A Theory of Overloading Part II: Semantics and Coherence

Andreas Rossberg
Programming Systems Lab
Universität des Saarlandes
Germany
rossberg@ps.uni-sb.de

Martin Sulzmann
Department of Computer Science
The University of Melbourne
Australia
sulzmann@cs.mu.oz.au

March 21, 2002

Abstract

This paper is a complement to “A Theory of Overloading” by Stuckey and Sulzmann. The authors introduce a general overloading framework based on Constraint Handling Rules (CHR). The main focus is on type inference issues. Here, we provide a rigorous study of semantic meanings for programs containing overloaded identifiers. Source expressions are translated by inserting appropriate evidence parameters which represent actual values of overloaded identifiers. We provide a general coherence results under some sufficient conditions on CHRs. We observe that the target language is typable in the Hindley/Milner system extended with polymorphic recursion. The coherence property is lost in case of a strict language. Surprisingly, in case we impose condition similar to the value-restriction found in ML, we can recover coherence for strict languages.

1 Introduction

In “A Theory of Overloading”, Stuckey and Sulzmann introduce a general theory for overloading. The main focus is on type inference. In this paper, we provide a thorough study of the semantic meaning of programs containing overloaded identifiers.

We follow the common approach [3] to give a semantic meaning for overloaded identifiers by passing around evidence values as additional function parameters. Evidence values represent the actual definitions of overloaded functions. Consider the following program.

overload eq :: Int → Int → Bool
eq = primEqInt

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

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

where tail :: [a] → [a] takes the tail of a list. In a first step we translate overloaded definitions. We find
eqEqInt = primEqInt
eqEqList eq = let eqL [] [] = True
            eqL (x::xs) [] = False
            eqL [] (y::ys) = False
            eqL (x::xs) (y::ys) = (eq x y) \&\& (eqL xs ys)
in eq

Constraints in the type component of source expressions are turned into function parameters, we also say evidence parameters, in the target language. In the above example, the function parameter \(eq\) represents evidence for the equality function on type \(a \to a \to \text{Bool}\). Note that this is represented by the constraint \(Eq (a \to a \to \text{Bool})\) in the type scheme of the source expression.

What remains is to insert the appropriate evidence values for expression \(\text{exp}\).

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

\[
\text{Eq} \ ([a] \to b \to c), \text{Eq} \ (\text{Int} \to \text{Int} \to d)
\]

where we assume \(\text{xs} :: [a], \text{ys} :: b, 1 :: \text{Int} \) and \(3 :: \text{Int}\). In our framework, constraint solving is defined in terms of Constraint Handling Rules (CHR). CHRs are a declarative constraint language to write incremental constraint solvers. Each of the two overloaded definitions gives rise for a CHR

simplification rule:

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

Rule (Eq1) states that a definition exists on type \(\text{Int} \to \text{Int} \to \text{Bool}\) and rule (Eq2) states a definition on type \([a] \to [a] \to \text{Bool}\) requires a definition on type \(a \to a \to \text{Bool}\). (Note that rule (Eq2) can be straightforwardly derived from the type annotation \(\forall a. \text{Eq} \ (a \to a \to \text{Bool}) \Rightarrow [a] \to [a] \to \text{Bool} \). Logically, \(\iff\) states an if-and-only-if relation. Operationally, a simplification rule can be read as follows. Whenever we find a term which matches the left-hand side, then this term can be simplified (replaced) by the right-hand side. We assume that substitutions represented by equations have been applied to user-defined constraints such as \(\text{Eq}\).

In addition to simplification rules, we also find propagation rules which allow us to impose stronger conditions. Consider the two propagation rules:

\[
\begin{align*}
\text{(Eq3)} & \quad \text{Eq} \ (\text{Int} \to \text{Int} \to a) \Rightarrow a = \text{Bool} \\
\text{(Eq4)} & \quad \text{Eq} \ ([a] \to b \to c) \Rightarrow b = [a], c = \text{Bool}
\end{align*}
\]

Rule (Eq3) states that any definition of equality which takes two arguments of type \(\text{Int}\) must have result type \(\text{Bool}\). A similar statement is expressed by rule (Eq4). Logically, the \(\Rightarrow\) symbol stands for boolean implication. Operationally, we propagate the right-hand side if there is a term matching the left-hand side. We commonly use \(P\) to refer to the set of CHRs. We identify by \(P_{\text{Simp}}\) the subset of all simplification rules in \(P\). Similarly, we write \(P_{\text{Prop}}\) for the subset of all propagation rules in \(P\). Write \(C \Rightarrow P, C'\) if constraint \(C\) can be reduced to \(C'\) by application of the set \(P\) of CHRs.

Consider the constraints generated from the program text of expression \(\text{exp}\). We find the following CHR derivation:

\[
\begin{align*}
\Rightarrow_{\text{Eq3}} \quad & \quad \text{Eq} \ ([a] \to b \to c), \text{Eq} \ (\text{Int} \to \text{Int} \to d) \\
\Rightarrow_{\text{Eq1}} \quad & \quad \text{Eq} \ ([a] \to b \to c), \text{Eq} \ (\text{Int} \to \text{Int} \to d), d = \text{Bool} \\
\Rightarrow_{\text{Eq4}} \quad & \quad \text{Eq} \ ([a] \to b \to c), b = [a], c = \text{Bool}, d = \text{Bool} \\
\Rightarrow_{\text{Eq2}} \quad & \quad \text{Eq} \ (a \to a \to \text{Bool}), b = [a], c = \text{Bool}, d = \text{Bool}
\end{align*}
\]

Resolution of remaining equalities via unification yields
\( \text{exp} :: \forall a. \text{Eq} \ (a \rightarrow a \rightarrow \text{Bool}) \Rightarrow [a] \rightarrow [a] \rightarrow (\text{Bool, Bool}) \)

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

Expression \( \text{exp} \)'s type states that we can implement \( \text{exp} \) given we can provide evidence for \( \text{Eq} \ (a \rightarrow a \rightarrow \text{Bool}) \). We commonly use \( \text{eq} \) to refer to evidence provided for a constraint \( C \).

The novelty of our approach is that the construction of evidence can be straightforwardly derived from the CHR derivation. Reading the CHR derivation backwards tells us exactly how this can be achieved. Constraint \( \text{Eq} \ (\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}) \) was reduced to \( \text{True} \) via rule (Eq1), therefore, we insert \( \text{eq} \) at the instantiation site. Constraint \( \text{Eq} \ ([a] \rightarrow [a] \rightarrow \text{Bool}) \) was reduced to \( \text{Eq} \ (a \rightarrow a \rightarrow \text{Bool}) \). That is, applying \( \text{eq} \) to evidence \( \text{eq} \ (a \rightarrow a \rightarrow \text{Bool}) \) yields evidence \( \text{eq} \ (\forall a. \text{Eq} \ (a \rightarrow a \rightarrow \text{Bool})) \). The final translated expression is as follows:

\[
\text{exp2} :: \forall a. (a \rightarrow a \rightarrow \text{Bool}) \Rightarrow [a] \rightarrow [a] \rightarrow (\text{Bool, Bool}) \\
\text{exp2} \ e \ xs \ ys = (\text{eq} \ (\text{eq} \ \text{List} \ e) \ (\text{tail} \ xs) \ ys, \ \text{eq} \ \text{Eq} \ \text{Int} \ 1 \ 3)
\]

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

In the remainder of this paper we will give a thorough description of how to translate source into target expressions including a general coherence result. As we saw in the example above, the translation depends on a programs typing. \( \text{Coherence} \) states that the translation is independent of a programs typing. The main result of this paper is that the translation is coherent if the CHRs involved are terminating, confluent, range-restricted and simplification rules are single-headed and non-overlapping. We will call such a set of CHRs a \( \text{good} \) set of CHRs. For terminology, definitions and related results we refer the reader to “A Theory of Overloading”.

2 Evidence-Passing Translation

The evidence-passing translation is driven by a programs typing. Source expressions are translated into target expressions by turning constraints in type schemes of source expressions into function parameters in target expressions. At instantiation sites we need to provide the appropriate evidence arguments.

Recall the rule for quantifier elimination of source expressions:

\[
P, C_1, \Gamma \vdash e : \forall \bar{\alpha}. C_2 \Rightarrow \tau \\
P, C_1, \Gamma \vdash [P] = C_1 \rightarrow [\bar{\alpha}/\bar{\alpha}] C_2
\]

We can eliminate a quantifier if the constraint component can be satisfied by the global context w.r.t. the current program theory, i.e. \( [P] \models C_1 \rightarrow [\bar{\alpha}/\bar{\alpha}] C_2 \). In the target language, we need to construct evidence \( \text{eq} \ (\forall \bar{\alpha}. C_2) \) out of \( \text{eq} \). This is done by a evidence constructing function written \( \text{eq} : C_1 \xrightarrow{P} [\bar{\alpha}/\bar{\alpha}] C_2 \). As we will see, such a function must exist because of premise \( [P] \models C_1 \rightarrow [\bar{\alpha}/\bar{\alpha}] C_2 \).

The target rule corresponding to quantifier elimination of source expressions is as follows:

\[
P, C_1, \Gamma \vdash e : \forall \bar{\alpha}. C_2 \Rightarrow \tau \leadsto E \\
[P] = C_1 \rightarrow [\bar{\alpha}/\bar{\alpha}] C_2 \\
\text{ee} :: C_1 \xrightarrow{P} [\bar{\alpha}/\bar{\alpha}] C_2 \\
P, C_1, \Gamma \vdash e : [\bar{\alpha}/\bar{\alpha}] \tau \leadsto E \ (\text{ee} \ \text{ee} \ C_1)
\]

Expression \( E \) is the translation of \( e \). Note that the translation depends on \( e \)'s type.
\[ \frac{(x : \sigma) \in \Gamma}{P, C, \Gamma \vdash x : \sigma} \quad \frac{\varepsilon : \sigma}{P, C, \Gamma_x \vdash e : \tau} \quad \frac{\lambda x.e : \tau \rightarrow \tau'}{P, C, \Gamma_x \vdash e : \tau'} \quad \frac{\sigma \notin \text{fr}(\Gamma)}{P, C_1, \Gamma \vdash e : \forall \alpha.C_2 \Rightarrow \tau} \]

\[ \frac{f_{\alpha} = \emptyset}{P, C, \Gamma \vdash (e :: \sigma) : \sigma} \quad \frac{P, C, \Gamma \vdash e : \forall \alpha.C_2 \Rightarrow \tau}{P, C_1, \Gamma \vdash e : \forall \alpha.C_2 \Rightarrow \tau} \quad \frac{\phi \text{ mgu of } h_{C_f}, \ F \phi_f \iff \phi_{C_f} \in P}{P, C, \Gamma \vdash p : \tau} \]

\[ \frac{P, C, \Gamma \vdash \text{overload } f \in p | \ldots}{P, C, \Gamma \vdash \text{overload } f :: \sigma \text{ instead of } \text{overload } f = (e :: \sigma) \text{ in } \ldots} \]

Figure 1: Typing Rules

### 2.1 Source Language

**Programs**

\[ p ::= \text{overload } f = e \text{ in } p | \ldots \]

**Expressions**

\[ e ::= x | \lambda x.e | e \ e | \text{let } x = e \text{ in } e \mid (e :: \sigma) \]

**Types**

\[ \tau ::= \alpha | \tau \rightarrow \tau | T \]

**Constraints**

\[ C ::= \tau | U \mid C \land C \]

**Type Schemes**

\[ \sigma ::= \tau | \forall \alpha.C \Rightarrow \tau \]

where \( \cdot \rightarrow \cdot \) is the function type constructor, \( T \) is a user-defined n-ary type constructor, \( \cdot = \cdot \) denotes (syntactic) equality among types, \( U \) is a user-defined n-ary predicate symbol, and \( \land \) denotes conjunction among constraints. Note that we use \( \wedge \) for conjunction among constraints in CHR rules and example CHR derivations.

For syntactic convenience, we write example programs using

\[ \text{overload } f :: \sigma \text{ instead of } \text{overload } f = (e :: \sigma) \text{ in } \ldots \]

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

The typing rules can be found in Figure 1. For explanations and inference results we refer to the related paper.
\[
\mathcal{V} = \mathbf{W} + \mathcal{V} \rightarrow \mathcal{V} + \sum_{k \in \mathcal{K}} (k \mathcal{V}_1 \ldots \mathcal{V}_\text{arity}(k))
\]

where \(\mathbf{W}\) is the error element and \(\mathcal{K}\) is the set of value constructors.

\[
[x]\eta = \eta(x)
\]

\[
[\lambda u. e]\eta = \lambda v.[e][u := v]
\]

\[
[e'']\eta = \begin{cases} & \text{if } [e]\eta \in \mathcal{V} \land [e']\eta \neq \mathbf{W} \\ & \text{then } ([e']\eta) ([e']\eta) \\ & \text{else } \mathbf{W} \\ \end{cases}
\]

\[
[\text{let } x = e'\text{ in } e''\eta = \begin{cases} & \text{if } [e]\eta \neq \mathbf{W} \\ & \text{then } [e']\eta[x := [e]\eta] \\ & \text{else } \mathbf{W} \\ \end{cases}
\]

\[
[\mu_1 \rightarrow \mu_2] = \{ f \in \mathcal{V} \rightarrow \mathcal{V} | v \in [\mu_1] \Rightarrow f v \in [\mu_2] \}
\]

\[
[T \mu_1 \ldots \mu_m] = (\bot) \cup \bigcup \{ [k [\mu'_1] \ldots [\mu'_n]] | k : \mu'_1 \rightarrow \ldots \rightarrow \mu'_n \rightarrow T \mu_1 \ldots \mu_m \}
\]

\[
[\forall \alpha. \tau] = \bigcap_{\eta} [\alpha/\eta]\tau
\]

where for monotypes \(\mu\) we require that \(\text{fe}() = \emptyset\) and \(\Gamma_{\text{init}}\) contains the set of value constructors used in this context.

---

**Figure 2: Target Semantics**

### 2.2 Target Language

We work with the following target language.

\[
\text{Target Expressions } E ::= x | \lambda x. E | \text{let } x = E \text{ in } E
\]

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

We interpret expressions in an untyped denotational semantics [2]. We refer to Figure 2 for definitions of domain equations and meaning functions on expressions and types.

### 2.3 Evidence

We give meaning to evidence variables via a variable environment \(\eta\) mapping variables to elements in \(\mathcal{V}\). We will always assume that \(\eta(e_{\text{True}}) = ()\) and \(\eta(e_{\tau_1 \rightarrow \tau_2}) = ()\) where \(\tau_1\) and \(\tau_2\) are syntactically equivalent and () represents a distinct value in \(\mathcal{V}\).

Variable environments \(\eta\) must satisfy program theories \(P\). We define \(\eta\) satisfies \(P\) iff for each user-defined atom \(U \tau\) where \(U \tau \rightarrow^p P \text{ True}\) we have that \(\eta(e_U \tau) \in [\forall \alpha_U \tau].\tau\). That means, evidence values of user-defined constraints must satisfy the given type specification. For example, \(E_\text{q} (\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}) \rightarrow^p P \text{ True}\), therefore, \(\eta(e_{E_\text{q} (\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool})}) \in [\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}]\) where \(P\) consists of rules (Eq1-4) from the Introduction.

Let \(C = U_1 \tau_1, \ldots, U_n \tau_n\) be a constraint such that \(C \rightarrow^p P \text{ True}\). Then, we require that \(\eta(C) = \langle \eta(e_{U_1 \tau_1}), \ldots, \eta(e_{U_n \tau_n}) \rangle\) as a further condition for variable environments. Note that we assume an ordering on \(U_1 \tau_1, \ldots, U_n \tau_n\).
$(\text{Var}) \quad \frac{(x : \sigma) \in \Gamma}{P, C, \Gamma \vdash x : \sigma \leadsto x}$ 

$(\text{Let}) \quad \frac{P, C, \Gamma, x : \tau \vdash e : \tau' \leadsto E}{P, C, \Gamma, x : \sigma \vdash e' : \tau' \leadsto E'}$ 

$(\text{Abs}) \quad \frac{P, C, \Gamma, x : \tau \vdash e : \tau' \leadsto E}{P, C, \Gamma, \lambda x.e : \tau \leadsto \lambda x.E}$ 

$(\text{App}) \quad \frac{P, C, \Gamma \vdash e_1 : \tau_1 \leadsto \tau_2 \leadsto E_1}{P, C, \Gamma \vdash e_2 : \tau_1 \leadsto E_2}$ 

$(\text{VI}) \quad \frac{P, C_1 \land C_2, \Gamma \vdash e : \tau \leadsto E}{\alpha \not\in f(r(C_1, \Gamma))} \quad \frac{P, C_1, \Gamma \vdash e : \forall \alpha. C_2 \Rightarrow \tau \leadsto \lambda \alpha C_2.E}{P, C_1, \Gamma \vdash e : \forall \alpha. C_2 \Rightarrow \tau \leadsto \lambda \alpha C_2.E}$ 

$(\text{VE}) \quad \frac{[P] \models C_1 \Rightarrow [\bar{\tau}/\alpha]C_2}{\alpha \not\in f(r(C_1))}$ 

$\frac{P, C_1 \vdash e : [\bar{\tau}/\alpha] \Rightarrow E(e C_1)}{P, C_1, \Gamma \vdash e : [\bar{\tau}/\alpha] \Rightarrow E(e C_1)}$ 

Figure 3: Evidence-passing translation

Let $C_1$ and $C_2$ be two constraints, $\eta$ a variable environment and $P$ be a good set of CHR rules such that $\eta$ satisfies $P$ and $[P] \models C_1 \Rightarrow C_2$. We say ec is an evidence constructor if for each substitution $\phi$ such that $\phi(C_1) \leadsto^p \text{true}$ we have that $\text{ec}(\phi(C_1)) = \eta(\phi(C_2))$. In such a situation we write $\text{ec} :: C_1 \Rightarrow^p C_2$. Note that the conditions imply that $\phi(C_1)$ (and $\phi(C_2)$) doesn’t contain any equality constraints. The Canonical Normal Form Lemma implies that $\phi(C_2) \Rightarrow^p \text{true}$ (if $\phi(C_1) \Rightarrow^p \text{true}$).

We maintain that if there exists $\eta$ which satisfies $P$, then the definition of evidence constructing functions is straightforward. However, one of the challenges will be to construct a variable environment $\eta$ which satisfies $P$.

2.4 Translation

Figure 3 describes the evidence-passing translation from source to target expressions via judgments of the form $P_1, C, \Gamma \vdash e : \sigma \leadsto E$. The translation is driven by a programs typing derivation. Note that we omitted the straightforward rule (Anom).

The most interesting rules are (VI) where we abstract over evidence variables and (VE) where we provide the proper evidence values. In case $\text{ec} :: C_1 \Rightarrow^p [\bar{\tau}/\alpha]C_2$ where $[\bar{\tau}/\alpha]C_2 \Rightarrow^p \text{true}$, we simply provide $e[\bar{\tau}/\alpha]C_2$ directly instead of $e C_1$ (see the final translated expression of exp from the Introduction).

The problem of translating overloaded definitions and building evidence constructors will be reduced to translating expressions. Both translations schemes are mutually recursive. For convenience, we will present both constructions in two separate statements (see Lemmas 1 and 2). Importantly, the translation schemes preserve typing as stated in Lemma 3.

We call $P_1, C, \Gamma \vdash e : \sigma$ a normalized derivation if no equality constraints appear in the constraint component of subderivations and type schemes and each entailment check $[P] \models C_1 \Rightarrow C_2$ holds iff $C_1, C_2 \Rightarrow^p C_1$.

Lemma 1 (Translation of Programs) Let $p$ be a program, $P_0$ be the set of simplification rules arising from $p$ and $P_p$ be a set of propagation rules such that $P = P_0 \cup P_p$ is good and $P, C, \Gamma \vdash e : \sigma$
is normalized for some $C$, $\Gamma$ and $\sigma$. Then, there exists a variable environment $\eta$ which satisfies $P$.

**Proof:** Consider each overloaded definition overload $f :: (\forall \alpha . C_f \Rightarrow \tau_f) = e$ in plus their associated simplification rule

$$F \tau \iff U_1 \tau_1, \ldots, U_n \tau_n$$

Note that $C_f = U_1 \tau_1, \ldots, U_n \tau_n$.

We translate $e$ yielding target expression $E$ i.e. $P, C, \Gamma \vdash e : \forall \alpha . C_f \Rightarrow \tau_f \Rightarrow E$. We generate

$$f_F \tau, U_1 \tau_1, \ldots, U_n \tau_n = E$$

Target expression $E$ might contain evidence constructors. For each $e : C_1 \Rightarrow C_2$ we generate an equation as described in Lemma 2 (therefore the process of translating programs and constructing evidence is mutually recursive).

Finally, for each overloaded identifier $f$ we generate $f = \lambda x. x$.

We record in $\eta$ the least fix point of the above set of equations.

In the following, we will use $\llbracket \cdot \rrbracket$ as a short-hand for $\llbracket \cdot \rrbracket_{\eta}$.

Consider $F \tau$ such that $F \tau \Rightarrow_p True$ (implies $F \tau \Rightarrow_p^{p_{simp}} True$). We provide an inductive construction of $\eta(e_F \tau)$.

For all $F \tau \Rightarrow_p^{p_{simp}} True$, we define $\eta(e_F \tau) = \eta(f_F \tau, True)$ where there exists $\phi$ such that $\tau \equiv \phi^2$ and $f_F \tau, True$ is as above. Note that $\eta(e_F \tau) \in \llbracket \cdot \rrbracket_{\eta}$ by construction (follows from Lemmas 2 and 3). Note that the CHRs are non-overlapping, therefore, there is only one such $f_F \tau, True$.

For all $F \tau \Rightarrow_p^{p_{simp}} U_1 \tau_1, \ldots, U_n \tau_n$, we have that $U_1 \tau_1 \Rightarrow_p^{p_{simp}} True$. Therefore, $\eta(e_{U_1 \tau_1})$ is defined. We define

$$\eta(e_F \tau) = \eta(f_F \tau, U_1 \tau_1, \ldots, U_n \tau_n) \ (\eta(e_{U_1 \tau_1}), \ldots, \eta(e_{U_n \tau_n}))$$

Note that $\eta(f_F \tau, U_1 \tau_1, \ldots, U_n \tau_n) \in \llbracket \cdot \rrbracket_{\eta}$ and $\eta(e_{U_1 \tau_1}) \in \llbracket \cdot \rrbracket_{\eta}$ (follows from Lemmas 2 and 3), therefore, $\eta(e_F \tau) \in \llbracket \cdot \rrbracket_{\eta}$.

Note that all simplification rules are non-overlapping, hence, for each $F \tau \Rightarrow_p^{p_{simp}} True$ the derivation is unique. Therefore, the above construction is deterministic.

A note on the ordering among user-defined constraints $U_1 \tau_1$:

We denote by $\leq$ the ordering relation among constraints. Consider (R) $F (a \Rightarrow (b \Rightarrow b)) \iff U a, U (b \Rightarrow b)$ in some set $P$ of CHRs where $U a \leq U (b \Rightarrow b)$. Assume $F (\tau_1 \Rightarrow \tau_2) \Rightarrow_p True$ for some types $\tau_1$ and $\tau_2$ such that $U \tau_2 \leq U \tau_1$. Clearly, $\leq$ is not stable under substitution. We find that $F (\tau_1 \Rightarrow \tau_2) \Rightarrow_R U \tau_2, U \tau_1 \Rightarrow_p True$. However, note that $f_F (a \Rightarrow (b \Rightarrow b)), U a, U (b \Rightarrow b) = \lambda(e_U a, e_U (b \Rightarrow b)) \ldots$. Therefore, we assume that each simplification rule step keeps the constraints in the same order as in the corresponding function definition. We have that $F (\tau_1 \Rightarrow \tau_2) \Rightarrow_R U \tau_1, U \tau_2 \Rightarrow_p True$. 

$\square$

Let $C = U_1 \tau_1 \land \ldots U_n \tau_n$ be a constraint. We define $\text{typeof}(C) = (\tau_1, \ldots, \tau_n)$. Recall that we assume an ordering on constraints.
Lemma 2 (Construction of Evidence) Let $p$ be a program, $P_i$ be the set of simplification rules arising from $p$ and $P_p$ be a set of propagation rules such that $P = P_i \cup P_p$ is good and $P, C, \Gamma \vdash p : \sigma$ is normalized for some $C$, $\Gamma$ and $\sigma$. Let $C_1$ and $C_2$ be two sets of user-defined constraints such that $C_1, C_2 \Rightarrow^*_p C_1$. Then, there exists a closed definition for $ec :: C_1 \xrightarrow{P} C_2$. In particular, the definition of $ec$ has type $\forall \Delta. \text{typoef}(C_1) \Rightarrow \text{typoef}(C_2)$ where $\Delta = f(\text{typoef}(C_1) \Rightarrow \text{typoef}(C_2))$.

**Proof:** For simplicity, we only consider the case if $C_2 = U \tau_1 \Rightarrow C_1$. We first try to find a closed definition $ec : C_1 \xrightarrow{P} C_2$. 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 $(R) \ U \tau' \Rightarrow C_3 \in P$ and there exists $\phi$ such that $\phi\tau' \equiv \tau$.

We find that $C_1, C_2 \Rightarrow_R C_1, \phi C_3$. For each $U_i \tau_i \in \phi \Delta$ we have that $C_1, U_i \tau_i \Rightarrow^* C_1$. By induction, there exist closed definitions $ec_i :: C_1 \xrightarrow{P} U_i \tau_i$. We define $ec(e_1, \ldots, e_m) = (ecU \tau' \phi \Delta (ec(e_1, \ldots, e_m)), \ldots, (ecU \tau' \phi \Delta (ec(e_1, \ldots, e_m))))$ where $ecU \tau' \phi \Delta$ is as in Lemma 1 and are done. Note that we yet again have to be aware of the “ordering” among constraints (see Lemma 1).

The statement about typability of $ec$ follows from Lemma 3. □

Note that the above statement excludes evidence constructors such as

$$ec_1 :: Eq ([a] \rightarrow [a] \rightarrow \text{Bool} \ xrightarrow{P} Eq (a \rightarrow a \rightarrow \text{Bool})$$

We can’t give a closed definition for $ec_1$, however, $ec_1$ must exist. For any $\tau$ such that $Eq ([\tau] \rightarrow [\tau] \rightarrow \text{Bool}) \Rightarrow^*_p \text{True}$ we can construct evidence for $\text{ec}_q (\tau \text{arrow Bool})$.

We state that the translation of normalized expression preserves typing.

Lemma 3 (Typability) Let $P, C', \Gamma' \vdash e : \forall \alpha. C \Rightarrow \tau \Rightarrow E$ be a normalized derivation. Let $\Gamma$ derived from $\Gamma'$ by replacing each $f : \forall \alpha. F \alpha \Rightarrow \alpha \in \Gamma'$. Let $f : \forall \alpha. \alpha \Rightarrow \alpha$. Then $\Gamma \vdash E : \text{typoef}(C) \Rightarrow \tau$ is derivable in the Hindley/Milner system with polymorphic recursion.

Consider

overload eq :: Int \rightarrow Int \rightarrow Bool

$\text{eq} = \text{primEqInt}$

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

$\text{eq} = \text{let eqList :: $\forall a. Eq (a \rightarrow a \rightarrow \text{Bool}) \Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool}$

eqList [] [] = True

eqList (xs:ys) [] = False

eqList [] (ys:ys) = False

eqList (xs:ys) (ys:ys) = (xs \times ys) \&\& (eqList xs ys)

\&\& (\text{ordList}[1][2])

\text{in eqList overload ord :: Int \rightarrow Int \rightarrow Bool}$

$\text{ord} = \text{primOrdInt}$

overload ord :: $\forall a. Ord (a \rightarrow a \rightarrow \text{Bool}) \Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool}$

$\text{ord} = \text{let ordList :: $\forall a. Ord (a \rightarrow a \rightarrow \text{Bool}) \Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool}$

ordList [] [] = True

ordList (xs:ys) [] = False

ordList [] (ys:ys) = False


8
ordList (xxs) (y:ys) = (ord x y) && (ordList xs ys)
&& (eq [1] [2])

in ordList

Our translation yields

eq = Int -> Int -> Bool

eq = primEqInt

eqL :: \forall a. (a -> a -> Bool) -> [a] -> [a] -> Bool
eqL eq = let eqList :: \forall a. [a] -> [a] -> Bool
           eqList [] [] = True
           eqList (x:xs) [] = False
           eqList [] (y:ys) = False
           eqList (x:xs) (y:ys) = (eq x y) && (eqL eq xs ys)
&& (ordL ordl [1] [2])
in eqList ordl :: Int -> Int -> Bool

ordl = primOrdInt

ordL :: \forall a. (a -> a -> Bool) -> [a] -> [a] -> Bool

ordL ord = let ordList :: \forall a. [a] -> [a] -> Bool
            ordList [] [] = True
            ordList (x:xs) [] = False
            ordList [] (y:ys) = False
            ordList (x:xs) (y:ys) = (ord x y) && (ordL ordl xs ys)
&& (eqL eqL [1] [2])
in ordList

Note that we have already simplified slightly the translated expressions. Instead of \f \in e\ e we simply write \in e e.

For example, we find ordL ordl because of Ord ([Int] -> [Int] -> Bool), Eq (a -> Bool) \Rightarrow^* p

Eq (a -> Bool).

We observe that the translated expression is typeable in the Hindley/Milner system extended
with polymorphic recursion.

2.5 Unambiguity

Unambiguity of a type scheme \forall \alpha. C \Rightarrow \tau implies that any ground instance of the type
component \tau uniquely determines the free variables in the constraint component C.

**Lemma 4 (Uniqueness)** Let P be a good set of CHRs, \forall \alpha. C \Rightarrow \tau an unambiguous type scheme,
\phi a mapping from type variables \text{fs}(C, \tau) to ground types and \phi' a mapping from type variables
\text{fs}(\tau) to ground types such that \phi' \leq \phi and \phi C \Rightarrow^* p. Then, C, \phi \Rightarrow^*_p \phi' such that \phi' \cup \phi'' = \phi.

**Proof:** Note that we interpret substitutions as equality constraints. Then, the \leq relation among substitutions can be expressed as constraint entailment, and \cup corresponds to conjunction.

By assumption \forall \alpha. C \Rightarrow \tau is unambiguous. Let \rho be a variable renaming. We have that

\[ [P] = (C \land \rho(C) \land \tau = \rho(\tau)) \Rightarrow (\alpha = \rho(\alpha)) \]

for variables \alpha \in \text{fs}(C, \tau). Then, we find that

\[ [P] = (C \land \phi(C) \land \tau = \phi(\tau)) \Rightarrow (\alpha = \phi(\alpha)) \]

for variables \alpha \in \text{fs}(C, \tau). From that, we conclude

\[ [P] = (C \land \phi(C) \land \phi) \leftrightarrow (C \land \phi(C) \land \phi') \]
By assumption we know that $\phi(C) \models \uparrow, True$. The Canonical Form Lemma enforces that $C, \phi' \models \uparrow, \phi''$. □

2.6 Coherence

The exercise in Section 2.4 was to ensure that there exists a variable environment $\eta$ which satisfies $P$. In fact, for normalized expressions we could provide a compilation scheme which preserves typing.

We will now establish a coherence result under the assumption that we have an $\eta$ such that $\eta$ satisfies $P$. The technical challenge is to introduce coercion (ordering) relations between different typing derivations.

Recall the example from the Introduction.

\[ \text{exp} :: \forall a. \text{Eq} \ (a \to a \to \text{Bool}) \Rightarrow [\alpha] \to [\alpha] \to (\text{Bool}, \text{Bool}) \]
\[ \text{exp} \ x \ y \ s = (\text{eq} \ (\text{tail} \ x) \ y, \ \text{eq} \ 1 \ 3) \]

and

\[ \text{exp}' :: \forall \cdot [\text{Int}] \to [\text{Int}] \to (\text{Bool}, \text{Bool}) \]
\[ \text{exp}' \ x \ y \ s = (\text{eq} \ (\text{tail} \ x) \ y, \ \text{eq} \ 1 \ 3) \]

are two valid typings for the same expression. We find the following two target expressions:

\[ \text{exp}2 :: \forall a. (a \to a \to \text{Bool}) \Rightarrow [\alpha] \to [\alpha] \to (\text{Bool}, \text{Bool}) \]
\[ \text{exp}2 \ e \ x \ y \ s = (\text{eq} \ (\text{ecEqList} \ e) \ (\text{tail} \ x) \ y, \ \text{eq} \ \text{ecEqInt} \ 1 \ 3) \]

and

\[ \text{exp}2' :: \forall \cdot [\text{Int}] \to [\text{Int}] \to (\text{Bool}, \text{Bool}) \]
\[ \text{exp}2' \ x \ y \ s = (\text{eq} \ (\text{ecEqList} \ \text{ecEqInt}) \ (\text{tail} \ x) \ y, \ \text{eq} \ \text{ecEqInt} \ 1 \ 3) \]

Clearly, expressions $\text{exp}$ type and translation is more general than that of $\text{exp}'$. Technically, we introduce 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 Lemma 8). Unambiguity becomes now important.

**Definition 1** Let $P$ be a set of CHR, $C$ a constraint, $\sigma_1 = \forall x_1. C_1 \Rightarrow \tau_1$ and $\sigma_2 = \forall x_2. C_2 \Rightarrow \tau_2$ two type schemes, $E_1$ and $E_2$ two target expressions, and $\eta_1$ and $\eta_2$ two variable environments. We define $P, C \models_{\text{tv}} (\sigma_1, E_1, \eta_1) \preceq (\sigma_2, E_2, \eta_2)$ iff

- $P$ is good and $\sigma_1$ is unambiguous,
- $\eta_1$ and $\eta_2$ satisfy $P$,
- $[P] \land C \vdash_1 \sigma_1 \preceq \sigma_2$
- for any $\phi$ such that $[P] \models (C \land C_2 \land \tau_1 = \tau_2)$, we have that $[\pi E_1] \eta_1 (\phi_{C_1}) = [\pi E_2] \eta_2 (\phi_{C_2})$

where $\phi'$ is the unique extension of $\phi$ such that $[\phi'] = \phi \land C_1$ and $\pi$ is mapping on evidence variables with $\pi(e_C) = e_{C'}$.

We define $P, C \models_{\text{tv}} (\Gamma_1, \eta_1) \preceq (\Gamma_2, \eta_2)$ iff $P, C \models_{\text{tv}} (\sigma_1, x_1, \eta_1) \preceq (\sigma'_1, x_1, \eta_2), \ldots, P, C \models_{\text{tv}} (\sigma_n, x_n, \eta_1) \preceq (\sigma'_n, x_n, \eta_2)$ 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 $\phi'$ is ensured by Lemma 4.

We consider $(\tau, E, \eta)$ as a short-hand for $(\forall \alpha. \alpha = \tau \Rightarrow \alpha, \lambda x. E, \eta)$ where $\alpha$ and $x$ are fresh variables. In this particular case, we find that if $P, C \vdash^{ue} (\tau_1, E_1, \eta_1) \preceq (\tau_2, E_2, \eta_2)$ where $f(\tau_1, \tau_2) = \emptyset$ and $C \Rightarrow^* P, \text{True}$, then $\llbracket E_1 \rrbracket_\eta = \llbracket E_2 \rrbracket_{\eta_2}$.

Coming back to the example above we find that

$$P, \text{True} \vdash^{ue} (\forall a. \text{Eq} (a \rightarrow a \rightarrow \text{Bool}) \Rightarrow [a] \rightarrow [a] \rightarrow (\text{Bool}, \text{Bool}), \exp 2, \eta) \preceq ([\text{Int}] \rightarrow [\text{Int}] \rightarrow [\text{Int}] \rightarrow (\text{Bool}, \text{Bool}), \exp 2, \eta)$$

where $P$ consists of rules (Eq1-4) from the Introduction and $\eta$ is the variable environment as induced by Lemma 1.

**Lemma 5 (Transitivity)** The above relation is transitive.

**Lemma 6 (Instantiation)** Let $P$ be a set of CHR$s, C_2$ and $C'_2$ two constraints, $\sigma_1$ a type scheme, $E_1$ and $E_2$ two target expressions, $\eta_1$ and $\eta_2$ two variable environments, $\tilde{\alpha}_2$ a sequence of type variables, $eC_2$ a evidence variable and $ec : C_2 \Rightarrow [a] \rightarrow [a] \rightarrow (\text{Bool}, \text{Bool}), \exp 2, \eta$.

Then,

$$P, C_2 \vdash^{ue} (\sigma_1, E_1, \eta_1) \preceq ([a] \rightarrow [a] \rightarrow [\text{Int}] \rightarrow (\text{Bool}, \text{Bool}), \exp 2, \eta).$$

**Lemma 7 (Generalization)** Let $P$ be a set of CHR$s, C_2$ and $C'_2$ two constraints, $\tilde{\alpha}_2$ a sequence of type variables, $\sigma_1$ a type scheme, $\tau_2$ a type, $E_1$ and $E_2$ two target expressions and $\eta_1$ and $\eta_2$ two variable environments such that $P, C_2 \land C'_2 \vdash^{ue} (\sigma_1, E_1, \eta_1) \preceq (\sigma_2, E_2, \eta_2)$ and $\tilde{\alpha}_2 \notin f(C_2, \sigma_1)$.

Then,

$$P, C_2 \vdash^{ue} (\sigma_1, E_1, \eta_1) \preceq (\forall \tilde{\alpha}_2, C'_2 \Rightarrow \tau_2, \lambda eC_2, E_2, \eta_2).$$

In order to define on ordering on typing derivations, we will first need to define levels of derivations trees.

Let $P, C, \Gamma \vdash e : \sigma \rightarrow E$ be a judgment. The derivation tree of $P, C, \Gamma \vdash e : \sigma \rightarrow E$ is an up-side down tree where all leave 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 $P, C, \Gamma \vdash e : \sigma \rightarrow E$. We refer to the level of $P, C, \Gamma \vdash e : \sigma \rightarrow E$ as the sum of all of the paths in the derivation tree.

**Definition 2** Let $P$ be a set of CHR$s, C_1$ and $C_2$ two constraints, $\Gamma_1$ and $\Gamma_2$ two type environments, $\sigma_1$ and $\sigma_2$ two type schemes, $E_1$ and $E_2$ two target expressions and $e$ a source expression. Then, we define

$$(P, C_1, \Gamma_1 \vdash e : \sigma_1 \rightarrow E_1) \preceq (P, C_2, \Gamma_2 \vdash e : \sigma_2 \rightarrow E_2)$$

iff

- $P$ is good and $\sigma_1$ is unambiguous,

- $[P] \Downarrow (\exists \Gamma_2 (C_1)$,

- $[P] \land C_2 \vdash^{i} \sigma_1 \preceq \sigma_2, [P] \land C_2 \vdash^{i} \Gamma_1 \preceq \Gamma_2,$ and
\[ \text{let } D_1 \text{ be the derivation tree of } P, C_1, \Gamma_1 \vdash e : \sigma_1 \leadsto E_1, \text{ and let } D_2 \text{ be the derivation tree of } P, C_2, \Gamma_2 \vdash e : \sigma_2 \leadsto E_2. \text{ Then relation } \preceq \text{ holds for all sub-derivations } P, C_1', \Gamma_1' \vdash e' : \sigma_1' \leadsto E_1' \text{ in } D_1 \text{ and } P, C_2', \Gamma_2' \vdash e' : \sigma_2' \leadsto E_2' \text{ in } D_2 \text{ where we assume that } P, C_1', \Gamma_1' \vdash e' : \sigma_1' \leadsto E_1' \text{ has a lower level in } D_1 \text{ than } P, C_2', \Gamma_2' \vdash e : \sigma_2 \leadsto E_2 \text{ in } D_2. \]

**Lemma 8 (Confluent Translations)** Let \( P \) be a good set of CHRs, \( P, C_1, \Gamma_1 \vdash e : \sigma_1 \leadsto E_1, \) \( P, C_2, \Gamma_2 \vdash e : \sigma_2 \leadsto E_2 \) be two judgments, \( \eta_1 \) and \( \eta_2 \) two variable environments such that \( (P, C_1, \Gamma_1 \vdash e : \sigma_1 \leadsto E_1) \preceq (P, C_2, \Gamma_2 \vdash e : \sigma_2 \leadsto E_2) \) and \( P, C_2 \vdash^{eq} (\Gamma_1, \eta_1) \preceq (\Gamma_2, \eta_2). \) Then \( P, C_2 \vdash^{eq} (\sigma_1, E_1, \eta_1) \preceq (\sigma_2, E_2, \eta_2). \)

**Proof:** We proceed by induction over the derivation \( P, C_2, \Gamma_2 \vdash e : \sigma_2 \leadsto E_2. \) We only show some of the interesting cases.

**Case (Var)** We find the following situation:

\[
\begin{array}{c}
(x : \sigma_2) \in \Gamma_2 \\
\hline \\
P, C_2, \Gamma_2 \vdash x : \sigma_2 \leadsto x
\end{array}
\]

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

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

\[
\begin{array}{c}
P, C_2, \Gamma_2 \vdash e : \forall \tilde{a}. C_2' \Rightarrow \tau_2 \leadsto E_2 \\
\hline \\
\text{ By assumption, we have that } (P, C, \Gamma_2 \vdash e : \sigma_2 \leadsto E_2) \text{ and } P, C_2 \vdash^{eq} (\Gamma_1, \eta_1) \preceq (\Gamma_2, \eta_2). \text{ Then } P, C_2 \vdash^{eq} (\sigma_2, E_1, \eta_1) \preceq (\sigma_2, E_2, \eta_2). \\
\end{array}
\]

We can apply the induction hypothesis to the premise. Application of Lemma 6 yields the desired result.

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

\[
\begin{array}{c}
P, C_2, C_2', \Gamma_2 \vdash e : \tau_2 \leadsto E_2 \\
\hline \\
\text{ Application of the induction hypothesis and Lemma 7 yields the desired result.}
\end{array}
\]

**Case (Let)** We find the following situation:

\[
\begin{array}{c}
P, C_2, (\Gamma_2)_x \vdash e : \sigma_2 \leadsto E_2 \\
\hline \\
P, C_2, (\Gamma_2)_x \vdash e : \sigma_2 \leadsto E_2'
\end{array}
\]

where

\[
(P, C_1, (\Gamma_1)_x \vdash e : \sigma_1 \leadsto E_1) \preceq (P, C_2, (\Gamma_2)_x \vdash e : \sigma_2 \leadsto E_2)
\]

12
for some $\sigma_1$ and $E_1$.

Application of the induction hypothesis to the top premise yields

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

We set $\eta'_1 = \eta_1[x := \lambda e_{C_1, [x]} E_1] \eta_1$ and $\eta'_2 = \eta_2[x := \lambda e_{C_2, [x]} E_2] \eta_2$. The induction hypothesis applied to the bottom premise yields

$$P, C_2 \vdash^u (\tau'_1, E'_1, \eta'_1) \preceq (\tau'_2, E'_2, \eta'_2)$$

We can conclude that

$$P, C_2 \vdash^u (\tau'_1, \text{let } x = E_1 \text{ in } E'_1, \eta_1) \preceq (\tau'_2, \text{let } x = E_2 \text{ in } E'_2, \eta_2)$$

and we are done. \qed

We say a derivation $P, C, \Gamma \vdash e : \sigma \leadsto E$ is principal and unambiguous iff for any other $P, C', \Gamma \vdash e : \sigma' \leadsto E'$ we have that $(P, C, \Gamma \vdash e : \sigma \leadsto E) \preceq (P, C', \Gamma \vdash e : \sigma' \leadsto E')$.

It remains to give a meaning to primitive functions. Let $\eta$ be a variable environment and $\Gamma$ a closed type environment $(f_\eta(\Gamma) = \emptyset)$. We define $\eta \vdash \Gamma$ iff (1) for each $f : \forall a. F a \Rightarrow a \in \Gamma$ we have that $\eta(f) = \lambda x. a$, and (2) for all other $x : \sigma \in \Gamma$ we have that $\eta(x) \in [\sigma]$.

**Theorem 1 (Coherence)** Let $P, C_1, \Gamma \vdash e : \tau \leadsto E_1$ and $P, C_2, \Gamma \vdash e : \tau \leadsto E_2$ be two judgments and $\eta$ be a variable environment such that the principal derivation is unambiguous, $C_1 \vdash^u \text{ True, } C_2 \vdash^u \text{ True, } \eta \vdash \Gamma$, $\eta$ satisfies $P$, $f_\eta(\Gamma, \sigma) = \emptyset$ and $P$ is a terminating, confluent set of CHR$\text{s}$ where each simplification rule is single-headed and non-overlapping. Then $[E_1]_\eta = [E_2]_\eta$.

**Proof:** The principal derivation is unambiguous. That means, we find that

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

and

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

for some $C_3$ and $\tau'$.

Application of Lemma 8 yields

$$P, C_1 \vdash^u (\tau', E_3, \eta) \preceq (\tau, E_1, \eta)$$

and

$$P, C_2 \vdash^u (\tau', E_3, \eta) \preceq (\tau, E_2, \eta)$$

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. \qed

We note that we will immediately lose coherence in case of a strict language.

Consider

**overload** $f :: \text{Unit} \rightarrow \text{Int}$

```haskell
f x = f x
in let a = f ()
in ()
```

The let-bound expression $a$ has two possible typings, $\forall b. F (\text{Unit} \rightarrow b) \Rightarrow b$ and $\text{Int}$. Note that the translation under $a :: \text{Int}$ doesn’t terminate. Relates to value restriction in ML.
3 Conclusion

We could provide a compilation scheme for normalized programs which preserves typing. We believe that it is also possible to provide an interpreter-style translation scheme where evidence values will be constructed at run-time.

The translation scheme is coherent under the assumption that the set of CHRs is terminating, confluent, range-restricted and simplification rules are single-headed and non-overlapping. This extends previously known coherence results [1].

References

