A Theory of Overloading

Peter J. Stuckey and Martin Sulzmann *
Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
{pjs,sulzmann}@cs.mu.oz.au

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 (CHR). 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 CHR 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.

Categories and Subject Descriptors

D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) languages; D.3.3 [Programming Languages]: Language Constructs and Features—Polymorphism; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

General Terms

Languages, Theory

Keywords

overloading, type classes, type inference, constraints

1 Introduction

The study of overloading, a.k.a. ad-hoc polymorphism, in the context of the Hindley/Milner system [23] dates back to Kais [21], Wadler and Blott [35]. 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 re-investigation 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.

load leq :: Int → Int → Bool
leq = primLeqInt
load leq :: Float → Float → Bool
leq = primLeqFloat
load ins :: ∀a. Leq (a → a → Bool) ⇒ a → a → [a]
ins = let insList [] y = [y]
      in 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 → a → 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 (CHR) [7]. In case of the above example, we find the following set of CHR simplification rules:

\[(Leq1) \quad \text{Leq}(\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}) \iff \text{True} \]
\[(Leq2) \quad \text{Leq}(\text{Float} \rightarrow \text{Float} \rightarrow \text{Bool}) \iff \text{True} \]
\[(Ins1) \quad \text{Ins}([a] \rightarrow a \rightarrow [a]) \iff \text{Leq}(a \rightarrow a \rightarrow \text{Bool}) \]

Rule (Leq1) states that leq is defined on type Int → Int → Bool. A similar property is stated by rule (Leq2). Rule (Ins1) states that ins on type [a] → a → [a] is defined iff leq is defined on type a → a → 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:

\[(Leq3) \quad \text{Leq}(\text{Int} \rightarrow \text{Int} \rightarrow a) \implies a = \text{Bool} \]
\[(Ins2) \quad \text{Ins}([a] \rightarrow b \rightarrow c) \implies b = a, c = [a] \]
For example, rule (Leq3) states that given both input arguments of \text{Int}'s, then the result must be of type \text{Bool}. The logical meaning of \( \iff \) 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 an 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 overloading and closed definitions of overloaded identifiers. Section 9 discusses related work. We conclude in Section 10.

Proofs can be found in an accompanying technical report [30].

2 Preliminaries

We shall be interested in manipulating constraints on types. A type is a variable \( \alpha \) 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 \( c_1 \land \ldots \land c_n \) instead of \( \{ c_1, \ldots, c_n \} \) where \( c_i \) are primitive constraints. We use \text{True} as an abbreviation for the empty constraint which denotes the true formula. We use \text{False} as an abbreviation for \( T_1 = T_2 \) which denotes the unsatisfisfiable 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 \( \theta = [\bar{t}/\bar{a}] \) simultaneously replaces each \( a \) by its corresponding \( t \). A unifier of conjunction of equations \( C \) of the form \( \tau_{i1} = \tau_{i2} \wedge \ldots \wedge \tau_{in} = \tau_{in+1} \) is a substitution \( \theta \) 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 \( \theta \) such that for each other unifier \( \theta' \) of \( C \) there exists substitution \( \rho \) such that \( \theta' = \rho(\theta) \).

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

Let \( f(v) \) 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 \( \exists \alpha_1 \ldots \exists \alpha_n F \). We let \( \exists F \) denote \( \exists_W f(v) F \). Similarly for \( \forall_W F \) and \( \forall F \). We let \( \exists_W F \) denote the formula \( \exists \alpha_1 \ldots \exists \alpha_n F \) where \( \{ \alpha_1, \ldots, \alpha_n \} = f(v) - W \). Note that formulas \( F \) are always implicitly universally quantified. We write \( F_1 \vdash F_2 \) to denote that \( F_2 \) holds in any model of \( F_1 \) where \( F_1 \) and \( F_2 \) are first-order formulae.

A type scheme is of the form \( \forall \alpha. C \Rightarrow \tau \) where \( \alpha \) are the bound variables, \( C \) is a constraint and \( \tau \) a type. Note that we can always view \( \tau \) as \( \forall \alpha. \alpha = \tau \Rightarrow \alpha \) where \( \alpha \) is fresh. We commonly use \( \sigma \) to refer to type schemes. We introduce an ordering among type schemes. We define \( F \vdash \tau_1 \Rightarrow \tau_2 \) iff \( F \vdash C_1 \Rightarrow \exists c \exists \tau_1 \land \tau_2 = \tau_2 \) where we assume there are no name clashes between \( \tau_1 \) and \( \tau_2 \) and \( F \) is a first-order formula. We define \( F \vdash \tau_1 \Rightarrow \tau_2 \) iff \( \tau_1 \Rightarrow \tau_2 \).

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

\[
\begin{align*}
\text{simplification} & \quad \text{(Rule 1)} \quad c_1, \ldots, c_n \iff d_1, \ldots, d_m \\
\text{propagation} & \quad \text{(Rule 2)} \quad c_1, \ldots, c_n \Rightarrow d_1, \ldots, d_m
\end{align*}
\]

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 \( \llbracket \cdot \rrbracket \) from CHR rules to first-order formulas is:

\[
\begin{align*}
\llbracket c_1, \ldots, c_n \iff d_1, \ldots, d_m \rrbracket &= \forall \alpha(c_1 \land \cdots \land c_n \iff (\exists \beta d_1 \land \cdots \land d_m)) \\
\llbracket c_1, \ldots, c_n \Rightarrow d_1, \ldots, d_m \rrbracket &= \forall \alpha(c_1 \land \cdots \land c_n \Rightarrow (\exists \beta d_1 \land \cdots \land d_m))
\end{align*}
\]

where \( \alpha = f(v(c_1 \land \cdots \land c_n)) \) and \( \beta = f(v(d_1 \land \cdots \land d_m)) - \alpha \). 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 \( \theta \) grounding \( c_1, \ldots, c_n \) also is such that \( \theta \cdot \theta' \) grounds all variables in \( d_1, \ldots, d_m \) for any \( \theta' \) 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 \( C \longrightarrow C' \) or \( C \longrightarrow^* C' \), see Appendix A for details.

A derivation, denoted \( C \longrightarrow^* C' \) is a sequence of derivation steps using rules in \( P \) where no derivation step is applicable to \( C' \). A derivation \( C \longrightarrow^* C' \) is successful iff \( h_C \) is satisfiable. A set \( P \) of
CHR's is terminating iff for any constraint \( C \) there exists a constraint \( C' \) such that \( C \rightarrow C' \).

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

\[
\begin{align*}
\text{Ins} & (a \rightarrow b \rightarrow c), \text{Leq} (\text{Int} \rightarrow \text{Int} \rightarrow d) \\
\rightarrow \text{Ins} & (a \rightarrow b \rightarrow c), \text{Leq} (\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}), d = \text{Bool} \\
\rightarrow \text{Ins} & (a \rightarrow b \rightarrow c), d = \text{Bool} \\
\rightarrow \text{Ins} & (a \rightarrow b \rightarrow c), d = \text{Bool} \\
\rightarrow \text{Leq} & (a \rightarrow b \rightarrow \text{Bool}), b = a, c = [a], d = \text{Bool}
\end{align*}
\]

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

A CHR program \( P \) is confluent if for each constraint \( C_0 \) for any two possible derivation steps applicable to \( C_0 \), say \( C_0 \rightarrow_P C_1 \) and \( C_0 \rightarrow_P C_2 \), then there exist derivations \( C_1 \rightarrow_P C_3 \) and \( C_2 \rightarrow_P C_4 \) such that \( C_3 \) is equivalent (modulo new variables introduced) to \( C_4 \), i.e. \( \models (\exists f\iota(C_0), C_3) \leftrightarrow (\exists f\iota(C_0), C_4) \).

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

\[
\begin{align*}
\text{Ins} & (a \rightarrow b \rightarrow c), \text{Leq} (\text{Int} \rightarrow \text{Int} \rightarrow d) \\
\rightarrow \text{Ins} & (a \rightarrow b \rightarrow c), d = \text{Bool} \\
\rightarrow \text{Ins} & (a \rightarrow b \rightarrow c), b = a, c = [a], d = \text{Bool} \\
\rightarrow \text{Ins} & (a \rightarrow b \rightarrow c), b = a, c = [a], d = \text{Bool} \\
\rightarrow \text{Leq} & (a \rightarrow b \rightarrow \text{Bool}), b = a, c = [a], d = \text{Bool}
\end{align*}
\]

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 \( \models (\exists P) \wedge C \).

**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 \( C \rightarrow\star C' \). Then \( \models (\exists P) \wedge C \) iff \( \exists \bar{C} \).

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 \( \models (\exists P) \wedge D \rightarrow D' \) iff \( D \rightarrow\star P C \) and \( D' \rightarrow\star P C' \) such that \( \models (\exists f\iota(D), C) \leftrightarrow (\exists f\iota(D), C') \).

Note that for this result we require the CHRs \( P \) to be terminating, that is \( \forall \bar{C} \bar{C}' \rightarrow \hat{P} \bar{C}' \). 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 [34, 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** \( p ::= \text{load} f = e \in p \mid e \)
- **Expressions** \( e ::= x \mid \lambda x.e \mid e_1 + e_2 \mid \text{let} x = e_1 \in e_2 \mid (e :: \sigma) \)
- **Types** \( \tau ::= \alpha \mid \tau \rightarrow \tau \mid \tau \rightarrow \tau \)
- **Constraints** \( C ::= \tau = \tau \mid \text{U} \tau \mid \forall \alpha.C \Rightarrow \tau \)
- **Type Schemes** \( \sigma ::= \tau \mid \forall \alpha.C \Rightarrow \tau \)

where \( f \) ranges over overloaded identifiers, \( \cdot \rightarrow \cdot \rightarrow \cdot \) is the function type constructor, \( \tau \rightarrow \tau \) is a user-defined n-ary type constructor. \( \cdot \rightarrow \cdot \) denotes (syntactic) equality among types, \( \cdot \rightarrow \cdot \rightarrow \cdot \) is a user-defined n-ary predicate symbol, and \( \forall \cdot \) denotes conjunction among constraints.

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, \( \Gamma \) a typing environment, \( e \) an expression and \( \tau \) a type. We will always require that constraints \( C \) appearing in typing judgments \( P,C,\Gamma \vdash e : \sigma \) and type schemes \( \forall \alpha.C \Rightarrow \tau \) 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.

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

In rule (\forall) the statement \( \models [P] \models C_1 \rightarrow [\tau/\alpha]C_2 \) requires that the constraint \( C_1 \) implies constraint \( [\tau/\alpha]C_2 \) (with \( \alpha \) replaced by \( \tau \)) in any model of \( [P] \).

Our formulation of rule (\forall) 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 \( \exists \alpha.C \) on the left-hand side. This has some advantages as discussed in [34]. For the purpose of this paper, the present formulation of rule (\forall) 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-
(Var) \[ \frac{(x : \sigma) \in \Gamma}{P.C, \Gamma \vdash x : \sigma} \]

(Abs) \[ \frac{P.C, \Gamma, x : \tau \vdash e : \tau'}{P.C, \Gamma, \lambda x. e : \tau \rightarrow \tau'} \]

(\forall I) \[ \frac{\forall a \not\in \text{fv}(C_1) \cup \text{fv}(\Gamma)}{P.C_1 \land C_2, \Gamma \vdash e : \tau} \]

(Annot) \[ \frac{\text{fv}(\sigma) = \emptyset}{P.C, \Gamma \vdash (e : \sigma) : \sigma} \]

(Over) \[ \frac{(f : \forall a.A \Rightarrow a) \in \Gamma}{\text{fv}(\forall a.A_f \Rightarrow \tau_f) = \emptyset} \]

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:

(Super) \[ \text{Leq} \ (a \rightarrow a \rightarrow \text{Bool}) \implies \text{Eq} \ (a \rightarrow a \rightarrow \text{Bool}) \]

For the remainder of the paper, we adopt the convention that \( P \) 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 \( P = P_p \cup P_p \) 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 we require that for each type scheme \( \forall a.C \Rightarrow \tau \) we have that \( \text{fv}(C) \cap a \subseteq \text{fv}(\tau) \cap a \). 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 \( \forall \alpha.C \Rightarrow \tau \) be a type scheme, \( P \) be the program theory used in this context and \( \rho \) be a variable renaming on \( \alpha \). Then \( \forall \alpha.C \Rightarrow \tau \) is unambiguous iff for \( P \) \( |(C \wedge \rho(C) \wedge (\tau = \rho(\tau))) \supset (\alpha = \rho(\alpha)) \) for each \( \alpha \in \alpha \). We also say that \( \tau \) is unambiguous if \( e::\sigma \) is a well-typed expression and \( \sigma \) 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 \( a \) cannot be determined from the constraint component alone. We find that \( |H(a \rightarrow b) \wedge H(a' \rightarrow b') \wedge b = b' \supset a = a' \). Assume that our program theory consists of the following CHR (note that this CHR mimics a functional dependency):

\[
(FH) \quad H(a \rightarrow b), H(a' \rightarrow b') \implies a = a' 
\]

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

### 4.2 Improvement

The programmer-specifiable set \( P \) 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

\[
\text{Leq} \ (a \rightarrow b \rightarrow c) \ | \ (a, b) \sim c \text{ states that both input arguments uniquely determine the result where } (a, b) \sim c \text{ is a functional dependency. This behavior can be modeled by the following CHR propagation rule:}
\]

\[
(FD) \quad \text{Leq} \ (a \rightarrow b \rightarrow c), \text{Leq} \ (a \rightarrow b \rightarrow d) \implies c = d
\]

In general, any declaration with functional dependencies can be translated into a set of CHR propagation rules. Assume we have \( F | f_{d1}, \ldots, f_{dn} \) where \( f_i(\tau) = \alpha \) and \( f_i \) is a functional dependency of the form \((\alpha_{i1}, \ldots, \alpha_{in}) \sim \alpha_{o} \). The functional dependency asserts that given fixed values of \( \alpha_{i1}, \ldots, \alpha_{in} \) then there is only one value of \( \alpha_{o} \) for which the constraint \( F \) can hold. Note that in [19] the right-hand side of the \( \sim \) 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:

\[
F \tau, F \theta(\rho\tau) \Rightarrow \alpha_{o} = \beta_{o}
\]

where \( \rho \) is a renaming on \( \alpha \) such that \( \rho(\alpha_{i}) = \beta_{i} \) and \( \theta \) maps each \( \beta_{i} \) to \( \alpha_{i} \) and each other \( \beta_{j} \) 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.**

\[
\begin{align*}
\text{overload } & f_1 :: \text{Float} \rightarrow \text{Float} \\
\text{overload } & f_2 :: \text{Int} \rightarrow \text{Float} \\
\text{overload } & f_3 :: \text{Int} \rightarrow \text{Int}
\end{align*}
\]

We find the following set \( P \).

\[
\begin{align*}
(F1) \quad & F (\text{Float} \rightarrow \text{Float}) \iff \text{True} \\
(F2) \quad & F (\text{Int} \rightarrow \text{Float}) \iff \text{True} \\
(F3) \quad & F (\text{Int} \rightarrow \text{Int}) \iff \text{True}
\end{align*}
\]

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

\[
(F4) \quad F (\text{Float} \rightarrow a) \implies a = \text{Float}
\]

Such behavior cannot be specified by functional dependencies.

\(^1\)The term improvement was coined by Jones [18].

### 4.3 Confluence

We require that program theories must be confluent.

**EXAMPLE 6. Consider the following program**

\[
\begin{align*}
\text{overload } & \text{eq} :: \text{Int} \rightarrow \text{Bool} \\
\text{eq} & = \text{primEqInt}
\end{align*}
\]

\[
\begin{align*}
\text{overload } & \text{eq} :: \forall a.\text{Eq} (a \rightarrow a \rightarrow \text{Bool}) \Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool} \\
\text{eq} & = \text{let } [\text{eq}] [] \Rightarrow \text{True} \\
\text{eq} & \text{[(x,y)] = False} \\
\text{eq} & \text{[(x,y)] = False} \\
\text{eq} & \text{[(x,y)] = (eq x y) \&\& (eq x ys) ys}
\end{align*}
\]

\[
\begin{align*}
\text{overload } & \text{eq} :: \text{Int} \rightarrow \text{Bool} \\
\text{eq} & = \lambda x. \lambda y. \text{True}
\end{align*}
\]

where \( \text{primEqInt} \) is a primitive equality function of type \( \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \).

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

\[
\begin{align*}
(\text{Eq1}) \quad & \text{Eq} (\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}) \iff \text{True} \\
(\text{Eq2}) \quad & \text{Eq} (\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}) \iff \text{Eq} (\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}) \\
(\text{Leq1}) \quad & \text{Leq} (\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}) \iff \text{True} \\
(\text{Leq4}) \quad & \text{Leq} (\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}) \iff \text{True}
\end{align*}
\]

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

\[
\begin{align*}
(\text{Super}) \quad & \text{Leq} (\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}) \iff \text{Eq} (\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool})
\end{align*}
\]

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

\[
\begin{align*}
\text{Leq} ([a] \rightarrow [a] \rightarrow \text{Bool}) \implies_{\text{Leq4}} \text{True}
\end{align*}
\]

and

\[
\begin{align*}
\text{Leq} ([a] \rightarrow [a] \rightarrow \text{Bool}) \implies_{\text{Super}} \text{Leq} ([a] \rightarrow [a] \rightarrow \text{Bool}) \\
\text{Leq} ([a] \rightarrow [a] \rightarrow \text{Bool}) \implies_{\text{Leq4}} \text{Eq} ([a] \rightarrow [a] \rightarrow \text{Bool}) \\
\text{Eq} ([a] \rightarrow [a] \rightarrow \text{Bool}) \implies_{\text{Eq2}} \text{Eq} (a \rightarrow a \rightarrow \text{Bool})
\end{align*}
\]

Clearly, there is a problem among the set of overloaded definitions and the super class relationship of \( \text{eq} \) and \( \text{leq} \).

There are however cases where it is safe to add some propagation rules to “complete” a non-confluent program theory.

**EXAMPLE 7. Consider the following program**

\[
\begin{align*}
\text{overload } & \text{ins} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{ins} & = \lambda x. \lambda y. x + y
\end{align*}
\]

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

\[
\begin{align*}
(\text{Ins1}’) \quad & \text{Ins} ([\text{Int}] \rightarrow \text{Int} \rightarrow [\text{Int}]) \iff \text{True} \\
(\text{Func}) \quad & \text{Ins} ([c \rightarrow e_1 \rightarrow c], \text{Ins} (c \rightarrow e_2 \rightarrow c) \implies e_1 = e_2)
\end{align*}
\]

The above program theory is non-confluent.

\[
\begin{align*}
\text{Ins} ([\text{Int}] \rightarrow \text{Int} \rightarrow [\text{Int}]), \text{Ins} ([\text{Int}] \rightarrow a \rightarrow [\text{Int}]) \\
\implies_{\text{Ins1}} \text{Ins} ([\text{Int}] \rightarrow a \rightarrow [\text{Int}])
\end{align*}
\]
are two distinct, non-joinable derivations. Adding the following propagation rule yields a confluent program theory.

\[
(\text{InsFunc1}) \quad \text{Ins} \left( [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Int} \right) \quad \implies \quad a = \text{Int}
\]

Note that rule (Func) states a general property which must hold for all its 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 \( \text{eq} \) of Example 6 extended with the following definition.

\[
\text{overload} \ \text{eq} :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool}
\]

We find the following program theory:

\[
\begin{align*}
\text{(Eq1)} \quad & \text{Eq} \left( [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool} \right) \iff \text{True} \\
\text{(Eq2)} \quad & \text{Eq} \left( [a] \rightarrow [a] \rightarrow \text{Bool} \right) \iff \text{Eq} \left( a \rightarrow a \rightarrow \text{Bool} \right) \\
\text{(Eq3)} \quad & \text{Eq} \left( [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool} \right) \iff \text{True}
\end{align*}
\]

The above program theory is confluent. However, note that the second and third definition of \( \text{eq} \) are overlapping. We say two definitions are overlapping if there exists a substitution \( \emptyset \) which unifies the head atoms in the respective simplification rules. In case we require a definition of \( \text{eq} \) at type \( [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool} \), we must take an indeterministic choice between the 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 \( \Gamma \) where all overloaded identifiers \( f \) are recorded, i.e., \( f :: \forall a.F \ a \Rightarrow a \) \( \in \Gamma \).

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

\[
\begin{align*}
\text{(Exp)} & \quad e \vdash_{\inf} \emptyset \\
\text{(Over)} & \quad P_\text{s} \vdash_{\inf} \emptyset, \text{ mgu of } \text{h}_C \\
\quad & \quad P'_s = P_s \cup \{ F \vdash_{\phi} C \}
\end{align*}
\]

In addition, we are given a set \( P_\text{s} \) of programmer-specifiable propagation rules. Together, \( P = P'_s \cup P_\text{s} \), where \( P \vdash_{\inf} P_\text{s} \), 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 where 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 [32] 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

\[
P_\Gamma, p \vdash_{\inf} (C \vdash \tau)
\]

where program theory \( P \), type environment \( \Gamma \) and program \( p \) are input values, a constraint \( C \) and a type \( \tau \) 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 \( \Gamma \) be a type environment, \( C \) a constraint and \( \tau \) a type. Then, we define \( \text{gen} (\Gamma, C, \tau) = (\text{True} \mid \sigma) \) where \( \bar{\alpha} = \text{fs} (C, \tau) / \text{fs} (\Gamma) \) and \( \sigma = \forall \bar{\alpha} . C \Rightarrow \tau \).

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

\[
\begin{align*}
\text{sat} (P, C) & \quad = \quad \text{True} & \quad \text{if } C \vdash_{\tau} \bar{C}' \quad \text{such that } |\bar{C}'| = h_C & \quad \text{False} & \quad \text{otherwise}
\end{align*}
\]

Note that the condition \(|\bar{C}'| = h_C\) 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:

\[
\text{unambig} (P, \forall \bar{\alpha} . C \Rightarrow \tau) =
\begin{align*}
\text{True} & \quad \text{if } C \vdash \bar{C}' \quad \text{such that } |\bar{C}'| = h_C & \quad \text{False} & \quad \text{otherwise}
\end{align*}
\]

The condition \(|\bar{C}'| = h_C\) is decidable (it holds iff the mgu of \( h_C \) unifies \( \bar{\alpha} \) and \( \rho (\bar{\alpha}) \)) which ensures that the above procedure is decidable for terminating CHRs.

**Example 9.** Consider the type scheme \( \forall \alpha, \alpha'. H \ (\alpha \rightarrow \alpha') \Rightarrow \alpha' \).

The program theory consists of rule (FH) (see 4.1). We assume that \( \rho (\bar{\alpha}) = \alpha'' \) and \( \rho (\alpha') = \alpha''' \). We check unambiguity via the derivation

\[
\begin{align*}
& \quad \vdash_{\text{FH}} H \ (\alpha \rightarrow \alpha'), H \ (\alpha'' \rightarrow \alpha'''), \alpha' = \alpha''', \alpha = \alpha'' \\
& \quad \quad \vdash_{\text{FH}} H \ (\alpha \rightarrow \alpha'), H \ (\alpha'' \rightarrow \alpha'''), \alpha' = \alpha''', \alpha = \alpha''
\end{align*}
\]

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 \textit{entail} performs an entailment check to check the validity of the annotated type. The definition of \textit{entail} is as follows:

\[
P_\Gamma, p \vdash_{\inf} (C \vdash \tau)
\]
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, $\forall \alpha.C \Rightarrow \tau$ be a type scheme and $\rho$ be a variable renaming on $\alpha$ such that $C \land p(C) \land \tau \Rightarrow p(\tau) \Rightarrow p(C')$ where $\models C' \supset (\alpha = p(\alpha))$ for each $\alpha \in \mathcal{A}$. Then for each $\alpha \in \mathcal{A}$ $\models P = (C \land p(C) \land (\tau = p(\tau))) \supset (\alpha = p(\alpha))$.

**LEMMA 4 (SOUNDNESS OF ENTAILMENT).** Let $P$ be a set of CHRs, $\alpha$ and $\alpha'$ two fresh variables and $\mathcal{S} = \forall \alpha.C \Rightarrow \tau$ and $\mathcal{S}' = \forall \alpha'.C' \Rightarrow \tau$ where $C' \land \tau' = \alpha' \land \alpha = \alpha' \land \alpha \land C \Rightarrow p.C_2$ such that $\models (\mathcal{S}_\alpha C_1) \Rightarrow (\mathcal{S}_\alpha C_2)$, where $V = \mathcal{F}C'(\alpha' \land \tau' = \alpha' \land \alpha \land C) \Rightarrow p.C_2$ such that $\models (\mathcal{S}_\alpha C_1) \Rightarrow (\mathcal{S}_\alpha C_2)$. Then $\models P \Rightarrow \models \mathcal{S} \supset \mathcal{S}'$. 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, $\Gamma$ a type environment, $P_p$ be a set of propagation rules, $P_0$ be a set of CHRs, $C$ a constraint and $\tau$ a type such that $\models p \Rightarrow P_p$ and $P_0 \cup P_0 \Rightarrow P, e : e : \tau$ 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, $\forall \alpha.C \Rightarrow \tau$ be a type scheme and $\rho$ be a variable renaming on $\alpha$ such that $\models P = (C \land p(C) \land (\tau = p(\tau))) \supset (\alpha = p(\alpha))$ for each $\alpha \in \mathcal{A}$. Then $C \land p(C) \land (\tau = p(\tau)) \Rightarrow p.C'$ where $\models C' \supset (\alpha = p(\alpha))$ for each $\alpha \in \mathcal{A}$.

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 CHR rules whose simplification rules are single-headed, \( \alpha, \alpha' \) two fresh variables and \( \sigma \equiv \forall \alpha. C \Rightarrow \tau \) and \( \sigma' \equiv \forall \alpha'. C' \Rightarrow \tau' \) such that \([P] \vdash \alpha \leq \sigma' \) and \( \sigma \) is unambiguous. Then \( C' \land \tau' = \alpha' \land \alpha = \alpha' \rightarrow P \land C \land \tau = \alpha' \land \alpha \Rightarrow P \land C \) and \( P' \land \tau' = \alpha' \land \alpha = \alpha' \rightarrow P'C \land \tau' = \alpha' \land \alpha \Rightarrow P'C \).

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 CHR rules, \( C \) a constraint, \( \Gamma \) an environment, \( e \) an expression and \( \sigma \) a type scheme. We say \((C, \sigma)\) is the principal constrained type (w.r.t. \( P, \Gamma \) and \( e \)) if (1) \( P, C, \Gamma \vdash e : \sigma \) and (2) for each \( P, C', \Gamma \vdash e : \sigma' \) we have that (a) \([P] \models C \cup (\exists V_{\forall \alpha} C_1)\), and (b) \([P] \land C' \models \sigma \leq \sigma' \). We say \((C, \sigma)\) is unambiguous if \( \sigma \) is unambiguous. A judgment \( P, C, \Gamma \vdash e : \sigma \) is principally unambiguous iff for each subexpression in \( e \) the principal constrained type is unambiguous.

**Theorem 2 (Weak Completeness of Type Inference).** Let \( P, C, \Gamma \vdash e : \tau \) be a principally unambiguous judgment such that \( P \) is terminating, confluent, range-restricted and all simplification rules are single-headed. Then \( \Gamma, P, e \vdash \inf (C \land \tau) \) for some constraint \( C \) and type \( \tau \) such that \([P] \models C \cup (\exists V_{\forall \alpha} \tau') \) where \( V = fV(\sigma) \cup fV(\tau') \cup fV(C' \land \tau = \tau') \).

### 6 Evidence Translation

We follow the common approach (e.g. [35]) 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 program 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.

**Target Expressions**

\[
E ::= x \mid \lambda x.E \mid \text{let } x = E \text{ in } E
\]

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

We introduce judgments of the form \( P, \Gamma, C \vdash e : \sigma \rightarrow E \) where \( E \) is the result of translating a well-typed expression \( e \) with type \( \sigma \) under program theory \( P \), environment \( \Gamma \) and constraint \( C \). The most interesting rules are (\( \forall \)) where we abstract over evidence variables and (\( \forall E \)) where we provide the proper evidence values. We assume that \( ec : C \rightarrow \overline{\tau/\alpha}C_2 \) is an evidence constructing function. Note that the context only provides evidence for \( ec_1 \). However, the instantiation site requires evidence \( ec_1(\overline{\alpha}/C_2) \). The premise states that \([P] \models C \cup \overline{\tau/\alpha}C_2 \). In fact, this is sufficient to ensure that function \( ec \) must exist. Assume we have a substitution \( \phi \) such that \( \phi C_1 =_{\overline{\alpha}} e \). Then, we also have that \( \phi(\overline{\tau/\alpha}C_2) =_{\overline{\alpha}} e \). In such a situation, we find that \( \phi(\overline{\tau/\alpha}C_2) =_{\overline{\alpha}} e \). This is allows us to construct evidence \( e_{\overline{\tau/\alpha}C_2} \) by reading the CHR derivation backwards.

Recall the definitions of \( eq \) in Example 6.

overload \( eq : \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \)

\[
eq \text{primEqInt}
\]

overload \( eq : \forall a. \text{Eq} \ (a \rightarrow a \rightarrow \text{Bool}) \Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool} \)

\[
eq \text{let } eq [] [a] = \text{True} \]

\[
eq \text{eq} [\text{xs}] [\text{ys}] = \text{False} \]

\[
eq \text{eq} [\text{ys}] = \text{False} \]

\[
eq \text{eq} [\text{xs}] [\text{ys}] = (\text{eq \ (tail \ xs)} \ (\text{ys}) \ 1 \ 3)\]

where \( \text{tail} \ [a] \rightarrow [a] \) takes the tail of a list. In a first step we translate overloaded definitions. We find

\[
\text{ecEqInt} = \text{primEqInt} \]

\[
\text{ecEqList} \ \text{eq} = \text{let } eq [] [a] = \text{True} \]

\[
\text{eq} [\text{xs}] [\text{ys}] = \text{False} \]

\[
\text{eq} [\text{ys}] = \text{False} \]

\[
\text{eq} [\text{xs}] [\text{ys}] = (\text{eq \ x \ y)) \ & \ & (\text{eq \ xs \ ys})\]

in eq.

Consider

\[
\text{eq} \ \text{xs} \ \text{ys} = (\text{eq \ (tail \ xs)} \ (\text{ys}) \ 1 \ 3)\]

Note that parameter eq represents evidence for the equality function on type \( a \rightarrow a \rightarrow \text{Bool} \). What remains is to insert the appropriate evidence values for expression \( \text{exp} \).

Expression \( \text{exp} \) gives rise to the following constraints

\[
E \ (\text{where we assume } x :: [a], y :: b, 1 :: \text{Int} \text{ and } 3 :: \text{Int}) \text{ we find that}
\]

\[
\text{exp} :: \forall a. \text{Eq} \ (a \rightarrow a \rightarrow \text{Bool}) \Rightarrow [a] \rightarrow [a] \text{ (Bool)}
\]

\[
\text{exp} \ \text{xs} \ \text{ys} = (\text{eq \ (tail \ xs)} \ (\text{ys}) \ 1 \ 3)
\]

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

The two instantiation sites of \( \text{eq} \) imply the existence of evidence constructors:

\[
\text{ec1} :: \text{Eq} \ (a \rightarrow a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \rightarrow \text{Bool} \]

\[
\text{ec2} :: \text{Eq} \ (a \rightarrow a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \rightarrow \text{Bool}
\]

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

\[
\text{Eq} ([\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool})
\]

\[
\text{Eq} ([\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool})
\]

\[
\text{Eq} ([\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool})
\]

\[
\text{Eq} \ 	ext{True}
\]

we conclude that

\[
\text{ecEq} ([\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool}) = \text{ecEqList} \ (\text{ecEqList} \ ecEqInt)
\]

Rule (Eq1) reduces to True. That is, \( \text{ecEq} ([\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool}) \) must be present. Each (Eq2) rule application corresponds to applying \( \text{ecEqList} \) to the previously constructed evidence. We find that

\[
\text{ec1 eEq} ([\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool}) = \text{ecEqList} \ (\text{ecEqList} \ ecEqInt)
\]

Similarly, for the second instantiation site we have that

\[
\text{ec2 eEq} ([\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Bool}) = \text{ecEqInt}
\]
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\( ec1 = ec_eqList \) and\( ec2 x = ec_eqInt \).

\[ \text{eqList} \]

\[ \text{exp} :: \text{and} \]

\[ \text{ec2} x \rightarrow \text{Bool} \]

\[ \phi \]

\[ \text{where} \]

\[ \text{found in [31].} \]

\[ \text{and proofs go clearly beyond the scope of the present paper. A} \]

\[ \text{identifiers allow us to provide type inference for some interesting} \]

\[ \text{function.} \]

\[ \text{11.} \]

\[ \forall a = \overload \text{zip} :: \text{a} \rightarrow \text{a} \rightarrow \text{Bool} \]

\[ \ast \]

\[ \text{where} \]

\[ \text{eq} x = x \]. We assume each overloaded identifier simply passes

\[ \text{on the appropriate calculated definition. Note that Eq (a \rightarrow a \rightarrow} \]

\[ \text{Bool) has been turned into a matching function type.} \]

\[ \text{Theorem 3 (Coherence). Let P be a confluent set of range-} \]

\[ \text{restricted CHRs where each simplification rule is single-headed} \]

\[ \text{and non-overlapping. Then the translation scheme in Figure 3} \]

\[ \text{yields a coherent system.} \]

\[ \text{The exact details of the translation process including formal results} \]

\[ \text{and proofs go clearly beyond the scope of the present paper. A} \]

\[ \text{formal development including a concise coherence result can be} \]

\[ \text{found in [31].} \]

\section{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

\[ \text{zip :: } \forall a,b, [a \rightarrow [b] \rightarrow [(a,b)]} \]

\[ \text{zip3 :: } \forall a,b,c, [a \rightarrow [b] \rightarrow [c] \rightarrow [(a,b,c)]} \]

\[ \ldots \]

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

\[ \text{Example 11. Consider the following generic definition of the zip-} \]

\[ \text{function.} \]

\[ \text{zip2 :: } \forall a,b, [a \rightarrow [b] \rightarrow [(a,b)]} \]

\[ \text{zip2 [] [] = []} \]

\[ \text{zip2 (a:as) (b:bs) = (a,b):(zip2 as bs)} \]

\[ \text{zip2 [] (b:bs) = []} \]

\[ \text{zip2 (a:as) [] = []} \]

\[ \text{overload zip :: } \forall a,b, [a \rightarrow [b] \rightarrow [(a,b)]} \]

\[ \text{overload zip :: } \forall a,b,c,e. \text{Zip ([a,b]) \rightarrow [c] \rightarrow e} \Rightarrow \]

\[ \text{[a] \rightarrow [b] \rightarrow [c] \rightarrow e} \Rightarrow \]

\[ \text{zip as bs cs = zip (zip2 as bs) cs} \]

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

\[ \text{Zip1} \]

\[ \text{Zip ([a] \rightarrow [b] \rightarrow [(a,b)])} \]

\[ \text{Zip2} \]

\[ \text{Zip ([a] \rightarrow [b] \rightarrow [(a,b)])} \]

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

\[ \text{Zip3} \]

\[ \text{Zip ([a] \rightarrow [b] \rightarrow [c])} \Rightarrow \]

\[ \text{Zip ([a] \rightarrow [b] \rightarrow [c])} \Rightarrow \]

Consider the following (partially) type-annotated expression.

\[ \text{e :: } [(\text{Int,Bool},\text{Char})} \]

\[ \text{e = zip1 [1,2,3] [True,False] [a',b',c']} \]

From the program text, we generate

\[ \text{Zip ([Int] \rightarrow [Bool] \rightarrow [Char] \rightarrow [((\text{Int,Bool}),\text{Char})]} \]

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

\[ \text{Zip ([Int] \rightarrow [Bool] \rightarrow [Char] \rightarrow [((\text{Int,Bool}),\text{Char})]} \]

\[ \Rightarrow \text{zip2} \]

\[ \text{Zip ([Int] \rightarrow [Bool] \rightarrow [Char] \rightarrow [((\text{Int,Bool}),\text{Char})]} \]

\[ \Rightarrow \text{zip1} \]

\[ \text{True} \]

Therefore, the above expression is translated into

\[ \text{e :: } [(\text{Int,Bool},\text{Char})} \]

\[ \text{e = zip2 [zip2 [1,2,3] [True,False] [a',b',c']} \]

\section{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.

\subsection{Closing Definitions}

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

\[ \text{overload eq :: } \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \]

\[ \text{eq = ...} \]

\[ \text{overload eq :: } \forall a. \text{Eq} \ (a \rightarrow a \rightarrow \text{Bool}) \Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool} \]

\[ \text{eq = ...} \]
We find the following set of CHRs:

(Eq1)  \[ Eq \left( \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \right) \iff \text{True} \]
(Eq2)  \[ Eq \left( [a] \rightarrow [a] \rightarrow \text{Bool} \right) \iff Eq \left( a \rightarrow a \rightarrow \text{Bool} \right) \]

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

\[ \text{true} :: \forall a. Eq \left( \text{Tree} \ a \rightarrow \text{Tree} \ a \rightarrow \text{Bool} \right) \Rightarrow \text{Tree} \ a \rightarrow \text{Tree} \ a \rightarrow \text{Bool} \]

\[ \text{true} \ x \ y = \text{eq} \ x \ y \]

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

\[ \text{(CloseEq)} \quad Eq \ a \implies (a = \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}) \lor (a = [b] \rightarrow [b] \rightarrow \text{Bool}) \]

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 \( \implies \) 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 cannot provide a coherent translation because we must take an indeterministic choice in case we require \text{eq} on type \[ \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \].

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, \( a \neq \text{Int} \), added to rule (Eq2), states that this rule only fires if the instance type is different from \text{Int}.

\begin{align*}
\text{(Eq1)} & \quad Eq \left( \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \right) \iff \text{True} \\
\text{(Eq2')} & \quad Eq \left( [a] \rightarrow [a] \rightarrow \text{Bool} \right) \mid a \neq \text{Int} \iff Eq \left( a \rightarrow a \rightarrow \text{Bool} \right) \\
\text{(Eq3)} & \quad Eq \left( \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \right) \iff \text{True}
\end{align*}

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

\[ \text{(Eq4)} \quad Eq \left( \text{Char} \rightarrow \text{Char} \rightarrow \text{Bool} \right) \iff \text{True} \]

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

\[ \text{(Eq5)} \quad Eq \left( a \rightarrow b \rightarrow c \right) \Rightarrow a = b \]

Type inference for expression

\[ g \times y = (\text{eq} \ [x] \ [y]) : \text{Bool} \]

yields the constraint \( Eq \left( [a] \rightarrow [a] \rightarrow \text{Bool} \right) \) where \( x : a \) and \( y : a \). Note that no further reduction steps are applicable at this point. We find that

\[ g :: \forall a. Eq \left( [a] \rightarrow [a] \rightarrow \text{Bool} \right) \Rightarrow a \rightarrow a \rightarrow \text{Bool} \]

\[ g \times y = (\text{eq} \ [x] \ [y]) : \text{Bool} \]

\[ \text{eq} \ x \ y = (\text{eq} \ [x] \ [y]) \]

where \( \text{eq} \) is \( g \)'s translation. Consider expression

\[ \text{exp} = (g \ [1] \ [2], g \ [[a] \ [b]] [[a] \ [b]]) \]

where we use \( g \) at two different instantiation sites. In context \( g \ [1] \ [2] \) we require the constraint \( Eq \left( \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \right) \) whereas in context \( g \ [[a] \ [b]] [[a] \ [b]] \) we require \( Eq \left( \text{Char} \rightarrow \text{Char} \rightarrow \text{Bool} \right) \). We find the following derivations:

\[ Eq \left( \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \right) \Rightarrow Eq^3 \text{True} \quad \text{and} \]

\[ Eq \left( \text{Char} \rightarrow \text{Char} \rightarrow \text{Bool} \right) \Rightarrow Eq^4 \text{True} \]

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

\[ \text{exp} = ((g \ [1] \ [2], g \ [[a] \ [b]] [[a] \ [b]]) \]

where \( \text{eq} \ [a] \ [b] \) represents evidence for \( Eq \left( \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \right) \) and \( \text{eq} \ [a] \ [b] \) represents evidence for \( Eq \left( \text{Char} \rightarrow \text{Char} \rightarrow \text{Bool} \right) \) and \( \text{eq} \ [a] \ [b] \) is defined as in Section 6.

We make the following observations. Overlapping definitions have no impact on typeability as long as the program theory is still confluent. Note that \( Eq \left( [a] \rightarrow [a] \rightarrow \text{Bool} \right) \) and \( Eq \left( a \rightarrow a \rightarrow \text{Bool} \right) \) 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 \( P = P_s' \cup P_s \) 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 \text{eq} and \text{leq}:

\[ \text{(Super)} \quad \text{Leq} \left( a \rightarrow a \rightarrow \text{Bool} \right) \Rightarrow Eq \left( a \rightarrow a \rightarrow \text{Bool} \right) \]

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

\[ h :: \forall a. Eq \left( a \rightarrow a \rightarrow \text{Bool} \right) \land \text{Leq} \left( a \rightarrow a \rightarrow \text{Bool} \right) \Rightarrow a \rightarrow a \rightarrow \text{Bool} \]

\[ h \times y = (\text{eq} \ x \ y \land \text{leq} \ x \ y) \]

Note that

\[ \left[ P \right] \models (Eq \left( a \rightarrow a \rightarrow \text{Bool} \right) \land Eq \left( a \rightarrow a \rightarrow \text{Bool} \right) ) \Leftarrow \text{Leq} \left( a \rightarrow a \rightarrow \text{Bool} \right) \]
where $P$ consists of (Eq1-2), (Leq1-2) and (Super). Therefore, we could assign to expression $h$ the equivalent but simpler type scheme
\[
\forall a. \text{Leq} (a \to a \to \text{Bool}) \Rightarrow a \to a \to (\text{Bool}, \text{Bool})
\]
Such form of "simplification" can always be achieved by turning a rule such as (Super) into a multi-headed simplification rule of the form
\[
\text{Leq} (a \to a \to \text{Bool}), \text{Eq} (a \to a \to \text{Bool}) \iff \text{Leq} (a \to a \to \text{Bool})
\]

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 which 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 CHR features. Although we are seeking a general formal framework within which we can reason about overloading in a concise way. CHRs serve as a meta-language to describe relations among overloaded identifiers. We can describe precisely in terms of CHR features. Under which conditions type inference is decidable (Section 5) and the semantics of programs is well-defined (Section 6). The framework presented here can be seen as a formal basis for 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, Gabeichler 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 systems 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.

Camaro 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 inference on 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 which we can study implicit parameters and Haskell style overloading.

It is also worth mentioning the work by Yang [36]. 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 has not been any proposal so far to make this connection concrete.

In this paper, we have 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 [31].

The design space for overloading systems is huge. In Section 8 we elaborate on some possible variations. We refer to [33] 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 [32].

Acknowledgements

We thank Kevin Glynn, Simon Peyton-Jones, Andreas Rossberg, Peter Thiemann, Jeremy Wazny, Matthias Zenger and the referees for their comments.

11 References

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 $\overline{e} | g \leftrightarrow \overline{d}$ where the guard constraints $g$ is a conjunction of disequality constraints.

(Solve) $C_u \cup C_e \longrightarrow_p \phi C_u \cup C'_e$

if $\models C'_e \leftarrow C_e$ and $\phi$ mgu of $C_e$

(Simp) $C_u \cup C_e \longrightarrow_p (C_u - \overline{e'}) \cup \theta(\overline{d}) \cup C_e$

if $\overline{e} | g \leftrightarrow \overline{d} \in P$ and there exists $\overline{e'} \in C_u$

and a substitution $\theta$ on variables in $r$

such that $\theta(\overline{e'}) \equiv \overline{e}$ and $C_e \models \theta(g)$

(Prop) $C_u \cup C_e \longrightarrow_p C_u \cup C_e \cup \theta(\overline{d})$

if $\overline{e} \models \overline{d} \in P$ and there exists a subset $\overline{e'} \subseteq C_u$

and a substitution $\theta$ on variables in $r$

such that $\theta(\overline{e'}) \equiv \overline{e}$

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 $C \longrightarrow_p C'$. Then, $[P] \models C \iff [P] \vdash^2 [C] C'$.