A Theory of Overloading

PETER J. STUCKEY
University of Melbourne
and
MARTIN SULZMANN
National University of Singapore

We present a novel approach to allow for overloading of identifiers in the spirit of type classes. 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, that provide our scheme with a form of programmable type language. 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 provides a common formal basis for many type class extensions such as multi-parameter type classes and functional dependencies.

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
Additional Key Words and Phrases: Constraints, overloading, type classes, type inference, evidence translation, coherence

1. INTRODUCTION

The study of overloading, a.k.a. ad-hoc polymorphism, in the context of the Hindley/Milner system [Milner 1978] dates back to Kaes [1988] and Wadler and Blott [1989]. Since then, it became a powerful programming feature in languages such as Haskell [Peyton Jones et al. 1999], Mercury [Henderson et al. 2001; Jeffery et al. 2000] , HAL [Demoen et al. 1999] and Clean [Plasmeijer and van Eekelen 1998]. In particular, Haskell provides through its type-class system [Jones 1992] 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 [Jones 1993b], multi-parameter classes [Jones et al. 1997] and most recently functional dependencies [Jones 2000]. Each of these extensions required a careful reinvestigation

Author's addresses: P.J. Stuckey. Dept of Comp. Sci. & Soft. Eng., University of Melbourne 3010, AUSTRALIA. pjs@cs.mu.oz.au M. Sulzmann. School of Computing, National University of Singapore, S16 Level 5, 3 Science Drive 2, Singapore 117543. sulzmann@comp.nus.edu.sg

of essential properties such as decidable type inference and coherent semantics. There is also a significant body of further closely related work, for example [Gasbichler et al. 2002; Duggan and Ophel 2002b; Shields and Jones 2001; Camarao and Figueiredo 1999; Odersky et al. 1995; Nipkow and Prehofer 1995; Chen et al. 1992].

Here, we present an extension of the Hindley/Milner system in the style of type classes to allow for overloading of identifiers. The aim of the system is to provide a *programmable* type language that has decidable type inference and a coherent evidence translation. We use *Constraint Handling Rules* (CHRs) [Frühwirth 1995] as the type programming language. CHRs are a rule based language for specifying constraints solvers. The benefit of using CHRs is that they are a form of logical statement which is also an executable specification. Hence we have a firm semantic basis to the type language as well as a clear operational semantics.

This framework is implemented and available as part of the Chameleon system [Sulzmann and Wazny 2003]. We assume the reader is familiar with Haskell [Peyton Jones et al. 1999] style type classes. In order to introduce our approach we first give examples in Haskell notation, before rewriting them to an equivalent form in Chameleon.

EXAMPLE 1. *Consider the following simple Haskell type class program.*

```
class Leq a where
   leq :: a->a->Bool
class Insert ce e | ce -> e where
   ins :: ce->e->ce
instance Leq Int where leq = primLeqInt
instance Leq Float where leq = primLeqFloat
instance Leq a => Insert [a] a where
   ins [] y = [y]
   ins (x:xs) y = if leq x y then x:(ins xs y) else y:x:xs
```

Our intention is that the `Leq` type class denotes the family of less-than-equal functions whereas `Insert` denotes the family of functions which allow us to insert an element of type `e` into a collection of type `ce`. We provide two instances of `Leq` where we assume that `primLeqInt` and `primLeqFloat` are primitive functions, testing for less-than-equal on integers and floats. The third instance states that we can provide an instance `Insert [a] a` if we provide an instance `Leq a`. Indeed, in the instance body we implement a simple insertion algorithm by inserting an element in a list such that elements preceding the inserted element are less-than-equal.

The `Insert` class also imposes a functional dependency `ce -> e`, that is the collection type `ce` uniquely determines the element type `e`. Hence, there cannot be another instance `Insert [Int] Bool` defined since then the element type `e` for a collection `[Int]` could be either `Int` or `Bool`.

EXAMPLE 2. *In our framework, the above example is written as follows*

```
overload leq :: Int->Int->Bool where
         leq = primLeqInt
overload leq :: Float->Float->Bool where
         leq = primLeqFloat
```

```
overload ins :: (Leq (a->a->Bool)) => [a]->a->[a] where
         ins [] y = [y]
         ins (x:xs) y = if leq x y then x:(ins xs y)
                        else y:x:xs

rule Leq t ==> t = a->a-> Bool
rule Ins t ==> t = ce->e->ce
rule Ins (ce->e1->ce), Ins (ce->e2->ce) ==> e1 = e2
rule Ins ([a]->b->[a]) ==> b = a
```

The above program introduces (overloaded) definitions for identifiers `leq` and `ins`. Note that our form of overloading is simplified compared to Haskell style type classes. We only overload single identifiers whereas in Haskell related "methods" can be grouped together in a type class. Furthermore, there are no explicit class declarations necessary. The dependencies among overloaded definitions manifests in the type annotation associated to each definition. For example, the definition of `ins` depends on `leq`. This is reflected in the *constrained* type $\forall a.Leq\ (a \to a \to Bool) \Rightarrow [a] \to [a] \to [a]$ of `ins`. The constraint $Leq\ (a \to a \to Bool)$ restricts the set of types variable a is allowed to range over. Note that the predicate symbol *Leq* refers to overloaded identifier `leq`. In our framework we introduce for each overloaded identifier `method` a predicate symbol $Method$.[1] Note that in Haskell we find `instance Leq a => Insert [a] a` instead of `overload ins :: (Leq (a->a->Bool)) => [a]->a->[a]`.

The novelty of our approach is that relationships among overloaded identifiers are defined in terms of the meta-language of Constraint Handling Rules. Each `overload` definition gives rise to a CHR *simplification* rule. The above definitions gives rise to the following set:

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

Rule (Leq1) is derived from the first definition. This rules states that `leq` is defined on type $Int \to Int \to Bool$. A similar property is stated by rule (Leq2). Rule (Ins1) states that `ins` on type $[a] \to a \to [a]$ is defined iff `leq` is defined on type $a \to a \to Bool$. 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.

In addition to CHR simplification rules, we also allow for user-specifiable CHR *propagation* rules. Propagation rules are introduced with the `rule` keyword. The four rules in Example 2 correspond to the CHR propagation rules

$$\begin{array}{rl} \text{(ShLeq)} & Leq\ t \Longrightarrow t = a \to a \to Bool \\ \text{(ShIns)} & Ins\ t \Longrightarrow t = ce \to e \to ce \\ \text{(FD)} & Ins\ (ce \to e1 \to ce), Ins\ (ce \to e2 \to ce) \Longrightarrow e1 = e2 \\ \text{(FDIns1)} & Ins\ ([a] \to b \to [a]) \Longrightarrow b = a \end{array}$$

[1] We could equally well write $(leq :: a \to a \to Bool)$ instead of $Leq\ (a \to a \to Bool)$. However, we prefer our current notation.

The rule (ShLeq) enforces that type of `leq` is always of the form $a \to a \to Bool$, corresponding to declaration in the Haskell class. The rule (ShIns) enforces that the type of `ins` is always of the form $ce \to e \to ce$. Rule (FD) states that the ce uniquely determines e, that is two Ins constraints with the same argument ce must have the the same e argument. This mimics the functional dependency of `Insert` in the Haskell program. The second rule (FDIns1) enforces the functional dependency on the instance Ins ($[a] \to a \to [a]$). As soon as we see the collection type $[a]$ we enforce that the element type must be a.

Operationally, a propagation rule can be read as follows. Whenever we see a set of constraints matching the left hand side of the rule we *propagate* (add) the right-hand side constraints. CHR propagation rules allow us to impose stronger conditions on the set of overloaded definitions. In this way we support a programmable type language.

Type inference in our approach simply consists of collecting the constraints arising from the program, and executing them with respect to the CHR program.

EXAMPLE 3. *Consider the definition*

```
f y x = ins [y] x
```

The inferred type (using Haskell) is `f :: Leq a => a->a->[a]`. *Using our approach type inference proceeds as follows. We collect the constraints from the definition, repeatedly apply CHR rules until no more are applicable. The constraints collected are given in the first line of the derivation below, where* `y` $:: a$, `x` $:: b$, `f` $:: c$ *and* `ins [y] x` $:: d$. *The derivation is*

$$\begin{array}{ll} & \underline{Ins\ ([a] \to b \to d)}, c = a \to b \to d \\ \longrightarrow_{ShIns} & \underline{Ins\ ([a] \to b \to d)}, c = a \to b \to d, [a] \to b \to d = ce \to e \to ce \\ \leftrightarrow & \underline{Ins\ ([a] \to b \to [a])}, c = a \to b \to [a] \\ \longrightarrow_{FDIns1} & \underline{Ins\ ([a] \to a \to [a])}, c = a \to a \to [a] \\ \longrightarrow_{Ins1} & Leq\ (a \to a \to Bool), c = a \to a \to [a] \end{array}$$

In the first step the rule (ShIns) is applied, to the Ins constraint and the equation $[a] \to b \to d = ce \to e \to ce$ is added. In the next step, we simplify the constraints by applying the most general unifier of the new equation to the remainder of the constraints, and eliminating any variables no longer of interest. For the following steps we automatically apply this step. In the next step we apply the the (FDIns1) rule to the Ins constraint, effectively adding that $b = a$. Finally we apply the (Ins) rule to replace the Ins constraint with an Leq constraint. The resulting inferred type `f :: Leq (a->a->Bool) => a->a->[a]` *corresponds to the type inferred by Haskell.*

The example above illustrates how we can encode functional dependency extensions using our approach to overloading. We now illustrate an extension that goes beyond functional dependencies.

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

```
zip ::  [a]->[b]->[(a,b)]
```

```
zip3 ::  [a]->[b]->[c]->[((a,b),c)]
zip4 ::  ...
```

Recently, Firdlender and Indrika [Fridlender and Indrika 2000] showed how to define such a generic family of zip functions in Haskell. Here, we present an alternative definition which nicely takes advantage of our general overloading framework.

EXAMPLE 4. *Consider the following generic definition of the zip function.*

```
zip2 ::  [a]->[b]->[(a,b)]
zip2 [] [] = []
zip2 (a:as) (b:bs) = (a,b):(zip2 as bs)
zip2 [] (b:bs) = []
zip2 (a:as) [] = []

overload zip ::  Zip ([(a,b)]->cs->e) => [a]->[b]->cs->e where
         zip as bs cs = zip (zip2 as bs) cs
overload zip ::  [a]->[b]->[(a,b)]
         zip = zip2

rule Zip ([a]->[b]->[c]) ==> c = (a,b)
rule Zip t ==> t = [a]->[b]->c
```

The corresponding CHRs for the above program are as follows:

$$\begin{array}{lrcl}
\textit{(Zip1)} & Zip\ ([a] \to [b] \to cs \to e) & \Longleftrightarrow & Zip\ ([(a,b)] \to cs \to e) \\
\textit{(Zip2)} & Zip\ ([a] \to [b] \to [(a,b)]) & \Longleftrightarrow & True \\
\textit{(Zip3)} & Zip\ ([a] \to [b] \to [c]) & \Longrightarrow & c = (a,b) \\
\textit{(Zip4)} & Zip\ t & \Longrightarrow & t = [a] \to [b] \to c
\end{array}$$

The role of the rule (Zip3) is to improve the type to allow the rule (Zip2) to be applied as soon as we know the last argument is a list. The role of rule (Zip4) is to immediately add the information that the first two arguments are list types (the common information of the overloaded definitions).

Consider the following expression.

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

We assume `head :: [a]->a` returns the head of a list. From the program text we generate the following constraint, where `e` :: a

$$Zip\ ([Int] \to [Bool] \to [Char] \to [Bool] \to [a])$$

where equalities have been resolved by unification. The type inference derivation is

$$\begin{array}{ll}
 & Zip\ ([Int] \to [Bool] \to [Char] \to [Bool] \to [a]) \\
\longrightarrow_{Zip1} & Zip\ ([(Int, Bool)] \to [Char] \to [Bool] \to [a]) \\
\longrightarrow_{Zip1} & Zip\ ([((Int, Bool), Char)] \to [Bool] \to [a]) \\
\longrightarrow_{Zip3} & Zip\ ([((Int, Bool), Char)] \to [Bool] \to [(((Int, Bool), Char), Bool)]), \\
 & a = (((Int, Bool), Char), Bool) \\
\longrightarrow_{Zip2} & a = (((Int, Bool), Char), Bool)
\end{array}$$

So the type of `e` is inferred as $(((Int, Bool), Char), Bool)$. Without rule (Zip3) type inference gives is $e :: Zip\ ([((Int, Bool), Char)] \to [Bool] \to [a]) \Rightarrow a$.

It is not possible to mimic the above example using functional dependencies since the rule (Zip3) does not correspond to a functional dependency.

We observe that CHR simplification rules allow us to describe dependencies among overloaded definitions whereas CHR propagation rules allow us to impose additional conditions on those overloaded identifiers. In our framework, we require that CHRs must be confluent. Confluence demands that no matter in what order we apply the CHR rules the result is the same. Each of the sets of CHRs in the previous examples is confluent. A non-confluent set of CHRs indicates a possible problem among the set of overloaded identifiers.

EXAMPLE 5. *Consider the following set of Haskell class and instance declarations. For simplicity, we leave out the instance bodies.*

```
instance Eq Int
instance Eq t => Eq [t]
instance Ord [t]
class Eq t => Ord t
```

According to the conditions [Peyton Jones et al. 1999] imposed on Haskell class and instance declarations the above program is not valid since the `Ord [t]` instance does not have a corresponding `Eq [t]` instance. The corresponding Chameleon program is as follows.

```
overload eq :: Int->Int->Bool
overload eq :: Eq (a->a->Bool) => [a]->[a]->Bool
overload ord :: [a]->[a]->Bool
rule Ord t ==> Eq t
```

Each superclass relation such as `class Eq t => Ord t` imposes an additional condition which is translated as a CHR propagation rule. The translation to CHRs is

$$\begin{array}{ll}
\text{(Eq1)} & Eq\ (Int \to Int \to Bool) \Longleftrightarrow True \\
\text{(Eq2)} & Eq\ ([a] \to [a] \to Bool) \Longleftrightarrow Eq\ (a \to a \to Bool) \\
\text{(Ord1)} & Ord\ ([a] \to [a] \to Bool) \Longleftrightarrow True \\
\text{(Super)} & Ord\ t \Longrightarrow Eq\ t
\end{array}$$

This set of CHRs is non-confluent. There are two derivations for constraint $Ord\ ([a] \to [a] \to Bool)$ which do not arrive at the same answer

$$Ord\ ([a] \to [a] \to Bool) \quad \longrightarrow_{Ord1} \quad True$$

and

$$\begin{array}{ll}
 & Ord\ ([a] \to [a] \to Bool) \\
\longrightarrow_{Super} & Ord\ ([a] \to [a] \to Bool), Eq\ ([a] \to [a] \to Bool) \\
\longrightarrow_{Ord1} & Eq\ ([a] \to [a] \to Bool) \\
\longrightarrow_{Eq2} & Eq\ (a \to a \to Bool)
\end{array}$$

This indicates that the overloaded program is problematic.

Confluence implies that the logical theory described by CHR is consistent. Hence, we have a general criteria to describe the validity of the set of overloaded identifiers. Not surprisingly, confluence is an essential assumption for most of our technical results.

Our approach to overloading models an open world understanding of user-defined constraints. This is usual for type class systems since it supports separate compilation.

EXAMPLE 6. *Consider the following Chameleon program defining a common ad-hoc overloading of addition (the actual function definitions are left out for simplicity).*

```
overload plus :: Int->Int->Int
overload plus :: Float->Int->Float
overload plus :: Int->Float->Float
rule Plus (a->b->Int) ==> a = Int, b = Int
rule Plus (a->Int->Float) ==> a = Float
rule Plus (a->Float->Float) ==> a = Int
e = plus (1::Int) True
```

The role of the propagation rules is to propagate type information from the result of `plus` to its arguments. We find that expression `e` has type `Plus (Int->Bool->b) => b` rather than reporting a type error. Indeed, there is nothing that prevents us from adding in

```
overload plus :: Int->Bool->Int
```

at some later stage. Interestingly, we could however use additional CHRs to enforce that no such instance can be added (see e.g. Example 27)

The original ideas of employing CHRs to deal with overloading were first described in [Glynn et al. 2000]. The current paper is an extended version of [Stuckey and Sulzmann 2002]. In particular, we provide concise proofs under which conditions in terms of CHRs type inference is decidable and the semantics given to programs in our framework is coherent. We also give a reformulation of type classes in terms of CHRs. The contributions of this work are:

—We introduce an extension of the Hindley/Milner system in style of type classes with user-definable CHR-based overloading.

—We identify a large class of CHRs (CHRs must be confluent, range-restricted and simplification rules must be single-headed) which enjoy a complete satisfiability check and a complete check for testing equivalence among constraints.

—We define complete procedures for checking *satisfiability* and *subsumption* among constrained types.

—We provide a general definition of *ambiguity* for constrained types, and a complete procedure for checking ambiguity.

—We provide a complete type inference algorithm based on the above three procedures.

—We give a semantic meaning to programs by employing the evidence translation scheme. Under the assumption that CHRs are confluent and simplification rules are single-headed and non-overlapping we can state a general coherence result.

—We give a reformulation of type classes in terms of CHRs.

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. Section 7 shows how our ideas apply to Haskell style type classes. Section 8 discusses extensions of our framework such as overlapping and closed definitions of overloaded identifiers. Section 9 discusses related work. We conclude in Section 10. Proofs of key lemmas and theorems can be found in the Appendix.

2. PRELIMINARIES

In this section, we introduce the following syntactic domains:

Types	$\tau ::= a \mid \tau \rightarrow \tau \mid T\ \bar{\tau}$
Constraints	$C ::= \tau = \tau \mid U\ \bar{\tau} \mid C \wedge C$
Type Schemes	$\sigma ::= \tau \mid \forall \bar{a}.C \Rightarrow \tau$
Substitutions	$\theta ::= [\bar{\tau}/\bar{a}]$
First Order Formulas	$F ::= C \mid F \wedge F \mid F \vee F \mid F \supset F \mid \neg F \mid \forall a\ F \mid \exists a\ F$

We shall be interested in manipulating constraints on types. A *type* is a variable a 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. We write the function type constructor $\cdot \rightarrow \cdot$ infix as usual.

A *primitive constraint* is an equation $\tau_1 = \tau_2$ denoting (syntactic) equality among types, or a user-defined constraint $U\ \tau_1 \ldots \tau_n$ where U is a user-defined n-ary predicate symbol. (In fact we restrict ourselves to unary user-defined constraints). A *constraint* C is a conjunction of primitive constraints. We shall often treat is as a set. Note that we use ',' for conjunction among constraints in CHR rules and example CHR derivations. We use $True$ as an abbreviation for the empty constraint which denotes the true formula. We use $False$ as an abbreviation for $T_1 = T_2$, where T_1 and T_2 are two distinct constructor symbols, which denotes the unsatisfiable equation. Given a constraint C, we use the notation h_C to refer to the set (or conjunction) of equations in C.

We write $\bar{x}$ to denote a sequence of objects x. Sometimes we treat these sequences as sets. A *substitution* $\theta = [\bar{\tau}/\bar{a}]$ simultaneously replaces each a by its corresponding τ. A *unifier* of a constraint 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 $\theta' = \rho(\theta)$.

We assume the reader is familiar with the basics of first-order logic. See for example [Shoenfield 1967]. The syntax for formulae F is given above. Note, we use the $\supset$ symbol to denote logical implication to distinguish it from the function type constructor $\rightarrow$. Formulae F are assumed to be implicitly universally quantified. The statement $\models F$ denotes that F is universally true, while $F_1 \models F_2$ denotes that F_2 holds in any model of F_1.

Let $fv(t)$ take a syntactic term t and return the set of free variables in t. We introduce additional notation for quantification as follows. We let $\forall W\ F$, where W is a set of variables $a_1, \ldots, a_n$, denote $\forall a_1 \cdots \forall a_n F$, similarly for $\exists W\ F$ We let $\tilde{\exists} F$ denote the existential closure of F, that is $\exists fv(F)\ F$. We let $\bar{\exists}_W F$ denote the

formula $\exists a_1 \cdots \exists a_n F$ where $\{a_1, \ldots, a_n\} = fv(F) - W$.

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

We introduce an ordering $\preceq$ among type schemes with respect to some first order formula F. Intuitively $F \vdash \sigma_1 \preceq \sigma_2$ means that σ_1 is a more general type than σ_2 where the meaning of the user-defined constraints in σ_1 and σ_2 is given by F. We define $F \vdash (\forall \bar{a}_1.C_1 \Rightarrow \tau_1) \preceq (\forall \bar{a}_2.C_2 \Rightarrow \tau_2)$ iff $F \models C_2 \supset \exists \bar{a}_1.(C_1 \wedge \tau_1 = \tau_2)$ where we assume there are no name clashes between a_1 and a_2 (i.e. $\bar{a}_1 \cap \bar{a}_2 = \emptyset$) and F is a first-order formula. We say σ_1 *subsumes* σ_2 w.r.t. F if $F \vdash \sigma_1 \preceq \sigma_2$. We define $F \vdash \sigma_1 \simeq \sigma_2$ iff $F \vdash \sigma_1 \preceq \sigma_2$ and $F \vdash \sigma_2 \preceq \sigma_1$. The interested reader is referred to [Sulzmann 2000] where we prove soundness of subsumption among type schemes w.r.t. a standard denotational semantics.

3. CONSTRAINT HANDLING RULES

Constraint handling rules [Frühwirth 1995] (CHRs) are a multi-headed concurrent constraint language for writing incremental constraint solvers. In effect, they define derivation steps from one constraint to an equivalent constraint. Derivation steps serve to simplify constraints and detect satisfiability and unsatisfiability.

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

$$\begin{array}{rrlll} \textbf{Simplification} & \text{(Rule1)} & c_1, \ldots, c_n & \Longleftrightarrow & d_1, \ldots, d_m \\ \textbf{Propagation} & \text{(Rule2)} & c_1, \ldots, c_n & \Longrightarrow & d_1, \ldots, d_m \end{array}$$

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 formulae. The translation function $[\![\cdot]\!]$ from CHR rules to first-order formulae is:

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

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

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

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[2] the left hand side of the rule

[2]The original CHR operational semantics is based on multi-set rewriting, while the logical semantics must treat constraints as sets. We use a set based operational semantics to more closely match the logical semantics.

(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 $C \longrightarrow_r C'$ or $C \longrightarrow_P C'$, see Appendix A for details.

A *derivation*, denoted $C \longrightarrow_P^* C'$ is a sequence of derivation steps using rules in P where no derivation step is applicable to C'. A derivation $C \longrightarrow_P^* C'$ 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 $C \longrightarrow_P^* C'$.

EXAMPLE 7. *Consider the set of CHRs defined in Example 2 in the introduction. Then the following CHR derivation is possible (we underline the constraint matching the left hand side of the rule):*

$$\begin{array}{ll} & Ins\ ([Bool] \to a \to b), \underline{Leq\ (Int \to Int \to a)} \\ \longrightarrow_{ShLeq} & Ins\ ([Bool] \to Bool \to b), \underline{Leq\ (Int \to Int \to Bool)}, a = Bool \\ \longrightarrow_{Leq1} & \underline{Ins\ ([Bool] \to Bool \to b)}, a = Bool \\ \longrightarrow_{ShIns} & \underline{Ins\ ([Bool] \to Bool \to [Bool])}, b = [Bool], a = Bool \\ \longrightarrow_{Ins1} & Leq\ (Bool \to Bool \to Bool), b = [Bool], a = Bool \end{array}$$

Confluence of CHR programs is a vital property. Confluence implies that the order of the derivation steps 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 $C_0 \longrightarrow_P C_1$ and $C_0 \longrightarrow_P C_2$, then there exist derivations $C_1 \longrightarrow_P^* C_3$ and $C_2 \longrightarrow_P^* C_4$ such that C_3 is equivalent (modulo new variables introduced) to C_4, i.e. $\models (\bar{\exists}_{fv(C_0)} C_3) \leftrightarrow (\bar{\exists}_{fv(C_0)} C_4)$.

EXAMPLE 8. *For example, another derivation for the goal in Example 7 is*

$$\begin{array}{ll} & \underline{Ins\ ([Bool] \to a \to b)}, Leq\ (Int \to Int \to a) \\ \longrightarrow_{Ins2} & \underline{Ins\ ([Bool] \to Bool \to [Bool])}, b = [Bool], a = Bool, Leq\ (Int \to Int \to Bool) \\ \longrightarrow_{Ins1} & Leq\ (Bool \to Bool \to Bool), b = [Bool], a = Bool, \underline{Leq\ (Int \to Int \to Bool)} \\ \longrightarrow_{Leq1} & Leq\ (Bool \to Bool \to Bool), b = [Bool], a = Bool \end{array}$$

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 will restrict ourselves to CHRs made up of propagation rules and single-headed simplification rules. Since simplification rules are only used to translate instance definitions, they never need to match multiple constraints, and hence can be single-headed.

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 prevents us from introducing a new unconstrained type variable during a derivation. This is not an onerous condition, for our purposes, but certainly does restrict the class of CHRs we allow.

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 $C \longrightarrow_P^* C'$. Then $\models \tilde{\exists}([\![P]\!] \wedge C)$ iff $\models \tilde{\exists} h_{C'}$.*

A proof can be found in the Appendix.

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. From an overloading perspective this means that we could add some further instances in order to make the constraint universally true in 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 and terminating set of range-restricted CHRs where each simplification rule is single-headed. Then $[\![P]\!] \models D \leftrightarrow D'$ iff $D \longrightarrow_P^* C$ and $D' \longrightarrow_P^* C'$ such that $\models (\bar{\exists}_{fv(D)} C) \leftrightarrow (\bar{\exists}_{fv(D')} C')$.*

A proof can be found in the Appendix.

Note that for this result we require the CHRs P to be terminating, that is $\forall C \exists C' \;\; C \longrightarrow_P^* C'$. 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 [Frühwirth 1998a]. For a terminating set of CHRs we have a decidable confluence test [Abdennadher 1997]. 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 [Sulzmann 2000; Odersky et al. 1999] 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.

$$\begin{array}{ll} \textbf{Programs} & p ::= \mathsf{overload}\ f = e\ \mathsf{in}\ p \mid e \\ \textbf{Expressions} & e ::= x \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid (e :: \sigma) \end{array}$$

We assume that f ranges over overloaded identifiers, For syntactic convenience, we write example programs using

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

We will also make use of pattern matching syntax and recursive functions. The straightforward description of these extensions 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*. Note that in Section 5 we will impose some conditions on P to allow for decidable and complete type inference. Typing judgments are of the form $P, C, \Gamma \vdash e : \tau$ where P is the program theory,

$$\text{(Var)}\ \frac{(x:\sigma)\in\Gamma}{P,C,\Gamma\vdash x:\sigma} \qquad \text{(Annot)}\ \frac{P,C,\Gamma\vdash e:\sigma \quad fv(\sigma)=\emptyset}{P,C,\Gamma\vdash (e::\sigma):\sigma}$$

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

$$\text{(}\forall\text{I)}\ \frac{P,C_1\wedge C_2,\Gamma\vdash e:\tau \quad \bar{a}\notin fv(C_1)\cup fv(\Gamma)}{P,C_1,\Gamma\vdash e:\forall\bar{a}.C_2\Rightarrow\tau} \qquad \text{(}\forall\text{E)}\ \frac{P,C_1,\Gamma\vdash e:\forall\bar{a}.C_2\Rightarrow\tau \quad [\![P]\!]\models C_1\supset[\bar{\tau}/\bar{a}]C_2}{P,C_1,\Gamma\vdash e:[\bar{\tau}/\bar{a}]\tau}$$

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

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

Fig. 1. Typing Rules

C a constraint, Γ a type 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 $\forall\bar{a}.C\Rightarrow\tau$ are weakly satisfiable. Recall that weak satisfiability implies the constraint is satisfiable in some model. Therefore, 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 Γ_x denotes the typing environment obtained from Γ by excluding the variable x.

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

Our formulation of rule ($\forall$I) follows [Jones 1992]. 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. The standard HM(X) quantifier introduction rule keeps the constraint $\exists\bar{a}.C_2$ on the left-hand side. This has some advantages as discussed in [Sulzmann 2000]. 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 lies in rule (Over) which introduces overloaded identifiers (note that overload definitions 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 $f :: \forall a.F\ a \Rightarrow a$ is part of some initial type environment.

Note that we require that overloaded definitions are closed and are annotated with their type. Each `overload` 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. These are the only simplification rules appearing in the program theory P.

EXAMPLE 9. *Consider parts of the program in Example 2 from the Introduction, but with a different typing.*

```
overload leq ::  (a=Bool) => Int->Int->a where
         leq = primLeqInt
overload ins ::  [Int]->Int->[Int] where
         ins = ...
```

Note that we do not impose any conditions on constraints appearing in type schemes besides being weakly satisfiable. Therefore, $\forall a.a = Bool \Rightarrow Int \to Int \to a$ is perfectly valid.

The above definitions give rise to the following set P'_s of CHRs:

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

Note that the first definition does not generate the following rule:

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

The side conditions in rule (Over) (we build an mgu of all constraints and apply the mgu to the right hand side of the CHR we generate) prevent equality constraints appearing on the right hand side of simplification CHRs.

Recall the set P_s of CHRs arising from Example 2:

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

We find that the set P_s subsumes P'_s. The more general typing in Example 2 allows for a larger set of overloaded definitions. That is, the set of available definitions depends on a programs typing.

In addition to CHRs arising from overload definitions, the user can provide CHR propagation rules to impose stronger conditions on the set of constraints allowed to appear.

EXAMPLE 10. *Recall the definitions of Example 6 from the introduction.*

```
overload plus :: Int->Int->Int
overload plus :: Float->Int->Float
overload plus :: Int->Float->Float
```

```
rule Plus (a->b->Int) ==> a = Int, b = Int
rule Plus (a->Int->Float) ==> a = Float
rule Plus (a->Float->Float) ==> a = Int
```

The propagation rules strengthen the constraints. The initial type generated out of the expression

```
f x y z = (x::Int) == (plus y z)
```

is given below, where y :: a, z :: b,

$$Plus(a \to b \to Int) \Rightarrow Int \to a \to b \to Bool$$

The propagation rule $Plus(a \to b \to Int) \Longrightarrow a = Int, b = Int$ *in combination with* `overload plus :: Int->Int->Int` *allows us to strengthen this to* $Int \to Int \to Int \to Bool$.

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 is an essential requirement to ensure a well-defined semantics for programs. Programs with ambiguous types can lead to operationally nondeterministic behavior (see Example 19 later).

The original definition of unambiguity was as follows. Let $\forall \bar{a}.C \Rightarrow \tau$ be a type scheme, it is unambiguous if $fv(C) \cap \bar{a} \subseteq fv(\tau) \cap \bar{a}$. This means that all variables are fixed if we fix the type τ. This definition had to be extended when functional dependencies were introduced. Our introduction captures the intuitive definition of unambiguity, that is, all variables appearing in C are fixed if all variables in τ are fixed, taking into account the program theory. It naturally extends both previous definitions.

Let P be the program theory and ρ be a variable renaming on $\bar{a}$. Then $\forall \bar{a}.C \Rightarrow \tau$ is *unambiguous* iff $[\![P]\!] \models (C \wedge \rho(C) \wedge (\tau = \rho(\tau))) \supset (a = \rho(a))$ for each $a \in \bar{a}$. This simply says that all variables appearing in the constraint component can be uniquely determined from the type component w.r.t. program theory P. We also say that e is unambiguous if $e :: \sigma$ is a well-typed expression and σ is unambiguous. Henceforth, all valid type schemes must be unambiguous in addition to weakly satisfiable.

EXAMPLE 11. *Consider the type scheme* $\forall a, b.H\ (a \to b) \Rightarrow b$. *Under the empty program theory this type scheme is ambiguous. The variable* a *cannot be determined from the constraint component alone. We find that* $\models H\ (a \to b) \wedge H\ (a' \to b') \wedge b = b' \not\supset a = a'$. *Assume that our program theory consists of the following CHR (note that this CHR mimics a functional dependency):*

$$(FH)\ H\ (a \to b), H\ (a' \to b) \implies a = a'$$

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

Our definition of unambiguity subsumes previous definitions. More discussion on this topic can be found in Section 6.

4.2 Confluence

We require that program theories must be confluent. A non-confluent program theory indicates a possible problem among the set of overloaded definitions. Example 5 in the introduction illustrates this behavior.

We generally require that the programmer provides a confluent set of CHR rules. This can be checked automatically (assuming the CHR rules are terminating). However, this can be burdensome. There are cases where it is safe to add some propagation rules to "complete" a non-confluent program theory to become confluent.

EXAMPLE 12. *Consider the following program*

```
overload ins :: [Int]->Int->[Int] where
         ins xs x = x :  xs
rule Ins (ce->e1->ce), Ins (ce->e2->ce) ==> e1=e2
```

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

$$\begin{array}{rl} \textit{(Ins1')} & \mathit{Ins}\ ([\mathit{Int}] \to \mathit{Int} \to [\mathit{Int}]) \iff \mathit{True} \\ \textit{(FD)} & \mathit{Ins}\ (ce \to e_1 \to ce), \mathit{Ins}\ (ce \to e_2 \to ce) \implies e_1 = e_2 \end{array}$$

The above program theory is non-confluent.

$$\begin{array}{rl} & \mathit{Ins}\ ([\mathit{Int}] \to \mathit{Int} \to [\mathit{Int}]), \mathit{Ins}\ ([\mathit{Int}] \to a \to [\mathit{Int}]) \\ \longrightarrow_{Ins1'} & \mathit{Ins}\ ([\mathit{Int}] \to a \to [\mathit{Int}]) \end{array}$$

and

$$\begin{array}{rl} & \mathit{Ins}\ ([\mathit{Int}] \to \mathit{Int} \to [\mathit{Int}]), \mathit{Ins}\ ([\mathit{Int}] \to a \to [\mathit{Int}]) \\ \longrightarrow_{FD} & \mathit{Ins}\ ([\mathit{Int}] \to \mathit{Int} \to [\mathit{Int}]), \mathit{Ins}\ ([\mathit{Int}] \to \mathit{Int} \to [\mathit{Int}]), a = \mathit{Int} \\ \longrightarrow_{Ins1'} & \mathit{Ins}\ ([\mathit{Int}] \to \mathit{Int} \to [\mathit{Int}]), a = \mathit{Int} \\ \longrightarrow_{Ins1'} & a = \mathit{Int} \end{array}$$

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

$$\text{(FDIns1')}\ \ \mathit{Ins}\ ([\mathit{Int}] \to a \to [\mathit{Int}]) \implies a = \mathit{Int}$$

Note that rule (FD) 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 guarantees that the types of all expressions are well defined. It does not necessarily guarantee that correctness on the value level.

EXAMPLE 13. *Consider the definition of* eq *of Example 5 extended with the following definition.*

```
overload eq :: [Int]->[Int]->Bool where
         eq = ...  special treatment on integers ...
```

Among others, the program theory contains the following CHRs:

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

The above three rules are 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 8.1). Confluence also provides us with a general notion to speak about the validity of Haskell class and instance declarations (see Section 7.2).

5. TYPE INFERENCE

We assume that we are given a program p and an initial environment Γ 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_s of simplification out of the annotated program text via a simple translation. We introduce judgments of the form $p \vdash_{inf} P_s$:

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

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

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. The situation seems similar to the problem of providing annotations in case of polymorphic recursion [Henglein 1993]. Hence, we believe it is undecidable to infer type annotations of overloaded definitions such that the resulting CHRs are terminating.

To obtain complete type inference we require that P is terminating, confluent and range-restricted. The upcoming soundness and completeness results rely on Lemmas 1 and 2 which assume that CHRs are terminating, confluent and range-restricted. We generally assume that CHRs satisfy these three properties. Range-restrictedness can easily verified. Grounding all variables on the left-hand side must ground all variables on the right-hand side. Note that any termination check for CHRs must be necessarily incomplete. Given a terminating set of CHRs we can decide the confluence of this set [Abdennadher 1997]. We simply build "critical pairs" and test whether they are joinable.

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

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

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

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

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

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

Fig. 2. Inference Rules

Stage (2) of type inference proceeds by inferring the type 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. Some readers might prefer to write $P, \Gamma \vdash_{inf} p : (C \mid \tau)$ instead. However, this notation hides which values are input and which 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. Note that in our

formulation substitutions are expressed by equality constraints, e.g. see rule (App).

Rules (Annot), (Let), and (Over) use a generalization procedure. Let Γ be a type environment, C a constraint and τ a type. Then, we define $gen(\Gamma, C, \tau) = \forall \bar{a}.C \Rightarrow \tau$ where $\bar{a} = fv(C, \tau) \backslash fv(\Gamma)$. Note that we push the whole constraint C into the type scheme. We could be more efficient by pushing in only the *affected* constraints. That is, we split C into two parts. C_1 consists of all constraints in C which have a type variable in common with $\bar{a}$. C_2 consists of the remaining constraints. We define $gen'(\Gamma, C, \tau) = (C_2, \forall \bar{a}.C_1 \Rightarrow \tau)$. For simplicity, we only show the necessary adjustments for the (Let) rule.

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

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

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

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 (Annot) and (Let) enforce unambiguity of type schemes. The procedure for checking of unambiguity of type schemes is defined as follows:

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

Note that w.l.o.g. we assume that $\bar{a} \subseteq fv(C, \tau)$. The condition $\models C' \supset (a = \rho(a))$ is decidable (it holds iff the mgu of $h_{C'}$ unifies a and $\rho(a)$) which ensures that the above procedure is decidable for terminating CHRs.

EXAMPLE 14. *Consider the type scheme* $\forall a, a'.H\ (a \to a') \Rightarrow a'$. *The program theory consists of rule (FH) (from Example 11)*

$$(FH)\ H\ (a \to b), H\ (a' \to b) \implies a = a'$$

We assume that $\rho(a) = a''$ *and* $\rho(a') = a'''$. *We check unambiguity via the derivation*

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

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. We also do not explicitly enforce unambiguity of σ in rule (Over). In case σ is ambiguous the resulting CHR will not be range-restricted. This is a sufficient condition for completeness but not a necessary condition for soundness.

In rule (Annot) procedure *subsumes* performs a subsumption check to check the validity of the annotated type. Note that we also ensure that the inferred type is unambiguous. This is a necessary condition for completeness of subsumption. Unambiguity of the annotated type is necessary to ensure a well-defined semantics.

The definition of *subsumes* is as follows:

$$
\begin{array}{ll}
& \mathit{subsumes}(P, \forall \bar{a}.C \Rightarrow \tau, \forall \bar{a}'.C' \Rightarrow \tau') \\
= & \mathit{True} \quad \text{if } C' \wedge \tau' = a' \wedge a = a' \longrightarrow_P^* C_1, \text{ and} \\
& \qquad\quad C' \wedge \tau' = a' \wedge a = a' \wedge \tau = a \wedge C \longrightarrow_P^* C_2 \\
& \qquad\quad \text{such that } \models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2) \\
& \qquad\quad \text{where } a,\ a' \text{ are new variables and} \\
& \qquad\quad V = \mathit{fv}(C' \wedge \tau' = a' \wedge a = a') \cup \mathit{fv}(\forall \bar{a}.C \Rightarrow \tau) \\
= & \mathit{False} \quad \text{otherwise}
\end{array}
$$

The idea is to rewrite $\forall \bar{a}.C \Rightarrow \tau$ into the equivalent type scheme $\forall \bar{a}, a.C \wedge \tau = a \Rightarrow a$ where a is a new variable, and then use equivalence testing (possible through Lemma 2) to test implication. We will see that it is important to rewrite the type scheme (see upcoming Example 16). Note that the condition $\models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2)$ is decidable. We check that $\models (\bar{\exists}_V h_{C_1}) \leftrightarrow (\bar{\exists}_V h_{C_2})$ holds. Furthermore, user-defined constraints in $\phi_1 C_1$ and $\phi_2 C_2$ must be renamings of each other (modulo variables V), where ϕ_1 is a unifier of h_{C_1} and ϕ_2 is a unifier of h_{C_2}.

EXAMPLE 15. *Consider the inference for the second definition of ins in Example 9. The inferred type for the expression is* $\forall b.Leq\ (b \to b \to Bool) \Rightarrow [b] \to [b] \to Bool$, *while the declared type is* $[Int] \to [Int] \to Bool$. *The subsumption test determines*

$$
\begin{array}{rl}
& [Int] \to [Int] \to Bool = a', a = a' \\
\longrightarrow_P^* & [Int] \to [Int] \to Bool = a' = a
\end{array}
$$

and

$$
\begin{array}{ll}
& [Int] \to [Int] \to Bool = a', a = a', [b] \to [b] \to Bool = a, Leq\ (b \to b \to Bool) \\
\leftrightarrow & [Int] \to [Int] \to Bool = a' = a, b = Int, Leq\ (Int \to Int \to Bool) \\
\longrightarrow_{Leq1} & [Int] \to [Int] \to Bool = a' = a, b = Int
\end{array}
$$

Hence the subsumption condition holds.

5.1 Soundness Results

We can state that procedures *unambig* and *subsumes* are sound. Proofs can be found in the Appendix. Soundness of *sat* follows from Lemma 1.

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

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

Note that the introduction of variables a and a' is necessary. Testing for equivalence among normal forms of $C' \wedge \tau' = \tau$ and $C' \wedge \tau' = \tau \wedge C$ would be unsound.

EXAMPLE 16. *Consider $\sigma = \forall c.(c, Int)$ and $\sigma' = \forall b.(Int, b)$. The program theory is assumed to be empty. We have that $\not\vdash \sigma \preceq \sigma'$, however, the subsumption check without the extra variables succeeds. In such a case, we find final stores $c = Int, Int = b$ and $c = Int, Int = b$ which are both logically equivalent.*

With the correct test we have

$$(Int, b) = a, a = a' \quad \longrightarrow \quad (Int, b) = a, a = a'$$
$$(Int, b) = a, a = a', (c, Int) = a' \quad \longrightarrow \quad c = Int, b = Int, a = (Int, Int), a = a'$$

where

$$\not\models ((Int, b) = a, a = a') \quad \leftrightarrow \quad (\exists c.c = Int, b = Int, a = (Int, Int), a = a')$$

That is, the subsumption check (correctly) reports failure.

The above example shows that it is necessary to effectively rewrite $\forall \bar{a}.C \Rightarrow \tau$ into $\forall \bar{a}.C \wedge \tau = a \Rightarrow a$ for the purposes of the subsumption test.

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

THEOREM 5 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 $p \vdash_{inf} P_s$ and $P_s \cup P_p, \Gamma, e \vdash_{inf} (C \mid \tau)$. 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. Proofs of the following two lemmas can be found in the Appendix.

LEMMA 6 COMPLETENESS OF UNAMBIGUITY. *Let P be a confluent set of range-restricted CHRs where each simplification rule is single-headed, $\forall \bar{a}.C \Rightarrow \tau$ be a type scheme and ρ be a variable renaming on $\bar{a}$ such that $[\![P]\!] \models (C \wedge \rho(C) \wedge (\tau = \rho(\tau))) \supset (a = \rho(a))$ for each $a \in \bar{a}$. Then $C \wedge \rho(C) \wedge \tau = \rho(\tau) \longrightarrow_P^* C'$ where $\models C' \supset (a = \rho(a))$ for each $a \in \bar{a}$.*

From Lemma 2 we can derive completeness of subsumption 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 7 COMPLETENESS OF SUBSUMPTION. *Let P be a terminating, confluent set of range-restricted CHRs whose simplification rules are single-headed, a, a' two fresh variables and $\sigma = \forall \bar{a}.C \Rightarrow \tau$ and $\sigma' = \forall \bar{a}'.C' \Rightarrow \tau'$ such that $[\![P]\!] \vdash \sigma \preceq \sigma'$ and σ is unambiguous. Then $C' \wedge \tau' = a' \wedge a = a' \longrightarrow_P^* C_1$ and $C' \wedge \tau' = a' \wedge a = a' \wedge C \wedge \tau = a \longrightarrow_P^* C_2$ such that $\models (\bar{\exists}_V C_1) \leftrightarrow (\bar{\exists}_V C_2)$, where $V = fv(\sigma) \cup fv(\sigma') \cup fv(C' \wedge a = a')$.*

Range-restrictedness and unambiguity of type scheme σ are necessary conditions.

EXAMPLE 17. *Consider the following program theory P:*

$$\begin{array}{rcl} Q & \Longleftrightarrow & R\ a \\ R\ Int & \Longleftrightarrow & True \end{array}$$

Note that the first rule is not range-restricted. Variable a appears on the right hand side but not on the left hand side. We have that $[\![P]\!] \vdash (\forall c.Q \Rightarrow c) \preceq Int$. Note that $[\![P]\!] \models Q$. However, the subsumption check fails. We find that

$$Q, c = a', a = a' \longrightarrow^* R\ d, c = a', a = a'$$

and

$$Q, c = a', a = a', Int = a \longrightarrow^* R\ d', c = Int, a' = Int, a = Int$$

Note that the two final stores (constraints on right hand side of $\longrightarrow^*$) are not logically equivalent. We have that

$$\not\models \forall c, a, a'.\exists d, d'.((R\ d, c = a', a = a') \leftrightarrow (R\ d', c = Int, a' = Int, a = Int))$$

EXAMPLE 18. *Consider*

$$\begin{array}{rcl} Q\ (Int, Bool) & \Longleftrightarrow & True \\ Q\ (Int, Float) & \Longleftrightarrow & True \end{array}$$

We have that $[\![P]\!] \models (\forall a, b.Q\ (b, c) \Rightarrow b) \preceq Int$. Note that $\forall b, c.Q\ (b, c) \Rightarrow b$ is ambiguous. In this case, we find that

$$Q\ (b, c), b = a', a = a'$$

and

$$Q\ (b, c), b = a', a = a', Int = a$$

are not logically equivalent. Therefore, the subsumption check fails.

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. Therefore, we only 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) $[\![P]\!] \wedge C' \vdash \sigma \preceq \sigma'$. 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 8 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. [Wadler and Blott 1989]) 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. The novelty of our approach is that evidence can be constructed out of a CHR derivation. This section may be opaque for readers not familiar with the dictionary-passing implementation of evidence [Wadler and Blott 1989], but can be skipped.

We motivate our approach with an example. Consider the definitions of `eq` below.

```
overload eq :: Int->Int->Bool where
         eq = primEqInt
overload eq :: (Eq (a->a->Bool)) => [a]->[a]->Bool where
         eq [] [] = True
         eq (x:xs) [] = False
         eq [] (y:ys) = False
         eq (x:xs) (y:ys) = (eq x y) && (eq xs ys)

rule Eq x ==> x=a->a->Bool
```

The program theory consists of the following CHRs:

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

Rules (Eq1) and (Eq2) arise from the two overloaded definitions above. Rule (Eq3) enforces that the two arguments of `eq` must be of equal type and the result is of type `Bool`.

We consider the evidence translation of the following expression.

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

We assume `tail :: [a]->[a]` takes the tail of a list. The evidence translation is driven by a programs typing. Among others, expression `exp` gives rise to the following constraints arising from the two application sites of overloaded identifier `eq`.

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

We assume `xs` $:: [a]$, `ys` $:: b$, `1` $:: Int$ and `3` $:: Int$. We perform type inference by CHR solving.

$$\begin{array}{rl} & Eq\ ([a] \to b \to c), Eq\ (Int \to Int \to d) \\ \longrightarrow_{Eq3} & Eq\ ([a] \to b \to c), Eq\ (Int \to Int \to Bool), d = Bool \\ \longrightarrow_{Eq1} & Eq\ ([a] \to b \to c), d = Bool \\ \longrightarrow_{Eq3} & Eq\ ([a] \to [a] \to Bool), b = [a], c = Bool, d = Bool \\ \longrightarrow_{Eq2} & Eq\ (a \to a \to Bool), b = [a], c = Bool, d = Bool \end{array}$$

Resolution of remaining equalities via unification yields the following type inference result.

```
exp :: Eq (a->a->Bool) => [a]->[a]->(Bool,Bool)
exp xs ys = (eq (tail xs) ys, eq (1::Int) (3::Int))
```

What remains is to replace the two occurrences of `eq` in the body of `exp` by some appropriate evidence value. Fortunately, evidence can be constructed out of the above CHR derivation. The left most occurrence gave rise to the constraint $Eq\ ([a] \to b \to c)$ and the following CHR derivation.

$$\begin{array}{rl} & Eq\ ([a] \to b \to c) \\ \longrightarrow_{Eq3} & Eq\ ([a] \to [a] \to Bool), b = [a], c = Bool \\ \longrightarrow_{Eq2} & Eq\ (a \to a \to Bool), b = [a], c = Bool \end{array}$$

In fact, we are only interested in CHR simplification steps. We apply the mgu of all equality constraints found in the final store to the initial store and re-run the CHR solver.

$$\begin{array}{rl} & Eq\ ([a] \to [a] \to Bool) \\ \longrightarrow_{Eq2} & Eq\ (a \to a \to Bool) \end{array}$$

The final store tells us which evidence values are necessary to construct evidence values in the initial store. Construction of evidence is performed by reading the CHR derivation backwards. Assume that e_C refers to the evidence representation of constraint C. Then, we can construct evidence $e_{Eq\ ([a]\to[a]\to Bool)}$ by applying the "evidence constructor" associated to (Eq2) to $e_{Eq\ (a\to a\to Bool)}$. In detail, assume that evidence constructor associated to rule (Eq2) is as follows.

```
ecEqList e [] [] = True
ecEqList e (x:xs) [] = False
ecEqList e [] (y:ys) = False
ecEqList e (x:xs) (y:ys) = (e x y) && (ecEqList e xs ys)
```

Note that parameter `e` represents evidence for the equality function on type $a \to a \to Bool$. Then, we can replace the left most occurrence of `eq` in `exp` by `ecEqList e'` where `e'` represents the evidence parameter of `exp`. Similarly, we find the following CHR derivation for the right most occurrence of `eq`.

$$Eq\ (Int \to Int \to Bool) \longrightarrow_{Eq1} True$$

Therefore, we replace the right most occurrence of `eq` by `ecEqInt` where

```
ecEqInt = primEqInt
```

In summary, the evidence translation of `exp` yields the following result.

```
expT :: (a->a->Bool)->[a]->[a]->(Bool,Bool)
expT e' xs ys = (ecEqList e' (tail xs) ys, ecEqInt (1::Int) (3::Int))
```

Evidence construction becomes a subtle issue in the presence of type annotations. Consider the following expression.

```
exp2 :: Eq ([a]->[a]->Bool) => a->a->Bool
exp2 x y = eq x y
```

According to our type inference rules the above expression is type correct. Inference yields type $\forall a.Eq\ (a \to a \to Bool) \Rightarrow a \to a \to Bool$. Indeed, we have that

$$\begin{array}{rl} [\![P]\!] \vdash & (\forall a.Eq\ (a \to a \to Bool) \Rightarrow a \to a \to Bool) \\ & \preceq \\ & (\forall a.Eq\ ([a] \to [a] \to Bool) \Rightarrow a \to a \to Bool) \end{array}$$

where P consists of rules (Eq1–3). The question is how can we construct evidence for $Eq\ (a \to a \to Bool)$ (as requested in the body of `exp2`) given that we provide only evidence for $Eq\ ([a] \to [a] \to Bool)$ (as stated by the annotation)? It seems that it should be rather the other way around! The important insight is that in such a case we simply cannot provide a "compilation" scheme. However, we can construct evidence once we apply `exp2` to some monomorphic arguments. That is, evidence construction is postponed until run-time. Hence, in our approach evidence construction is similar to method look-up in object-oriented languages. Our translation scheme yields

```
exp2T e x y = (ec e) x y
```

where ec :: $Eq\ ([a] \to [a] \to Bool) \xrightarrow{P} Eq\ (a \to a \to Bool)$ is a "function" constructing evidence for $Eq\ (a \to a \to Bool)$ provided we have evidence for $Eq\ ([a] \to [a] \to Bool)$. The program theory P consists of rules (Eq1-3). Note that $[\![P]\!] \models Eq\ ([a] \to [a] \to Bool) \supset Eq\ ([a] \to [a] \to Bool)$. Assume we apply `expT` to some monomorphic arguments of type t, i.e. $fv(t) = \emptyset$. We find that $Eq\ ([t] \to [t] \to Bool) \longrightarrow_P^* True$. Hence, $Eq\ (t \to t \to Bool) \longrightarrow_P^* True$. In fact, we even know that $Eq\ (t \to t \to Bool) \longrightarrow_{P_s}^* True$. That is, $Eq\ (t \to t \to Bool)$ can be reduced to $True$ by applying simplifications rules only. Now we are in the position to construct evidence for $Eq\ (t \to t \to Bool)$ by reading the CHR derivation $Eq\ (t \to t \to Bool) \longrightarrow_{P_s}^* True$ backwards. We maintain that function ec is only defined for monomorphic arguments. This is sufficient for our purposes. Details can be found in Section 6.1.

The ultimate goal of this section is to establish a general *coherence* result [Breazu-Tannen et al. 1990]. That is, the semantic meaning of a translated expression should be independent of its typing. For example, assume we provide another definition of `eq`.

```
overload eq :: [Int]->[Int]->Bool where
         eq _ _ = True
```

Two definitions are now applicable on values of type list of integers. Note that the program theory becomes overlapping. There are two rules applicable to resolve the constraint $Eq\ ([Int] \to [Int] \to Bool)$. We can circumvent such problems by requiring program theories must be confluent and all simplification rules are single-headed and non-overlapping.. Details of the coherence result can be found in Section 6.2.

6.1 Translation and Evidence Construction

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

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

$$\mathcal{V} \;=\; \mathbf{W}_\perp \;+\; \mathcal{V} \to \mathcal{V} \;+\; \textstyle\sum_{k \in \mathcal{K}} \left(k\, \mathcal{V}_1 \ldots \mathcal{V}_{\mathrm{arity}(k)}\right)_\perp$$

$$\begin{array}{lcl}
[\![x]\!]\eta & = & \eta(x) \\
[\![\lambda u.e]\!]\eta & = & \lambda v.[\![e]\!]\eta[u := v] \\
[\![e\, e']\!]\eta & = & \mathsf{if}\, [\![e]\!]\eta \in \mathcal{V} \to \mathcal{V} \\
 & & \quad \mathsf{then}\, ([\![e]\!]\eta)\, ([\![e']\!]\eta) \\
 & & \quad \mathsf{else}\, \mathbf{W} \\
[\![\mathsf{let}\, x = e\, \mathsf{in}\, e']\!]\eta & = & [\![e']\!]\eta[x := [\![e]\!]\eta]
\end{array}$$

$$\begin{array}{lcl}
[\![\mu_1 \to \mu_2]\!] & = & \{\; f \in \mathcal{V} \to \mathcal{V} \mid v \in [\![\mu_1]\!] \Rightarrow f\, v \in [\![\mu_2]\!] \;\} \\
[\![T\, \mu_1 \ldots \mu_m]\!] & = & \{\perp\} \cup \bigcup \{k\, [\![\mu'_1]\!] \ldots [\![\mu'_n]\!] \mid \\
 & & \qquad True, \Gamma_{init} \vdash k : \mu'_1 \to \ldots \to \mu'_n \to T\, \mu_1 \ldots \mu_m\} \\
[\![\forall \bar{a}.\tau]\!] & = & \bigcap_{\bar{\mu}} [\![[\bar{\mu}/\bar{a}]\tau]\!]
\end{array}$$

where for monotypes μ we require that $fv(\mu) = \emptyset$ and Γ_{init} contains the set of value constructors used in this context.

Fig. 3. Expression Semantics

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 follow [Odersky et al. 1995] by interpreting target expressions in an untyped denotational semantics. See Figure 3 for details. Note that $+$ and $\sum$ denote coalesced sums and $\mathcal{V} \to \mathcal{V}$ is the continuous function space. In general, we leave injection and projection operators for sums implicit. The value $\mathbf{W}$ is the error element and $\mathcal{K}$ is the set of value constructors. In particular, we assume the availability of the family of tuple constructors $(\ldots)_n :: \forall \bar{a}.a_1 \to \ldots \to a_n \to (a_1, \ldots, a_n)$ which can easily be provided by choice of $\mathcal{K}$. The corresponding projection operators $ith :: \forall \bar{a}.(a_1, \ldots, a_n) \to a_i$ are recorded in some initial type environment. We assume that variable environments η map variables to elements in $\mathcal{V}$ We will always assume that $\eta(e_{True}) = ()$ and $\eta(e_{\tau=\tau}) = ()$ where $() \in K$ with arity 0.

Variable environments η must satisfy program theories P. We define η *satisfies* P iff for each user-defined atom $U\ \tau$ where $U\ \tau \longrightarrow_P^* True$ we have that $\eta(e_{U\ \tau}) \in [\![\forall fv(\tau).\tau]\!]$. That means, evidence values of user-defined constraints must satisfy the given type specification. For example, $Eq\ (Int \to Int \to Bool) \longrightarrow_P^* True$, therefore, $\eta(e_{Eq\ (Int \to Int \to Bool)}) \in [\![Int \to Int \to Bool]\!]$ where P consists of the above rules (Eq1-3).

Let $C = U_1\ \tau_1, \ldots, U_n\ \tau_n$ be a constraint such that $C \longrightarrow_P^* True$. Then, we require that $\eta(e_C) = (\eta(e_{U_1\ \tau_1}), \ldots, \eta(e_{U_n\ \tau_n}))$ as a further condition for variable environments. Note that we assume an ordering on $U_1\ \tau_1, \ldots, U_n\ \tau_n$.

Let C_1 and C_2 be two constraints, η a variable environment and P be a set

of CHRs such that η satisfies P and $[\![P]\!] \models C_1 \supset C_2$. We say ec is an *evidence constructor* iff for each substitution ϕ such that $\phi C_1 \longrightarrow_P^* True$ we have that $\eta(ec)\ (\eta(e_{\phi C_1})) = \eta(e_{\phi C_2})$. In such a situation we write $ec :: C_1 \stackrel{P}{\rightarrow} C_2$. Note that we only provide an interpretation of ec on ground instances.

As already mentioned, a closed definition for an evidence constructor might not necessarily exist. Recall

$$ec ::\ Eq\ ([a] \rightarrow [a] \rightarrow Bool) \stackrel{P}{\rightarrow} Eq\ (a \rightarrow a \rightarrow Bool)$$

We cannot give a closed definition for ec. Evidence can only be constructed once we know the grounding substitution. However, we can classify a class of evidence constructing function for which we can provide a compilation scheme, i.e. closed definition.

LEMMA 9 COMPILATION OF EVIDENCE. *Let P be a confluent set of CHRs where each simplification rule is single-headed and non-overlapping and η a variable environment which satisfies P. Let C_1 and C_2 be two sets of user-defined constraints such that $C_1, C_2 \longrightarrow_P^* C_1$. Then, there exists a closed definition for $ec ::\ C_1 \stackrel{P}{\rightarrow} C_2$.*

A proof can be found in the Appendix. Note that in case of $ec :: True \stackrel{P}{\rightarrow} C$ we can always provide a closed definition of ec

For simplicity, we assume that overloaded definitions are prerecorded in the variable environment η. Consider a overloaded definition

$$\mathsf{overload}\ f :: (\forall \bar{a}.U_1\ \tau_1 \wedge \cdots \wedge U_n\ \tau_n \Rightarrow \tau) = e$$

We assume that we find

$$ec_{F\ \tau} :: U_1\ \tau_1, \ldots, U_n\ \tau_n \stackrel{P}{\rightarrow} F\ \tau$$

In Figure 4, we introduce judgments of the form $P, \Gamma, C \vdash e : \sigma \rightsquigarrow E$ 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. Rule (Over) is a special instance of rule ($\forall$E) to deal with overloaded identifiers. We assume that $ec : C_1 \stackrel{P}{\rightarrow} [\bar{\tau}/\bar{a}]C_2$ is an evidence constructing function. Note that the context only provides evidence for e_{C_1}. However, the instantiation site requires evidence $e_{[\bar{\tau}/\bar{a}]C_2}$. The premise states that $[\![P]\!] \models C_1 \supset [\bar{\tau}/\bar{a}]C_2$. In fact, this is sufficient to ensure that function ec must exist. Assume we have a substitution ϕ such that $\phi C_1 \longrightarrow_P^* True$. Then, we also have that $\phi[\bar{\tau}/\bar{a}]C_2 \longrightarrow_P^* True$. In such a situation, we find that $\phi[\bar{\tau}/\bar{a}]C_2 \longrightarrow_{P_s}^* True$, i.e. $\phi[\bar{\tau}/\bar{a}]C_2$ can be solely reduced by simplification rules. Therefore, evidence $e_{\phi[\bar{\tau}/\bar{a}]C_2}$ can be constructed by reading the CHR derivation backwards.

Note that our translation scheme requires run-time type information (the type of monomorphic arguments). This is implicitly recorded in the typing derivation which drives the translation process.

For simplicity, we leave out the translation of type-annotated expressions. Their treatment is similar to rules ($\forall$I) and ($\forall$E).

$$\text{(Var)}\ \frac{(x:\sigma)\in\Gamma}{P,C,\Gamma\ \vdash\ x:\sigma\rightsquigarrow x} \qquad \text{(Over)}\ \frac{\begin{array}{c}P,C,\Gamma\ \vdash\ f:\forall a.F\ a\Rightarrow a\\ [\![P]\!]\models C\supset F\ \tau\\ ec::C\overset{P}{\rightarrow}F\ \tau\end{array}}{P,C,\Gamma\ \vdash\ f:\tau\rightsquigarrow ec\ e_C}$$

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

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

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

Fig. 4. Evidence Translation Rules

6.2 Coherence

We establish a coherence result under the assumption that CHRs are confluent and simplification rules are single-headed and non-overlapping, and the principal typing derivation is unambiguous. In previous work [Jones 1993a], coherence has been established under the assumption that for each type scheme $\forall\bar{a}.C\Rightarrow\tau$ we have that $fv(C)\cap\bar{a}\subseteq fv(\tau)\cap\bar{a}$. This means that we can determine from the type component alone, each of the types occurring in the constraint part. The recent addition of functional dependencies [Jones 2000] to Haskell made it necessary to adjust the unambiguity condition. Our notion of unambiguity (see Section 4.1) subsumes previous definitions.

The following lemma is crucial in establishing coherence. We need to ensure that evidence values can unambiguously be constructed. We define $\phi\leq\phi'$ iff there exists ψ such that $\psi.\phi=\phi'$. We denote by $\sqcup$ the least upper bound among substitutions.

LEMMA 10 UNIQUENESS. *Let P be a confluent set of CHRs whose simplification rules are single-headed, $\forall\bar{a}.C\Rightarrow\tau$ an unambiguous type scheme, ϕ a mapping from type variables $fv(C,\tau)$ to ground types and ϕ' a mapping from type variables $fv(\tau)$ to ground types such that $\phi'\leq\phi$ and $\phi C\longrightarrow_P^* True$. Then $C,\phi'\longrightarrow_P^*\phi''$ for some ϕ'' such that $\phi'\sqcup\phi''=\phi$.*

A proof can be found in the Appendix.

Note that an ambiguous type scheme indicates that we have to take an indeter-

ministic choice in deciding which evidence value we pass in as an argument.

EXAMPLE 19. *Consider the following program (where the actual function definitions are omitted).*

```
overload c ::  Int->Bool where -- c1
         c = ...
overload c ::  Int->Int where -- c2
         c = ...
overload d ::  Bool->Char where -- d1
         d = ...
overload d ::  Int->Char where -- d2
         d = ...

exp3 ::  (C (Int->a), D (a->Char)) => Int->Char
exp3 = c . d
```

The type of `exp3` *is ambiguous. This implies that we cannot decide which actual definitions of* `c` *and* `d` *(either (c1,d1) or (c2,d2)) to use.*

A further technical challenge is to introduce coercion (ordering) relations between different typing derivations. Consider the following two expressions.

```
exp ::  Eq (a->a->Bool) => [a]->[a]->(Bool,Bool)
exp xs ys = (eq (tail xs) ys, eq (1::Int) (3::Int))

exp' ::  [Int]->[Int]->(Bool,Bool)
exp' xs ys = (eq (tail xs) ys, eq (1::Int) (3::Int))
```

Both expressions are identical modulo their type annotation. We find the following target expressions.

```
expT ::  (a->a->Bool)->[a]->[a]->(Bool,Bool)
expT e xs ys = (ecEqList e (tail xs) ys, ecEqInt (1::Int) (3::Int))

expT' ::  [Int]->[Int]->(Bool,Bool)
expT' xs ys = (ecEqList ecEqInt (tail xs) ys, ecEqInt (1::Int) (3::Int))
```

Note that in this case we can provide closed definitions for evidence constructors. Both target expressions yield the same result for any ground instance, however, expressions `exp`'s type and translation is "more general" than `exp`''s type and translation.

We define an ordering relation among types, target expressions and variable environments. Variable environments need to be included because in case of let expressions the environment will be changed (see the upcoming Lemma 14).

DEFINITION 1. *Let P be a confluent set of CHRs whose simplification rules are single-headed and non-overlapping, C a constraint, $\sigma_1 = \forall \bar{a}_1.C_1 \Rightarrow \tau_1$ and $\sigma_2 = \forall \bar{a}_2.C_2 \Rightarrow \tau_2$ two unambiguous type schemes, E_1 and E_2 two target expressions, and η_1 and η_2 two variable environments. We define $P, C \vdash^{ttv} (\sigma_1, E_1, \eta_1) \preceq (\sigma_2, E_2, \eta_2)$ iff*

—η_1 *and* η_2 *satisfy* P,
—$[\![P]\!] \wedge C \vdash \sigma_1 \preceq \sigma_2$
—*for any* ϕ *such that* $[\![P]\!] \models \phi(C \wedge C_2 \wedge \tau_1 = \tau_2)$, *we have that*

$$[\![\phi(E_1)]\!]\eta_1 \ \eta_1(e_{\phi' C_1}) = [\![\phi(E_2)]\!]\eta_2 \ \eta_2(e_{\phi C_2})$$

where ϕ' *is the unique extension of* ϕ *such that* $[\![P]\!] \models \phi' C_1$. *We define* ϕ *on target expression as follows:* $\phi(e_C) = e_{\phi C}$ *and* $\phi(e) = e$ *otherwise.*

We define $P, C \vdash^{ttv} (\Gamma_1, \eta_1) \preceq (\Gamma_2, \eta_2)$ *iff* $P, C \vdash^{ttv} (\sigma_1, x_1, \eta_1) \preceq (\sigma'_1, x_1, \eta_2), \ldots, P, C \vdash^{ttv} (\sigma_n, x_n, \eta_1) \preceq (\sigma'_n, x_n, \eta_2)$ *where* $\Gamma_1 = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ *and* $\Gamma_2 = \{x_1 : \sigma'_1, \ldots, x_n : \sigma'_n\}$.

Note that uniqueness of extension ϕ' *is ensured by the Lemma 10.*

For future calculations, we consider (τ, E, η) as a short-hand for $(\forall a.a = \tau \Rightarrow a, \lambda x.E, \eta)$ where a and x are fresh variables. Note that the lambda bound variable x refers to the evidence parameter for $a = \tau$. In this particular case, we find that if $P, C \vdash^{ttv} (\tau_1, E_1, \eta_1) \preceq (\tau_1, E_2, \eta_2)$ where $fv(\tau_1, \tau_2) = \emptyset$ and $C \longrightarrow^*_P True$, then $[\![E_1]\!]\eta_1 = [\![E_2]\!]\eta_2$.

In case of the above expressions, we find that

$$P, True \vdash^{ttv} (\forall a.Eq\ (a \to a \to Bool) \Rightarrow [a] \to [a] \to (Bool, Bool), \texttt{expT}, \eta) \preceq ([Int] \to [Int] \to [Int] \to (Bool, Bool), \texttt{expT}', \eta)$$

where P consists of rules (Eq1-3) and η is any variable environment that satisfies P.

The following three lemmas fall out by some straightforward calculations.

LEMMA 11 TRANSITIVITY. *The above relation is transitive.*

In the following two lemmas we will assume that type schemes are unambiguous and CHRs are confluent and simplification rules are single-headed and non-overlapping.

LEMMA 12 INSTANTIATION. *Let* P *be a set of CHRs,* C_2 *and* C'_2 *two constraints,* σ_1 *a type scheme,* E_1 *and* E_2 *two target expressions,* η_1 *and* η_2 *two variable environments,* $\bar{\tau}_2$ *a sequence of types,* $\bar{a}_2$ *a sequence of type variables,* e_{C_2} *a evidence variable and* $ec :: C_2 \xrightarrow{P} [\bar{\tau}_2/\bar{a}_2]C'_2$ *a evidence constructor such that* $P, C_2 \vdash^{ttv} (\sigma_1, E_1, \eta_1) \preceq (\forall \bar{a}_2.C'_2 \Rightarrow \tau_2, E_2, \eta_2)$. *Then,*

$$P, C_2 \vdash^{ttv} (\sigma_1, E_1, \eta_1) \preceq ([\bar{\tau}_2/\bar{a}_2]\tau_2, E_2\ (ec\ e_{C_2}), \eta_2).$$

LEMMA 13 GENERALIZATION. *Let* P *be a set of CHRs,* C_2 *and* C'_2 *two constraints,* $\bar{a}_2$ *a sequence of type variables,* σ_1 *a type scheme,* τ_2 *a type,* E_1 *and* E_2 *two target expressions and* η_1 *and* η_2 *two variable environments such that* $P, C_2 \wedge C'_2 \vdash^{ttv} (\sigma_1, E_1, \eta_1) \preceq (\tau_2, E_2, \eta_2)$ *and* $\bar{a}_2 \notin fv(C_2, \sigma_1)$. *Then,*

$$P, C_2 \vdash^{ttv} (\sigma_1, E_1, \eta_1) \preceq (\forall \bar{a}_2.C'_2 \Rightarrow \tau_2, \lambda e_{C'_2}.E_2, \eta_2).$$

In order to define an ordering on typing derivations, we assign weights to derivations trees.

Let $P, C, \Gamma \vdash e : \sigma \rightsquigarrow E$ be a judgment. The *derivation tree* of $P, C, \Gamma \vdash e : \sigma \rightsquigarrow E$ is a reversed tree where all leaf nodes are associated with (Var) rule application,

intermediate nodes are associated with other valid rule applications and the root node (i.e. the bottom most node) is $P, C, \Gamma \vdash e : \sigma \rightsquigarrow E$. We refer to the *weight* of $P, C, \Gamma \vdash e : \sigma \rightsquigarrow E$ as the number of nodes in the derivation tree, i.e. the number of typing rules necessary to derive the final judgment.

Definition 2. *Let P be a program theory, C_1 and C_2 two constraints, Γ_1 and Γ_2 two type environments, σ_1 and σ_2 two type schemes, E_1 and E_2 two target expressions and e a source expression. Then, we define*

$$\begin{array}{c}(P, C_1, \Gamma_1 \vdash e : \sigma_1 \rightsquigarrow E_1)\\ \preceq \\ (P, C_2, \Gamma_2 \vdash e : \sigma_2 \rightsquigarrow E_2)\end{array}$$

iff

—$[\![P]\!] \models C_2 \supset (\bar{\exists}_{fv(\Gamma_2)} C_1)$,

—$[\![P]\!] \wedge C_2 \vdash \sigma_1 \preceq \sigma_2$, $[\![P]\!] \wedge C_2 \vdash \Gamma_1 \preceq \Gamma_2$, *and*

—*let D_1 be the derivation tree of $P, C_1, \Gamma_1 \vdash e : \sigma_1 \rightsquigarrow E_1$, and let D_2 be the derivation tree of $P, C_2, \Gamma_2 \vdash e : \sigma_2 \rightsquigarrow E_2$. Then relation $\preceq$ holds for all sub-derivations $P, C_1', \Gamma_1' \vdash e' : \sigma_1' \rightsquigarrow E_1'$ in D_1 and $P, C_2', \Gamma_2' \vdash e' : \sigma_2' \rightsquigarrow E_2'$ in D_2 where we assume that $P, C_1', \Gamma_1' \vdash e' : \sigma_1' \rightsquigarrow E_1'$ has a lower weight in D_1 than $P, C_2', \Gamma_2' \vdash e' : \sigma_2' \rightsquigarrow E_2'$ in D_2.*

Note that we can always *normalize* typing derivations by combining (Var) and (∀E) rule applications and (∀I) and (Let) rule applications. In particular, given a fixed environment Γ and expression e we find that for any $(P, C_2, \Gamma_2 \vdash e : \sigma_2 \rightsquigarrow E_2)$ there is always a $(P, C_1, \Gamma \vdash e : \sigma_1 \rightsquigarrow E_1)$ for some C_1 and σ_1 such that $(P, C_1, \Gamma \vdash e : \sigma_1 \rightsquigarrow E_1) \preceq (P, C_2, \Gamma_2 \vdash e : \sigma_2 \rightsquigarrow E_2)$. That is, we have principal derivations for a fixed Γ and e (follows from completeness of inference).

Lemma 14 Confluent Translations. *Let $P, C_1, \Gamma_1 \vdash e : \sigma_1 \rightsquigarrow E_1$, $P, C_2, \Gamma_2 \vdash e : \sigma_2 \rightsquigarrow E_2$ be two valid judgments, η_1 and η_2 two variable environments such that $(P, C_1, \Gamma_1 \vdash e : \sigma_1 \rightsquigarrow E_1) \preceq (P, C_2, \Gamma_2 \vdash e : \sigma_2 \rightsquigarrow E_2)$ and $P, C_2 \vdash^{ttv} (\Gamma_1, \eta_1) \preceq (\Gamma_2, \eta_2)$. Then $P, C_2 \vdash^{ttv} (\sigma_1, E_1, \eta_1) \preceq (\sigma_2, E_2, \eta_2)$.*

A proof can be found in the Appendix.

In the final step toward establishing coherence, we will need to prove that if the principal derivation is unambiguous then any other derivation will be unambiguous as well. Maybe surprisingly, this is not always true as the following example will show.

Consider the expression $(f, 1)$ where we assume that we have an overloaded identifier f in scope. The principal judgment is $P, F\ a, \Gamma \vdash (f, 1) : (a, Int) \rightsquigarrow (f, 1)$ for some appropriate P and Γ. The following is another valid judgment:

$$P, F\ a, \Gamma \vdash (f, 1) : (a, Int) \rightsquigarrow (f, \lambda e_{F\ b}.1)$$

Note that we assume a sub-derivation of the form $P, F\ a \wedge F\ b, \Gamma \vdash 1 : Int \rightsquigarrow 1$ from which we derive (by applying rule (∀I)) $P, F\ a, \Gamma \vdash 1 : \forall a.F\ a \Rightarrow Int \rightsquigarrow \lambda e_{F\ b}.1$. Clearly, this example shows that although the principal derivation is unambiguous, other derivations might be ambiguous. However, as we also can see the evidence parameter $e_{F\ b}$ is not used inside the function body. Therefore, we rule out such non-sensical derivations. Under these additional assumptions it is straightforward

$$\text{(Class)}\quad \frac{\begin{array}{c} TC\ \bar{\tau} \Longrightarrow\ c_1, \ldots, c_m \in P \\ \bar{a} = fv(\bar{\tau}) \backslash fv(\Gamma) \\ P, C, \Gamma_x . x : \forall \bar{a}, \bar{b}. TC\ \bar{\tau} \wedge C' \Rightarrow \tau \vdash p : \sigma' \end{array}}{P, C, \Gamma_x \vdash \mathsf{class}\ c_1, \ldots, c_m \Rightarrow TC\ \bar{\tau}\ \mathsf{where}\ x : \forall \bar{b}. C' \Rightarrow \tau\ \mathsf{in}\ p : \sigma'}$$

$$\text{(Inst)}\quad \frac{\begin{array}{c} TC\ \bar{\tau}' \Longleftrightarrow\ c_1, \ldots, c_m \in P \\ P, C, \Gamma \vdash p : \sigma'' \\ P, C, \Gamma \vdash e : \forall \bar{a}'. C' \Rightarrow \tau' \quad (x : \forall \bar{a}. TC\ \bar{\tau} \wedge C'' \Rightarrow \tau) \in \Gamma \\ [\![P]\!] \models (\forall \bar{a}'. (TC\ \bar{\tau}' \wedge C') \Rightarrow \tau') \preceq (\forall \bar{a}. (TC\ \bar{\tau} \wedge C'' \wedge \bar{\tau}' = \bar{\tau}) \Rightarrow \tau) \end{array}}{P, C, \Gamma \vdash \mathsf{instance}\ c_1, \ldots, c_m \Rightarrow TC\ \bar{\tau}'\ \mathsf{where}\ x = e\ \mathsf{in}\ p : \sigma''}$$

Fig. 5. Type Class Typing Rules

to verify that if the principal derivation is unambiguous then any (besides nonsensical ones) other derivation must be unambiguous as well.

It remains to give a meaning to primitive functions. Let η be a variable environment and Γ a closed type environment ($fv(\Gamma) = \emptyset$). We define $\eta \models \Gamma$ iff for each non-overloaded identifier $x : \sigma \in \Gamma$ we have that $\eta(x) \in [\![\sigma]\!]$.

THEOREM 15 COHERENCE. *Let $P, C_1, \Gamma \vdash e : \tau \rightsquigarrow E_1$ and $P, C_2, \Gamma \vdash e : \tau \rightsquigarrow E_2$ be two valid judgments and η be a variable environment such that the principal derivation is unambiguous, $C_1 \longrightarrow_P^* True$, $C_2 \longrightarrow_P^* True$, $\eta \models \Gamma$, η satisfies P, P is a confluent set of CHRs where each simplification rule is single-headed and non-overlapping. Then $[\![E_1]\!]\eta = [\![E_2]\!]\eta$.*

A proof can be found in the Appendix.

7. TYPE CLASSES AND CONSTRAINT HANDLING RULES

We consider a particular instance of Haskell style type classes where type class relations are described by CHRs. The development is similar to Section 4. Expressions are extended to programs which also contain class and instance declarations.

$$\begin{array}{lrl} \textbf{Expressions} & e & ::= x \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\ \textbf{Programs} & p & ::= e \mid \mathsf{class}\ c_1 \ldots c_m \Rightarrow TC\ \bar{\tau}\ \mathsf{where}\ x : \sigma\ \mathsf{in}\ p \mid \\ & & \quad\ \mathsf{instance}\ c_1 \ldots c_m \Rightarrow TC\ \bar{\tau}\ \mathsf{where}\ x = e\ \mathsf{in}\ p \end{array}$$

We assume that $TC\ \bar{\tau}$ is a type class constraint expressing membership of type tuple $\bar{\tau}$ to the type class TC. Constraints $c_1, \ldots, c_m$ refer to type class constraints. For simplicity, we assume that class and instance declarations consist of only one member function.

Typing judgments are as before. However, in this particular context we refer to P as the *type class theory*. In Figure 5, we provide typing rules for class and instance declarations. We adopt rules (Var–Let) from Figure 1.

Rule (Class) introduces a new type class TC. The class declaration $\mathsf{class}\ c_1, \ldots, c_m \Rightarrow TC\ \bar{\tau}\ \mathsf{where}\ \ldots$ constrains any instance of the class TC to also satisfy the constraints

$c_1, \ldots, c_m$. This is expressed by the CHR propagation rule

$$TC\ \bar{\tau} \implies c_1, \ldots, c_m$$

which must be in P.

EXAMPLE 20. *Consider the Haskell standard prelude definitions of* `Ord` *and its translation:*

```
class Eq t => Ord t where ...
```
(S1) $Ord\ t \Longrightarrow Eq\ t$

This ensures that whenever we assert $Ord\ t$ *we must also have* $Eq\ t$.

Rule (Inst) defines an instance of class TC. The instance declaration instance $c_1, \ldots, c_m \Rightarrow TC\ \bar{\tau}'$ where ... maintains that $\bar{\tau}'$ is an instance of TC if and only if the constraints $c_1, \ldots, c_m$ are satisfied. This is expressed by the CHR simplification rule

$$TC\ \bar{\tau}' \iff c_1, \ldots, c_m$$

which must be in P.

EXAMPLE 21. *The instances of Ord and Eq for Lists and their translations are:*

`instance Eq t => Eq [t] where ...`	*(S2)* $Eq\ [t] \Longleftrightarrow Eq\ t$
`instance Ord t => Ord [t] where ...`	*(S3)* $Ord\ [t] \Longleftrightarrow Ord\ t$
`instance Eq Int where ...`	*(S4)* $Eq\ Int \Longleftrightarrow True$

This means that we can prove that a type $[t]$ *is an instance of the class Eq or Ord if we can prove that* t *is an instance, and that Int is an instance of Eq.*

As a further condition, we require that the type of the member function for a particular instance is subsumed by the type specified by the class declaration.

EXAMPLE 22. *Consider the program*

```
class C a where f :: a->a
instance C Int where f x = x
instance C Bool where f x = 1
```

The class declaration states that x *has type* $\forall a.C\ a \Rightarrow a \to a$. *The first instance declaration implements an instance of* C *at type Int where the member function* x *has type* $\forall a.a \to a$. *This is more general than actually required, but still safe because* $\forall a.a \to a$ *subsumes* $Int \to Int$. *The situation is different in case of the second instance declaration. At instance type Bool the member function has type* $\forall a.a \to Int$. *This will certainly cause problems at run-time. If function* x *is applied in context C Bool, we expect* x *to be applicable to values of type Bool and resulting in a value of type Bool. However, the actual implementation has type* $\forall a.a \to Int$ *which does not subsume the expected instance type. Therefore, we need to reject the above program. Indeed, the second instance is not type correct w.r.t. rule (Inst).*

7.1 Examples of type class systems

It follows immediately that single and multi–parameter type classes as described in [Jones 1992] can be expressed in our formulation of type classes. The program theory P simply consists of the simplification and propagation rules P_p which arise from class and instance declarations.

We continue with examples of type class systems where additional type class features can be expressed by the programmer specifiable set P_p of CHR propagation rules.

Functional Dependencies. Jones [Jones 2000] extends multi-parameter type classes to include functional dependencies (FDs) among class arguments. From our perspective, functional dependencies just extend the proof requirements for an instance. The extra constraints are represented straightforwardly using CHRs.

EXAMPLE 23. *Consider the following class declaration.*

```
class Collects e ce | ce -> e where
            empty ::  ce
            insert ::  e->ce->ce
```

The functional dependency `ce -> e` *enforces that* `ce` *uniquely determines* `e`. *From this class declaration we generate the following additional CHR:*

$$\textit{(FD)}\ \ \textit{Collects}\ e\ ce, \textit{Collects}\ e'\ ce \implies e = e'$$

The above rule states that whenever we encounter the two constraints `Collects e ce` *and* `Collects e' ce` *we enforce that* `e` *and* `e'` *must be equal.*

In fact, this is not sufficient to enforce the functional dependency as found in Haskell in all possible situations.

EXAMPLE 24. *Consider the following instance declarations.*

```
instance Collects Int [Int] where ...
instance Collects Float [Float] where ...
```

Assume out of an expression we generate the constraint $Collects\ a\ [Int]$. *According to the functional dependency imposed on Collects type variable a is uniquely determined by* $[Int]$. *Therefore, Haskell "improves" this constraint by substituting variable a by Int. In our framework, we have to enforce the functional dependency specifically for each instance. The following CHRs will do the job.*

$$\begin{array}{ll} \textit{(FD1)} & Collects\ a\ [Int] \implies a = Int \\ \textit{(FD2)} & Collects\ a\ [Float] \implies a = Float \end{array}$$

We observe that CHRs are sufficient to model functional dependencies. Recall the above class and instance declarations. In our framework, the above can be represented as follows.

```
class Collects e ce
instance Collects Int [Int]
instance Collects Float [Float]

-- general FD
rule Collects e ce, Collects e' ce ==> e=e'
-- instance specific FDs
rule Collects a [Int] ==> a=Int
rule Collects a [Float] ==> a=Float
```

In general, a class declaration with functional dependencies has the form

`class C => TC` $\mathtt{a}_1$ `...` $\mathtt{a}_n$ `|` fd_1`, ...,` fd_m

where fd_i is a *functional dependency* of the form $a_{i_1}, \ldots, a_{i_{k_i}} \rightarrow a_{i_0}$. Note that variables a_{i_j} are distinct. The functional dependency asserts that given fixed values of $a_{i_1}, \ldots, a_{i_{k_i}}$ then there is only one value of a_{i_0} for which the class constraint `TC` $\mathtt{a}_1$ `...` $\mathtt{a}_n$ can hold. In [Jones 2000] the right hand side of the $\rightarrow$ can have a list of variables. We use this simpler form, the expressiveness is equivalent. The CHR translation creates, for each functional dependency in a class declaration, an additional propagation rule of the form:

$$\text{(FD-TC)}\ \ TC\ a_1 \ldots a_n, TC\ \theta(b_1) \ldots \theta(b_n) \implies a_{i_0} = b_{i_0}$$

where variables $b_1, \ldots, b_n$ are distinct and θ maps each b_{i_j} to a_{i_j} and each other b_l to itself.

For each instance declaration of the form

`instance C => TC` $\mathtt{t}_1$ `...` $\mathtt{t}_n$

and functional dependency $a_{i_1}, \ldots, a_{i_{k_i}} \rightarrow a_{i_0}$ we generate the following CHR:

$$\text{(FD-Inst)}\ \ TC\ \theta'(b_1) \ldots \theta'(b_n) \implies t_{i_0} = b_{i_0}$$

where θ' maps each b_{i_j} to t_{i_j} and each other b_l to itself.

As already observed by Jones, functional dependencies subsume parametric type classes [Chen et al. 1992]. For example, the (parametric) declaration `class a :: Foo b` can be expressed by the functional dependency `class Foo a b | a->b`. The parameter `a` uniquely determines `b`. Note that with parametric type classes we can only describe uni-directional dependencies. E.g. with functional dependencies we can state `class Foo a b | a->b, b->a`. That is, there is a mutual dependency between both parameters Duggan and Ophel [Duggan and Ophel 2002b] describe yet another type checking strategy for multi-parameter type classes. In essence, they infer functional dependencies from the set of available instances. Surprisingly, there are some inconsistencies when it comes to termination of the respective inference algorithms.

EXAMPLE 25. *Here is a reformulation of the example on page 17 from [Duggan and Ophel 2002b] in terms of functional dependencies. For simplicity, we leave out the actual instance bodies.*

```
class Foo a b | a->b, b->a where foo :: a->b->Int
instance Foo Int Float
instance Foo a b => Foo [a] [b]
g x y = (foo [x] y) + (foo [y] x)
```

During type inference for function `g` we encounter, among others, constraints $Foo\ [a]\ b, Foo\ [b]\ a$. Duggan and Ophel's inference algorithm will not terminate in this case. In fact, `instance Foo a b => Foo [a] [b]` is not legal according to the restrictions imposed on functional dependencies. For any instance `instance ... => Foo t1 t2`, free variables in `t2` must also be found in `t1`. This is clearly violated. For details, we refer to page 12, Section 6.1 in [Jones 2000].

But GHC allows for a more relaxed form of functional dependencies. Surprisingly, GHC [GHC 2003] reports the following type for g.

$$\forall a, b.\ (Foo\ [[a]]\ [b], Foo\ [[b]]\ [a]) \Rightarrow [b] \rightarrow [a] \rightarrow Int$$

That is, type inference in GHC terminates! The key to understand the difference in behavior between [GHC 2003] and [Duggan and Ophel 2002b] is to translate the above class and instance declarations into CHRs.

$$\begin{array}{rrcl}
\text{(Inst1)} & Foo\ [a]\ [b] & \Longleftrightarrow & Foo\ a\ b \\
\text{(Inst2)} & Foo\ Int\ Float & \Longleftrightarrow & True \\
\text{(FD1)} & Foo\ a\ b, Foo\ a\ c & \Longrightarrow & b = c \\
\text{(FD2)} & Foo\ a\ b, Foo\ c\ b & \Longrightarrow & a = c \\
\text{(FD1I1)} & Foo\ [a]\ b & \Longrightarrow & b = [c] \\
\text{(FD2I1)} & Foo\ a\ [b] & \Longrightarrow & a = [c] \\
\text{(FD1I2)} & Foo\ Int\ a & \Longrightarrow & a = Float \\
\text{(FD2I2)} & Foo\ a\ Float & \Longrightarrow & a = Int
\end{array}$$

Rules (Inst1) and (Inst2) result from the instance declarations. Rules (FD1) and (FD2) formulated the functional dependencies `class Foo a b | a -> b` and `class Foo a b | b -> a`. The other rules are instance specific FDs. The above CHRs are non-terminating. For example, consider the following (non-terminating) derivation.

$$\begin{array}{ll}
 & Foo\ [a]\ b, Foo\ [b]\ a \\
\longrightarrow_{FD1I1} & Foo\ [a]\ [c], Foo\ [[c]]\ a, b = [c] \\
\longrightarrow_{Inst1} & Foo\ a\ c, Foo\ [[c]]\ a, b = [c] \\
\longrightarrow_{FD2I1} & Foo\ [d]\ c, Foo\ [[c]]\ [d], b = [c], a = [d] \\
\longrightarrow_{Inst1} & Foo\ [d]\ c, Foo\ [c]\ d, b = [c], a = [d] \\
\ldots &
\end{array}$$

GHC terminates because of "lazy" context reduction [Jones et al. 1997]. That is, not necessary all simplification rules are applied during type inference of function g. This prevents the GHC type checker from non-termination. Note that the CHR solver will terminate for any ground constraint. Indeed, GHC will only detect that function g is unsatisfiable once we apply g to some monomorphic arguments.

Disjoint and Negative Class Constraints. Once we apply our approach to the Haskell type classes formulation we can easily express extensions.

The following example illustrates that it might be useful to to express disjointness of type classes.

EXAMPLE 26. *The following program is accepted by Haskell.*

```
f x y = x / y + x `div` y
```

Function `f` *has an inferred type of*

```
f ::  (Integral a, Fractional a) => a->a->a
```

rather than immediately causing a type error. A CHR expressing that the `Integral` *and* `Fractional` *type classes are disjoint is simply.*

$$\textit{(Disjoint)}\ Integral\ a, Fractional\ a \Longrightarrow False$$

Another useful feature would allow type classes to represent negative information:

EXAMPLE 27. *The intention of the* `Num` *class is to describe numeric types. We might insist that functional types are never numbers by adding the rule*

$$(Negative)\ Num\ (a \to b) \Longrightarrow False$$

Then, the declaration `instance Num (a->b)` *will cause an immediate error to be detected.*

7.2 Properties of type class systems

Class and instance declarations must satisfy certain conditions. For example, according to the Haskell 98 [Peyton Jones et al. 1999] language report Section 4.3.2. (Instance Declarations):

Assume that the type variables in the instance type (T u1 ... uk) satisfy the constraints i n the instance context cx'. Under this assumption, the following two conditions must also be satisfied:

(*1*) *The constraints expressed by the superclass context cx[(T u1 ... uk)/u] of C must be satisfied. In other words, T must be an instance of each of C's superclasses and the contexts of all superclass instances must be implied by cx'.*

(*2*) *Any constraints on the type variables in the instance type that are required for the class method declarations in d to be well-typed must also be satisfied.*

In our framework, this can be expressed by requiring that the type class theory must be confluent.

EXAMPLE 28. *Reconsidering Example 5 from the introduction, but directly translating the Haskell class and instance declarations to CHRs we obtain*

`class Eq t => Ord t where ...`	$(S1)\ Ord\ t \Longrightarrow Eq\ t$
`instance Eq t => Eq [t] where ...`	$(S2)\ Eq\ [t] \Longleftrightarrow Eq\ t$
`instance Eq Int where ...`	$(S4)\ Eq\ Int \Longleftrightarrow True$
`instance Ord [t] where ...`	$(S5)\ Ord\ [t] \Longleftrightarrow True$

We would have a non-confluent type class theory, since `Ord [u]` *has two derivations whose result is not joinable:*

$$Ord\ [u] \longrightarrow_{S1} Ord\ [u], Eq\ [u] \longrightarrow_{S5} Eq\ [u] \longrightarrow_{S2} Eq\ u$$

and simply $Ord\ [u] \longrightarrow_{S5} True$ *using rule (S5).*

The non-confluence arises exactly because the requirement that each `Ord` *instance is also a* `Eq` *instance is not satisfied.*

Confluence provides us with a general notion to ensure validity of class and instance declarations. Note that confluence is essential to ensure type correctness (see Section 5) and a coherent semantics (see Section 6). The techniques and methods developed in these sections also apply to Haskell style type classes.

8. EXTENSIONS

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

8.1 Overlapping Definitions

EXAMPLE 29. *Consider the following adaptation of Example 13 from Section 4.2.*

```
overload eq :: Int->Int->Bool
overload eq :: Char->Char->Bool
overload eq :: Eq (a->a->Bool) => [a]->[a]->Bool
overload eq :: [Int]->[Int]->Bool
rule Eq x ==> x = t->t->Bool
```

The following CHRs arise.

$$\begin{array}{lrcl}
\textit{(Eq1)} & Eq\ (Int \to Int \to Bool) & \Longleftrightarrow & True \\
\textit{(Eq2)} & Eq\ (Char \to Char \to Bool) & \Longleftrightarrow & True \\
\textit{(Eq3)} & Eq\ ([a] \to [a] \to Bool) & \Longleftrightarrow & Eq\ (a \to a \to Bool) \\
\textit{(Eq4)} & Eq\ ([Int] \to [Int] \to Bool) & \Longleftrightarrow & True \\
\textit{(Eq5)} & Eq\ x & \Longrightarrow & x = t \to t \to Bool
\end{array}$$

Although, the program theory is confluent, it seems difficult to provide a coherent translation because we must take an indeterministic choice in case we require `eq` *on type* $[Int] \to [Int] \to Bool$.

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 guard constraint into simplification rules when constructing evidence. The guard constraint, $a \neq Int$, added to rule (Eq3), states that this rule only fires if the instance type is different from Int.

$$\begin{array}{lrcl}
\text{(Eq1)} & Eq\ (Int \to Int \to Bool) & \Longleftrightarrow & True \\
\text{(Eq2)} & Eq\ (Char \to Char \to Bool) & \Longleftrightarrow & True \\
\text{(Eq3')} & Eq\ ([a] \to [a] \to Bool) & \Longleftrightarrow & a \neq Int \mid Eq\ (a \to a \to Bool) \\
\text{(Eq4)} & Eq\ ([Int] \to [Int] \to Bool) & \Longleftrightarrow & True
\end{array}$$

Note that our approach to providing a meaning for programs containing overloaded identifiers relies on run-time evidence construction. Hence, for type inference purposes we employ rules (Eq1-5) whereas for the actual evidence construction we employ rules (Eq1-2), (Eq3') and (Eq4). Consider

```
g x y = eq [x] [y]
e = (g (1::Int) (2::Int), g 'b' 'a')
```

Type inference yields that `g` has type `Eq (a->a->Bool) => a->a->Bool` and `e` has type `(Bool, Bool)`. Hence, evidence translation yields

```
g e x y = (ec1 e) [x] [y]
e = (g (ec2 eTrue) (1::Int) (2::Int), g (ec3 eTrue) 'b' 'a')
```

where

$$\begin{array}{l}
ec_1 :: Eq\ (a \to a \to Bool) \xrightarrow{P} Eq\ ([a] \to [a] \to Bool) \\
ec_2 :: True \xrightarrow{P} Eq\ ([Bool] \xrightarrow{P} [Bool] \to Bool) \\
ec_3 :: True \xrightarrow{P} Eq\ ([Char] \xrightarrow{P} [Char] \to Bool)
\end{array}$$

and P consists of rules (Eq1-5). At run-time, that is when we actually construct evidence values, we exchange P by P' where P' consists of rules (Eq1-2), (Eq3') and (Eq4). This ensures that evidence will be constructed unambiguously. For example, when evaluating `g (1::Int) (2::Int)` we pass in the information to `g` that we demand evidence for $Eq\ ([Int] \to [Int] \to Bool)$. By employing P' we pick the more specific instance.

We observe that our dealing with overlapping definitions crucially relies on the fact that we postpone evidence construction until run-time. Obviously, such a method prohibits separate compilaton. In summary, assume the set $P_s \cup P_p$ is confluent but P_s (all simplification rules) is overlapping. That is, we find $M\ t_1 \iff C_1$ and $M\ t_2 \iff C_2$ such that t_1 and t_2 are unifiable. We perform type inference and evidence translation w.r.t. $P_s \cup P_p$. At run-time, i.e. evidence construction time, we employ P'_s which is derived from P_s by choosing an appropriate set of guard constraints to ensure that P'_s w.r.t. guard constraints is non-overlapping. For example, $M\ t_1 \iff g_1 \mid C_1$ and $M\ t_2 \iff g_2 \mid C_2$ are non-overlapping w.r.t. guard constraints if for any unifier ϕ of t_1 and t_2 either guard constraint ϕg_1 or ϕg_2 is satisfied. Note that we can always make P'_s non-overlapping by e.g. setting g_1 to $False$.

8.2 Closing Definitions

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

```
overload eq ::  Int->Int->Bool where
         eq = ...
overload eq ::  Eq (a->a->Bool) => [a]->[a]->Bool where
         eq = ...
```

We find the following set of CHRs:

$$\text{(Eq1)}\ \ Eq\ (Int \to Int \to Bool) \iff True$$
$$\text{(Eq2)}\ \ Eq\ ([a] \to [a] \to Bool) \iff Eq\ (a \to a \to Bool)$$

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

```
f ::  Eq (Tree a->Tree a->Bool) => Tree a->Tree a->Bool
f x y = eq x y
```

Although there is no equality definition on trees in scope at the moment, this does not 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 [Abdennadher and Schütz 1998] 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

$$\text{(CloseEq)}\ \ Eq\ a \implies (a = Int \to Int \to Bool) \vee (a = [b] \to [b] \to Bool)$$

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 $\Longrightarrow$ 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 soundness 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 [Shields and Jones 2001] by disallowing recursive dependencies for "closed" definitions. This requirement is sufficient to ensure termination of CHRs involving closed definitions. Completeness results should also carry over except that we need a much more complex definition of *subsumes* to handle the disjunctive results.

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. Recall the super class relation from Example 5.

$$\text{(Super)}\ \ Ord\ a \implies Eq\ a$$

Consider the expression

```
h ::  (Eq (a->a->Bool), Ord (a->a->Bool) => a->a->(Bool,Bool)
h x y = (eq x y, ord x y)
```

Note that

$$[\![P]\!] \models (Eq\ (a \to a \to Bool) \wedge Ord\ (a \to a \to Bool)) \leftrightarrow Ord\ (a \to a \to Bool)$$

where P consists of (Super) among others. Therefore, we could assign to expression `h` the equivalent but simpler type scheme $\forall a.Ord\ (a \to a \to Bool) \Rightarrow a \to a \to (Bool, Bool)$.

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

$$Ord\ (a \to a \to Bool), Eq\ (a \to a \to Bool) \iff Ord\ (a \to a \to Bool)$$

9. RELATED WORK

Our approach is clearly inspired by Haskell style type classes and its various extensions. The theory of qualified types [Jones 1992] provides a general framework for type classes. However, essential questions such as decidability of type inference need to be reconsidered for each new extension [Jones 2000; Jones et al. 1997; Jones 1993b]. In our overloading framework, we have established some precise conditions in terms of CHRs under which we achieve decidable type inference and the meaning of programs is unambiguous.

Moreover, we gave a reformulation of type classes in terms of CHRs. In particular, we considered functional dependencies [Jones 2000] and some of its variations [Duggan and Ophel 2002b; Chen et al. 1992]. CHRs do not solve the termination problem in case of Example 25. However, CHRs provide some insights between the differences of [Jones 2000] and [Duggan and Ophel 2002b]. In turn, we take a look at some other closely related work.

Constructor classes [Jones 1993b] are another popular extension to type classes where we can parameterize a type class in terms of a type constructor. This requires extending the inference algorithm to "kinded" unification. That is, in addition to the kind of all types, usually referred to as $\star$, we also allow for the kind $\star \rightarrow \star$ and so on. For example, the list constructor would be of kind $\star \rightarrow \star$. Kinded unification ensures that equations are kind correct. We believe that constructor classes can be incorporated into our framework by providing some appropriate CHRs. Details are left for future work.

Shields and Jones [2001] gave 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 [2002]. 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 [Hanus 1994]. 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 et al. [1995] 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 [1999] 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 8.2).

Implicit parameters [Lewis et al. 2000] introduced by Lewis, Shields, Meijer and Launchbury are a complement to Haskell style overloading. A implicit parameter can be seen as a special form of overloaded identifier which is allowed to be defined locally. To ensure decidable type inference and coherence some sufficient conditions are imposed such as each implicit parameter must always be of the same monomorphic type. Note that we could enforce such conditions via CHRs. However, our semantics as described in Section 6 would need to be extended to deal with locally overloaded identifiers. Currently, we assume that all overloaded definitions appear on the top-most level.

Duggan and Ophel [Duggan and Ophel 2002a] studied an extension of Haskell style type classes with open and closed scope. Type classes with open scope follow the common open world assumption and are only allowed to be defined on the top-most level. Type classes with closed scope follow the closed world assumption and may be defined locally. Their system seems to provide a unifying framework for implicit parameters and Haskell style overloading. In future work, we intend to investigate how the CHR approach fits into their system.

Augustsson [1998] introduced a dependently-typed language of great expressiveness. In his system decidable type inference is left to the user whereas we could establish some precise conditions under which type inference is sound and complete. Additionally, CHRs allow us to give a precise characterization under which a program is unambiguous. An interesting topic is to lift the phase distinction between type inference and the operational semantics of programs in our framework while yet retaining decidable type inference and unambiguity of programs.

It is also worth mentioning the work by Yang [1998]. 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. The framework can gracefully handle many extensions to Hindley/Milner types but certainly not all of them.

The ideas of this paper have been implemented as part of the Chameleon system [Sulzmann and Wazny 2003]. Chameleon can be seen as an experimental version of Haskell where the programmer can design her own type extensions via CHRs.

The current requirement that the CHRs supplied are confluent, can be burdensome. There are methods to automatically complete a set of CHRs [Abdennadher and Frühwirth 1998], but they add multi-headed simplification rules and require an explicit termination order. In future work, we plan to consider automatic methods to complete a set of CHRs (see Example 12) that fit into our scheme. In particular, this seems to be an issue in the context of functional dependencies (see Example 25).

APPENDIX

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 $\bar{c} \iff g \mid \bar{d}$ where the guard constraints g is a conjunction of disequality constraints.

(Solve) $C_u \cup C_e \longrightarrow_P \phi C_u \cup con(\phi)$
ϕ mgu of C_e

(Simp) $C_u \cup C_e \longrightarrow_P (C_u - \bar{c}') \cup C_e \cup \theta(\bar{d})$
if (R) $\bar{c} \Longleftrightarrow g \mid \bar{d} \in P$ and there exists a subset $\bar{c}' \subseteq C_u$
and a substitution θ on variables in (R)
such that $\theta(\bar{c}') \equiv \bar{c}$ and $\models C_e \supset \theta(g)$

(Prop) $C_u \cup C_e \longrightarrow_P C_u \cup C_e \cup \theta(\bar{d})$
if (R) $\bar{c} \Longrightarrow \bar{d} \in P$ and there exists a subset $\bar{c}' \subseteq C_u$
and a substitution θ on variables in (R)
such that $\theta(\bar{c}') \equiv \bar{c}$

In rule (Solve), we assume that we normalize stores by building most general unifiers. The observant reader will notice that we have to prevent the infinite application of CHR propagation rules. We refer to [Abdennadher 1997] for more details.

We find the following result, see [Frühwirth 1998b] for details.

THEOREM 16 SOUNDNESS. *Let P be a CHR program and C, C' constraints such that $C \longrightarrow_P^* C'$. 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 $C_0 \longrightarrow_P C_1$ and $C_0 \longrightarrow_P C_2$, then there exist derivations $C_1 \longrightarrow_P^* C_3$ and $C_2 \longrightarrow_P^* C_4$ 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 all 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 17 MODEL EXTENSION. *Let P be a confluent set of range-restricted CHRs where each simplification rule is single-headed. Suppose $D \longrightarrow_P^* C$ 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 $S \longrightarrow_P^* S'$ 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 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 use a_C to refer to all user-defined constraints in C.

We augment M using the following rules.

—If there exists a rule $h \iff d_1, \ldots, d_m$ 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 $h \iff d_1, \ldots, d_m$ 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 $h_1, \ldots, h_n \implies d_1, \ldots, d_m$ 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 $S \longrightarrow_P^* S'$ 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 $\phi(h) \longrightarrow_P \phi(d_1) \wedge \cdots \wedge \phi(d_m)$ and by induction we have a derivation from $\phi(d_1) \wedge \cdots \wedge \phi(d_m) \longrightarrow_P^* S'$ 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 $\phi(h) \longrightarrow_P^* S'$ where $S' \subseteq \theta(a_C)$. Hence, by the confluence of P it must be that $\phi(d_1) \wedge \cdots \wedge \phi(d_m) \longrightarrow_P^* S'$. (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 $\phi(d_i) \longrightarrow_P^* S_i'$ 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 $\phi(h_1) \wedge \cdots \wedge \phi(h_n) \longrightarrow_P^* S'$ where $S' \subseteq \theta(a_C)$. Hence, by confluence of P it must be that $\phi(h_1) \wedge \cdots \wedge \phi(h_n) \wedge \phi(d_1) \wedge \ldots \wedge \phi(d_m) \longrightarrow_P^* S'$. Again using the same argument as the previous case we have that each $\phi(d_i) \longrightarrow_P^* S_i'$ where $S_i' \subseteq \theta(a_C)$. □

LEMMA 2 CANONICAL FORM. *Let P be a confluent and terminating set of range-restricted CHRs where each simplification rule is single-headed. Then $[\![P]\!] \models D \leftrightarrow D'$ iff $D \longrightarrow_P^* C$ and $D' \longrightarrow_P^* C'$ 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 $D \longrightarrow_P^* C$ 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 17 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 Skolem 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 17 we construct a model M of $[\![P]\!]$ and $\theta(C)$ such that for each subset $S \subseteq M$ we have that $S \longrightarrow_P^* S'$ 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 $\theta(a_{C'}) \longrightarrow_P^* S'$ 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 $[\![P]\!] \models D \leftrightarrow D'$. □

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 $C \longrightarrow_P^* C'$. 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 17 to build a model M of $[\![P]\!]$ and $\theta(C')$ where θ is as defined in Lemma 17. Since $[\![P]\!] \models C \leftrightarrow \bar{\exists}_{fv(C)} C'$, clearly $M \models \tilde{\exists} C$. □

B.3 Ambiguity

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

PROOF. Follows immediately from soundness of CHR derivations. □

LEMMA 6 COMPLETENESS OF UNAMBIGUITY. *Let P be a confluent set of range-restricted CHRs where each simplification rule is single-headed, $\forall \bar{a}.C \Rightarrow \tau$ be a type*

scheme and ρ be a variable renaming on $\bar{a}$ such that $[\![P]\!] \models (C \wedge \rho(C) \wedge (\tau = \rho(\tau))) \supset (a = \rho(a))$ for each $a \in \bar{a}$. Then $C \wedge \rho(C) \wedge \tau = \rho(\tau) \longrightarrow_P^* C'$ where $\models C' \supset (a = \rho(a))$ for each $a \in \bar{a}$.

PROOF. Suppose to the contrary $[\![P]\!] \models C \wedge \rho(C) \wedge \tau = \rho(\tau) \supset a = \rho(a)$ but $\not\models C' \supset a = \rho(a)$ or equivalently $C' \not\models a = \rho(a)$. 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 $\theta(a) \neq \theta(\rho(a))$.

By Lemma 17 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 $\theta(a) \neq \theta(\rho(a))$ and we have a contradiction. □

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 $\forall \bar{a}.C \Rightarrow \tau$ is equivalent to $\forall \bar{a}.C \wedge \tau = a \Rightarrow a$ where a is new.

LEMMA 4 SOUNDNESS OF ENTAILMENT. *Let P be a set of CHRs and $\sigma = \forall \bar{a}.C \Rightarrow \tau$ and $\sigma' = \forall \bar{a}'.C' \Rightarrow \tau'$ (by assumption τ and τ' are type variables) where $C' \wedge \tau = \tau' \longrightarrow_P^* C_1$ and $C' \wedge \tau = \tau' \wedge C \longrightarrow_P^* C_2$ 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 $\bar{a} \cap \bar{a}' = \emptyset$.

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

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

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

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

We apply the following reasoning

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

From B.1 and B.2 we conclude that

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

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

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

and finally

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

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 $\wedge_{a \in fv\phi} a = \phi(a)$. 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 $\phi_{|\bar{a}}$ the substitution resulting from ϕ where $\models con(\phi_{|\bar{a}}) \leftrightarrow (\bar{\exists}_{\bar{a}} con(\phi))$. Similarly, we denote by $\phi_{\backslash \bar{a}}$ the substitution such that $\models con(\phi_{\backslash \bar{a}}) \leftrightarrow \exists \bar{a}.con(\phi)$.

LEMMA 18 CANONICAL EXTENSION. *Let $\forall \bar{a}.C \Rightarrow \tau$ 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 $[\![P]\!] \models (C' \wedge \tau = \tau') \leftrightarrow (\exists \bar{b}.C' \wedge \tau = \tau' \wedge C)$ where $\bar{b} = \bar{a} \backslash \{\tau\}$. 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 $\bar{a} = fv(C, \tau)$.

From $[\![P]\!] \models (C' \wedge \tau = \tau') \leftrightarrow (\exists \bar{b}.C' \wedge \tau = \tau' \wedge C)$ 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,

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

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

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

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

From B.4, we conclude

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

The "$\leftarrow$" direction follows immediately. The other direction holds because B.4 implies that $[\![P]\!] \models (C \wedge \phi'(C) \wedge con(\phi)) \leftarrow con(\phi')$. Note that the same holds for ϕ''. We apply substitution ϕ to this formula (note $\models \phi(con(\phi')) \leftrightarrow con(\phi'_{|\bar{b}})$) and find

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

From B.3 and B.5 we conclude

$$\begin{aligned} M \models\ & (\phi(C') \wedge \phi(C) \wedge \phi''(C) \wedge \phi(\tau = \tau') \wedge con(\phi''_{|\bar{b}})) \\ & \leftrightarrow \\ & (\phi(C') \wedge \phi(C) \wedge \phi'(C) \wedge \phi(\tau = \tau') \wedge con(\phi'_{|\bar{b}})) \end{aligned}$$

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

LEMMA 7 COMPLETENESS OF ENTAILMENT. *Let P be a terminating confluent set of range-restricted CHRs whose simplification rules are single-headed, $\sigma = \forall \bar{a}.C \Rightarrow \tau$ and $\sigma' = \forall \bar{a}'.C' \Rightarrow \tau'$ where w.l.o.g. τ and τ' are type variables*

such that $[\![P]\!] \vdash \sigma \preceq \sigma'$ *and* σ *is unambiguous. Then* $C' \wedge \tau = \tau' \longrightarrow_P^* C_1$ *and* $C' \wedge \tau = \tau' \wedge C \longrightarrow_P^* C_2$ *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

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

We obtain that

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

where $\bar{b} = fv(C) \backslash fv(\tau)$. 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 $fv(C' \wedge \tau = \tau' \wedge C) \backslash \bar{b}$) substitution ϕ. From B.6 we follow that $M \models \phi(\exists \bar{a}.C \wedge \tau = \tau')$. 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 B.7 we get

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

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

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 B.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

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

From B.7 we can conclude that

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

for some extension θ' of θ. Note that θ and θ' might only differ on $\bar{b}$. 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 B.9 and B.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 $S \longrightarrow_P^* S'$ where $S' \subseteq \theta(C_1)$.

We have that $\theta(a_{C_2}) \subseteq M$. But then $\theta(a_{C_2}) \longrightarrow_P^* S'$ 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. □

LEMMA 9 COMPILATION OF EVIDENCE. *Let P be a confluent set of CHRs where each simplification rule is single-headed and non-overlapping and η a variable environment which satisfies P. Let C_1 and C_2 be two sets of user-defined constraints such that $C_1, C_2 \longrightarrow_P^* C_1$. Then, there exists a closed definition for $ec :: C_1 \xrightarrow{P} C_2$.*

PROOF. For simplicity, we only consider the case if $C_2 = U\ \tau$ and $C_1 = U_1\ \tau_1, \ldots, U_m\ \tau_m$.

Note that $C_1, C_2 \longrightarrow_P^* C_1$ implies $C_1, C_2 \longrightarrow_{P_{Simp}}^* C_1$.

The proof proceeds by induction on the derivation $C_1, C_2 \longrightarrow_P^* C_1$. We distinguish among the following cases:

(1) $U = U_j$ for some $j = 1 \ldots m$: We immediately find that $ec(e_1, \ldots, e_j, \ldots, e_m) = e_j$.

(2) We have that $(R)\ \ U\ \tau' \Longleftrightarrow C_3 \in P$ and there exists ϕ such that $\phi\tau' \equiv \tau$.

We find that $C_1, C_2 \longrightarrow_R C_1, \phi C_3$. For each $U_i'\ \tau_i' \in \phi C_3$ we have that $C_1, U_i'\ \tau_i' \longrightarrow^* C_1$ (as usual we assume an ordering on $U_i'\ \tau_i'$ in ϕC_3). By induction, there exist closed definitions $ec_i :: C_1 \xrightarrow{P} U_i'\ \tau_i'$. We define

$$ec(e_1, \ldots, e_m) = (ec_{U\ \tau}\ (ec_1(e_1, \ldots, e_m)), \ldots, (ec_k(e_1, \ldots, e_m)))$$

where $ec_{U\ \tau}$ is as defined above. □

LEMMA 10 UNIQUENESS. *Let P be a confluent set of CHRs whose simplification rules are single-headed, $\forall \bar{a}.C \Rightarrow \tau$ an unambiguous type scheme, ϕ a mapping from*

type variables $fv(C,\tau)$ to ground types and ϕ' a mapping from type variables $fv(\tau)$ to ground types such that $\phi' \leq \phi$ and $\phi C \longrightarrow_P^* True$. Then $C, \phi' \longrightarrow_P^* \phi''$ for some ϕ'' such that $\phi' \sqcup \phi'' = \phi$.

PROOF. Note that we interpret substitutions as equality constraints. Then, the $\leq$ relation among substitutions can be expressed as constraint entailment, and $\sqcup$ corresponds to conjunction.

By assumption $\forall \bar{a}.C \Rightarrow \tau$ is unambiguous. Let ρ be a variable renaming. We have that

$$[\![P]\!] \models (C \wedge \rho(C) \wedge \tau = \rho(\tau)) \supset (a = \rho(a))$$

for variables $a \in fv(C,\tau)$. Then, we find that

$$[\![P]\!] \models (C \wedge \phi(C) \wedge \tau = \phi(\tau)) \supset (a = \phi(a))$$

for variables $a \in fv(C,\tau)$. From that, we conclude

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

By assumption we know that $\phi(C) \longrightarrow_P^* True$. The Canonical Form Lemma enforces that $C, \phi' \longrightarrow_P^* \phi''$. □

LEMMA 14 CONFLUENT TRANSLATIONS. *Let $P, C_1, \Gamma_1 \vdash e : \sigma_1 \rightsquigarrow E_1$, $P, C_2, \Gamma_2 \vdash e : \sigma_2 \rightsquigarrow E_2$ be two valid judgments, η_1 and η_2 two variable environments such that $(P, C_1, \Gamma_1 \vdash e : \sigma_1 \rightsquigarrow E_1) \preceq (P, C_2, \Gamma_2 \vdash e : \sigma_2 \rightsquigarrow E_2)$ and $P, C_2 \vdash^{ttv} (\Gamma_1, \eta_1) \preceq (\Gamma_2, \eta_2)$. Then $P, C_2 \vdash^{ttv} (\sigma_1, E_1, \eta_1) \preceq (\sigma_2, E_2, \eta_2)$.*

PROOF. We proceed by induction over the derivation $P, C_2, \Gamma_2 \vdash e : \sigma_2 \rightsquigarrow E_2$. We only show some of the interesting cases.

Case (Var) We find the following situation:

$$\frac{(x : \sigma_2) \in \Gamma_2}{P, C_2, \Gamma_2 \vdash x : \sigma_2 \rightsquigarrow x}$$

where $(P, C_2, \Gamma_1 \vdash x : \sigma_1 \rightsquigarrow x) \preceq (P, C_2, \Gamma_2 \vdash x : \sigma_2 \rightsquigarrow x)$. By assumption, we have that $P, C_2 \vdash^{ttv} (\sigma_1, x, \eta_1) \preceq (\sigma_2, x, \eta_2)$ and we are done.

Case ($\forall E$) We find the following situation:

$$\frac{\begin{array}{c} P, C_2, \Gamma_2 \vdash e : \forall \bar{a}.C_2' \Rightarrow \tau_2 \rightsquigarrow E_2 \\ [\![P]\!] \models C_2 \supset [\bar{\tau}_2/\bar{a}_2]C_2' \\ ec :: C_2 \xrightarrow{P} [\bar{\tau}_2/\bar{a}_2]C_2' \end{array}}{P, C, \Gamma_2 \vdash e : [\bar{\tau}_2/\bar{a}_2]\tau_2 \rightsquigarrow E_2\ (ec\ e_{C_2})}$$

We can apply the induction hypothesis to the premise. Application of Lemma 12 yields the desired result.

Case ($\forall I$) We find the following situation:

$$\frac{\begin{array}{c} P, C_2 \wedge C_2', \Gamma_2 \vdash e : \tau_2 \rightsquigarrow E_2 \\ \bar{a}_2 \notin fv(C_2, \Gamma_2) \end{array}}{P, C_2, \Gamma_2 \vdash e : \forall \bar{a}_2.C_2' \Rightarrow \tau_2 \rightsquigarrow \lambda e_{C_2'}.E_2}$$

Application of the induction hypothesis and Lemma 13 yields the desired result.

Case (Let) We find the following situation:

$$\frac{P, C_2, (\Gamma_2)_x \vdash e : \sigma_2 \rightsquigarrow E_2 \quad P, C_2, (\Gamma_2)_x.x : \sigma_2 \vdash e' : \tau_2' \rightsquigarrow E_2'}{P, C_2, (\Gamma_2)_x \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau_2' \rightsquigarrow \mathsf{let}\ x = E_2\ \mathsf{in}\ E_2'}$$

where

$$(P, C_1, (\Gamma_1)_x \vdash e : \sigma_1 \rightsquigarrow E_1) \preceq (P, C_2, (\Gamma_2)_x \vdash e : \sigma_2 \rightsquigarrow E_2)$$
$$(P, C_1, (\Gamma_1)_x.x : \sigma_1 \vdash e' : \tau_1' \rightsquigarrow E_1') \preceq (P, C_2, (\Gamma_2)_x.x : \sigma_2 \vdash e' : \tau_2' \rightsquigarrow E_2')$$

for some σ_1 and E_1.

Application of the induction hypothesis to the top premise yields

$$P, C_2 \vdash^{ttv} (\sigma_1, E_1, \eta_1) \preceq (\sigma_2, E_2, \eta_2)$$

We set $\eta_1' = \eta_1[x := \lambda e_{C_1}.[\![\pi E_1]\!]\eta_1]$ and $\eta_2' = \eta_2[x := \lambda e_{C_2}.[\![\pi E_2]\!]\eta_2]$. The induction hypothesis applied to the bottom premise yields

$$P, C_2 \vdash^{ttv} (\tau_1', E_1', \eta_1') \preceq (\tau_2', E_2', \eta_2').$$

We can conclude that

$$P, C_2 \vdash^{ttv} (\tau_1', \mathsf{let}\ x = E_1\ \mathsf{in}\ E_1', \eta_1) \preceq (\tau_2', \mathsf{let}\ x = E_2\ \mathsf{in}\ E_2', \eta_2)$$

and we are done. □

THEOREM 15 COHERENCE. *Let $P, C_1, \Gamma \vdash e : \tau \rightsquigarrow E_1$ and $P, C_2, \Gamma \vdash e : \tau \rightsquigarrow E_2$ be two valid judgments and η be a variable environment such that the principal derivation is unambiguous, $C_1 \longrightarrow_P^* True$, $C_2 \longrightarrow_P^* True$, $\eta \models \Gamma$, η satisfies P, P is a confluent set of CHRs where each simplification rule is single-headed and non-overlapping. Then $[\![E_1]\!]\eta = [\![E_2]\!]\eta$.*

PROOF. The principal derivation must exist. That means, we find that

$$(P, C_3, \Gamma \vdash e : \tau' \rightsquigarrow E_3) \preceq (P, C_1, \Gamma \vdash e : \tau \rightsquigarrow E_1)$$

and

$$(P, C_3, \Gamma \vdash e : \tau' \rightsquigarrow E_3) \preceq (P, C_2, \Gamma \vdash e : \tau \rightsquigarrow E_2)$$

for some C_3 and τ' where all derivations are unambiguous.

Application of Lemma 14 yields

$$P, C_1 \vdash^{ttv} (\tau', E_3, \eta) \preceq (\tau, E_1, \eta)$$

and

$$P, C_2 \vdash^{ttv} (\tau', E_3, \eta) \preceq (\tau, E_2, \eta).$$

In particular, we find that $[\![E_3]\!]\eta = [\![E_1]\!]\eta$ and $[\![E_3]\!]\eta = [\![E_2]\!]\eta$. Therefore, $[\![E_1]\!]\eta = [\![E_2]\!]\eta$ and we are done. □

ACKNOWLEDGEMENTS

We thank Kevin Glynn, Andreas Rossberg, Peter Thiemann, Jeremy Wazny and Matthias Zenger for their comments. In particular, we thank Simon Peyton-Jones for encouragement and helpful feedback. We are grateful to our referees for detailed comments and constructive criticism which has helped to significantly improve the presentation.

REFERENCES

Abdennadher, S. 1997. Operational semantics and confluence of constraint propagation rules. In *Proc. of the Third International Conference on Principles and Practice of Constraint Programming, CP'97*. LNCS. Springer, 252–266.

Abdennadher, S. and Frühwirth, T. 1998. On completion of constraint handling rules. In *Fourth International Conference on Principles and Practice of Constraint Programming (CP98)*. Lecture Notes in Computer Science, vol. 1520. Springer, 25–39.

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

Augustsson, L. 1998. Cayenne - a language with dependent types. In *Proc. International Conference on Functional Programming (ICFP 1998)*. 239–250.

Breazu-Tannen, V., Gunter, C. A., and Scedrov, A. 1990. Computing with coercions. In *Proc. of Conference on LISP and Functional Programming*. ACM, Nice, France, 44–60.

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

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

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

Duggan, D. and Ophel, J. 2002a. Open and closed scopes for constrained genericity. *Theoretical Computer Science 275,* 1–2, 215–258.

Duggan, D. and Ophel, J. 2002b. Type-checking multi-parameter type classes. *Journal of Functional Programming 12,* 2, 133–158.

Fridlender, D. and Indrika, M. 2000. Do we need dependent types? *Journal of Functional Programming 10,* 4, 409–415.

Frühwirth, T. 1995. Constraint handling rules. In *Constraint Programming: Basics and Trends*. LNCS, vol. 910. Springer.

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

Frühwirth, T. 1998b. Theory and practice of constraint handling rules. *Journal of Logic Programming 37,* 1–3, 95–138.

Gasbichler, M., Neubauer, M., Sperber, M., and Thiemann, P. 2002. Functional logic overloading. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, OR, 233–244.

GHC 2003. Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

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

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

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

Henglein, F. 1993. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems 15,* 1 (April), 253–289.

JEFFERY, D., HENDERSON, F., AND SOMOGYI, Z. 2000. Type classes in Mercury. In *Proc. Twenty-Third Australasian Computer Science Conf.* Australian Computer Science Communications, vol. 22. IEEE Computer Society Press, 128–135.

JONES, M. P. 1992. Qualified types: Theory and practice. Ph.D. thesis, Oxford University.

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

JONES, M. P. 1993b. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 52–61.

JONES, M. P. 2000. Type classes with functional dependencies. In *Proc. of the 9th European Symposium on Programming Languages and Systems, ESOP 2000*. LNCS, vol. 1782. Springer.

JONES, S. P., JONES, M. P., AND MEIJER, E. 1997. Type classes: an exploration of the design space. In *Haskell Workshop*.

KAES, S. 1988. Parametric overloading in polymorphic programming languages. In *ESOP'88 Programming Languages and Systems*. LNCS, vol. 300. Springer, 131–141.

LEWIS, J., SHIELDS, M., MEIJER, E., AND LAUNCHBURY, J. 2000. Implicit parameters: Dynamic scoping with static types. In *Proc. of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts*. 108–118.

MILNER, R. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*, 348–375.

NIPKOW, T. AND PREHOFER, C. 1995. Type reconstruction for type classes. *Journal of Functional Programming 5,* 2, 201–224.

ODERSKY, M., SULZMANN, M., AND WEHR, M. 1999. Type inference with constrained types. *Theory and Practice of Object Systems 5,* 1, 35–55.

ODERSKY, M., WADLER, P., AND WEHR, M. 1995. A second look at overloading. In *Proc. of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*. ACM Press, La Jolla, California, 135–146.

PEYTON JONES ET AL., S. 1999. Report on the programming language Haskell 98. http://haskell.org.

PLASMEIJER, M. AND VAN EEKELEN, M. 1998. Language report Concurrent Clean. Technical Report CSI-R9816, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands. June. ftp://ftp.cs.kun.nl/pub/Clean/Clean13/doc/refman13.ps.gz.

SHIELDS, M. AND JONES, S. P. 2001. Object-oriented overloading for Haskell. In *Workshop on Multi-Language Infrastructure and Interoperability*.

SHOENFIELD, J. 1967. *Mathematical Logic*. Addison-Wesley.

STUCKEY, P. J. AND SULZMANN, M. 2002. A theory of overloading. In *Proc. of ICFP'02*. 167–178.

SULZMANN, M. 2000. A general framework for Hindley/Milner type systems with constraints. Ph.D. thesis, Yale University, Department of Computer Science.

SULZMANN, M. AND WAZNY, J. 2003. The Chameleon system. `http://www.comp.nus.edu.sg/~sulzmann/chameleon`.

WADLER, P. AND BLOTT, S. 1989. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. 16th ACM Symposium on Principles of Programming Languages (POPL)*. 60–76.

YANG, Z. 1998. Encoding types in ML-like languages. In *Proc. of ACM SIGPLAN International Conference on Functional Programming*. 289–300.