more usual CHR syntax.

```
      add(s(0), M, N1), add(s(0), M, N2), M = 0
>->A3 add(s(0), M, N2), M = 0, N1 = N2
<-->  add(s(0), 0, N2), M = 0, N1 = N2
>->A5 add(s(0), 0, s(N3)), N2 = s(N3), M = 0, N1 = N2
>->A1 add(0, 0, N3), N2 = s(N3), M = 0, N1 = N2
>->A4 add(0, 0, 0), N3 = 0, N2 = s(N3), M = 0, N1 = N2
>->A2 N3 = 0, N2 = s(N3), M = 0, N1 = N2
```

For the rest of the paper, we will however use the "type class" syntax of CHRs.

3 Advanced Typing via Implication Constraints

We briefly motivate the need for implication constraints whose syntax is as follow.

$$\text{Implication Constraints } F ::= C \mid \forall \bar{a}.(C \supset \exists \bar{b}.F) \mid F \wedge F$$

Note that we consider a restricted set of first-order formula. E.g., assumptions (left-hand side of $\supset$) only consist of sets of primitive constraints. In a single implication constraint $\forall \bar{a}.(C \supset \exists \bar{b}.F)$ we find universal and existential quantifiers. Note that F may contain again a mix of universal and existential quantifiers. Hence, we can immediately encode the unification problem under a mixed prefix [Mil92] (if we take all assumptions to be $True$ and equations are the only primitive constraint).

For our purposes, implication constraints have a lot of other uses. Implication constraints allow us to capture precisely the typing problem for some typing features found in Haskell [Has] with extensions. Note that similar arguments can be made for other languages such as Mercury [Hen01], HAL [DGH$^+$99] etc. For concreteness, we use Haskell syntax in the following.

Example 3. Here is a (contrived) program making use of type annotations.

```
class Foo a b where foo::a->b->Int
instance Foo a Int  where foo _ _ = 1    -- (F1)
instance Foo a Bool where foo _ _ = 2    -- (F2)
f x = (x::Int, let g :: a->Int
                   g y = foo y x
               in g x)
```

Note that variable `a` in `g`'s annotation is (implicitly) universally quantified. Hence, we need to check whether `g`'s program text satisfies the annotation. The following implication constraint represents the check for correctness of `g`'s type annotation.

$$\forall a.(t_g = a \rightarrow Int \supset \exists t_y.(t_g = t'_g, t'_g = t_y \rightarrow Int, Foo\ t_y\ t_x))$$

The above constraint holds in combination with $t_x = Int$ (generated from the surrounding context), $t_f = t_x \rightarrow (t_x, Int)$ and the instance (F). Hence, the expected type of `f` is `Int->(Int,Int)`.

Note that without implication constraints we would be forced to perform the annotation check locally. Hence, we might miss the information that $t_x = Int$. Hence, the annotation check fails and this is indeed the case for the standard Haskell implementations [GHC,HUG].

In the next example, we constrain the possible types of a data type constructor [Läu96].

Example 4. Consider

```
class Key a where getKey :: a->Int
instance Key Bool where
  getKey True = 1
  getKey False = 0
instance Key Char where
  getKey = charToInt
data KEY = forall a. Key a => Mk a
h (Mk x) = getKey x
```

The `forall a` keyword indicates that the argument type of constructor `Mk` does not appear in the type of the data type `KEY`. Hence, values of type `a` are "abstract". Keyword `Key a =>` constrains values of type `a`. This implies that when accessing a `KEY` value we may make use of the `Key` class and its associated methods. Consider function `f` which pattern matches over `KEY`. The implication constraint representing the typing problem here is as follows.

$$t_f = KEY \rightarrow t_1, \forall a.(Key\ a \supset (Key\ a, t_1 = Int))$$

Thus, we are able to represent the temporary assumption $Key\ a$ when typing the function body. The universal quantifier $\forall a$ ensures that the `KEY` values remains abstract. We can verify that $t_f = KEY \rightarrow Int$ solves the above constraint. Hence, `h` can be given type `KEY -> Int`.

4 Solving of Implication Constraints

We consider the task of solving an implication constraint F whose meaning of user-defined constraints is defined in terms of some CHR program P. We are interested in finding a constraint C (consisting of sets of primitive constraints) such that $P \models C \supset F$. Commonly, we refer to C as a *solution* of F w.r.t. P. Note that $False$ is a trivial solution which could obviously be excluded. In the following, we lift the CHR solving mechanism to the domain of implication constraints. We write $F \gg_P^* C$ to denote that solving of F yields solution C.

Our idea is to turn the implication constraint solving problem into an equivalence checking problem by making use of the fact that $C_1 \supset C_2$ iff $C_1 \leftrightarrow C_1, C_2$. Then, we execute $C_1 \rightarrowtail_P^* D_1$ and $C_1, C_2 \rightarrowtail_P^* D_2$ and check whether D_1 and D_2 are logically equivalent. If yes, then $C_1 \supset C_2$ is solved and $True$ is the solution. Hence, $C_1 \supset C_2 \gg True$. Otherwise, we observe final stores to guess a (partial) solution S. Hence, $C_1 \supset C_2 \gg S, (C_1 \supset C_2)$

The obvious guess is to take $S = D_2 - D_1$, i.e. we take the difference between both final stores. However, we allow for universal quantifications. E.g., consider $\forall a.(Bar\ a \supset Foo\ a)$. Clearly, $Foo\ a$ is not a valid solution (assuming no CHRs are imposed) because the quantified variable a escapes. Hence, we need to exclude partial solutions which are bound by some universally quantified variables.

We also allow for nested implications. E.g., consider $\forall a.(Foo\ a \supset \forall b.(Bar\ a\ b \supset ...))$. We treat such cases by recursive solving. In case, we encounter a sequence of implication constraints. We pick one single implication constraint at a time.

4.1 CHR Implication Solver with Abductive Reasoning

The description of our generic implication solving scheme is as follow.

Definition 2 (CHR Implication Solver). *Let P be a set of CHRs and F an implication constraint.*

Primitive: *We define $F \gg_P C'$ where $C \rightarrowtail_P^* C'$ if $F \equiv \exists \bar{a}.C$.*

General: *Otherwise* $F \equiv C_0, (\forall \bar{a}.D \supset \exists \bar{b}.F_1), F_2$ *where* C_0 *is a conjunction of primitive constraints,* D *is a set of assumptions and* F_1 *and* F_2 *are implication constraints.*

We compute (1) $C_0, D \rightarrowtail_P^* D'$ *and (2)* $C_0, D, F_1 \gg_P^* C'$ *for some* D' *and* C'*. We distinguish among the following cases.*

Solved: *We define* $F \gg_P C_0, F_2$ *if* $\models (\bar{\exists}_V.D') \leftrightarrow (\bar{\exists}_V.C')$ *where* $V = fv(C_0, D, \bar{a})$.

Add: *We define* $F \gg_P S, F$ *if* $\models (\bar{\exists}_V.D') \not\leftrightarrow (\bar{\exists}_V.C')$ *where* $V = fv(C_0, D, \bar{a})$ *and* $False \notin C'$ *and* S *is a conjunction of primitive constraints such that (3)* $P \models C' \supset S$*, (4)* $fv(S) \cap \bar{a} = \emptyset$ *and (5)* $P \models D' \not\supset S$.

Failure: *We define* $F \gg_P False$ *in all other cases.*

We assume that $\gg^*$ *denotes the exhaustive application of CHR implication solving steps.*

In the **Primitive** step we apply standard CHR solving. No surprises here. In the **General** step, we split the constraint store into C_0 containing sets of primitive constraints, a single implication constraint $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$ and F_2 containing the remaining implication constraints. Strictly speaking, F_2 itself could be a set of primitive constraints. Silently, we assume that all sets of primitive constraints are collected in C_0. The important point to note here is that we do not specify an order in which to solve constraints. We come back to this issue shortly. Also note that we inductively solve nested implication constraints, see (2).

The **Solved** step applies if the equivalence check succeeds. Hence, the constraint $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$ is removed from the store. Note that w.l.o.g. we assume that $\bar{b} = fv(F_2) - fv(\bar{a}, C_0)$. Any variable not bound by a universal quantifier is (implicitly) existentially bound.

The **Add** step is the most interesting solving step and applies if equivalence checking failed. We perform some abductive reasoning to find a constraint S which may help us to eventually solve the implication. Often, we refer to S as a *partial solution.* Conditions (3-5) give only a characterization which subset of primitive constraints in the final store of derivation (2) may prove useful to solve the implication constraint $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$. The least we require is the following. Condition (3) ensures that we only add in useful constraints. Condition (4) prevents universally bound variables $\bar{a}$ from escaping. Condition (5) avoids adding in redundant constraints.

4.2 Examples

We consider a couple of examples to get a feel for CHR implication solving.

Example 5. Consider $P = \{(\mathrm{F}) \quad Foo\ Int\ b \Longleftrightarrow True\}$ and

$$F = \left\{ \begin{array}{ll} t = Erk\ a \rightarrow t_3, t_3 = t_1, t_3 = t_2, & (C_0) \\ \forall b.(Bar\ a\ b \supset Bar\ a\ b, t_1 = a), & (\forall b.D_1 \supset C_1) \\ \forall b.(True \supset Foo\ a\ b, t_2 = Int) & (\forall b.D_2 \supset C_2) \end{array} \right\}$$

Solving proceeds as follow:
Step 1: Solve $C_0, (\forall b.D_1 \supset C_1)$: Add $t_1 = a$. Hence, $F \gg F, t_1 = a$.
Step 2: Solve $C_0, t_1 = a, (\forall b.D_2 \supset C_2)$: Add $t_2 = Int$. Hence, $F, t_1 = a \gg F, t_1 = a, t_2 = Int$.
Step 3: Solve $C_0, t_1 = a, t_2 = Int, (\forall b.D_1 \supset C_1), (\forall b.D_2 \supset C_2)$: Solved. Hence, $F, t_1 = a, t_2 = Int \gg t = Erk\ a \to t_3, t_3 = t_1, t_3 = t_2, t_1 = a, t_2 = Int$.

In the above example, we have been solving implications in turn, combining results until we have solved all implications or reached failure. In the following, we employ a different solving strategy.

Example 6. We assume the definitions from Example 5. We consider solving of each implication individually. That is, we perform solving until we have solved the implication or reached failure. We do not combine results here.
Step 1: Solve $C_0, (\forall b.D_2 \supset C_2)$: Add $t_2 = Int$. Hence, $F \gg F, t_2 = Int$.
Step 2: Solve $C_0, t_2 = Int, (\forall b.D_2 \supset C_2)$:

$$\begin{array}{ll} C_0, t_2 = Int, D_2 & \rightarrowtail^* t = Erk\ a \to t_3, t_3 = t_1, t_3 = t_2, t_2 = Int \\ C_0, t_2 = Int, D_2, C_2 & \rightarrowtail^* t = Erk\ a \to t_3, t_3 = t_1, t_3 = t_2, t_2 = Int, \\ & \qquad Foo\ a\ b, t_2 = Int \end{array}$$

As Example 5 shows we could solve the implication constraint by adding $t_1 = a$. Clearly, we need some reasonably sophisticated reasoning about the set of CHRs P and the constraint store to find that $t_1 = a$ is a "good" guess. A naive implementation of the add step will be unable to find any constraint which could be added. Hence, we fail.

In general, the best solving strategy seems to iterate over the set of implications. However, this may lead to a high solving cost (in terms of the underlying CHR solver).

Example 7. Consider $P = \{(\text{F})\ F\ Int\ Int\ b \Longleftrightarrow True \mid (\text{G1})\ G\ Int\ Int\ b \Longleftrightarrow True \mid (\text{G2})\ G\ Int\ c\ b \Longrightarrow c = Int\}$ and

$$F = \left\{ \begin{array}{ll} t = Erk\ a_1\ a_2 \to (Int, Int), & (C_0) \\ \forall b.(True \supset F\ a_1\ a_2\ b, a_1 = Int), & (F_1) \\ \forall b.(True \supset G\ a_1\ a_2\ b) & (F_2) \end{array} \right\}$$

Solving by iterating over the set of implications proceeds as follows.
Step 1: Solve C_0, F_1: Add $a_1 = Int$. Hence, $F \gg F, a_1 = Int$.
Step 2: Solve $C_0, a_1 = Int, F_2$: Add $a_2 = Int$ (rule (G2) applies). Hence, $F, a_1 = Int \gg F, a_1 = Int, a_2 = Int$.
Step 3: Solve $C_0, a_1 = Int, a_2 = Int, F_1$: Solved! (rule (F) applies). Hence, $F, a_1 = Int, a_2 = Int \gg C_0, a_1 = Int, a_2 = Int, F_2$.
Step 4: Solve $C_0, a_1 = Int, a_2 = Int, F_2$: Solved! (rule (1) applies). Hence, $C_0, a_1 = Int, a_2 = Int, F_2 \gg C_0, a_1 = Int, a_2 = Int$.

Notice how the sub-results affect each other. Each time we add some partial solution for one implication, we achieve progress (i.e. add a new partial solution) for the other implication. Obviously, we could generalize this example such that we need to jump back and forth between implications almost an arbitrary number of times.

Another issue is that the order of solving may affect the final result.

Example 8. Consider $P = \{Foo\ b\ Int \Longrightarrow Bar\ b\}$ and

$$(Foo\ b\ a \supset Bar\ b), (Foo\ b\ a \supset a = Int)$$

Solving the second implication constraint first yields the final result $a = Int$. In case, we switch the order we first find the partial solution $Bar\ b$ which then yields the final result $Bar\ b, a = Int$.

4.3 Results

We can state the following result for any concrete solver instance satisfying Definition 2.

Lemma 1 (Soundness). *Let P be a set of CHRs, F be an implication constraint and C be a set primitive constraints such that $F \gg_P^* C$. Then, $P \models C \supset F$.*

The above result is important to guarantee that type inference is sound. Less obvious is how to achieve "best" solutions.

Definition 3 (Principal Solutions). *Let P be a set of CHRs, F be an implication constraint. We say that C is a* solution *of F w.r.t. P iff $P \models C \supset F$.*

We say that C is a principal *solution iff C is a solution and for any other solution C' we have that $P \models C' \supset C$.*

We say that P satisfies the principal implication constraint property *iff each implication constraint has a principal solution or no solution exists.*

Immediately, we find that the principal implication constraint property cannot be satisfied in general.

Example 9. Consider $P = \{Foo\ a\ Int \Longleftrightarrow True \mid Foo\ a\ Bool \Longleftrightarrow True\}$ and $\forall a.True \supset Foo\ a\ t_x$. We find that $t_x = Int$ and $t_x = Bool$ are two incomparable solutions and there is no more general solution. In fact, this shows that Haskell does not enjoy principal types. E.g., the following variation of Example 3

```
f' x = (x, let g :: a->Int
               g y = foo y x
           in g x)
```

can be given types `f'::Int->(Int,Int)` and `f'::Bool->(Bool,Int)` but there is no more general type.

For a restricted class of CHR programs and implication constraints however we can guarantee principal solutions.

We say a CHR program P belongs to the multi-parameter type class ($MPTC$) program class if the following three restrictions are satisfied.

1. CHRs are single-headed simplification rules where we only find user-defined constraints or $True$ on right-hand sides.
2. The rules are range-restricted. We say a CHR is *range-restricted* iff grounding all variables on the left-hand side of a CHR, grounds all variables on the right-hand side.
3. The CHRs are terminating and confluent. We say a set of CHRs is *confluent* iff any order of rule applications yields to the same final store.

We say a implication constraint F is *unambiguous* iff for any sub-formula $C_0, (\forall \bar{a}.D \supset \exists \bar{b}.F_1)$ encountered during solving we have that all variables in F_1 are either uniquely determined by C_0 and D or part of $\bar{a}$.

We say a implication constraint F is *pure* iff for each sub-formula $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$ we have that (a) D consists only of user-defined constraints, and (b) any user-defined constraint in F_1 consists of only universally bound variable.

MPTC programs in combination with the unambiguity condition ensure that the logical equivalence check in the **Solved** step holds iff the considered formula holds under the CHR program. The purity condition is important to ensure that we can locally compute principal solutions. E.g., consider Example 9 where condition (c) is violated.

Lemma 2 (Completeness). *Let P be a CHR program in MPTC. Then, P satisfies the principal implication constraint property for any unambiguous and pure implication constraint.*

Termination is another important property. Obviously, the CHR program must be terminating. However, this is not sufficient to guarantee termination. A subtle problem is that our current CHR implication solver description allows to add repeated, renamed copies of constraints. E.g., consider solving of $Bar\ a \supset Foo\ a$ where $P = \{Foo\ a \supset a = [b]\}$. We find that $Bar\ a \supset Foo\ a \gg a = [b_1], (Bar\ a \supset Foo\ a) \gg a = [b_1], a = [b_2], (Bar\ a \supset Foo\ a), ...$ is a valid derivation.

We can state termination under the condition that partial solutions are variable-restricted. Formally, partial solution S in Definition 2 is *variable-restricted* iff (6) $fv(S) \subseteq fv(C_0, D) - \bar{a}$. In essence, we prevent the addition of fresh variable.

Lemma 3 (Termination). *Let P be a terminating CHR program and F an implication constraint. We assume all partial solutions are variable-restricted. Then, $F \gg_P^* C$ for some C or we report failure.*

Unfortunately, the restriction to variable-restricted (partial) solutions is too restrictive in practice.

Example 10. Consider type inference for

```
data List n = (n=Zero) => Empty
            | forall m. (n=Succ m) => List Int m
f Empty = Empty
```

which gives rise to

$$t_f = List\ n \to t_1, (n = Zero \supset (t_1 = List\ m, m = Zero))$$

In an intermediate solving step, we find final stores

$$t_f = List\ n \to t_1, n = Zero$$

and

$$t_f = List\ n \to t_1, n = Zero, t_1 = List\ m, m = Zero$$

Taking $S = t_1 = List\ m, m = Zero$ leads to success but this partial solution is not variable-restricted.

Hence, we do not enforce variable-restriction in our actual implementation. An alternative termination condition may be the restriction to terminating and range-restricted CHR programs. However, also range-restriction is something which is often not satisfied by CHRs (see rule (A5) in Example 2). A rather simple method (that in practice seems to work well) is to impose a bound on the number of solving steps.

5 Implementation

We have implemented a prototype solver as part of the Chameleon system [SW]. Our generic implication solver is written in Haskell and built upon an optimized CHR solver [Duc]. It can handle not only stand-alone implication goals, but also implications which appear in the body of CHRs rules.

Our CHR solver uses an efficient, untyped term representation (WAM style), to encode both types and constraints. This allows for efficient implementations of term matching, unification, and equivalence checking. The fact that these algorithms directly update the terms they are passed, however, complicates the abduction step.

In solving an implication constraint under CHR rules P, $\forall \bar{a}.(D \supset \exists \bar{b}.C)$ in the context of primitives C_0, we perform two CHR derivations $C_0, D \rightarrowtail_P^* D'$, and $D', C \rightarrowtail_P^* C'$. If C' is equivalent to D', wrt variables $\bar{a}, fv(C_0, D')$, then we can eliminate this implication constraint, as it is trivially *True* in this context. Otherwise, we attempt to solve it, via abduction.

Performing abduction involves comparing the C' store against C_0, and extracting any additional information it contains about the variables $fv(C_0)$, which is not already part of D'. We then project C' onto $fv(C_0)$, essentially building a set of equations $S_{C'}$, which represent part of C_0 which was *potentially* updated in the step from D' to C'. We say that $S_{C'}$ represents only a possible update, because it may contain equations that merely rename the original C_0 variables, add therefore do not add any substantial information.

All that remains is to backtrack to the point before either derivation, and test whether adding $S_{C'}$ instantiates anything. If so, $S_{C'}$ represents a partial solution of the original implication. If not, then we could find no partial solution, and we fail.

A sketch of the implication solving algorithm can be found in Figure 1. We assume that solve is called whenever a normal CHR derivation, indicated by the usual arrow, $\rightarrowtail^*$ ends, with implications left over. It is invoked with two arguments, the first of which, F represents the implication and user constraints in the store, and the second P, the current CHR program. It returns any remaining user constraints, or implications. Herbrand constraints are treated implicitly, and updated directly as the result of unification. If F is primitive, we return it immediately, since there are no implications to process. Otherwise we split F into user constraints C_0 and implications Fs , and attempt to solve each implication in turn. We use variable V_{C_0} to record the variables in C_0 at that time, S to represent the current solution to the implications, and $F's$ to record the implications which have been only partially solved.

For each implication, we first set a choice point, and then perform the left and right-hand derivations. If the result of both derivations is equivalent, we can forget about this implication, and so we break out of the loop early. If either derivation failed, then we set S to *False* and also break out of the loop. Otherwise, we project the constraints in the current (otherwise implicit) Herbrand store onto the variables C_0, which gives us $S_{C'}$, our potential solution constraints. We then rewind, and check whether $S_{C'}$ adds anything to C_0. If not, the constraints we found represent no solution, so we set S to *False*, and break out of the loop. Otherwise, we add $S_{C'}$ to C_0, record the current implication in $F's$, to return to later, and continue to the next implication.

When we finally leave the loop, if S is *False*, then solving failed. If there remain no implications to solve in $F's$, then we can return the solution we have calculated, otherwise we recursively invoke solve with the partial solution we have so far, and all remaining implications. In practice, we may place a limit on the depth of this recursion, in order to guarantee termination.

Extending CHR solving to the domain of implication constraints allows to type check some interesting programs.

Example 11. Consider the following where we make use of the CHRs from Example 2.

```
data List a n = (n=Zero) => Nil
              | forall m. (n=Succ m) => Cons a (List a m)

append ::  Add l m n =>
           List a l -> List a m -> List a n
append Nil ys = Nil
append (Cons x xs) ys = Cons x (append xs ys)
```

The above is essentially the well-known append function. The difference is that we have refined the list data type which carries an additional type argument `n` describing the length of a specific list. The type of append carries the additional property that the sum of the length of the two input lists equals the length of the output list.

```
solve(F,P) =
    if primitive(F)
    then return F'
    else let (C_0,Fs) = F        (C_0 primitive, Fs nonprimitive)
         let V_{C_0} = fv(C_0)
         let S = True
         let F's = ∅
         for ∀ā.(D ⊃ ∃b̄.C) in Fs
             let cp_1 = set_choice_point()
             do C_0,D ↣*_P D'
             let V_{C_0,D'} = fv(D')
             do D',C ↣*_P C'
             let V = ā ∪ V_{C_0,D'}
             if C' = False then S = False, break
             if ⊨ (∃̄_V D') ↔ (∃̄_V C') then break
             let H = get_herbrand_store()
             S_{C'} := project(V_{C_0},H)
             do rewind(cp_1)
             if ⊨ (∃̄_{V_{C_0}} C_0) ↔ (∃̄_{V_{C_0}} C_0,S_{C'}) then S = False, break
             S = S ∪ S_{C'}
             F's = F's ∪ {∀ā.(D ⊃ ∃b̄.C)}
         if S = False then return False
         if F's = ∅ then return C_0,S
         solve((C_0,S,F's), P)
```

Fig. 1. Implication solver with abduction

6 Conclusion

We have pointed out new directions in the applications of CHRs. We have presented an extension of CHRs to support type inference for advanced systems. Our solving approach relies on a combination of CHR solving, equivalence checking and abductive reasoning. We could show that our generic implication solver is sound and could state completeness for a restricted class of programs and constraints. We could also identify a termination condition by disallowing the addition of fresh variables. We have implemented our approach as part of the Chameleon system [SW].

Solving of implication constraints turned out to be a hard problem. There are cases where the solution we find is not principal, or we do not find a solution even if a solution exists (see Examples in Section 4.2 and Example 9). This is mainly due to the fact that local principal solutions may not exist. Hence, some more sophisticated abductive reasoning will be necessary which not only observes final stores but also the CHR program. This is a direction we plan to pursue in the future. We also intend to study the special case where the only primitive constraints are equations. Some recent work [Mah05] may prove to be useful here.

References

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

[CF02] E. Coquery and F. Fages. TCLP: Overloading, subtyping and parametric polymorphism made practical for CLP. In *Proc. of ICLP '02*, volume 2401, page 480. Springer-Verlag, 2002.

[DGH+99] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In *Proc. of the Fourth International Conference on Principles and Practices of Constraint Programming*, LNCS, pages 174–188. Springer, October 1999.

[Duc] G. J. Duck. A CHR/WAM implementation for Haskell. http://www.cs.mu.oz.au/˜ gjd/haskellchr/.

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

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

[GJDS04] P. J. Stuckey G. J. Duck, S. Peyton-Jones and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *Proc. of ESOP'04*, volume 2986 of *LNCS*, pages 49–63. Springer-Verlag, 2004.

[Has] Haskell 98 language report. http://research.microsoft.com/Users/simonpj/haskell98-revised/haskell98-report-html/.

[Hen01] F. Henderson et al. The Mercury language reference manual, 2001. http://www.cs.mu.oz.au/research/mercury/.

[HUG] Hugs home page. haskell.cs.yale.edu/hugs/.

[JJM97] S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[Jon00] M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[Läu96] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, 1996.

[Mah05] M. Maher. Herbrand constraint abduction. In *Proceedings of Logic in Computer Science Symposium (LICS)*, page to appear, 2005.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[Mil92] Dale Miller. Unification under a mixed prefix. *J. Symb. Comput.*, 14(4):321–358, 1992.

[Pei65] C.S. Peirce. *Collected Papers of Charles Saunders Peirce*. Belknap Press of Harvard University Press, 1965.

[SAC00] S S. Abdennadher and H. Christiansen. An experimental clp platform for integrity constraints and abduction. In *Proceedings of FQAS2000, Flexible Query Answering Systems*, Advances in Soft Computing series, Physica-Verlag, pages 141–152. Springer, 2000.

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

[SW] M. Sulzmann and J. Wazny. Chameleon. http://www.comp.nus.edu.sg/˜ sulzmann/chameleon.

[SWS05] M. Sulzmann, J. Wazny, and P. J. Stuckey. A framework for extended algebraic data types. http://www.comp.nus.edu.sg/˜ sulzmann, 2005.

[Wan87] Mitchell Wand. A simple algorithm and proof for type inference. In *Fundamenta Informaticae X*, pages 115–122. North-Holland, 1987.