A Theory of Overloading

Peter J. Stuckey and Martin Sulzmann
Dept. of Computer Science and Software Engineering,
The University of Melbourne,
Parkville 3052, Australia.
{pjs,sulzmann}@cs.mu.oz.au

June 3, 2002

Abstract

We present a minimal extension of the Hindley/Milner system to allow for overloading of identifiers. Our approach relies on a combination of the HM(X) type system framework with Constraint Handling Rules (CHRs). CHRs are a declarative language for writing incremental constraint solvers. CHRs allow us to precisely describe the relationships among overloaded identifiers. Under some sufficient conditions on the CHRs we achieve decidable type inference and the semantic meaning of programs is unambiguous. Our approach allows us to combine open and closed world overloading. We also show how to deal with overlapping definitions.

1 Introduction

The study of overloading, a.k.a. ad-hoc polymorphism, in the context of the Hindley/Milner system [23] dates back to Kaes [21], Wadler and Blott [34]. Since then, it became a powerful programming feature in languages such as Haskell [27], Mercury [13, 14] , HAL [6] and Clean [28]. In particular, Haskell provides through its type-class system [15] one of the most powerful overloading mechanisms. There have been a number of significant extensions of Haskell's type class mechanism such as constructor classes [17], multi-parameter classes [20] and most recently functional dependencies [19]. Each of these extensions required a careful reinvestigation of essential properties such as decidable type inference and coherent semantics. There is also a significant body of further closely related work, for example [10, 29, 4, 26, 24, 5].

Here, we present a minimal extension of the Hindley/Milner system to allow for overloading of identifiers.

Example 1 *Consider the following type-annotated program where we provide definitions for overloaded functions* leq *and* ins.

```
overload leq :: Int → Int → Bool
         leq = primLeqInt
overload leq :: Float → Float → Bool
         leq = primLeqFloat
overload ins :: ∀a.Leq (a → a → Bool) ⇒ [a] → a → [a]
         ins = let insList [] y = [y]
                   insList (x:xs) y = if leq x y then x:(insList xs y)
                                      else y:x:xs
               in insList
```

We assume that primLeqInt *(*primLeqFloat*) is a primitive function, testing for less-than-equal on integers (floats).*

Note that the definition of ins depends on leq. This is reflected in ins type where we find the constraint $Leq\ (a \rightarrow a \rightarrow Bool)$. For simplicity, we omit further definitions of ins on other data structures such as trees etc.

The novelty of our approach is that relationships among overloaded identifiers are defined in terms of the meta-language of Constraint Handling Rules (CHRs) [7]. In case of the above example, we find the following set of CHR *simplification* rules:

$$\begin{array}{rl} \text{(Leq1)} & Leq\ (Int \rightarrow Int \rightarrow Bool) \iff True \\ \text{(Leq2)} & Leq\ (Float \rightarrow Float \rightarrow Bool) \iff True \\ \text{(Ins1)} & Ins\ ([a] \rightarrow a \rightarrow [a]) \iff Leq\ (a \rightarrow a \rightarrow Bool) \end{array}$$

Rule (Leq1) states that leq is defined on type $Int \rightarrow Int \rightarrow Bool$. A similar property is stated by rule (Leq2). Rule (Ins1) states that ins on type $[a] \rightarrow a \rightarrow [a]$ is defined iff leq is defined on type $a \rightarrow a \rightarrow Bool$. Logically, the $\iff$ symbol states an if-and-only-if relation. Operationally, a simplification rule can be read as follows. Whenever there is a term which matches the left-hand side, then this term can be *simplified* (replaced) by the right-hand side.

CHRs also allow us to impose stronger constraints on the set of overloaded definitions via *propagation* rules:

$$\begin{array}{rrcl} \text{(Leq3)} & Leq\ (Int \rightarrow Int \rightarrow a) & \Longrightarrow & a = Bool \\ \text{(Ins2)} & Ins\ ([a] \rightarrow b \rightarrow c) & \Longrightarrow & b = a, c = [a] \end{array}$$

For example, rule (Leq3) states that given both input arguments of leq are Int's, then the result must be of type $Bool$. The logical meaning of $\Longrightarrow$ corresponds to Boolean implication. Operationally, we *propagate* (add) the right-hand side if there is a term matching the left-hand side.

The original ideas of employing CHRs to deal with overloading were first described in [11]. The main contribution of the current paper is that we establish some sufficient conditions in terms of CHRs under which we achieve decidable type inference. Compare this to systems such as Cayenne [3], where decidability of type inference is left to the user. Additionally, CHRs allow us to give a precise characterization under which a program is unambiguous.

The rest of this paper is structured as follows. Section 2 introduces some basic notation used throughout the paper. Section 3 gives on overview of CHRs. Section 4 introduces our CHR-based overloading system. Type inference issues are discussed in Section 5. Section 6 shows how to resolve overloading on the value-level. In Section 7 we present an extended example, defining a generic family of zip-functions. Section 8 discusses extensions such as overlapping and closed definitions of overloaded identifiers. Section 9 discusses related work. We conclude in Section 10.

2 Preliminaries

We shall be interested in manipulating constraints on types. A *type* is a variable α or of the form $T\ \tau_1 \ldots \tau_n$ where T is an n-ary type constructor and $\tau_1, \ldots, \tau_n$ are types.

A *primitive constraint* is an equation $\tau_1 = \tau_2$, or a user-defined constraint $U\ \tau_1 \ldots \tau_n$ where U is a predicate symbol (In fact we restrict ourselves to unary user-defined constraints). A *constraint* C is a set of primitive constraints. Sometimes, we write $c_1 \wedge \ldots \wedge c_n$ instead of $\{c_1, \ldots, c_n\}$ where c_i are primitive constraints. We use $True$ as an abbreviation for the empty constraint which denotes the true formula. We use $False$ as an abbreviation for $T_1 = T_2$ which denotes the unsatisfiable equation where T_1 and T_2 are two distinct constructor symbols. Given a constraint C, we use the notation h_C to refer to the set of equations in C.

We write $\bar{x}$ to denote a sequence of objects x. A *substitution* $\theta = [\bar{\tau}/\bar{\alpha}]$ simultaneously replaces each α by its corresponding τ. A *unifier* of conjunction of equations C of the form $\tau_{11} = \tau_{12} \wedge \cdots \wedge \tau_{n1} = \tau_{n2}$ is a substitution θ such that $\theta(\tau_{i1})$ is syntactically identical to $\theta(\tau_{i2})$ for $1 \leq i \leq n$. A *most general unifier* (*mgu*) for C is a unifier θ such that for each other unifier θ' of C there exists substitution ρ such that $\theta' = \rho(\theta)$.

A *type scheme* is of the form $\forall \bar{\alpha}.C \Rightarrow \tau$ where $\bar{\alpha}$ are the bound variables, C is a constraint and τ a type. Note that we can always view τ as $\forall \alpha.\alpha = \tau \Rightarrow \alpha$ where α is fresh. We commonly use σ to refer to type schemes.

We assume the reader is familiar with the basics of first-order logic. Note, we use the $\supset$ symbol to denote logical implication to distinguish it from the function type constructor $\rightarrow$. We use Haskell notation for our example programs.

Let $fv(t)$ take a syntactic term t and return the set of free variables in t. We let $\exists_W F$, where W is a set of variables $\alpha_1, \ldots, \alpha_n$, denote $\exists \alpha_1 \cdots \exists \alpha_n F$. We let $\tilde{\exists} F$ denote $\exists_{fv(F)} F$. Similarly for $\forall_W F$ and $\tilde{\forall} F$. We let $\bar{\exists}_W F$ denote the formula $\exists \alpha_1 \cdots \exists \alpha_n F$ where $\{\alpha_1, \ldots, \alpha_n\} = fv(F) - W$. Note that formulas F are always implicitly universally quantified. We write $F_1 \models F_2$ to denote that F_2 holds in any model of F_1 where F_1 and F_2 are first-order formulae.

We introduce an ordering among type schemes. We define $F \vdash (\forall \bar{\alpha}_1.C_1 \Rightarrow \tau_1) \preceq (\forall \bar{\alpha}_2.C_2 \Rightarrow \tau_2)$ iff $F \models C_2 \supset \exists \bar{\alpha}_1.(C_1 \wedge \tau_1 = \tau_2)$ where we assume there are no name clashes between α_1 and α_2 and F is a first-order formula. We define $F \vdash \sigma_1 \simeq \sigma_2$ iff $F \vdash \sigma_1 \preceq \sigma_2$ and $F \vdash \sigma_2 \preceq \sigma_1$.

3 Constraint Handling Rules

Constraint handling rules [7] (CHRs) are a multi-headed concurrent constraint language for writing incremental constraint solvers. In effect, they define transitions from one constraint to an equivalent constraint. Transitions serve to simplify constraints and detect satisfiability and unsatisfiability.

Constraint handling rules (*CHR rules*) are of two forms

$$\begin{array}{rlccc} \textbf{simplification} & (\text{Rule1}) & c_1, \ldots, c_n & \iff & d_1, \ldots, d_m \\ \textbf{propagation} & (\text{Rule2}) & c_1, \ldots, c_n & \Longrightarrow & d_1, \ldots, d_m \end{array}$$

In these rules Rule1 and Rule2 are unique identifiers for a rule, $c_1, \ldots, c_n$ are user-defined constraints and $d_1, \ldots, d_m$ are user-defined constraints or equations. The simplification rule states that given constraint $c_1, \ldots, c_n$ we can *replace* it by constraint $d_1, \ldots, d_m$. The propagation rule states that given constraint $c_1, \ldots, c_n$, we can *add* $d_1, \ldots, d_m$. We say a CHR is *single-headed* if the left hand side has exactly one user-defined constraint. A *CHR program* is a set of CHR rules.

CHR rules can also be interpreted as first-order formulas. The translation function $[\![\cdot]\!]$ from CHR rules to first-order formulas is:

$$\begin{array}{c} [\![c_1, \ldots, c_n \iff d_1, \ldots, d_m]\!] \\ = \\ \forall \bar{\alpha}(c_1 \wedge \cdots \wedge c_n \leftrightarrow (\exists \bar{\beta}\ d_1 \wedge \cdots \wedge d_m)) \end{array}$$

$$\begin{array}{c} [\![c_1, \ldots, c_n \Longrightarrow d_1, \ldots, d_m]\!] \\ = \\ \forall \bar{\alpha}(c_1 \wedge \cdots \wedge c_n \supset (\exists \bar{\beta}\ d_1 \wedge \cdots \wedge d_m)) \end{array}$$

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

For the purposes of this paper, we will restrict ourselves to CHRs made up of propagation rules and single-headed simplification rules. In Section 8 we investigate how we can make use of larger classes of CHRs, including guards and disjunction. We also require that the CHRs are *range-restricted.* A CHR is range-restricted if any substitution θ grounding $c_1, \ldots, c_n$ also is such that $\theta \cdot \theta'$ grounds all variables in $d_1, \ldots, d_m$ for any mgu θ' of the equations in $d_1, \ldots, d_m$. Range-restrictedness is not an onerous condition, for our purposes—we rarely want to introduce a new unconstrained type variable.

The operational semantics of CHRs are straightforward. We can apply a rule r in program P to a constraint C if C contains a subset matching a copy of the left hand side of the rule (we assume that substitutions represented by equations have already been applied, see examples below). The resulting constraint C' replaces this subset by the right hand side of the rule (if it is a simplification rule), or adds the right hand side of the rule to C (if it is a propagation rule). This *derivation step* is denoted $C \rightarrowtail_r C'$ or $C \rightarrowtail_P C'$, see Appendix A for details.

A *derivation*, denoted $C \rightarrowtail_P^* C'$ is a sequence of derivation steps using rules in P where no derivation step is applicable to C'. A derivation $C \rightarrowtail_P^* C'$ is *successful* iff $h_{C'}$ is satisfiable. A set P of CHRs is *terminating* iff for any constraint C there exists a constraint C' such that $C \rightarrowtail_P^* C'$.

Example 2 *Consider the set of CHRs defined in the introduction. Then the following CHR derivation is possible:*

$$
\begin{array}{ll}
 & \mathit{Ins}\ ([a] \rightarrow b \rightarrow c), \underline{\mathit{Leq}\ (\mathit{Int} \rightarrow \mathit{Int} \rightarrow d)} \\
\rightarrowtail_{Leq3} & \mathit{Ins}\ ([a] \rightarrow b \rightarrow c), \underline{\mathit{Leq}\ (\mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Bool})}, d = \mathit{Bool} \\
\rightarrowtail_{Leq1} & \underline{\mathit{Ins}\ ([a] \rightarrow b \rightarrow c)}, d = \mathit{Bool} \\
\rightarrowtail_{Ins2} & \underline{\mathit{Ins}\ ([a] \rightarrow a \rightarrow [a])}, b = a, c = [a], d = \mathit{Bool} \\
\rightarrowtail_{Ins1} & \mathit{Leq}\ (a \rightarrow a \rightarrow \mathit{Bool}), b = a, c = [a], d = \mathit{Bool}
\end{array}
$$

Confluence of CHR programs is a vital property. Confluence implies that the order of the transitions does not affect the final result. Confluent CHR programs are guaranteed to be *consistent* (in the usual sense of a theory).

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

Example 3 *For example, another derivation for the goal in Example 2 is*

$$
\begin{array}{ll}
 & \underline{\mathit{Ins}\ ([a] \rightarrow b \rightarrow c)}, \mathit{Leq}\ (\mathit{Int} \rightarrow \mathit{Int} \rightarrow d) \\
\rightarrowtail_{Ins2} & \underline{\mathit{Ins}\ ([a] \rightarrow [a] \rightarrow \mathit{Bool})}, b = [a], c = \mathit{Bool}, \\
 & \mathit{Leq}\ (\mathit{Int} \rightarrow \mathit{Int} \rightarrow d) \\
\rightarrowtail_{Ins1} & \mathit{Leq}\ (a \rightarrow a \rightarrow \mathit{Bool}), b = a, c = [a], \\
 & \underline{\mathit{Leq}\ (\mathit{Int} \rightarrow \mathit{Int} \rightarrow d)} \\
\rightarrowtail_{Leq3} & \mathit{Leq}\ (a \rightarrow a \rightarrow \mathit{Bool}), b = a, c = [a], \\
 & \underline{\mathit{Leq}\ (\mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Bool})}, d = \mathit{Bool} \\
\rightarrowtail_{Leq1} & \mathit{Leq}\ (a \rightarrow a \rightarrow \mathit{Bool}), b = a, c = [a], d = \mathit{Bool}
\end{array}
$$

CHRs transform one constraint into a constraint which is equivalent w.r.t. the CHR program. While confluence guarantees that the order of application of CHRs does not matter, in some cases we can obtain stronger results. We now define a class of CHRs which have a (weak) satisfiability test and generate a canonical form. We use this class to ensure decidable type inference.

We say a constraint C is *weakly* satisfiable w.r.t. a set P of CHRs iff $\models \tilde{\exists}([\![P]\!] \wedge C)$.

Lemma 1 (Weak Satisfiability) *Let P be a confluent set of range-restricted CHRs where each simplification rule is single-headed. Let C be a constraint and suppose $C \rightarrowtail_P^* C'$. Then $\models \tilde{\exists}([\![P]\!] \wedge C)$ iff $\models \tilde{\exists} h_{C'}$.*

We can test the (weak) satisfiability of a constraint C by executing the CHR program and testing if the resulting equational constraints are satisfiable. Note that this weak satisfiability implicitly codes an open world understanding of the user-defined constraints. The constraint is satisfiable in some model of P, not all models.

The following canonical form result will allow us to test equivalence of constraints using CHRs. It is the first canonical form result we know of for CHRs.

Lemma 2 (Canonical Form) *Let P be a confluent terminating set of range-restricted CHRs where each simplification rule is single-headed. Then $[\![P]\!] \models D \leftrightarrow D'$ iff $D \rightarrowtail_P^* C$ and $D' \rightarrowtail_P^* C'$ such that $\models (\bar{\exists}_{fv(D)} C) \leftrightarrow (\bar{\exists}_{fv(D')} C')$.*

Note that for this result we require the CHRs P to be terminating, that is $\forall C \exists C' \;\; C \rightarrowtail_P^* C'$. There are some simple syntactic criteria, e.g. no cyclic dependencies among CHRs, which ensure that P is terminating. There are also a number of other approaches to proving termination of CHR programs [8]. For a terminating set of CHRs we have a decidable confluence test [1]. In essence, we need to build "critical pairs" and test whether they are joinable.

4 HM(CHR) and Overloading

We employ the HM(X) type system framework [33, 25] as the type-theoretic basis of our CHR-based overloading system. We assume that the constraint domain X is described by a set P of CHRs. To support overloading we extend the language of expressions by allowing for overloaded definitions.

We work with the following syntactic domains:

Programs	$p ::= \mathsf{overload}\ f = e\ \mathsf{in}\ p \mid e$
Expressions	$e ::= x \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid (e :: \sigma)$
Types	$\tau ::= \alpha \mid \tau \rightarrow \tau \mid T\ \bar{\tau}$
Constraints	$C ::= \tau = \tau \mid U\ \bar{\tau} \mid C \wedge C$
Type Schemes	$\sigma ::= \tau \mid \forall \bar{\alpha}.C \Rightarrow \tau$

where f ranges over overloaded identifiers, $\cdot \rightarrow \cdot$ is the function type constructor, T is a user-defined n-ary type constructor, $\cdot = \cdot$ denotes (syntactic) equality among types, U is a user-defined n-ary predicate symbol. and $\wedge$ denotes conjunction among constraints. Note that we use ',' for conjunction among constraints in CHR rules and example CHR derivations. For syntactic convenience, we write example programs using

$$\begin{array}{lll} \mathsf{overload} & f :: \sigma & \text{instead of} \quad \mathsf{overload}\ f = (e :: \sigma)\ \mathsf{in}\ \ldots \\ & f = e & \\ & \ldots & \end{array}$$

We will also make use of pattern matching syntax. The straightforward description of this extension is omitted.

We will always assume that the relationship among constraints is specified by a set P of CHRs. We refer to P as the *program theory*. Typing judgments are of the form $P, C, \Gamma \vdash e : \tau$ where P is the program theory, C a constraint, Γ a typing environment, e an expression and τ a type. We will always require that constraints C appearing in typing judgments $P, C, \Gamma \vdash e : \sigma$ and type schemes $\forall \bar{\alpha}.C \Rightarrow \tau$ are weakly satisfiable. Note that this models an open world understanding of user-defined constraints. We will restrict our attention to *valid* judgments, i.e. those judgments which can be derived by the typing rules in Figure 1.

$$\text{(Var)}\quad \frac{(x:\sigma)\in\Gamma}{P,C,\Gamma\vdash x:\sigma} \qquad\qquad \text{(Let)}\quad \frac{\begin{array}{c}P,C,\Gamma_x\vdash e:\sigma\\ P,C,\Gamma_x.x:\sigma\vdash e':\tau'\end{array}}{P,C,\Gamma_x\vdash \mathsf{let}\ x=e\ \mathsf{in}\ e':\tau'}$$

$$\text{(Abs)}\quad \frac{P,C,\Gamma_x.x:\tau\vdash e:\tau'}{P,C,\Gamma_x\vdash \lambda x.e:\tau\to\tau'} \qquad\qquad \text{(App)}\quad \frac{\begin{array}{c}P,C,\Gamma\vdash e_1:\tau_1\to\tau_2\\ P,C,\Gamma\vdash e_2:\tau_1\end{array}}{P,C,\Gamma\vdash e_1\ e_2:\tau_2}$$

$$(\forall\text{I})\quad \frac{\begin{array}{c}P,C_1\wedge C_2,\Gamma\vdash e:\tau\\ \bar{\alpha}\not\in fv(C_1)\cup fv(\Gamma)\end{array}}{P,C_1,\Gamma\vdash e:\forall\bar{\alpha}.C_2\Rightarrow\tau} \qquad\qquad (\forall\text{E})\quad \frac{\begin{array}{c}P,C_1,\Gamma\vdash e:\forall\bar{\alpha}.C_2\Rightarrow\tau\\ [\![P]\!]\models C_1\supset[\bar{\tau}/\bar{\alpha}]C_2\end{array}}{P,C_1,\Gamma\vdash e:[\bar{\tau}/\bar{\alpha}]\tau}$$

$$\text{(Annot)}\quad \frac{\begin{array}{c}P,C,\Gamma\vdash e:\sigma\\ fv(\sigma)=\emptyset\end{array}}{P,C,\Gamma\vdash (e::\sigma):\sigma} \qquad \text{(Over)}\quad \frac{\begin{array}{c}(f:\forall a.F\ a\Rightarrow a)\in\Gamma\\ fv(\forall\bar{\alpha}.C_f\Rightarrow\tau_f)=\emptyset\\ P,C,\Gamma\vdash e:\forall\bar{\alpha}.C_f\Rightarrow\tau_f\\ \phi\ \text{mgu of}\ h_{C_f}\quad F\ \phi\tau_f\iff\phi C_f\in P\\ P,C,\Gamma\vdash p:\tau\end{array}}{P,C,\Gamma\vdash \mathsf{overload}\ f::(\forall\bar{\alpha}.C_f\Rightarrow\tau_f)=e\ \mathsf{in}\ p:\tau}$$

Figure 1: Typing Rules

The first six rules are the standard Hindley/Milner rules but extended with a program theory P and constraint component C. We note that Γ_x denotes the typing environment obtained from Γ by excluding the variable x.

In rule ($\forall$E) the statement $[\![P]\!]\models C_1\supset[\bar{\tau}/\bar{\alpha}]C_2$ requires that the constraint C_1 implies constraint $[\bar{\tau}/\bar{\alpha}]C_2$ (with $\bar{\alpha}$ replaced by $\bar{\tau}$) in any model of $[\![P]\!]$.

Our formulation of rule ($\forall$I) follows [15]. We push the "free" constraint C_2 into the type scheme. The now quantified constraint C_2 is simply erased from the left-hand side of the turnstile. Clearly, this rule is suitable for a lazy language. The standard HM(X) quantifier introduction rule keeps the constraint $\exists\bar{\alpha}.C_2$ on the left-hand side. This has some advantages as discussed in [33]. For the purpose of this paper, the present formulation of rule ($\forall$I) is sufficient.

Rule (Annot) is a straightforward extension of the standard Hindley/Milner rules to deal with type annotations.

The novelty of the typing rules resides in rule (Over) which introduces overloaded identifiers (recall they can only appear at the top-level). For each overloaded function f we introduce a new predicate symbol F. The identifier f is available in e, p or any surrounding part of the program. Therefore, we assume that the assumption $f::\forall a.F\ a\Rightarrow a$ is part of some initial type environment.

We require that overloaded definitions are closed and are annotated with their type. Each definition gives rise to a simplification rule $F\ \phi\tau_f\iff\phi C_f$, where the role of ϕ the mgu of h_{C_f} is to remove any equality constraints appearing in C_f.

Note that the set of available definitions depends on a programs typing.

Example 4 *Consider parts of the program in Example 1 from the Introduction, but with a different* *typing.*

$$\begin{array}{ll} \textsf{overload leq} :: \forall a.a = Bool \Rightarrow Int \to Int \to a \\ \qquad \textsf{leq} = \textsf{primLeqInt} \\ \textsf{overload ins} :: [Int] \to Int \to [Int] \\ \qquad \textsf{ins} = \ldots \end{array}$$

The set P'_s of CHRs arising is as follows:

$$\begin{array}{rrcl} \text{(Leq1)} & Leq\ (Int \to Int \to Bool) & \Longleftrightarrow & True \\ \text{(Ins1')} & Ins\ ([Int] \to Int \to [Int]) & \Longleftrightarrow & True \end{array}$$

Recall the set P_s of CHRs arising from Example 1:

$$\begin{array}{rrcl} \text{(Leq1)} & Leq\ (Int \to Int \to Bool) & \Longleftrightarrow & True \\ \text{(Ins1)} & Ins\ ([a] \to a \to [a]) & \Longleftrightarrow & Leq\ (a \to a \to Bool) \end{array}$$

We find that the set P_s subsumes P'_s. The more general typing in Example 1 allows for a larger set of overloaded definitions.

Note that not enforcing the side conditions in rule (Over), i.e. equalities on the right-hand side of simplification rules need not necessarily be resolved, would have resulted into the following rule instead of rule (Leq1):

$$\text{(Leq1')} \quad Leq\ (Int \to Int \to a) \quad \Longleftrightarrow \quad a = Bool$$

Such form of "improvement" might actually be desired by the programmer. However, we require that this must be explicitly specified.

We note that in our system we do not impose any hierarchies among overloaded identifiers. This is in contrast to Haskell where overloaded identifiers must be grouped into classes. Membership to a certain class and super class relationships can always be mimicked by some appropriate set of CHR propagation rules. Assume we provide definitions for two overloaded identifiers eq and leq modeling the equality and less-than-equal relation. A super-class relationship between the two can be expressed as follows:

$$\text{(Super)} \quad Leq\ (a \to a \to Bool) \quad \Longrightarrow \quad Eq\ (a \to a \to Bool)$$

Note that this rule doesn't imply that we can "extract" an implementation of eq out of a given implementation of leq. We only state that if a definition of leq on type $a \to a \to Bool$ is present for some a, then there must be also a definition of eq present on the same type.

For the remainder of the paper, we adopt the convention that P_s denotes the set of CHR simplification rules arising from overloaded definitions for a given program p. We denote by P_p the set of programmer-specifiable CHR propagation rules. The set $P = P_s \cup P_p$ forms the program theory.

4.1 Unambiguity

An important restriction usually made on constrained types is that they be *unambiguous.* This means that we can determine from the type component alone, each of the types occurring in the constraint part. Ambiguous types lead to difficulties in implementing the function since non-deterministic choices need to be made about which definitions to use. In Haskell 98 we require that for each type scheme $\forall \bar{\alpha}.C \Rightarrow \tau$ we have that $fv(C) \cap \bar{\alpha} \subseteq fv(\tau) \cap \bar{\alpha}$. That is, all bound variables found in the constraint component must also appear in the type component. The recent addition of functional dependencies [19] to Haskell made it necessary to adjust the unambiguity condition.

Here, we present a general definition of unambiguity of a type scheme w.r.t. a program theory which subsumes previous definitions.

Let $\forall\bar{\alpha}.C \Rightarrow \tau$ be a type scheme, P be the program theory used in this context and ρ be a variable renaming on $\bar{\alpha}$. Then $\forall\bar{\alpha}.C \Rightarrow \tau$ is *unambiguous* iff $[\![P]\!] \models (C \wedge \rho(C) \wedge (\tau = \rho(\tau))) \supset (\alpha = \rho(\alpha))$ for each $\alpha \in \bar{\alpha}$. We also say that e is unambiguous if $e :: \sigma$ is a well-typed expression and σ is unambiguous.

Consider the type scheme $\forall a, b.H\ (a \rightarrow b) \Rightarrow b$. Under the empty program theory this type scheme is ambiguous. The variable a can't be determined from the constraint component alone. We find that $\models H\ (a \rightarrow b) \wedge H\ (a' \rightarrow b') \wedge b = b' \not\supset a = a'$. Assume that our program theory consists of the following CHR (note that this CHR mimics a functional dependency):

$$\text{(FH)}\quad H\ (a \rightarrow b), H\ (a' \rightarrow b) \quad\Longrightarrow\quad a = a'$$

In the above type scheme, variable a is now determined by b.

4.2 Improvement

The programmer-specifiable set P_p of CHR propagation rules allows the programmer to impose stronger conditions on the set of constraints allowed to appear. It is also common to refer to this as *improvement*[1] of constraints. Functional dependencies are one example of improving constraints. For example, the declaration $Leq\ (a \rightarrow b \rightarrow c) \mid (a, b) \leadsto c$ states that both input arguments uniquely determine the result where $(a, b) \leadsto c$ is a *functional dependency.* This behavior can be modeled by the following CHR propagation rule:

$$\text{(FD)}\quad Leq\ (a \rightarrow b \rightarrow c), Leq\ (a \rightarrow b \rightarrow d) \quad\Longrightarrow\quad c = d$$

In general, any declaration with functional dependencies can be translated into a set of CHR propagation rules. Assume we have $F\ \tau \mid fd_1, \ldots, fd_m$ where $fv(\tau) = \bar{\alpha}$ and fd_i is a functional dependency of the form $(\alpha_{i_1}, \ldots, \alpha_{i_k}) \leadsto \alpha_{i_0}$. The functional dependency asserts that given fixed values of $\alpha_{i_1}, \ldots, \alpha_{i_k}$ then there is only one value of α_{i_0} for which the constraint $F\ \tau$ can hold. Note that in [19] the right-hand side of the $\leadsto$ can have a list of variables. For simplicity, we only allow for one variable on the right-hand side. The expressiveness is equivalent. The translation creates for each functional dependency a propagation rule of the form:

$$F\ \tau, F\ \theta(\rho\tau) \Longrightarrow \alpha_{i_0} = \beta_{i_0}$$

where ρ is a renaming on $\bar{\alpha}$ such that $\rho(\alpha_i) = \beta_i$ and θ maps each β_{i_j} to α_{i_j} and each other β_l to itself. This allows us to model full and faithfully functional dependencies via CHRs.

The ability to specify arbitrary programmer-definable CHR propagation rules clearly goes beyond functional dependencies.

Example 5 *Consider the three definitions.*

overload f = (e1 :: $Float \rightarrow Float$)
overload f = (e2 :: $Int \rightarrow Float$)
overload f = (e3 :: $Int \rightarrow Int$)

We find the following set P_s

$$\begin{array}{lrcl}
\textit{(F1)} & F\ (Float \rightarrow Float) & \Longleftrightarrow & True \\
\textit{(F2)} & F\ (Int \rightarrow Float) & \Longleftrightarrow & True \\
\textit{(F3)} & F\ (Int \rightarrow Int) & \Longleftrightarrow & True
\end{array}$$

[1]The term improvement was coined by Jones [18].

Our intention might be that every definition of f with argument type $Float$ must have result type $Float$. In our framework, this can be specified by an additional propagation rule

$$(F4) \quad F\ (Float \to a) \quad \Longrightarrow \quad a = Float$$

Such behavior can't be specified by functional dependencies.

4.3 Confluence

We require that program theories must be confluent.

Example 6 *Consider the following program*

$$
\begin{array}{lll}
\mathsf{overload\ eq} & :: & Int \to Int \to Bool \\
\mathsf{eq} & = & \mathsf{primEqInt} \\
\mathsf{overload\ eq} & :: & \forall a.Eq\ (a \to a \to Bool) \Rightarrow [a] \to [a] \to Bool \\
\mathsf{eq} & = & \mathsf{let\ eqL\ []\ []\ =\ True} \\
 & & \quad \mathsf{eqL\ (x{:}xs)\ []\ =\ False} \\
 & & \quad \mathsf{eqL\ []\ (y{:}ys)\ =\ False} \\
 & & \quad \mathsf{eqL\ (x{:}xs)\ (y{:}ys)\ =\ (eq\ x\ y)\ \&\&\ (eqL\ xs\ ys)} \\
 & & \mathsf{in\ eqL} \\
\mathsf{overload\ leq} & :: & Int \to Int \to Bool \\
\mathsf{leq} & = & \mathsf{primLeqInt} \\
\mathsf{overload\ leq} & :: & \forall a.[a] \to [a] \to Bool \\
\mathsf{leq} & = & \lambda l_1.\lambda l_2.\ \mathsf{True}
\end{array}
$$

where `primEqInt` *is a primitive equality function of type $Int \to Int \to Bool$.*

The following set of CHRs arise from the above overloaded definitions:

$$
\begin{array}{rl}
\text{(Eq1)} & Eq\ (Int \to Int \to Bool) \iff True \\
\text{(Eq2)} & Eq\ ([a] \to [a] \to Bool) \iff Eq\ (a \to a \to Bool) \\
\text{(Leq1)} & Leq\ (Int \to Int \to Bool) \iff True \\
\text{(Leq4)} & Leq\ ([a] \to [a] \to Bool) \iff True
\end{array}
$$

Furthermore, we assume we are given the following user-defined propagation rule:

$$\text{(Super)} \quad Leq\ (a \to a \to Bool) \quad \Longrightarrow \quad Eq\ (a \to a \to Bool)$$

states that whenever leq is defined on type $a \to a \to Bool$, then eq must be defined on type $a \to a \to Bool$ as well. Note that the above set of CHRs is non-confluent, since $Leq\ ([a] \to [a] \to Bool)$ has two derivations which are non-joinable.

$$Leq\ ([a] \to [a] \to Bool) \quad \rightarrowtail_{Leq4} \quad True$$

and

$$
\begin{array}{ll}
 & Leq\ ([a] \to [a] \to Bool) \\
\rightarrowtail_{Super} & Leq\ ([a] \to [a] \to Bool), Eq\ ([a] \to [a] \to Bool) \\
\rightarrowtail_{Leq4} & Eq\ ([a] \to [a] \to Bool) \\
\rightarrowtail_{Eq2} & Eq\ (a \to a \to Bool)
\end{array}
$$

Clearly, there is a problem among the set of overloaded definitions and the super class relationship of eq and leq.

There are however cases where it's safe to add some propagation rules to "complete" a non-confluent program theory.

Example 7 *Consider the following program*

$$\begin{array}{l} \textsf{overload ins} :: [Int] \to Int \to [Int] \\ \qquad\qquad \textsf{ins} = \lambda xs.\lambda x.\ x : xs \end{array}$$

The program theory consists of the following set of CHRs where rule (Func) states a functional dependency among the input values.

$$\begin{array}{ll} \textit{(Ins1')} & Ins\ ([Int] \to Int \to [Int]) \iff True \\ \textit{(Func)} & Ins\ (c \to e_1 \to c), Ins\ (c \to e_2 \to c) \implies e_1 = e_2 \end{array}$$

The above program theory is non-confluent.

$$\begin{array}{ll} & Ins\ ([Int] \to Int \to [Int]), Ins\ ([Int] \to a \to [Int]) \\ \rightarrowtail_{Ins1'} & Ins\ ([Int] \to a \to [Int]) \end{array}$$

and

$$\begin{array}{ll} & Ins\ ([Int] \to Int \to [Int]), Ins\ ([Int] \to a \to [Int]) \\ \rightarrowtail_{Func} & Ins\ ([Int] \to Int \to [Int]), \\ & Ins\ ([Int] \to Int \to [Int]), a = Int \\ \rightarrowtail_{Ins1'} & Ins\ ([Int] \to Int \to [Int]), a = Int \\ \rightarrowtail_{Ins1'} & a = Int \end{array}$$

are two distinct, non-joinable derivations. Adding the following propagation rule yields a confluent program theory.

$$\text{(InsFunc1)} \quad Ins\ ([Int] \to a \to [Int]) \quad \implies \quad a = Int$$

Note that rule (Func) states a general property which must hold for all ins definitions. Therefore, we had to add in an additional propagation rule per overloaded definition to complete the program theory.

Confluence becomes a subtle issue in case of overlapping definitions.

Example 8 *Consider the definition of* eq *of Example 6 extended with the following definition.*

$$\begin{array}{l} \textsf{overload eq} :: [Int] \to [Int] \to Bool \\ \qquad\quad \textsf{eq} = \ldots \textsf{ special treatment on integers } \ldots \end{array}$$

We find the following program theory:

$$\begin{array}{ll} \text{(Eq1)} & Eq\ (Int \to Int \to Bool) \iff True \\ \text{(Eq2)} & Eq\ ([a] \to [a] \to Bool) \iff Eq\ (a \to a \to Bool) \\ \text{(Eq3)} & Eq\ ([Int] \to [Int] \to Bool) \iff True \end{array}$$

The above program theory is confluent. However, note that the second and third definition of eq are overlapping. We say two definitions are *overlapping* if there exists a substitution ϕ which unifies the head atoms in the respective simplification rules. In case we require a definition of eq at type $[Int] \to [Int] \to Bool$, we must take an indeterministic choice between two possibilities.

As we will see in Section 5, confluence is a sufficient condition to ensure correctness on the level of types. Correctness on the value level, i.e. a coherent semantics, additionally requires that all simplification rules must be non-overlapping (see Section 6). In certain cases, it is possible to handle overlapping definitions via a simple extension of the CHRs (see Section 8.2).

5 Type Inference

We assume that we are given a program p and an initial environment Γ where all overloaded idenfiers f are recorded, i.e. $(f :: \forall a.F\ a \Rightarrow a) \in \Gamma$.

Stage (1) of type inference, extracts the set P_s of simplification out of the annotated program text via a simple translation. We introduce judgments of the form $p \vdash_{inf} Q$:

$$\text{(Exp)} \qquad e \vdash_{inf} \emptyset$$

$$\text{(Over)} \quad \frac{\begin{array}{c} p \vdash_{inf} Q \quad \phi \text{ mgu of } h_C \\ Q' = Q \cup \{F\ \phi\tau \Longleftrightarrow \phi C\} \end{array}}{\mathsf{overload}\ f :: (\forall\bar{\alpha}.C \Rightarrow \tau) = e\ \mathsf{in}\ p \vdash_{inf} Q'}$$

In addition, we are given a set P_p of programmer-specifiable propagation rules. Together, $P = P_s \cup P_p$, where $p \vdash_{inf} P_s$, forms the program theory. The curious reader might ask what happens if we do not provide type annotations for overloaded definitions. This would require to infer the set of simplification rules. We believe it is generally undecidable to find a terminating set of simplification rules which satisfies overloaded definitions.

To obtain complete type inference we require that P is terminating, confluent and range-restricted. We have already seen cases were an incomplete set can be completed, see Example 7. We will neglect the issue of testing for termination, confluence and completion of CHRs for the purpose of this paper and refer to [31] for more details.

Stage (2) of type inference proceeds by inference of expressions and checking that the annotated types match the actual implementation, see Figure 2. The inference algorithm is formulated as a deduction system with inference clauses of the form

$$P, \Gamma, p \vdash_{inf} (C \mid \tau)$$

where program theory P, type environment Γ and program p are input values, a constraint C and a type τ are output values. The set P consists of the CHRs collected in the previous stage and a user-defined set of propagation rules. Inference of expressions consists of (a) generating constraints from the program text and solving them w.r.t. the given program theory, (b) checking for unambiguity of type schemes, and (c) checking for validity of user-provided type annotations.

Rules (Let),(Annot) and (Over) use a generalization procedure. Let Γ be a type environment, C a constraint and τ a type. Then, we define $gen(\Gamma, C, \tau) = (True \mid \sigma)$ where $\bar{\alpha} = fv(C,\tau) \backslash fv(\Gamma)$ and $\sigma = \forall\bar{\alpha}.C \Rightarrow \tau$.

Rules (Let), (App) and (Over) make use of a procedure sat for checking satisfiability of constraints which is defined as follows:

$$\begin{array}{lll} sat(P, C) & & \\ = & True & \text{if } C \rightarrowtail_P^* C' \text{ such that} \models \tilde{\exists} h_{C'} \\ = & False & \text{otherwise} \end{array}$$

Note that the condition $\models \tilde{\exists} h_{C'}$ can be checked by a unification procedure. Immediately, it follows from Lemma 1 that the satisfiability test is decidable for terminating CHRs.

Rules (Let) and (Annot) either use or build type schemes. The procedure for checking of unambiguity of type schemes is defined as follows:

$$\begin{array}{lll} unambig(P, \forall\bar{\alpha}.C \Rightarrow \tau) & & \\ = & True & \text{if } C \wedge \rho(C) \wedge \tau = \rho(\tau) \rightarrowtail_P^* C' \\ & & \text{such that} \models C' \supset (\alpha = \rho(\alpha)) \text{ for each } \alpha \in \bar{\alpha} \\ & & \text{where } \rho \text{ is a variable renaming on } \bar{\alpha} \\ = & False & \text{otherwise} \end{array}$$

The condition $\models C' \supset (\alpha = \rho(\alpha))$ is decidable (it holds iff the mgu of $h_{C'}$ unifies α and $\rho(\alpha)$) which ensures that the above procedure is decidable for terminating CHRs.

$$\text{(Var)} \quad \frac{(x : \forall \bar{\alpha}.C \Rightarrow \tau) \in \Gamma \quad \bar{\beta} \text{ new}}{P, \Gamma, x \vdash_{inf} ([\bar{\beta}/\bar{\alpha}]C \mid [\bar{\beta}/\bar{\alpha}]\tau)}$$

$$\text{(Let)} \quad \frac{\begin{array}{c} P, \Gamma_x, e \vdash_{inf} (C_1 \mid \tau) \\ (C_2 \mid \sigma) = gen(\Gamma_x, C_1, \tau) \\ unambig(P, \sigma) \\ P, \Gamma_x.x : \sigma, e' \vdash_{inf} (C_3 \mid \tau') \\ C = C_2 \wedge C_3 \quad sat(P, C) \end{array}}{P, \Gamma_x, \mathsf{let}\ x = e\ \mathsf{in}\ e' \vdash_{inf} (C \mid \tau')}$$

$$\text{(Abs)} \quad \frac{P, \Gamma_x.x : \alpha, e \vdash_{inf} (C \mid \tau') \quad \alpha \text{ new}}{P, \Gamma_x, \lambda x.e \vdash_{inf} (C \mid \alpha \to \tau')}$$

$$\text{(App)} \quad \frac{\begin{array}{c} P, \Gamma, e_1 \vdash_{inf} (C_1 \mid \tau_1) \\ P, \Gamma, e_2 \vdash_{inf} (C_2 \mid \tau_2) \\ C' = C_1 \wedge C_2 \wedge (\tau_1 = \tau_2 \to \alpha) \\ \alpha \text{ new} \\ sat(P, C') \end{array}}{P, \Gamma, e_1 e_2 \vdash_{inf} (C' \mid \alpha)}$$

$$\text{(Annot)} \quad \frac{\begin{array}{c} P, \Gamma, e \vdash_{inf} (C_2 \mid \tau_2) \\ gen(\Gamma, C_2, \tau_2) = (\_ \mid \sigma_2) \\ unambig(P, \sigma_2) \quad unambig(P, \forall \bar{\alpha}.C_1 \Rightarrow \tau_1) \\ entail(P, \sigma_2, \forall \bar{\alpha}.C_1 \Rightarrow \tau_1) \\ fv(\forall \bar{\alpha}.C_1 \Rightarrow \tau_1) = \emptyset \\ \bar{\beta} \text{ new} \qquad C = C_2 \wedge [\bar{\beta}/\bar{\alpha}]C_1 \end{array}}{P, \Gamma, e :: \forall \bar{\alpha}.C_1 \Rightarrow \tau_1 \vdash_{inf} (C \mid [\bar{\beta}/\bar{\alpha}]\tau_1)}$$

$$\text{(Over)} \quad \frac{\begin{array}{c} P, \Gamma, e \vdash_{inf} (C_e \mid \tau_e) \quad sat(P, C_e) \\ P, \Gamma, p \vdash_{inf} (C_p \mid \tau_p) \\ fv(\sigma) = \emptyset \\ gen(\Gamma, C_e, \tau_e) = (\cdot \mid \sigma') \\ unambig(P, \sigma') \quad entail(P, \sigma', \sigma) \end{array}}{P, \Gamma, \mathsf{overload}\ f :: \sigma = e\ \mathsf{in}\ p \vdash_{inf} (C_p \mid \tau_p)}$$

Figure 2: Inference system

Example 9 *Consider the type scheme $\forall \alpha, \alpha'.H\ (\alpha \to \alpha') \Rightarrow \alpha'$. The program theory consists of rule (FH) (see 4.1). We assume that $\rho(\alpha) = \alpha''$ and $\rho(\alpha') = \alpha'''$. We check unambiguity via the derivation*

$$\begin{array}{rl} & H\ (\alpha \to \alpha'), H\ (\alpha'' \to \alpha'''), \alpha' = \alpha''' \\ \rightarrowtail_{FH} & H\ (\alpha \to \alpha'), H\ (\alpha'' \to \alpha'''), \alpha' = \alpha''', \alpha = \alpha'' \end{array}$$

Hence the type is unambiguous.

Note that we do not need to check for satisfiability and unambiguity of type schemes in rule (Var). Given that this holds for the initial type environment, our inference rules preserve these conditions.

In rule (Annot) procedure *entail* performs an entailment check to check the validity of the annotated type. The definition of *entail* is as follows:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{entail}(P, \forall\bar{\alpha}.C \Rightarrow \tau, \forall\bar{\alpha}'.C' \Rightarrow \tau')} \\
= & \mathit{True} & \text{if } C' \wedge \tau' = \alpha' \wedge \alpha = \alpha' \rightarrowtail_P^* C_1, \text{ and} \\
& & C' \wedge \tau' = \alpha' \wedge \alpha = \alpha' \wedge \tau = \alpha \wedge C \rightarrowtail_P^* C_2 \\
& & \text{such that } \models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2) \\
& & \text{where } \alpha, \alpha' \text{ are new variables and} \\
& & V = \mathit{fv}(C' \wedge \tau' = \alpha' \wedge \alpha = \alpha') \cup \mathit{fv}(\forall\bar{\alpha}.C \Rightarrow \tau) \\
= & \mathit{False} & \text{otherwise}
\end{array}
$$

The idea is to rewrite $\forall\bar{\alpha}.C \Rightarrow \tau$ into the equivalent type scheme $\forall\bar{\alpha},\alpha.C \wedge \tau = \alpha \Rightarrow \alpha$, and then use equivalence testing (possible through the Lemma 2) to test implication.

Note that the condition $\models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2)$ is decidable.

Example 10 *Consider the inference for the second definition of* ins *in Example 4. The inferred type for the expression is $\forall\beta.Leq\ (\beta \to \beta \to Bool) \Rightarrow [\beta] \to [\beta] \to Bool$, while the declared type is $[Int] \to [Int] \to Bool$. The entailment test determines*

$$
\begin{array}{ll}
 & [Int] \to [Int] \to Bool = \alpha', \alpha = \alpha' \\
\rightarrowtail_P^* & [Int] \to [Int] \to Bool = \alpha' = \alpha
\end{array}
$$

and

$$
\begin{array}{ll}
 & [Int] \to [Int] \to Bool = \alpha', \alpha = \alpha', \\
 & [\beta] \to [\beta] \to Bool = \alpha, Leq\ (\beta \to \beta \to Bool) \\
\leftrightarrow & [Int] \to [Int] \to Bool = \alpha' = \alpha, \\
 & \beta = Int, Leq\ (Int \to Int \to Bool) \\
\rightarrowtail_{Leq1} & [Int] \to [Int] \to Bool = \alpha' = \alpha, \beta = Int
\end{array}
$$

Hence the entailment condition holds.

5.1 Soundness Results

We can state that procedures *unambig* and *entail* are sound. Soundness of *sat* follows from Lemma 1.

Lemma 3 (Soundness of Unambiguity) *Let P be a set of CHRs, $\forall\bar{\alpha}.C \Rightarrow \tau$ be a type scheme and ρ be a variable renaming on $\bar{\alpha}$ such that $C \wedge \rho(C) \wedge \tau = \rho(\tau) \rightarrowtail_P^* C'$ where $\models C' \supset (\alpha = \rho(\alpha))$ for each $\alpha \in \bar{\alpha}$. Then for each $\alpha \in \bar{\alpha}$ $[\![P]\!] \models (C \wedge \rho(C) \wedge (\tau = \rho(\tau))) \supset (\alpha = \rho(\alpha))$.*

Lemma 4 (Soundness of Entailment) *Let P be a set of CHRs, α and α' two fresh variables and $\sigma = \forall\bar{\alpha}.C \Rightarrow \tau$ and $\sigma' = \forall\bar{\alpha}'.C' \Rightarrow \tau'$ where $C' \wedge \tau' = \alpha' \wedge \alpha = \alpha' \rightarrowtail_P^* C_1$ and $C' \wedge \tau' = \alpha' \wedge \alpha = \alpha' \wedge C \rightarrowtail_P^* C_2$ such that $\models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2)$, where $V = \mathit{fv}(\sigma) \cup \mathit{fv}(\sigma') \cup \mathit{fv}(C' \wedge \tau' = \alpha' \wedge \alpha = \alpha')$. Then $[\![P]\!] \vdash \sigma \preceq \sigma'$.*

We conclude that the inference system described in Figure 2 is sound w.r.t. the typing rules in Figure 1.

Theorem 1 (Soundness of Type Inference) *Let p be a program, Γ a type environment, P_p be a set of propagation rules, P_s be a set of CHRs, C a constraint and τ a type such that $p \vdash_{\mathit{inf}} P_s$ and $P_s \cup P_p, \Gamma, e \vdash_{\mathit{inf}} (C \mid \tau)$. Then $P_s \cup P_p, C, \Gamma \vdash e : \tau$ is valid.*

5.2 Completeness Results

Lemma 1 implies that our weak satisfiability test is complete.

Lemma 5 (Completeness of Unambiguity) *Let P be a confluent set of range-restricted CHRs where each simplification rule is single-headed, $\forall\bar{\alpha}.C \Rightarrow \tau$ be a type scheme and ρ be a variable renaming on $\bar{\alpha}$ such that $[\![P]\!] \models (C \wedge \rho(C) \wedge (\tau = \rho(\tau))) \supset (\alpha = \rho(\alpha))$ for each $\alpha \in \bar{\alpha}$. Then $C \wedge \rho(C) \wedge \tau = \rho(\tau) \rightarrowtail_P^* C'$ where $\models C' \supset (\alpha = \rho(\alpha))$ for each $\alpha \in \bar{\alpha}$.*

From Lemma 2 we can derive completeness of entailment checking. Note that completeness only holds under the additional assumption that type schemes are unambiguous, a natural condition imposed on type schemes in our system.

Lemma 6 (Completeness of Entailment) *Let P be a terminating, confluent set of range-restricted CHRs whose simplification rules are single-headed, α, α' two fresh variables and $\sigma = \forall\bar{\alpha}.C \Rightarrow \tau$ and $\sigma' = \forall\bar{\alpha}'.C' \Rightarrow \tau'$ such that $[\![P]\!] \vdash \sigma \preceq \sigma'$ and σ is unambiguous. Then $C' \wedge \tau' = \alpha' \wedge \alpha = \alpha' \rightarrowtail_P^* C_1$ and $C' \wedge \tau' = \alpha' \wedge \alpha = \alpha' \wedge \tau = \alpha \wedge C \rightarrowtail_P^* C_2$ such that $\models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2)$, where $V = fv(\sigma) \cup fv(\sigma') \cup fv(C' \wedge \alpha = \alpha')$.*

It is common knowledge that inference is incomplete in the presence of ambiguous types. Therefore, we require that all type schemes in the principal judgment must be unambiguous. This is sufficient to state *weak* completeness of inference.

Let P be a set of CHRs, C a constraint, Γ an environment, e an expression and σ a type scheme. We say (C, σ) is the *principal* constrained type (w.r.t. P, Γ and e) iff (1) $P, C, \Gamma \vdash e : \sigma$, and (2) for each $P, C', \Gamma \vdash e : \sigma'$ we have that (a) $[\![P]\!] \models C' \supset (\bar{\exists}_{fv(\Gamma)} C)$, and (b) $[\![P]\!] \wedge C' \models\vdash \sigma \preceq \sigma'$. We say (C, σ) is unambiguous iff σ is unambiguous. A judgment $P, C, \Gamma \vdash p : \sigma$ is *principally unambiguous* iff for each subexpression in p the principal constrained type is unambiguous.

Theorem 2 (Weak Completeness of Type Inference) *Let $P, C, \Gamma \vdash p : \tau$ be a principally unambiguous judgment such that P is terminating, confluent, range-restricted and all simplification rules are single-headed. Then $P, \Gamma, p \vdash_{inf} (C' \mid \tau')$ for some constraint C' and type τ' such that $[\![P]\!] \models C \supset \bar{\exists}_V (C' \wedge \tau = \tau')$ where $V = fv(\Gamma)$.*

6 Evidence Translation

We follow the common approach (e.g. [34]) for giving a semantic meaning for programs containing overloaded identifiers by passing around evidence values as additional function parameters. This translation process is driven by a programs typing. We wish to have a *coherent* system [16], i.e. the semantic meaning of a translated expression should be independent of its typing.

The input of the translation process is a well-typed program which is translated into a target language.

$$\textbf{Target Expressions} \quad E \quad ::= \quad x \mid \lambda x.E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E$$

In particular, we assume we are given a denumerable set of evidence variables e_C indexed by a constraint C. Evidence variables will carry the appropriate definitions of overloaded identifiers.

We introduce judgments of the form $P, \Gamma, C \vdash e : \sigma \leadsto E$ where E is the result of translating a well-typed expression e with type σ under program theory P, environment Γ and constraint C. The most interesting rules are ($\forall$I) where we abstract over evidence variables and ($\forall$E) where we provide the proper evidence values. We assume that $ec : C_1 \xrightarrow{P} [\bar{\tau}/\bar{\alpha}]C_2$ is an *evidence constructing* function. Note that the context only provides evidence for e_{C_1}. However, the instantiation site requires evidence $e_{[\bar{\tau}/\bar{\alpha}]C_2}$. The premise states that $[\![P]\!] \models C_1 \supset [\bar{\tau}/\bar{\alpha}]C_2$. In fact, this is sufficient to ensure

$$\text{(Var)} \quad \frac{(x : \sigma) \in \Gamma}{P, C, \Gamma \vdash x : \sigma \rightsquigarrow x} \qquad \text{(Let)} \quad \frac{\begin{array}{c} P, C, \Gamma_x \vdash e : \sigma \rightsquigarrow E \\ P, C, \Gamma_x.x : \sigma \vdash e' : \tau' \rightsquigarrow E' \end{array}}{P, C, \Gamma_x \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau' \rightsquigarrow \mathsf{let}\ x = E\ \mathsf{in}\ E'}$$

$$\text{(Abs)} \quad \frac{P, C, \Gamma_x.x : \tau \vdash e : \tau' \rightsquigarrow E}{P, C, \Gamma_x \vdash \lambda x.e : \tau \to \tau' \rightsquigarrow \lambda x.E} \qquad \text{(App)} \quad \frac{\begin{array}{c} P, C, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow E_1 \\ P, C, \Gamma \vdash e_2 : \tau_1 \rightsquigarrow E_2 \end{array}}{P, C, \Gamma \vdash e_1\ e_2 : \tau_2 \rightsquigarrow E_1\ E_2}$$

$$(\forall \mathrm{I}) \quad \frac{\begin{array}{c} P, C_1 \wedge C_2, \Gamma \vdash e : \tau \rightsquigarrow E \\ \bar{\alpha} \notin fv(C_1, \Gamma) \end{array}}{P, C_1, \Gamma \vdash e : \forall \bar{\alpha}.C_2 \Rightarrow \tau \rightsquigarrow \lambda e_{C_2}.E} \qquad (\forall \mathrm{E}) \quad \frac{\begin{array}{c} P, C_1, \Gamma \vdash e : \forall \bar{\alpha}.C_2 \Rightarrow \tau \rightsquigarrow E \\ [\![P]\!] \models C_1 \supset [\bar{\tau}/\bar{\alpha}]C_2 \\ ec :: C_1 \overset{P}{\to} [\bar{\tau}/\bar{\alpha}]C_2 \end{array}}{P, C_1, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau \rightsquigarrow E\,(ec\ e_{C_1})}$$

Figure 3: Evidence-passing translation

that function ec must exist. Assume we have a substitution ϕ such that $\phi C_1 \rightarrowtail_P^* True$. Then, we also have that $\phi[\bar{\tau}/\bar{\alpha}]C_2 \rightarrowtail_P^* True$. In such a situation, we find that $\phi[\bar{\tau}/\bar{\alpha}]C_2 \rightarrowtail_{P_s}^* True$, i.e. $\phi[\bar{\tau}/\bar{\alpha}]C_2$ can be solely reduced by simplification rules. This is allows us to construct evidence $e_{\phi[\bar{\tau}/\bar{\alpha}]C_2}$ by reading the CHR derivation backwards.

Recall the definitions of eq in Example 6.

```
overload eq :: Int → Int → Bool
         eq = primEqInt
overload eq :: ∀a.Eq (a → a → Bool) ⇒ [a] → [a] → Bool
         eq = let eqL [] [] = True
                  eqL (x:xs) [] = False
                  eqL [] (y:ys) = False
                  eqL (x:xs) (y:ys) = (eq x y) && (eqL xs ys)
              in eqL
```

Consider

```
exp xs ys = (eq (tail xs) ys, eq 1 3)
```

where $tail :: [a] \to [a]$ takes the tail of a list. In a first step we translate overloaded definitions. We find

```
ec_eqInt = primEqInt
ec_eqList eq = let eqL [] [] = True
                   eqL (x:xs) [] = False
                   eqL [] (y:ys) = False
                   eqL (x:xs) (y:ys) = (eq x y) && (eqL xs ys)
               in eqL
```

Note that parameter eq represents evidence for the equality function on type $a \to a \to Bool$. What remains is to insert the appropriate evidence values for expression exp.

Expression exp gives rise to the following constraints

$$Eq\ ([a] \to b \to c), Eq\ (Int \to Int \to d)$$

where we assume xs :: $[a]$, ys :: b, 1 :: Int and 3 :: Int. We find that

$$\begin{array}{ll} & Eq\ ([a] \rightarrow b \rightarrow c), Eq\ (Int \rightarrow Int \rightarrow d) \\ \rightarrowtail_{Eq3} & Eq\ ([a] \rightarrow b \rightarrow c), Eq\ (Int \rightarrow Int \rightarrow Bool), d = Bool \\ \rightarrowtail_{Eq1} & Eq\ ([a] \rightarrow b \rightarrow c), d = Bool \\ \rightarrowtail_{Eq4} & Eq\ ([a] \rightarrow [a] \rightarrow Bool), b = [a], c = Bool, d = Bool \\ \rightarrowtail_{Eq2} & Eq\ (a \rightarrow a \rightarrow Bool), b = [a], c = Bool, d = Bool \end{array}$$

Resolution of remaining equalities via unification yields

```
exp :: ∀a.Eq (a → a → Bool) ⇒ [a] → [a] → (Bool, Bool)
exp xs ys = (eq (tail xs) ys, eq 1 3)
```

Expression exp's type states that we can implement exp given we can provide evidence for $Eq\ (a \rightarrow a \rightarrow Bool)$.

The two instantiation sites of eq imply the existence of evidence constructors:

$$\begin{array}{l} ec_1 :: Eq\ (a \rightarrow a \rightarrow Bool) \xrightarrow{P} Eq\ ([a] \rightarrow [a] \rightarrow Bool) \quad \text{and} \\ ec_2 :: Eq\ (a \rightarrow a \rightarrow Bool) \xrightarrow{P} Eq\ (Int \rightarrow Int \rightarrow Bool) \end{array}$$

Consider a particular ground instance, say $a = [Int]$, of exp's type. Then, the first instantiation site of eq requires evidence $e_{Eq\ ([[Int]] \rightarrow [[Int]] \rightarrow Bool)}$. By reading the following CHR derivation backwards

$$\begin{array}{ll} & Eq\ ([[Int]] \rightarrow [[Int]] \rightarrow Bool) \\ \rightarrowtail_{Eq2} & Eq\ ([Int] \rightarrow [Int] \rightarrow Bool) \\ \rightarrowtail_{Eq2} & Eq\ (Int \rightarrow Int \rightarrow Bool) \\ \rightarrowtail_{Eq1} & True \end{array}$$

we conclude that

$$e_{Eq\ ([[Int]] \rightarrow [[Int]] \rightarrow Bool)} = ec\_eqList\ (ec\_eqList\ ec\_eqInt)$$

Rule (Eq1) reduces to $True$. That is, $e_{Eq\ (Int \rightarrow Int \rightarrow Bool)}$ must be present. Each (Eq2) rule application corresponds to applying ec_eqList to the previously constructured evidence. We find that

$$ec_1\ e_{Eq\ ([Int] \rightarrow [Int] \rightarrow Bool)} = ec\_eqList\ (ec\_eqList\ ec\_eqInt)$$

Similiarly, for the second instantiation site we have that

$$ec_2\ e_{Eq\ ([Int] \rightarrow [Int] \rightarrow Bool)} = ec\_eqInt$$

Note that this translation scheme requires run-time type information. Evidence can only be constructed once we have enough information available (the grounding substitution ϕ). However, in case of the above example we are able to provide some closed definitions for the evidence constructing functions. We define ec_1 = ec_eqList and $ec_2\ x$ = ec_eqInt Hence,

```
t_exp :: ∀a.(a → a → Bool) → [a] → [a] → (Bool, Bool)
t_exp2 e xs ys = (eq (ec1 e) (tail xs) ys, eq (ec2 ec_eqInt) 1 3)
```

where eq x = x. We assume each overloaded identifier simply passes on the appropriate calculated definition. Note that $Eq\ (a \rightarrow a \rightarrow Bool)$ has been turned into a matching function type.

Theorem 3 (Coherence) *Let P be a confluent set of range-retricted CHRs where each simplification rule is single-headed and non-overlapping. Then the translation scheme in Figure 3 yields a coherent system.*

The exact details of the translation process including formal results and proofs go clearly beyond the scope of the present paper. A formal development including a concise coherence result can be found in [30].

7 A Generic Family of Zip-Functions

The availability of a meta-language to reason about overloaded identifiers allow us to provide type inference for some interesting programs. Usually, we find the following family of zip-functions

$$\begin{array}{lll} zip & :: & \forall a, b.[a] \rightarrow [b] \rightarrow [(a, b)] \\ zip3 & :: & \forall a, b, c.[a] \rightarrow [b] \rightarrow [c] \rightarrow [(a, b, c)] \\ & \vdots & \end{array}$$

Although straightforward, we can not give a generic definition of the zip-function in a typed language such as Haskell.

Example 11 *Consider the following generic definition of the zip function.*

```
            zip2 :: ∀a,b.[a] → [b] → [(a,b)]
            zip2 [] [] = []
            zip2 (a:as) (b:bs) = (a,b):(zip2 as bs)
            zip2 [] (b:bs) = []
            zip2 (a:as) [] = []

overload zip :: ∀a,b.[a] → [b] → [(a,b)]
            zip = zip2
overload zip :: ∀a,b,cs,e. Zip ([(a,b)] → cs → e) ⇒
                  [a] → [b] → cs → e
            zip as bs cs = zip (zip2 as bs) cs
```

The corresponding CHR program for the above set of definitions is as follows:

$$\begin{array}{ll} (\mathrm{Zip1}) & Zip\ ([a] \rightarrow [b] \rightarrow [(a, b)]) \Longleftrightarrow True \\ (\mathrm{Zip2}) & Zip\ ([a] \rightarrow [b] \rightarrow cs \rightarrow e) \Longleftrightarrow \\ & \quad Zip\ ([(a, b)] \rightarrow cs \rightarrow e) \end{array}$$

In addition, we provide the following two propagation rules to enforce stronger properties on the set of overloaded identifiers allowed to appear.

$$\begin{array}{lrcl} (\mathrm{Zip3}) & Zip\ ([a] \rightarrow [b] \rightarrow [c]) & \Longrightarrow & c = (a, b) \\ (\mathrm{Zip4}) & Zip\ (a \rightarrow b \rightarrow c) & \Longrightarrow & a = [a'], b = [b'] \end{array}$$

Consider the following (partially) type-annotated expression.

```
e :: [((Int,Bool),Char)]
e= zip [1,2,3] [True,False] ['a','b','c']
```

From the program text, we generate

$$Zip\ ([Int] \rightarrow [Bool] \rightarrow [Char] \rightarrow [((Int, Bool), Char)]$$

Note that equalities have already been resolved by unification. We find that

$$\begin{array}{ll} & Zip\ ([Int] \rightarrow [Bool] \rightarrow [Char] \rightarrow [((Int, Bool), Char)] \\ \rightarrowtail_{Zip2} & Zip\ ([(Int, Bool)] \rightarrow [Char] \rightarrow [((Int, Bool), Char)] \\ \rightarrowtail_{Zip1} & True \end{array}$$

Therefore, the above expression is translated into

```
e :: [((Int,Bool),Char)]
e = zip2 (zip2 [1,2,3] [True,False]) ['a','b','c']
```

8 Extensions

We discuss how to handle closing and overlapping definitions. Both extensions fit into our framework by employing more expressive CHRs. We also introduce multi-headed simplifications.

8.1 Closing Definitions

Consider the following definitions. For simplicity, we omit the obvious function bodies.

$$\begin{array}{l} \mathsf{overload\ eq} :: Int \rightarrow Int \rightarrow Bool \\ \qquad\qquad \mathsf{eq} = \ldots \\ \mathsf{overload\ eq} :: \forall a. Eq\ (a \rightarrow a \rightarrow Bool) \Rightarrow [a] \rightarrow [a] \rightarrow Bool \\ \qquad\qquad \mathsf{eq} = \ldots \end{array}$$

We find the following set of CHRs:

$$\begin{array}{ll} \text{(Eq1)} & Eq\ (Int \rightarrow Int \rightarrow Bool) \iff True \\ \text{(Eq2)} & Eq\ ([a] \rightarrow [a] \rightarrow Bool) \iff Eq\ (a \rightarrow a \rightarrow Bool) \end{array}$$

In our current scheme the following expression would be still well-typed.

$$\begin{array}{l} \mathsf{f} :: \forall a. Eq\ (Tree\ a \rightarrow Tree\ a \rightarrow Bool) \Rightarrow \\ \quad Tree\ a \rightarrow Tree\ a \rightarrow Bool \\ \mathsf{f\ x\ y} = \mathsf{eq\ x\ y} \end{array}$$

Although there is no equality definition on trees in scope at the moment, this doesn't mean there might not be one available in the future. Compare this to a closed world approach which would rule out the above program.

Fortunately, there exists an extension of the CHR framework [2] presented so far that allows us to mix open and closed world style overloading. We introduce propagation rules where disjunction is allowed to appear on the right-hand side. By adding in the following propagation rule

$$\begin{array}{llll} \text{(CloseEq)} & Eq\ a & \Longrightarrow & (a = Int \rightarrow Int \rightarrow Bool) \vee \\ & & & (a = [b] \rightarrow [b] \rightarrow Bool) \end{array}$$

we enforce that the definitions corresponding to rules (Eq1) and (Eq2) are the only ones available.

Note that allowing for disjunction among equality constraints on the right-hand side of the $\Longrightarrow$ symbol influences the constraint solving process. In addition to simplification and propagation of constraints, we also now perform constraint solving by search.

While all of our results carry over to the extended set of CHRs, it is now much more difficult to ensure termination. The addition of rule (CloseEq) makes the above set of CHRs non-terminating. We follow [29] by disallowing recursive dependencies for "closed" definitions. This requirement is sufficient to ensure termination of CHRs involving closed definitions.

8.2 Overlapping Definitions

Recall Example 8 from Section 4.3. Although, the program theory is confluent, we can't provide a coherent translation because we must take an indeterministic choice in case we require eq on type $[Int] \rightarrow [Int] \rightarrow Bool$.

We can rely on yet another extension of the CHR framework to resolve the above ambiguity. Assume that by default we always want to choose the more specific definition. This can be modeled by incorporating guards constraint into simplification rules. The guard constraint, $a \neq Int$, added to rule (Eq2), states that this rule only fires if the instance type is different from Int.

$$
\begin{array}{rrcl}
\text{(Eq1)} & Eq\ (Int \to Int \to Bool) & \Longleftrightarrow & True \\
\text{(Eq2')} & Eq\ ([a] \to [a] \to Bool)\ \mid\ a \neq Int & \Longleftrightarrow & \\
 & Eq\ (a \to a \to Bool) & & \\
\text{(Eq3)} & Eq\ ([Int] \to [Int] \to Bool) & \Longleftrightarrow & True
\end{array}
$$

To make the example more interesting we assume that additionally a definition of eq on type $Char$ is available. We only give the resulting simplification rule:

$$\text{(Eq4)} \quad Eq\ (Char \to Char \to Bool) \quad \Longleftrightarrow \quad True$$

We avoid some clumsy type annotations by employing the following propagation rule:

$$\text{(Eq5)} \quad Eq\ (a \to b \to c) \quad \Longrightarrow \quad a = b$$

Type inference for expression

```
g x y = (eq [x] [y]) :: Bool
```

yields the constraint $Eq\ ([a] \to [a] \to Bool)$ where x : a and y : a. Note that no further reduction steps are applicable at this point. We find that

g :: $\forall a.Eq\ ([a] \to [a] \to Bool) \Rightarrow a \to a \to Bool$
```
g x y = (eq [x] [y]) :: Bool
t_g eq x y = (eq [x] [y])
```

where t_g is g's translation. Consider expression

```
exp = (g [1] [2], g ['a','b'] ['a','b'])
```

where we use g at two different instantiation sites. In context g [1] [2] we require the constraint $Eq\ ([Int] \to [Int] \to Bool)$ whereas in context g ['a','b'] ['a','b'] we require $Eq\ ([Char] \to [Char] \to Bool)$. We find the following derivations:

$$Eq\ ([Int] \to [Int] \to Bool) \rightarrowtail_{Eq3} True \quad \text{and}$$

$$
\begin{array}{rl}
 & Eq\ ([Char] \to [Char] \to Bool) \\
\rightarrowtail_{Eq2'} & Eq\ (Char \to Char \to Bool) \\
\rightarrowtail_{Eq4} & True
\end{array}
$$

Note that rule (Eq2') fired because $Char \neq Int$. Out of the two derivations above we can calculate which evidence parameters we need to pass to t_g. The translation of exp yields

```
expt = (t_g ec_eqList' [1] [2],
        t_g (ec_eqList ec_eqChar) ['a','b'] ['a','b'])
```

where ec_eqList' represents evidence for $Eq\ ([Int] \to [Int] \to Bool)$ and ec_eqChar represents evidence for $Eq\ (Char \to Char \to Bool)$ and ec_eqList is defined as in Section 6.

We make the following observations. Overlapping definitions have no impact on typability as long as the program theory is still confluent. Note that $Eq\ ([a] \to [a] \to Bool)$ and $Eq\ (a \to a \to Bool)$ are both equivalent w.r.t. the program theory represented by rules (Eq1-5). If the set P_s of simplification rules is overlapping, we can try to employ guard constraints to obtain a non-overlapping set P'_s. For example, in the above example we added a guard constraint to rule (Eq2) yielding rule (Eq2'). Then, we simply re-run all CHR derivations under $P' = P'_s \cup P_p$ to compute the proper evidence values. This allows us to provide type inference and a coherent semantics even in case of overlapping definitions.

8.3 Simplifying Constraints

In Haskell it is common to "simplify" constraints before presenting them to the user. One simple form of simplification is unification. That is, no explicit equality constraints are allowed to appear in constraints. Another form is to omit "redundant" constraints. Assume we have specified a super-class relation ship among eq and leq:

$$\text{(Super)} \quad Leq\ (a \rightarrow a \rightarrow Bool) \quad \Longrightarrow \quad Eq\ (a \rightarrow a \rightarrow Bool)$$

For details of overloaded definitions we refer to Example 6 in Section 4.3. Consider the expression

$$\begin{array}{l} \mathsf{h} :: \forall a. Eq\ (a \rightarrow a \rightarrow Bool) \wedge Leq\ (a \rightarrow a \rightarrow Bool) \Rightarrow \\ \quad a \rightarrow a \rightarrow (Bool, Bool) \\ \mathsf{h\ x\ y} = (\mathsf{eq\ x\ y},\ \mathsf{leq\ x\ y}) \end{array}$$

Note that

$$\begin{array}{lll} [\![P]\!] & \models & (Eq\ (a \rightarrow a \rightarrow Bool) \wedge Leq\ (a \rightarrow a \rightarrow Bool)) \leftrightarrow \\ & & Leq\ (a \rightarrow a \rightarrow Bool) \end{array}$$

where P consists of (Eq1-2), (Leq1-2) and (Super). Therefore, we could assign to expression h the equivalent but simpler type scheme $\forall a. Leq\ (a \rightarrow a \rightarrow Bool) \Rightarrow a \rightarrow a \rightarrow (Bool, Bool)$.

Such form of "simplification" can always be achieved by turning a rule such as (Super) into a *multi-headed* simplification rule of the form

$$\begin{array}{rl} Leq\ (a \rightarrow a \rightarrow Bool), Eq\ (a \rightarrow a \rightarrow Bool) & \Longleftrightarrow \\ Leq\ (a \rightarrow a \rightarrow Bool) & \end{array}$$

9 Discussion

Our approach is clearly inspired by Haskell style type classes and its various extensions. In contrast to previous work [19, 20, 17], we are seeking a general formal framework within we can reason about overloading in a concise way. CHRs turn out to be the perfect candidate. We have established some precise conditions in terms of CHRs under which we achieve decidable type inference and the meaning of programs is unambiguous.

The framework presented here can be seen as a formal basis for the ideas described by Jones [18]. He introduces the concept of improving and simplifying constraints by example whereas our work provides an actual proof system based on CHRs.

Shields and Peyton-Jones [29] give an extensive discussion of various possible extensions to Haskell style overloading. Their main motivation is to investigate which extensions are necessary to incorporate object-oriented classes into Haskell. In particular, they also discuss issues involving closed and overlapping definitions. As we have seen in Section 8, CHRs are able to cope with such additional features.

The work presented here shares ideas with the recent work by Neubauer, Thiemann, Gasbichler and Sperber [10]. Both works can be seen as a consequent refinement of the HM(X) framework by incorporating an actual programming language on the type-level. Whereas we employ CHRs, Neubauer *et al.* employ a functional-logic language [12]. The expressiveness of both system seems to be equivalent in power. One of the main differences is that we require confluence of CHRs whereas Neubauer *et al.* allow for a customizability of evaluation strategies.

Similarly to our proposal, Odersky, Wadler and Wehr [26] proposed a variation of overloading, named System O, where no class hierarchies are imposed on overloaded identifiers. Their motivation was mainly to provide an untyped semantics for overloading. This clearly results in a less expressive system.

Camarao and Figueiredo [4] considered an extension of System O which is close to our proposal. Their system seems to be even more liberal by allowing for "local" overloading. Overloaded identifiers can be defined via ordinary let-definitions at any arbitrary level. By default their system codes a closed world assumption. We suspect that this must put decidable type inference in danger in the presence of closed recursive definitions (see Section 8.1).

Implicit parameters [22] introduced by Lewis, Shields, Meijer and Launchbury are a complement to Haskell style overloading. Implicit parameters can be seen as a "mild" form of local overloading while still retaining decidable type inference and coherence. We are currently investigating how to incorporate local overloading into the present approach. This should provide then a unifying framework within we can study implicit parameters and Haskell style overloading.

It is also worth mentioning the work by Yang [35]. He shows how to express type-indexed values in languages based on the Hindley/Milner system. This is clearly related to overloading, though we yet have to work out the exact connections between his work and ours.

10 Conclusion

It is folklore knowledge that via Haskell's type class system it is possible to encode logic programs on the level of types. However, there hasn't been any proposal so far to make this connection concrete.

In this paper, we proposed a general overloading framework based on CHRs. CHRs serve as a meta-language to describe relations among overloaded identifiers. We can describe precisely in terms of CHRs under which conditions type inference is decidable (Section 5 and the semantics of programs is well-defined (Section 6). We believe that the range-restrictedness condition can be lifted as long as variables in the body of CHRs functionally depend on variables in the head. Due to space limitations we could only sketch the coherence result. For a detailed discussion we refer to [30].

The design space for overloading systems is huge. In Section 8 we elaborate on some possible variations. We refer to [32] for more examples on how our CHR-based overloading system helps to improve previous approaches. We are also in the process of implementing the CHR-based type inference engine for a Haskell like language including a translation scheme [31].

References

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

[2] S. Abdennadher and H. Schütz. $\text{CHR}^{\vee}$: A flexible query language. In *Proc. of Flexible Query Answering Systems, LNAI* 1495, pages 1–14. Springer, 1998.

[3] Lennart Augustsson. Cayenne - a language with dependent types. In *Proc. of ICFP'98*, pages 239–250, 1998.

[4] C. Camarao and L. Figueiredo. Type inference for overloading without restrictions, declarations or annotations. In *Proc. of the 4th Fuji International Symposium on Functional and Logic Programming, LNCS* 1722, pages 37–52. Springer, 1999.

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

[6] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In *Proc. of CP'99*, LNCS 1713, pages 174–188. Springer-Verlag, 1999.

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

[8] T. Frühwirth. A declarative language for constraint systems: Theory and practice of constraint handling rules, 1998. Habilitation.

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

[10] M. Gasbichler, M. Neubauer, M. Sperber, and P. Thiemann. Functional logic overloading. In *Proc. of POPL'02*, pages 233–244, 2002.

[11] K. Glynn, P. Stuckey, and M. Sulzmann. Type classes and constraint handling rules. First Workshop on Rule-Based Constraint Reasoning and Programming, July 2000.

[12] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[13] The Mercury language reference manual, 2001. http://www.cs.mu.oz.au/research/mercury/.

[14] D. Jeffery, F. Henderson, and Z. Somogyi. Type classes in Mercury. In *Proc. 23rd Australasian Computer Science Conf.*, pages 128–135. IEEE Press, 2000.

[15] M. P. Jones. *Qualified Types: Theory and Practice.* D.phil. thesis, Oxford University, September 1992.

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

[17] M. P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *FPCA '93*, pages 52–61. ACM Press, 1993.

[18] M. P. Jones. Simplifying and improving qualified types. In *FPCA '95*, pages 160–169. ACM Press, 1995.

[19] M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, pages 230–234 *LNCS* 1782. Springer-Verlag, 2000.

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

[21] S. Kaes. Parametric overloading in polymorphic programming languages. In *Proc. of ESOP'88*, *LNCS* 300, pages 131–141. Springer-Verlag, 1988.

[22] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Proc. of POPL'00*, pages 108–118, 2000.

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

[24] T. Nipkow and C. Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[25] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[26] M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *Proc. of FPCA'95*, pages 135–146, , ACM Press. 1995.

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

[28] M.J. Plasmeijer and M.C.J.D. van Eekelen. Language report Concurrent Clean. Tech. Report CSI-R9816, University of Nijmegen, June 1998. ftp://ftp.cs.kun.nl/pub/Clean/Clean13/doc/refman13.ps.gz.

[29] M. Shields and S. Peyton Jones. Object-oriented overloading for Haskell. In *Workshop on Multi-Language Infrastructure and Interoperability*, September 2001.

[30] M. Sulzmann and A. Rossberg. A theory of overloading part II: semantics and coherence. Technical report 2002/1, The University of Melbourne, 2002. `www.cs.mu.oz.au/~sulzmann/chr/`.

[31] M. Sulzmann, A. Rossberg and J. Wazny. Tie: Type inference engine based on CHRs. `http://www.cs.mu.oz.au/~sulzmann/chr/software.html`.

[32] M. Sulzmann and A. Rossberg. Beyond type classes. Working paper. `www.cs.mu.oz.au/~sulzmann/chr/`.

[33] M. Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints.* PhD thesis, Yale University, Department of Computer Science, May 2000.

[34] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. of POPL'89*, pages 60–76, 1989.

[35] Z. Yang. Encoding types in ML-like languages. In *Proc. of ICFP'98*, pages 289–300, 1998.

A Constraint Handling Rules

Individual rule application steps are formalized below. Each constraint C is split into a set of user-defined constraints C_u and a set of equations C_e, i.e. $C = C_u \cup C_e$. Variables in CHR rules r are renamed before rule application. Note that we allow for guarded simplification rules $\bar{c} \mid g \Longleftrightarrow \bar{d}$ where the guard constraints g is a conjunction of disequality constraints.

$$\begin{array}{ll}
\text{(Solve)} & C_u \cup C_e \rightarrowtail_P \phi C_u \cup C'_e \\
& \text{if } \models C'_e \leftrightarrow C_e \text{ and } \phi \text{ mgu of } C_e \\
\\
\text{(Simp)} & C_u \cup C_e \rightarrowtail_P (C_u - \bar{c}') \cup \theta(\bar{d}) \cup C_e \\
& \text{if } \bar{c} \mid g \Longleftrightarrow \bar{d} \in P \text{ and there exists } \bar{c}' \in C_u \\
& \text{and a substitution } \theta \text{ on variables in } r \\
& \text{such that } \theta(\bar{c}') \equiv \bar{c} \text{ and } \models C_e \supset \theta(g) \\
\\
\text{(Prop)} & C_u \cup C_e \rightarrowtail_P C_u \cup C_e \cup \theta(\bar{d}) \\
& \text{if } \bar{c} \Longrightarrow \bar{d} \in P \text{ and there exists a subset } \bar{c}' \subseteq C_u \\
& \text{and a substitution } \theta \text{ on variables in } r \\
& \text{such that } \theta(\bar{c}') \equiv \bar{c}
\end{array}$$

The observant reader will notice that we have to prevent the infinite application of CHR propagation rules. We refer to [1] for more details.

We find the following result, see [9] for details.

Theorem 4 (Soundness) *Let P be a CHR program and C, C' constraints such that $C \rightarrowtail_P^* C'$. Then, $[\![P]\!] \models C \leftrightarrow \bar{\exists}_{fv(C)} C'$.*

B Type Inference

B.1 Essential lemmas

An important property of range-restricted CHRs is that confluence for this class of CHRs, enforces a stronger property than the general case. Given $C_0 \rightarrowtail_P C_1$ and $C_0 \rightarrowtail_P C_2$, then there exist derivations $C_1 \rightarrowtail_P^* C_3$ and $C_2 \rightarrowtail_P^* C_4$ such that C_3 is equivalent (modulo new variables introduced) to C_4, i.e. $\models (\bar{\exists}_{fv(C_0)} C_3) \leftrightarrow (\bar{\exists}_{fv(C_0)} C_4)$. But any substitution ϕ that grounds variable in $fv(C_0)$ also grounds any variables in $\theta(C_3)$ where θ_3 is an mgu of h_{C_3}, and similarly for C_4. This means that $\phi(\theta_3(C_3))$ and $\phi(\theta_4(C_4))$ must be the same set of ground user-defined constraints.

Lemma 7 (Model Extension) *Let P be a confluent set of range-restricted CHRs where each simplification rule is single-headed. Suppose $D \rightarrowtail_P^* C$ where $\models \tilde{\exists} h_C$. Let $\theta = \theta_1 \cdot \theta_0$ be a substitution where θ_0 is an mgu of h_C, and θ_1 is a substitution mapping every variable to a distinct new type constant. Then, we can construct a model M of $\theta(C)$ and $[\![P]\!]$ such that for each subset $S \subseteq M$ we have that $S \rightarrowtail_P^* S'$ where $S' \subseteq \theta(C)$.*

Proof: Note that by the definition of θ, no further derivation steps are applicable to $\theta(C)$ since none were applicable to C and the effect of θ does not cause any variables that were not equal to be equal, nor any constraint $\theta(C)$ to now match the head of a rule it did not previously match.

We construct a model M, defined by a set of ground CHR constraints, of the program P and $\theta(C)$ iteratively as follows:

We iteratively construct a model M of the program P and $\theta(C)$, defined as a set of ground user-defined constraints, as follows:

Initially $M = \theta(a_C)$.

We augment M using the following rules.

- If there exists a rule $h \Longleftrightarrow d_1, \ldots, d_m$ and ground substitution ϕ such that $\phi(d_i) \in M$ if d_i is a user-defined constraint and $\phi(d_j)$ is true if d_j is an equation then add $\phi(h)$ to M.
- If there exists a rule $h \Longleftrightarrow d_1, \ldots, d_m$ and ground substitution ϕ such that $\phi(h) \in M$ and $\phi(d_j)$ is true if d_j is an equation then add the user-defined constraints $\phi(d_i)$ to M.
- If there exists a rule $h_1, \ldots, h_n \Longrightarrow d_1, \ldots, d_m$ and ground substitution ϕ such that $\{\phi(h_1), \ldots, \phi(h_n)\} \subseteq M$ and $\phi(d_j)$ is true for equations d_j then add the user-defined constraints $\phi(d_i)$ to M.

We prove by induction (on k such that $S \subseteq M_k$ where M_k is the kth iteration in the construction of M) that for any subset $S \subseteq M$ there is a derivation $S \rightarrowtail_P^* S'$ where $S' \subseteq \theta(a_C)$. Clearly the base case holds for $S \subseteq \theta(a_C)$.

Suppose $\phi(h)$ was introduced using the first rule above. Then $\phi(h) \rightarrowtail_P \phi(d_1) \wedge \cdots \wedge \phi(d_m)$ and by induction we have a derivation from $\phi(d_1) \wedge \cdots \wedge \phi(d_m) \rightarrowtail_P^* S'$ where $S' \subseteq \theta(a_C)$.

Suppose $\phi(d_i)$ was introduced by the second rule above. Then there exists $\phi(h) \in M$ such that by induction $\phi(h) \rightarrowtail_P^* S'$ where $S' \subseteq \theta(a_C)$. Hence, by the confluence of P it must be that $\phi(d_1) \wedge \cdots \wedge \phi(d_m) \rightarrowtail_P^* S'$. (since the rules are range-restricted the derivations must end identically). Clearly then if d_j is an equation we have that $\models \phi(d_j)$ since otherwise this derivation is impossible. Now consider the user-defined constraints

$\phi(d_i)$. Since the simplification rules are all single headed, for each $\phi(d_i) \notin \theta(a_C)$ it must be the case that it is rewritten using a simplification rule, eventually to a set of facts in $\theta(a_C)$. Using this sub-derivation we have that for each $\phi(d_i) \rightarrowtail_P^* S_i'$ where $S_i' \subseteq \theta(a_C)$.

Suppose $\phi(d_i)$ was introduced by the third rule above, then there exist $\{\phi(h_1), \ldots, \phi(h_n)\} \subseteq M$ such that by induction $\phi(h_1) \wedge \cdots \wedge \phi(h_n) \rightarrowtail_P^* S'$ where $S' \subseteq \theta(a_C)$. Hence, by confluence of P it must be that $\phi(h_1) \wedge \cdots \wedge \phi(h_n) \wedge \phi(d_1) \wedge \ldots \wedge \phi(d_m) \rightarrowtail_P^* S'$. Again using the same argument as the previous case we have that each $\phi(d_i) \rightarrowtail_P^* S_i'$ where $S_i' \subseteq \theta(a_C)$. $\square$

Lemma 2 (Canonical Form) *Let P be a confluent set of range-restricted CHRs where each simplification rule is single-headed. Then $[\![P]\!] \models D \leftrightarrow D'$ iff $D \rightarrowtail_P^* C$ and $D' \rightarrowtail_P^* C'$ such that $\models (\bar{\exists}_V C) \leftrightarrow (\bar{\exists}_V C')$, where $V = fv(D) \cup fv(D')$.*

Proof: We assume that variables in C and C' not in V are renamed apart.

" $\Rightarrow$ ": We proceed in two steps. First, we show that $\bar{\exists}_V h_C$ and $\bar{\exists}_V h_{C'}$ are either equivalent or both unsatisfiable. Assuming that $\bar{\exists}_V h_C$ and $\bar{\exists}_V h_{C'}$ are equivalent, we show that $\theta(a_C)$ and $\theta(a_{C'})$ are syntactically equivalent for any mgu θ of $h_C \wedge h_{C'}$.

(1) Since $D \rightarrowtail_P^* C$ implies that $[\![P]\!] \models D \leftrightarrow (\bar{\exists}_V C)$ we have $[\![P]\!] \models (\bar{\exists}_V C) \leftrightarrow (\bar{\exists}_V C')$.

We show that if the Herbrand constraints h_C of C are satisfiable, then $\models (\bar{\exists}_V h_C) \supset (\bar{\exists}_V h_{C'})$.

Let θ_0 be an mgu of the Herbrand constraints h_C in C. Let θ_1 be a valuation that maps each type variable in $\theta_0(V)$ to a distinct new type constant not appearing in the program. Let $\theta = \theta_1 \cdot \theta_0$. Note that since θ grounds V is also grounds all variables in C and $\theta'(C')$ where θ' is an mgu of $h_{C'}$.

By Lemma 7 we can construct a model M of $\theta(C)$ and $[\![P]\!]$. But, now $M \models (\bar{\exists}_V C) \supset (\bar{\exists}_V C')$ since M models $[\![P]\!]$ and $[\![P]\!] \models (\bar{\exists}_V C) \leftrightarrow (\bar{\exists}_V C')$.

But by construction $M \models \theta(C)$ and hence $M \models \bar{\exists}_V \theta(C')$ Now M is irrelevant to the equational constraints of C' in other words we have $\models (\bar{\exists}_V h_{C'})$. The new constants of θ_1 take the role of Skölem constants, hence we have shown that any solution of h_C implies $\bar{\exists}_V h_{C'}$. That is $\models (\bar{\exists}_V h_C) \supset (\bar{\exists}_V h_{C'})$.

Similarly for C'. Hence either the Herbrand constraints of C and C' are both unsatisfiable in which case we are done, or they are equivalent, i.e. $\models (\bar{\exists}_V h_C) \leftrightarrow (\bar{\exists}_V h_{C'})$.

(2) We assume $\models (\bar{\exists}_V h_C) \leftrightarrow (\bar{\exists}_V h_{C'})$. We show that $\theta(a_C)$ and $\theta(a_{C'})$ are syntactically equivalent where $\theta = \theta_1 \cdot \theta_0$, θ_0 is an mgu of $h_C \wedge h_{C'}$ and θ_1 a substitution mapping each type variable in $\theta_0(V)$ to a distinct new type constant.

Note that $\theta(C)$ and $\theta(C')$ are ground by construction.

Note that since $\bar{\exists}_V h_C$ and $\bar{\exists}_V h_{C'}$ are equivalent then no further derivation steps are applicable to $\theta(C)$ since none were applicable to C and the new values for variables given by θ do not cause any variables that were not equal to be equal, not any constraint $\theta(C)$ to now match the head of a rule it did not previously match. Similarly for C'.

Using Lemma 7 we construct a model M of $[\![P]\!]$ and $\theta(C)$ such that for each subset $S \subseteq M$ we have that $S \rightarrowtail_P^* S'$ where $S' \subseteq \theta(C)$. By definition $M \models (\bar{\exists}_V C) \leftrightarrow (\bar{\exists}_V C')$, and hence $M \models \theta(C) \leftrightarrow \theta(C')$, and hence $M \models \theta(C')$ and thus $\theta(a_{C'}) \subseteq M$.

But then $\theta(a_{C'}) \rightarrowtail_P^* S'$ where $S' \subseteq \theta(a_C)$. But $\theta(a_{C'})$ has no further derivation steps possible hence $\theta(a_{C'}) \subseteq \theta(a_C)$. The same argument applies starting from $\theta(C')$ hence $\theta(a_C) = \theta(a_{C'})$.

We conclude that $\models (\bar{\exists}_V C) \leftrightarrow (\bar{\exists}_V C')$.

" $\Leftarrow$ ": The CHR soundness result yields $[\![P]\!] \models D \leftrightarrow (\bar{\exists}_V C)$ and $[\![P]\!] \models D' \leftrightarrow (\bar{\exists}_V C')$. We immediately find that $[\![P]\!] \models D \leftrightarrow D'$. $\square$

B.2 Satisfiability

Lemma 1 (Weak Satisfiability) *Let P be a confluent set of range-restricted CHRs where each simplification rule is single-headed. Let C be a constraint and suppose $C \rightarrowtail_P^* C'$. Then $\models \tilde{\exists}([\![P]\!] \wedge C)$ iff $\models \tilde{\exists} h_{C'}$.*

Proof: " $\Rightarrow$ ": Now $[\![P]\!] \models C \leftrightarrow \bar{\exists}_{fv(C)} C'$, hence if $\models \neg\tilde{\exists} h_{C'}$ we have that $[\![P]\!] \models \neg\tilde{\exists} C$.

" $\Leftarrow$ ": We apply Lemma 7 to build a model M of $[\![P]\!]$ and $\theta(C')$. Since $[\![P]\!] \models C \leftrightarrow \bar{\exists}_{fv(C)} C'$, clearly $M \models \tilde{\exists} C$. $\square$

B.3 Ambiguity

Lemma 3 (Soundness of Unambiguity) *Let P be a set of CHRs, $\forall \bar{\alpha}.C \Rightarrow \tau$ be a type scheme and ρ be a variable renaming on $\bar{\alpha}$ such that $C \wedge \rho(C) \wedge \tau = \rho(\tau) \rightarrowtail_P^* C'$ where $\models C' \supset (\alpha = \rho(\alpha))$ for each $\alpha \in \bar{\alpha}$. Then for each $\alpha \in \bar{\alpha}$ $[\![P]\!] \models (C \wedge \rho(C) \wedge (\tau = \rho(\tau))) \supset (\alpha = \rho(\alpha))$.*

Proof: Follows immediately from soundness of CHR derivations. $\square$

Lemma 5 (Completeness of Unambiguity) *Let P be a confluent set of range-restricted CHRs where each simplification rule is single-headed, $\forall \bar{\alpha}.C \Rightarrow \tau$ be a type scheme and ρ be a variable renaming on $\bar{\alpha}$ such that $[\![P]\!] \models (C \wedge \rho(C) \wedge (\tau = \rho(\tau))) \supset (\alpha = \rho(\alpha))$ for each $\alpha \in \bar{\alpha}$. Then $C \wedge \rho(C) \wedge \tau = \rho(\tau) \rightarrowtail_P^* C'$ where $\models C' \supset (\alpha = \rho(\alpha))$ for each $\alpha \in \bar{\alpha}$.*

Proof: Suppose to the contrary $[\![P]\!] \models C \wedge \rho(C) \wedge \tau = \rho(\tau) \supset \alpha = \rho(\alpha)$ but $\not\models C' \supset \alpha = \rho(\alpha)$ or equivalently $C' \not\models \alpha = \rho(\alpha)$. Let V be the free variables in $C \wedge \rho(C) \wedge \tau = \rho(\tau)$.

Let θ_0 be the mgu of the Herbrand constraints $h_{C'}$ in C'. Let θ_1 be a valuation that maps each type variable in $\theta_0(C')$ to a distinct new type symbol not appearing in the program. Let $\theta = \theta_1 \cdot \theta_0$. By definition $\theta(\alpha) \neq \theta(\rho(\alpha))$.

By Lemma 7 we can construct a model M of $\theta(C')$ and $[\![P]\!]$. Now, since $[\![P]\!] \models (C \wedge \rho(C) \wedge \tau = \rho(\tau)) \leftrightarrow \bar{\exists}_V C'$ we have that $M \models \theta(C \wedge \rho(C) \wedge \tau = \rho(\tau))$ but $\theta(\alpha) \neq \theta(\rho(\alpha))$ and we have a contradiction. $\square$

B.4 Entailment

We assume without loss of generality that the all type schemes have a type variable on the r.h.s. of $\Rightarrow$. We can convert any type scheme to this form since $\forall \bar{\alpha}.C \Rightarrow \tau$ is equivalent to $\forall \bar{\alpha}.C \wedge \tau = \alpha \Rightarrow \alpha$ where α is new.

Lemma 4 (Soundness of Entailment) *Let P be a set of CHRs and $\sigma = \forall\bar{\alpha}.C \Rightarrow \tau$ and $\sigma' = \forall\bar{\alpha}'.C' \Rightarrow \tau'$ (by assumption τ and τ' are type variables) where $C' \wedge \tau = \tau' \rightarrowtail^*_P C_1$ and $C' \wedge \tau = \tau' \wedge C \rightarrowtail^*_P C_2$ such that $\models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2)$, where $V = fv(\sigma) \cup fv(\sigma') \cup fv(C' \wedge \tau = \tau')$. Then $[\![P]\!] \vdash \sigma \preceq \sigma'$.*

Proof: W.l.o.g. we assume that $\bar{\alpha} \cap \bar{\alpha}' = \emptyset$.

We continue by applying the CHR soundness result to our assumptions. We find that

$$\begin{array}{lll} [\![P]\!] & \models & (C' \wedge \tau = \tau') \leftrightarrow (\bar{\exists}_V C_1) \\ [\![P]\!] & \models & (C' \wedge \tau = \tau' \wedge C) \leftrightarrow (\bar{\exists}_{V \cup \{\bar{\alpha}\}} C_2) \end{array}$$

By assumption $\models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2)$. We conclude that

$$[\![P]\!] \quad \models \quad (C' \wedge \tau = \tau') \leftrightarrow (\bar{\exists}_V (C' \wedge \tau = \tau' \wedge C)) \tag{1}$$

We apply the following reasoning

$$\begin{array}{lll} [\![P]\!] & \models & (\bar{\exists}_V (C' \wedge \tau = \tau' \wedge C) \\ & & \text{(by weakening)} \\ & \supset & (\bar{\exists}_V (C \wedge \tau = \tau')) \\ & \supset & \exists\bar{\alpha}.(C \wedge \tau = \tau') \end{array} \tag{2}$$

From 1 and 2 we conclude that

$$\begin{array}{lll} [\![P]\!] & \models & (C' \wedge \tau = \tau')) \supset \\ & & \exists\bar{\alpha}.(C \wedge \tau = \tau') \end{array}$$

Let ρ be the renaming $[\tau'/\tau]$ (note that w.l.o.g. τ and τ' are variables). Then applying ρ to both sides, clearly

$$[\![P]\!] \models C' \supset \exists\bar{\alpha}.\rho(C)$$

and finally

$$[\![P]\!] \models C' \supset \exists\bar{\alpha}.(C \wedge \tau = \tau')$$

Thus we conclude that $[\![P]\!] \vdash \sigma \preceq \sigma'$. □

Note that we interpret substitutions as equality constraints, so ϕ is equivalent to $con(\phi)$ defined as $\wedge_{\alpha \in fv_\phi} \alpha = \phi(\alpha)$. Note that $\models \phi(con(\phi))$. Then, the $\leq$ relation among substitutions can be expressed as constraint entailment, and $\sqcup$ corresponds to conjunction. We say a substation ϕ' is an *extension* of ϕ, written $\phi \leq \phi'$ if $\models con(\phi') \supset con(\phi)$. We denote by $\phi_{|\bar{\alpha}}$ the substitution resulting from ϕ where $\models con(\phi_{|\bar{\alpha}}) \leftrightarrow (\bar{\exists}_{\bar{\alpha}} con(\phi))$. Similarly, we denote by $\phi_{\backslash\bar{\alpha}}$ the substitution such that $\models con(\phi_{\backslash\bar{\alpha}}) \leftrightarrow \exists\bar{\alpha}.con(\phi)$.

Lemma 9 (Canonical Extension) *Let $\forall\bar{\alpha}.C \Rightarrow \tau$ be an unambiguous type scheme, where τ is a type variable, P be a confluent and range-restricted set of CHRs whose simplification rules are single-headed, C' be a constraint and τ' be a type variable such that $[\![P]\!] \models (C' \wedge \tau = \tau') \leftrightarrow (\exists\bar{\beta}.C' \wedge \tau = \tau' \wedge C)$ where $\bar{\beta} = \bar{\alpha} \backslash \{\tau\}$. Then, for any substitution ϕ grounding $C' \wedge \tau = \tau'$ and any model M of $[\![P]\!]$ there exists a unique extension $\phi \leq \phi'$ such that $M \models \phi C' \wedge \tau = \tau') \leftrightarrow \phi'(C' \wedge \tau = \tau' \wedge C)$.*

Proof: W.l.o.g. we assume that $\bar{\alpha} = fv(C, \tau)$.

From $[\![P]\!] \models (C' \wedge \tau = \tau') \leftrightarrow (\exists\bar{\beta}.C' \wedge \tau = \tau' \wedge C)$ we conclude that for any model M of $[\![P]\!]$ there exists an extension $\phi \leq \phi'$ such that $M \models \phi(C' \wedge \tau = \tau') \leftrightarrow \phi'(C' \wedge \tau = \tau' \wedge C)$.

Assume M is a model of $[\![P]\!]$ and we have two different extensions $\phi \leq \phi'$ and $\phi \leq \phi''$. Hence,

$$M \models (\phi(C') \wedge \phi'(C) \wedge \phi(\tau = \tau')) \leftrightarrow (\phi(C') \wedge \phi''(C) \wedge \phi(\tau = \tau')) \tag{3}$$

The type scheme $\forall \bar{\alpha}.C \Rightarrow \tau$ is unambiguous by assumption. Therefore,

$$[\![P]\!] \models (C \wedge \rho(C) \wedge \tau = \rho(\tau)) \supset \alpha = \rho(\alpha) \tag{4}$$

where $\alpha \in fv(C, \tau)$ and ρ is a renaming on $\bar{\alpha}$.

From the above, we conclude

$$[\![P]\!] \models (C \wedge \phi'(C) \wedge con(\phi')) \leftrightarrow (C \wedge \phi'(C) \wedge con(\phi))$$

Note that the same holds for ϕ''. We apply substitution ϕ to this formula (note $\models \phi(con(\phi')) \leftrightarrow con(\phi'_{|\bar{\beta}})$) and find

$$[\![P]\!] \models (\phi(C) \wedge \phi'(C) \wedge con(\phi'_{|\bar{\beta}})) \leftrightarrow (\phi(C) \wedge \phi'(C)) \tag{5}$$

From 3 and 5 we conclude

$$\begin{array}{lll} M & \models & (\phi(C') \wedge \phi(C) \wedge \phi''(C) \wedge \phi(\tau = \tau') \wedge con(\phi''_{|\bar{\beta}})) \\ & & \quad\quad\quad\quad \leftrightarrow \\ & & (\phi(C') \wedge \phi(C) \wedge \phi'(C) \wedge \phi(\tau = \tau') \wedge con(\phi'_{|\bar{\beta}})) \end{array}$$

Immediately, we conclude that $\models con(\phi''_{|\bar{\beta}}) \leftrightarrow con(\phi'_{|\bar{\beta}})$. Therefore, the extension must be unique. $\square$

Lemma 6 (Completeness of Entailment) *Let P be a terminating confluent set of range-restricted CHRs whose simplification rules are single-headed, $\sigma = \forall \bar{\alpha}.C \Rightarrow \tau$ and $\sigma' = \forall \bar{\alpha}'.C' \Rightarrow \tau'$ where w.l.o.g. τ and τ' are type variables such that $[\![P]\!] \vdash \sigma \preceq \sigma'$ and σ is unambiguous. Then $C' \wedge \tau = \tau' \rightarrowtail_P^* C_1$ and $C' \wedge \tau = \tau' \wedge C \rightarrowtail_P^* C_2$ such that $\models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2)$, where $V = fv(\sigma) \cup fv(\sigma') \cup fv(C' \wedge \tau = \tau')$.*

Proof: Note that $[\![P]\!] \vdash \sigma \preceq \sigma'$ implies

$$[\![P]\!] \models C' \supset (\exists \bar{\alpha}.C \wedge \tau = \tau') \tag{6}$$

We obtain that

$$[\![P]\!] \models (C' \wedge \tau = \tau') \leftrightarrow (\exists \bar{\beta}.C' \wedge C \wedge \tau = \tau') \tag{7}$$

where $\bar{\beta} = fv(C) \backslash fv(\tau)$. The direction from right to left follows immediately. Otherwise, assume $M \models [\![P]\!]$ and $M \models \phi(C' \wedge \tau = \tau')$ for some model M and ground (on $fv(C' \wedge \tau = \tau' \wedge C) \backslash \bar{\beta}$) substitution ϕ. From 6 we follow that $M \models \phi(\exists \bar{\alpha}.C \wedge \tau = \tau')$. That is, there exists $\phi_{\backslash fv(\tau)} \leq \phi'$ such that $M \models \phi'(C)$ and $\phi'\tau \equiv \phi'\tau' \equiv \phi\tau'$. Note that $\phi\tau \equiv \phi\tau'$, therefore $\phi'\tau \equiv \phi\tau$. Hence, $\phi'_{|fv(\tau)} = \phi_{|fv(\tau)}$. We conclude that $\phi \leq \phi'$ such that $M \models \phi'(C' \wedge C \wedge \tau = \tau')$ which shows the direction from left to right.

Note that $\models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2)$ iff (1) $\models (\bar{\exists}_V h_{C_1}) \leftrightarrow (\bar{\exists}_V h_{C_2})$, and (2) $\phi_1(a_{C_1}) = \rho \cdot \phi_2(a_{C_2})$ where ϕ_1 mgu of h_{C_1}, ϕ_2 mgu of h_{C_2} and ρ is a variable renaming such that $\rho_{|V} = id$.

(1) We show that $\models (\bar{\exists}_V h_{C_1}) \supset (\bar{\exists}_V h_{C_2})$ (the other direction follows easily):
Assume the contrary, i.e. there exists ϕ such that $\models \phi(h_{C_1})$ and $\phi(h_{C_2})$ is unsatisfiable. W.l.o.g. ϕ grounds C', τ, τ'.

From 7 we get

$$[\![P]\!] \models (\phi(C') \wedge \phi(\tau = \tau')) \leftrightarrow (\phi'(C') \wedge \phi'(C) \wedge \phi'(\tau = \tau')) \tag{8}$$

where ϕ' is an appropriate extension of ϕ grounding variables $\bar{\beta}$.

From our assumptions and soundness of CHR solving we get

$$[\![P]\!] \models (\phi(C') \wedge \phi(\tau = \tau')) \leftrightarrow \phi(C_1)$$

$$[\![P]\!] \models (\phi'(C') \wedge \phi'(C) \wedge \phi'(\tau = \tau')) \leftrightarrow \phi'(C_2)$$

Note that range-restrictedness is a necessary condition for the above statement.

From the above and 8 we get

$$[\![P]\!] \models \phi(C_1) \leftrightarrow \phi'(C_2)$$

which contradicts our assumptions.

Therefore, $\models (\bar{\exists}_V h_{C_1}) \supset (\bar{\exists}_V h_{C_2})$.

(2) We show that $\phi_1(a_{C_1}) = \rho \cdot \phi_2(a_{C_2})$:

Let $\theta = \theta_1 \cdot \theta_0$ where θ_0 is an mgu of $h_C \wedge h_{C'}$ and θ_1 a substitution mapping each type variable in $\theta_0(V)$ to a distinct new type constant. Note that no further derivation steps are applicable to $\theta(C_1)$ and $\theta(C_2)$.

Using the Model Extension Lemma we can construct a model M of $[\![P]\!]$ and $\theta(C_1)$. The proof proceeds in the following two steps.

(2a) We will show that $M \models \theta(a_{C_2})$:

Soundness of CHR solving and range-restrictedness yields

$$[\![P]\!] \models \theta(C' \wedge C \wedge \tau = \tau') \leftrightarrow \theta(C_2) \tag{9}$$

From 7 we can conclude that

$$M \models \theta(C' \wedge \tau = \tau') \leftrightarrow \theta'(C' \wedge C \wedge \tau = \tau') \tag{10}$$

for some extension θ' of θ. Note that θ and θ' might only differ on $\bar{\beta}$. Assume $\theta \neq \theta'$.

We build a model M' as an extension of M such that $M' \models \theta(C_2)$ (construction as in Model Extension Lemma). Together with 9 and 10 we conclude that

$$M' \models \theta'(C' \wedge C \wedge \tau = \tau')$$

$$M' \models \theta(C' \wedge C \wedge \tau = \tau')$$

Note that $M' \models \theta(C' \wedge \tau = \tau')$.

In particular, we find that

$$M' \models \theta(C' \wedge \tau = \tau') \leftrightarrow \theta'(C' \wedge C \wedge \tau = \tau')$$

$$M' \models \theta(C' \wedge \tau = \tau') \leftrightarrow \theta'(C' \wedge C \wedge \tau = \tau')$$

However, this is a contradiction to the Canonical Extension Lemma. Therefore, $\theta = \theta'$ and we find that $M \models \theta(C_2)$.

(2b) We assume that $M \models \theta(a_{C_2})$:

Note that for each subset $S \subseteq M$ we have that $S \rightarrowtail_P^* S'$ where $S' \subseteq \theta(C_1)$.

We have that $\theta(a_{C_2}) \subseteq M$. But then $\theta(a_{C_2}) \rightarrowtail_P^* S'$ where $S' \subseteq \theta(a_{C_2})$. But $\theta(a_{C_2})$ has no further derivation steps possible hence $\theta(a_{C_2}) \subseteq \theta(a_{C_1})$.

Similarly, we can build a model M of $[\![P]\!]$ and $\theta(C_2)$. Applying the same arguments we find that $\theta(a_{C_1}) \subseteq \theta(a_{C_2})$. Hence, $\theta(a_{C_1}) = \theta(a_{C_2})$.

This shows that $\phi_1(a_{C_1}) = \rho \cdot \phi_2(a_{C_2})$ for some appropriate renaming ρ.

$\square$