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 (CHRs). The main focuss 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 focuss 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 :: $Int \to Int \to Bool$
         eq = primEqInt
overload eq :: $\forall a.Eq\ (a \to a \to Bool) \Rightarrow [a] \to [a] \to Bool$
         eq = let eqL :: $\forall a.Eq\ (a \to a \to Bool) \Rightarrow$
                           $[a] \to [a] \to Bool$
                  eqL [] [] = True
                  eqL (x:xs) [] = False
                  eqL [] (y:ys) = False
                  eqL (x:xs) (y:ys) = (eq x y) && (eqList xs ys)
              in  eqL

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

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

```
ecEqInt = primEqInt
ecEqList 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 $a \rightarrow a \rightarrow Bool$. Note that this is represented by the constraint $Eq\ (a \rightarrow a \rightarrow Bool)$ in the type scheme of the source expression. What remains is to insert the appropriate evidence values for expression exp.

Expression exp gives rise to the following constraints

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

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

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

Rule (Eq1) states that a definition exists on type $Int \rightarrow Int \rightarrow Bool$ and rule (Eq2) states a definition on type $[a] \rightarrow [a] \rightarrow Bool$ requires a definition on type $a \rightarrow a \rightarrow Bool$ (Note that rule (Eq2) can be straightforwardly derived from the type annotation $\forall a.Eq\ (a \rightarrow a \rightarrow Bool) \Rightarrow [a] \rightarrow [a] \rightarrow Bool$). Logically, $\Longleftrightarrow$ 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 Eq.

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

$$\begin{array}{lrcl}
\text{(Eq3)} & Eq\ (Int \rightarrow Int \rightarrow a) & \Longrightarrow & a = Bool \\
\text{(Eq4)} & Eq\ ([a] \rightarrow b \rightarrow c) & \Longrightarrow & b = [a], c = Bool
\end{array}$$

Rule (Eq3) states that any definition of equality which takes two arguments of type Int must have result type $Bool$. A similar statement is expressed by rule (Eq4). Logically, the $\Longrightarrow$ 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_{Simp} the subset of all simplification rules in P. Similarly, we write P_{Prop} for the subset of all propagation rules in P. We write $C \rightarrowtail_P^* C'$ 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 exp. We find the following CHR derivation:

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

Resolution of remaining equalities via unification yields

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

Expression exp's type states that we can implement exp given we can provide evidence for $Eq\ (a \rightarrow a \rightarrow Bool)$. We commonly use e_C 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 $Eq\ (Int \rightarrow Int \rightarrow Bool)$ was reduced to $True$ via rule (Eq1), therefore, we insert ecEqInt at the instantiation site. Constraint $Eq\ ([a] \rightarrow [a] \rightarrow Bool)$ was reduced to $Eq\ (a \rightarrow a \rightarrow Bool)$. That is, applying ecEqList to evidence $e_{Eq\ (a \rightarrow a \rightarrow Bool)}$ yields evidence $e_{Eq\ ([a] \rightarrow [a] \rightarrow Bool)}$. The final translated expression is as follows:

```
exp2 :: ∀a.(a → a → Bool) → [a] → [a] → (Bool, Bool)
exp2 e xs ys = (eq (ecEqList e) (tail xs) ys, eq ecEqInt 1 3)
```

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

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. *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 *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 paramters in target expressions. At instantiation sites we need to provide the appropriate evidence arguments.

Recall the rule for quantifier elimination of source expressions:

$$\frac{P, C_1, \Gamma \vdash e : \forall \bar{\alpha}.C_2 \Rightarrow \tau \qquad [\![P]\!] \models C_1 \rightarrow [\bar{\tau}/\bar{\alpha}]C_2}{P, C_1, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau}$$

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. $[\![P]\!] \models C_1 \rightarrow [\bar{\tau}/\bar{\alpha}]C_2$. In the target language, we need to construct evidence $e_{[\bar{\tau}/\bar{\alpha}]C_2}$ out of e_{C_1}. This is done by a evidence constructing function written $ec : C_1 \overset{P}{\rightarrow} [\bar{\tau}/\bar{\alpha}]C_2$. As we will see, such a function must exist because of premise $[\![P]\!] \models C_1 \rightarrow [\bar{\tau}/\bar{\alpha}]C_2$.

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

$$\frac{\begin{array}{c} P, C_1, \Gamma \vdash e : \forall \bar{\alpha}.C_2 \Rightarrow \tau \leadsto E \\ [\![P]\!] \models C_1 \rightarrow [\bar{\tau}/\bar{\alpha}]C_2 \\ ec :: C_1 \overset{P}{\rightarrow} [\bar{\tau}/\bar{\alpha}]C_2 \end{array}}{P, C_1, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau \leadsto E\,(ec\ e_{C_1})}$$

Expression E is the translation of e. Note that the translation dependes on e's type.

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

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

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

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

Figure 1: Typing Rules

2.1 Source Language

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

where $\cdot\to\cdot$ is the function type constructor, T is a user-defined n-ary type constructor, $\cdot=\cdot$ denotes (syntactic) equality among types, U is a user-defined n-ary predicate symbol. and $\wedge$ denotes conjunction among constraints. Note that we use ',' for conjunction among constraints in CHR rules and example CHR derivations.

For syntactic convenience, we write example programs using

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

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

The typing rules can be found in Figure 1. For explanations and inference results we refer to the related paper.

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

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

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

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

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

Figure 2: Target Semantics

2.2 Target Language

We work with the following target language.

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

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

We 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 η mapping variables to elements in $\mathcal{V}$. We will always assume that $\eta(e_{True}) = ()$ and $\eta(e_{\tau_1=\tau_2}) = ()$ where τ_1 and τ_2 are syntactically equivalent and $()$ represents a distinct value in $\mathcal{V}$.

Variable environments η must satisfy program theories P. We define η *satisfies* P iff for each user-defined atom $U\ \tau$ where $U\ \tau \rightarrowtail_P^* True$ 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, $Eq\ (Int \to Int \to Bool) \rightarrowtail_P^* True$, therefore, $\eta(e_{Eq\ (Int \to Int \to Bool)}) \in [\![Int \to Int \to Bool]\!]$ 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 $C \rightarrowtail_P^* True$. 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 $U_1\ \tau_1, \ldots, U_n\ \tau_n$.

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

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

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

Figure 3: Evidence-passing translation

Let C_1 and C_2 be two constraints, η a variable environment and P be a good set of CHRs such that η satisfies P and $[\![P]\!]\models C_1\to C_2$. We say ec is an *evidence constructor* iff for each substitution ϕ such that $\phi(C_1)\rightarrowtail_P^* \mathit{True}$ we have that $ec(\eta(e_{\phi(C_1)}))=\eta(e_{\phi(C_2)})$. In such a situation we write $ec::C_1\overset{P}{\to}C_2$. 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 $\phi(C_2)\rightarrowtail_P^*\mathit{True}$ (if $\phi(C_1)\rightarrowtail_P^*\mathit{True}$).

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

2.4 Translation

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

The most interesting rules are ($\forall$I) where we abstract over evidence variables and ($\forall$E) where we provide the proper evidence values. In case $ec::C_1\overset{P}{\to}[\bar\tau/\bar\alpha]C_2$ where $[\bar\tau/\bar\alpha]C_2\rightarrowtail_P^*\mathit{True}$, we simply provide $e_{[\bar\tau/\bar\alpha]C_2}$ directly instead of $ec\ e_{C_1}$ (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 seperate statements (see Lemmas 1 and 2). Importantly, the translation schemes preserve typing as stated in Lemma 3.

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

Lemma 1 (Translation of Programs) *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, Γ and σ. Then, there exists a variable environment η which satisfies P.

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

$$F\ \tau \Longleftrightarrow U_1\ \tau_1, \dots, U_n\ \tau_n$$

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

We translate e yielding target expression E, i.e. $P, C, \Gamma \vdash e : \forall\bar{a}.C_f \Rightarrow \tau_f \leadsto E$. We generate

$$f_{F\ \tau, U_1\ \tau_1, \dots, U_n\ \tau_n} = E$$

Target expression E might contain evidence constructors. For each $ec :: C_1 \overset{P}{\to} C_2$ 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 η the least fix point of the above set of equations.

In the following, we will use $[\![\forall.\tau]\!]$ as a short-hand for $[\![\forall fv(\tau).\tau]\!]$.

Consider $F\ \tau$ such that $F\ \tau \rightarrowtail_P^* True$ (implies $F\ \tau \rightarrowtail_{P_{Simp}}^* True$). We provide an inductive construction of $\eta(e_{F\ \tau})$.

For all $F\ \tau \rightarrowtail_{P_{Simp}} True$, we define $\eta(e_{F\ \tau}) = \eta(f_{F\ \tau', True})$ where there exists ϕ such that $\tau \equiv \phi\tau'$ and $f_{F\ \tau', True}$ is as above. Note that $\eta(e_{F\ \tau}) \in [\![\forall.\tau]\!]$ 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 $F\ \tau \rightarrowtail_{P_{Simp}} U_1\ \tau_1, \dots, U_n\ \tau_n$, we have that $U_i\ \tau_i \rightarrowtail_{P_{Simp}}^* True$. Therefore, $\eta(e_{U_i\ \tau_i})$ is defined. We define

$$\eta(e_{F\ \tau}) = \eta(f_{F\ \tau, U_1\ \tau_1, \dots, U_n\ \tau_n})\ (\eta(e_{U_1\ \tau_1}), \dots, \eta(e_{U_n\ \tau_n}))$$

Note that $\eta(f_{F\ \tau, U_1\ \tau_1, \dots, U_n\ \tau_n}) \in [\![\forall.(\tau_1, \dots, \tau_n) \to \tau]\!]$ and $\eta(e_{U_i\ \tau_i}) \in [\![\forall.\tau_i]\!]$ (follows from Lemmas 2 and 3), therefore, $\eta(e_{F\ \tau}) \in [\![\forall.\tau]\!]$.

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

A note on the ordering among user-defined constraints $U_i\ \tau_i$:

We denote by $\leq$ the ordering relation among constraints. Consider $(R)\ \ F\ (a \to (b \to b)) \Longleftrightarrow U\ a, U\ (b \to b)$ in some set P of CHRs where $U\ a \leq U\ (b \to b)$. Assume $F\ (\tau_1 \to \tau_2) \rightarrowtail_P^* True$ for some types τ_1 and τ_2 such that $U\ \tau_2 \leq U\ \tau_1$. Clearly, $\leq$ is not stable under substitution. We find that $F\ (\tau_1 \to \tau_2) \rightarrowtail_R U\ \tau_2, U\ \tau_1 \rightarrowtail_P^* True$. However, note that $f_{F\ (a \to (b \to b)), U\ a, U\ (b \to b)} = \lambda(e_{U\ a}, e_{U\ (b \to b)}).\dots$. Therefore, we assume that each simplification rule step keeps the constraints in the same order as in the corresponding function definition. We have that $F\ (\tau_1 \to \tau_2) \rightarrowtail_R U\ \tau_1, U\ \tau_2 \rightarrowtail_P^* True$.

□

Let $C = U_1\ \tau_1 \wedge \dots U_n\ \tau_n$ be a constraint. We define *typeof*$(C) = (\tau_1, \dots, \tau_n)$. 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, Γ and σ. Let C_1 and C_2 be two sets of user-defined constraints such that $C_1, C_2 \rightarrowtail^*_P C_1$. Then, there exists a closed definition for ec :: $C_1 \overset{P}{\rightarrow} C_2$. In particular, the definition of ec has type $\forall \bar{\alpha}.typeof(C_1) \rightarrow typeof(C_2)$ where $\bar{\alpha} = fv(typeof(C_1) \rightarrow typeof(C_2))$.*

> **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 $C_1, C_2 \rightarrowtail^*_P C_1$ implies $C_1, C_2 \rightarrowtail^*_{P_{Simp}} C_1$.
>
> The proof proceeds by induction on the derivation $C_1, C_2 \rightarrowtail^*_P C_1$. 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 $C_1, C_2 \rightarrowtail_R C_1, \phi C_3$. For each $U_i\ \tau_i \in \phi C_3$ we have that $C_1, U_i\ \tau_i \rightarrowtail^* C_1$. By induction, there exist closed definitions $ec_i :: C_1 \overset{P}{\rightarrow} U_i\ \tau_i$. We define $ec(e_1, \ldots, e_m) = (ec_{U\ \tau', C_3}\ (ec_1(e_1, \ldots, e_m)), \ldots, (ec_k(e_1, \ldots, e_m)))$ where $ec_{U\ \tau, C_3}$ 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

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

We can't give a closed definition for ec_1, however, ec_1 must exist. For any τ such that $Eq\ ([\tau] \rightarrow [\tau] \rightarrow Bool) \rightarrowtail^*_P True$ we can construct evidence for $e_{Eq\ (\tau \rightarrow \tau arrow Bool)}$.

We state that the translation of normalized expression preseves typing.

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

Consider

```
overload eq :: $Int \rightarrow Int \rightarrow Bool$
         eq = primEqInt
overload eq :: $\forall a.Eq\ (a \rightarrow a \rightarrow Bool) \Rightarrow [a] \rightarrow [a] \rightarrow Bool$
         eq = let eqList :: $\forall a.Eq\ (a \rightarrow a \rightarrow Bool) \Rightarrow$
                             $[a] \rightarrow [a] \rightarrow Bool$
                  eqList [] [] = True
                  eqList (x:xs) [] = False
                  eqList [] (y:ys) = False
                  eqList (x:xs) (y:ys) = (eq x y) && (eqList xs ys)
                                         && (ord [1] [2])
              in  eqList overload ord :: $Int \rightarrow Int \rightarrow Bool$
         ord = primOrdInt
overload ord :: $\forall a.Ord\ (a \rightarrow a \rightarrow Bool) \Rightarrow [a] \rightarrow [a] \rightarrow Bool$
         ord = let ordList :: $\forall a.Ord\ (a \rightarrow a \rightarrow Bool) \Rightarrow$
                              $[a] \rightarrow [a] \rightarrow Bool$
                   ordList [] [] = True
                   ordList (x:xs) [] = False
                   ordList [] (y:ys) = False
```

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

Our translation yields

```
eqI :: Int → Int → Bool
eqI = primEqInt
eqL :: ∀a.(a → a → Bool) → [a] → [a] → Bool
eqL eq = let eqList :: ∀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 ordI [1] [2])
         in eqList ordI :: Int → Int → Bool
ordI = primOrdInt
ordL :: ∀a.(a → a → Bool) → [a] → [a] → Bool
ordL ord = let ordList :: ∀a.[a] → [a] → Bool
               ordList [] [] = True
               ordList (x:xs) [] = False
               ordList [] (y:ys) = False
               ordList (x:xs) (y:ys) = (ord x y) && (ordL ord xs ys)
                                       && (eqL eqI [1] [2])
           in ordList
```

Note that we have already simplified slightly the translated expressions. Instead of f (ec e) we simply write ec e.

For example, we find ordL ordI because of $Ord\ ([Int] \to [Int] \to Bool), Eq\ (a \to\!\!\to Bool) \rightarrowtail_P^* Eq\ (a \to\!\!\to Bool)$.

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

2.5 Unambiguity

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

Lemma 4 (Uniqueness) *Let P be a good set of CHRs, $\forall\bar{\alpha}.C \Rightarrow \tau$ 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 $\phi C \rightarrowtail_P^* True$. Then $C, \phi' \rightarrowtail_P^* \phi''$ 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 $\forall\bar{\alpha}.C \Rightarrow \tau$ is unambiguous. Let ρ be a variable renaming. We have that

$$[\![P]\!] \models (C \wedge \rho(C) \wedge \tau = \rho(\tau)) \to (\alpha = \rho(\alpha))$$

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

$$[\![P]\!] \models (C \wedge \phi(C) \wedge \tau = \phi(\tau)) \to (\alpha = \phi(\alpha))$$

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

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

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

2.6 Coherence

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

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

Recall the example from the Introduction.

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

and

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

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

```
exp2 :: ∀a.(a → a → Bool) → [a] → [a] → (Bool, Bool)
exp2 e xs ys = (eq (ecEqList e) (tail xs) ys, eq ecEqInt 1 3)
```

and

```
exp2' :: [Int] → [Int] → (Bool, Bool)
exp2' xs ys = (eq (ecEqList ecEqInt) (tail xs) ys, eq ecEqInt 1 3)
```

Clearly, expressions exp type and translation is more general than that of 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 CHRs, C a constraint, $\sigma_1 = \forall\bar{\alpha}_1.C_1 \Rightarrow \tau_1$ and $\sigma_2 = \forall\bar{\alpha}_2.C_2 \Rightarrow \tau_2$ two 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*

- *P is good and σ_1 is unambiguous,*
- *η_1 and η_2 satisfy P,*
- *$[\![P]\!] \wedge C \vdash^i \sigma_1 \preceq \sigma_2$*
- *for any ϕ such that $[\![P]\!] \models \phi(C \wedge C_2 \wedge \tau_1 = \tau_2)$, we have that*

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

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

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 uniquess of extension ϕ' is ensured by Lemma 4.

We consider (τ, E, η) as a short-hand for $(\forall\alpha.\alpha = \tau \Rightarrow \alpha, \lambda x.E, \eta)$ where α and x are fresh variables. 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 $C \rightarrowtail^*_P True$, then $[\![E_1]\!]\eta_1 = [\![E_2]\!]\eta_2$.

Coming back to the example from above we find that

$$\begin{array}{c} P, True \vdash^{ttv} \\ (\forall a.Eq\ (a \rightarrow a \rightarrow Bool) \Rightarrow [a] \rightarrow [a] \rightarrow (Bool, Bool), exp2, \eta) \\ \preceq \\ ([Int] \rightarrow [Int] \rightarrow [Int] \rightarrow (Bool, Bool), exp2', \eta) \end{array}$$

where P consists of rules (Eq1-4) from the Introduction and η is the variable enviroment as induced by Lemma 1.

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

Lemma 6 (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, $\bar{\alpha}_2$ a sequence of type variables, e_{C_2} a evidence variable and $ec :: C_2 \overset{P}{\rightarrow} [\bar{\tau}_2/\bar{\alpha}_2]C_2'$ a evidence constructor such that $P, C_2 \vdash^{ttv} (\sigma_1, E_1, \eta_1) \preceq (\forall\bar{\alpha}_2.C_2' \Rightarrow \tau_2, E_2, \eta_2)$. Then,*

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

Lemma 7 (Generalization) *Let P be a set of CHRs, C_2 and C_2' two constraints, $\bar{\alpha}_2$ 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 $P, C_2 \wedge C_2' \vdash^{ttv} (\sigma_1, E_1, \eta_1) \preceq (\sigma_2, E_2, \eta_2)$ and $\bar{\alpha}_2 \notin fv(C_2, \sigma_1)$. Then,*

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

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

Let $P, C, \Gamma \vdash e : \sigma \rightsquigarrow E$ be a jugdment. The *derivation tree* of $P, C, \Gamma \vdash e : \sigma \rightsquigarrow E$ 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 $P, C, \Gamma \vdash e : \sigma \rightsquigarrow E$. We refer to the *level* of $P, C, \Gamma \vdash e : \sigma \rightsquigarrow E$ as the sum of all of the paths in the derivation true.

Definition 2 *Let P be a set of CHRs, 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*

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

- *P is good and σ_1 is unambiguous,*
- *$[\![P]\!] \models C_2 \rightarrow (\bar{\exists}_{fv(\Gamma_2)} C_1)$,*
- *$[\![P]\!] \wedge C_2 \vdash^i \sigma_1 \preceq \sigma_2$, $[\![P]\!] \wedge C_2 \vdash^i \Gamma_1 \preceq \Gamma_2$, and*

- *let D_1 be the derivation tree of $P, C_1, \Gamma_1 \vdash e : \sigma_1 \leadsto E_1$, and let D_2 be the derivation tree of $P, C_2, \Gamma_2 \vdash e : \sigma_2 \leadsto E_2$. Then relation $\preceq$ holds for all sub-derivations $P, C_1', \Gamma_1' \vdash e' : \sigma_1' \leadsto E_1'$ in D_1 and $P, C_2', \Gamma_2' \vdash e' : \sigma_2' \leadsto E_2'$ in D_2 where we assume that $P, C_1', \Gamma_1' \vdash e' : \sigma_1' \leadsto E_1'$ has a lower level in D_1 than $P, C_2', \Gamma_2' \vdash e' : \sigma_2' \leadsto E_2'$ in D_2 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, η_1 and η_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 $P, C_2 \vdash^{ttv} (\Gamma_1, \eta_1) \preceq (\Gamma_2, \eta_2)$. 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 $P, C_2, \Gamma_2 \vdash e : \sigma_2 \leadsto E_2$. We only show some of the interesting cases.

Case (Var) We find the following situation:

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

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 $P, C_2 \vdash^{ttv} (\sigma_1, x, \eta_1) \preceq (\sigma_2, x, \eta_2)$ and we are done.

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

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

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:

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

Application of the induction hypothesis and Lemma 7 yields the desired result.

Case (Let) We find the following situation:

$$\frac{\begin{array}{c} P, C_2, (\Gamma_2)_x \vdash e : \sigma_2 \leadsto E_2 \\ P, C_2, (\Gamma_2)_x.x : \sigma_2 \vdash e' : \tau_2' \leadsto E_2' \end{array}}{P, C_2, (\Gamma_2)_x \vdash \mathsf{let}\ x = e\, \mathsf{in}\ e' : \tau_2' \leadsto \mathsf{let}\ x = E_2\ \mathsf{in}\ E_2'}$$

where

$$\begin{array}{c} (P, C_1, (\Gamma_1)_x \vdash e : \sigma_1 \leadsto E_1) \preceq (P, C_2, (\Gamma_2)_x \vdash e : \sigma_2 \leadsto E_2) \\ (P, C_1, (\Gamma_1)_x.x : \sigma_1 \vdash e' : \tau_1' \leadsto E_1') \preceq (P, C_2, (\Gamma_2)_x.x : \sigma_2 \vdash e' : \tau_2' \leadsto E_2') \end{array}$$

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. □

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 η be a variable environment and Γ a closed type environment ($fv(\Gamma) = \emptyset$). We define $\eta \models \Gamma$ iff (1) for each $f : \forall a.F\ a \Rightarrow a \in \Gamma$ we have that $\eta(f) = \lambda x.x$, 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 η be a variable environment such that the principal derivation is unambiguous, $C_1 \rightarrowtail_P^* True$, $C_2 \rightarrowtail_P^* True$, $\eta \models \Gamma$, η satisfies P, $fv(\Gamma, \sigma) = \emptyset$ and P is a terminating, 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 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 τ'.

Application of Lemma 8 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. □

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

```
overload f :: Unit → Int
         f x = f x
         in let a = f ()
            in ()
```

The let-bound expression a has two possible typings, $\forall b.F\ (Unit \to b) \Rightarrow b$ and Int. Note that the translation under $a :: Int$ 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 a 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

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

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

[3] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. 16th ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–76, January 1989.