Type Inference with constrained Types

Martin Sulzmann, Martin Odersky, Martin Wehr
University of Karlsruhe
Institute for Program Structures and Data Organization
Am Fasanengarten 5, 76128 Karlsruhe, Germany
{sulzmann,odersky,wehr}@ira.uka.de

1 Introduction

There are many type systems that extend the Hindley/Milner[Mil78] system with constraints. Examples are found in record systems [Oho95, Rem89], overloading [Jon92, HHJW96, NP93, KC92, OWW95], and systems that support subtyping [CCH+89, BSvG95, AW93, EST95]. Extensions of Hindley/Milner with constraints are also increasingly popular in program analysis [DHM95, TJ92].

Even though these type systems use different constraint domains, they are largely alike in their type-theoretic aspects. In this paper we present a general framework HM(X) for Hindley/Milner style type systems with constraints, analogous to the CLP(X) framework in constraint logic programming [JM94]. Particular type systems can be obtained by instantiating the parameter X to a specific constraint system. The Hindley/Milner system itself is obtained by instantiating X to the trivial constraint system over a one point domain.

By and large, the treatment of constraints in type systems has been syntactic : constraints were regarded as sets of formulas, often of a specific form. On the other hand, constraint programming now generally uses a semantic definition of constraint systems, taking a constraint system as a cylindric algebra with some additional properties [HMT71, Sar93]. Cylindric algebras define a projection operator $\exists\bar{\alpha}$ that binds some subset of variables $\bar{\alpha}$ in the constraint. In the usual case where constraints are boolean algebras, projection corresponds to existential quantification.

Following the lead of constraint programming, we treat a constraint system as a cylindric algebra with a projection operator. Projection is very useful for our purposes for two reasons: First, projection allows us to formulate a logically pleasing and pragmatically useful rule ($\forall$ Intro) for quantifier introduction. Second, projection is an important source of opportunities for simplifying constraints [Jon95]. In our framework, *simplifying* means changing the syntactic representation of a constraint without changing its denotation. For example, the subtyping constraint

$$\exists\beta.\alpha \leq \beta, \beta \leq \gamma$$

can safely be simplified to

$$\alpha \leq \gamma$$

since the denotation is the same for both constraints. Without the projection operator, the two constraints would be different, since one restricts the variable β while the other does not.

Two of the main strengths of the Hindley/Milner system are the existence of a principal types theorem and a type inference algorithm. We present sufficient conditions on the constraint domain X so that the principal types property carries over to HM(X). The conditions are fairly simple and natural. For those constraint systems meeting the conditions, we present a generic type inference algorithm that will always yield the principal type of a term. The proofs for the properties can be found in the forthcoming technical report [Sul96].

The rest of this paper is structured as follows: In Section 2 we discuss some previous approaches to constrained type systems, and how they differ in their quantifier introduction rule. Section 3 presents a characterization of constraint systems. A general framework HM(X) for Hindley/Milner style type systems with constraints is discussed in Section 4. Some instantiations for parameter X in the frame HM(X) are examined in Sections 6, 5 and 7. In Section 5 we consider extensible records, in Section 6 type classes and overloading and in section 7 subtypes. Section 8 concludes.

2 Related Work

All constrained type systems we study extend the type judgments , $\vdash e : \sigma$ of the Hindley/Milner system with a constraint hypothesis on the left hand side of the turnstyle, written $C, , \vdash e : \sigma$. Furthermore, they extend the type schemes $\forall\bar{\alpha}.\tau$ of the Hindley/Milner system with a constraint component; we write

$$\forall\bar{\alpha}.C \Rightarrow \tau$$

to express that the constraint C restricts the types that can legally be substituted for the bound variables $\bar{\alpha}$.

All type systems have essentially the same rule for eliminating quantifiers, which we write as follows:

$$(\forall \text{ Elim}); \frac{C, , \ \vdash e : \forall\bar{\alpha}.D \Rightarrow \tau' \qquad C \vdash [\bar{\tau}/\bar{\alpha}]D}{C, , \ \vdash [\bar{\tau}/\bar{\alpha}]\tau'}$$

The rule is a refinement of the corresponding rule in the Hindley/Milner system. It says that only those types $\overline{\tau}$ that satisfy the constraint D can legally be substituted for the bound variables $\overline{\alpha}$ in a type scheme $\forall\overline{\alpha}.D \Rightarrow \tau$.

No satisfiability check [Jon92]:
$$\frac{C \cup D, , \ \vdash e : \tau \qquad \bar{\alpha} \notin tv(C) \cup tv(,)}{C, , \ \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau} \quad (\forall \text{ Intro-1})$$

Weak satisfiability check [AW93]:
$$\frac{C \cup D, , \ \vdash e : \tau \qquad \exists D \qquad \bar{\alpha} \notin tv(C) \cup tv(,)}{C, , \ \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau} \quad (\forall \text{ Intro-2})$$

Strong satisfiability check [Smi94]:
$$\frac{\begin{array}{c} C \cup D, , \ \vdash e : \tau \qquad C \vdash [\bar{\tau}/\bar{\alpha}]D \\ \bar{\alpha} \notin tv(C) \cup tv(,) \end{array}}{C, , \ \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau} \quad (\forall \text{ Intro-3})$$

Duplication [EST95]:
$$\frac{C \cup D, , \ \vdash e : \tau \qquad \bar{\alpha} \notin tv(C) \cup tv(,)}{C \cup D, , \ \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau} \quad (\forall \text{ Intro-4})$$

Figure 1: Versions of the quantifier introduction rule

While there is agreement about the proper technique for eliminating quantifiers in type schemes, there is remarkable disagreement about the proper way to introduce them. Figure 1 shows four different rules that have all been proposed in the literature. We have edited these rules somewhat to present them in a uniform style, and have attempted to compensate for the considerable variations in detail between published type systems. We now discuss each of the four schemes in turn.

In his work in qualified types [Jon92], Jones uses a general framework for type qualification with a rule equivalent to rule (∀ Intro-1). Any constraint can be shifted from the assumption on the left to the type scheme on the right of the turnstyle; it is not checked if the constraint so traded is satisfiable or not. This might lead to programs that are well-typed as a whole, even though some parts have unsatisfiable constraints.

To give an example, assume that our constraints are subtyping constraints ($\leq$) in a type system with records. Let us assume that there is a parameterized type $\mathsf{List}\,\alpha$ with a less field such that for all types τ,

$$\mathsf{List}\,\tau \leq \{\mathsf{less} : \mathsf{List}\,\tau \to \mathsf{Bool}\}.$$

Let us further assume that there is a value Nil of type $\forall\alpha.\mathsf{true} \Rightarrow \mathsf{List}\,\alpha$ that represents the empty list. Consider the following (nonsensical) program.

Example 1

```
let
    f: ∀α.(List α ≤ {less : Bool → Bool}) ⇒ List α → List α
    f x = if x.less(true) then x else Nil
in 1
end
```

We use a Haskell-style notation, adding type annotations for illustration purposes. Using rule (∀ Intro-1), the program in 1 is well-typed, even though we would not expect the constraint in function f's type scheme to have a solution, since the function type $\mathsf{List}\,\alpha$ would not have a less field of type $\mathsf{Bool} \to \mathsf{Bool}$.

In the ideal semantics of types [MPS86], which represents universal quantification by intersection, f's type would be an empty intersection, which is equal the whole type universe including the error element *wrong*. However, the whole program in 1 is still sound because every application of f must provide a valid instantiation of the constraint. Since the constraint is unsatisfiable, no application is possible. In essence, Jones treats constraints as proof obligations that have to be fulfilled by presenting "evidence" at the *instantiation* site. This scheme is clearly inspired by Haskell's implementation of overloading by dictionary passing. It runs into problems if one ever wants to compute a value of a constrained type without any instantiation sites, as in the following slight variation of Example 1.

Example 2

```
let
    y: ∀α.(List α ≤ {less : Bool → Bool}) ⇒ Bool
    y = Nil.less(true)
in 1
end
```

Jones excludes this code on the grounds that y's type is ambiguous, but it is unclear how to generalize this restriction to arbitrary constraint systems.

In the type system of Aiken/Wimmers [AW93], moving a constraint from the left hand side of the turnstyle to the right-hand side is allowed only if the constraint is satisfiable (i.e. has a solution). Hence, none of the previous examples would be typable with rule (∀ Intro-2), which they use. However, this example is typable.

Example 3

```
let
    f: ∀β.β → Int
    f x =
        let y: ∀α.(List α ≤ {less : β → Bool}) ⇒ Bool
            y = Nil.less(x)
        in 1
in f true
```

The constraint $\mathsf{List}\,\alpha \leq \{\mathsf{less} : \beta \to \mathsf{Bool}\}$ has a solution, namely $\beta = \mathsf{List}\,\alpha$. Therefore, using rule (∀ Intro-2) we can generalize y's type to

$$\forall\alpha.(\mathsf{List}\,\alpha \leq \{\mathsf{less} : \beta \to \mathsf{Bool}\}) \Rightarrow \mathsf{Bool}.$$

On the other hand, if we substitute the actual parameter true in f's definition, we get again Example 2 which is not typable under the system with ($\forall$ Intro-2). Hence, the system with ($\forall$ Intro-2) does not enjoy the property of subject reduction, which says that if a term is typable then its reduction instances are typable as well.

Where Aiken and Wimmers require only a weak form of satisfiability for traded constraints, G. Smith requires a strong one [Smi94]. In rule ($\forall$ Intro-3), the traded constraint D must be solvable by instantiation of only the quantified variables $\bar{\alpha}$. Hence, all three previous examples would be untypable under his system. However, ($\forall$ Intro-3) rule might seem overly restrictive, depending on the constraint system used. In a type system in which subtyping is by declaration, assume we have a record type

type Less a = {less: a→Bool}

with precisely the following four instances:

Int $\leq$ {less: Int $\rightarrow$ Bool}
Int $\leq$ {less: Float $\rightarrow$ Bool}
Float $\leq$ {less: Int $\rightarrow$ Bool}
Float $\leq$ {less: Float $\rightarrow$ Bool}

Now consider the following program:

Example 4

```
let
    f x =
        let g y = x.less(y)
        in g 1 && g 1.0
in f 1 && f 1.0
```

When typing the definition of g, Smith's system requires a solution of the constraint $\tau_x \leq \mathsf{Less}\,\tau_y$, where τ_x is x's type and τ_y is y's type. Solutions to this constraint systems exist for both $\tau_x = \mathsf{Int}$ and $\tau_x = \mathsf{Float}$. The problem is that we have to arbitrarily pick one of these instantiations for τ_x since there is no best type for x that improves on both solutions.

The system of the Hopkins Objects Group [EST95] differs from the previous three systems in that in rule ($\forall$ Intro-4) the constraint D is copied instead of moved; there are no restrictions on when the copying can take place. Under this scheme, the first three examples would be rejected and the fourth one would be accepted, which corresponds fairly well to our intuition. At the same time, rule ($\forall$ Intro-4) seems strange in that its conclusion contains two copies of the constraint D, one in which the type variables α are bound and one in which they are free. Actually, the Hopkins Objects Group use a slightly different system in which generalization is coupled with the let rule and one of the two constraints undergoes a variable renaming. Still, it is difficult to see how one could put this into a good logical formulation.

Another possible objection to rule ($\forall$ Intro-4) is pragmatic: Since constraints never disappear from the hypothesis, we will end with a large hypothesis when typing large programs. The Hopkins Objects Group do address this problem by investigating ways to simplify constraints.

The framework that we present is most closely related to the one of the Hopkins Objects Group, but instead of simply duplicating the constraint D we split it up into two versions, one existentially quantified, the other universally quantified. With this change, their system can be seen as a special instance of our framework that deals just with subtyping constraints. Furthermore, our framework gives a semantic justification for the consistency requirements and their simplification techniques.

3 Constraint Systems

We present a characterizations of constraint systems along the lines of Henkin [HMT71] and Saraswat [Sar93].

Definition 1 (Simple Constraint System) *A* simple constraint system *is a structure* $(\Omega, \vdash^e)$ *where* Ω *is a non–empty set of tokens or (primitive) constraints* [1] *and* $\vdash^e \subseteq p\Omega \times \Omega$ *is a decidable entailment relation where* $p\Omega$ *is the set of finite subsets of* Ω. *We call* $C \in p\Omega$ *a constraint set or simply a constraint.*

A constraint system $(\Omega, \vdash^e)$ *must satisfy for all constraints* $C, D \in p\Omega$*:*

C1 $C \vdash^e P$ *whenever* $P \in C$ *and*
C2 $C \vdash^e Q$ *whenever* $C \vdash^e P$ *for all* $P \in D$ *and* $D \vdash^e Q$

We extend $\vdash^e$ *to be a relation on* $p\Omega \times p\Omega$ *by:* $C \vdash^e D$ *:iff* $C \vdash^e P$ *for every* $P \in D$. *Define* $C =^e D$ *:iff* $C \vdash^e D$ *and* $D \vdash^e C$ *and* $\mathsf{true} := \{P | \emptyset \vdash^e P\}$.

Definition 2 (Cylindric Constraint System) *A* cylindric constraint system $\mathcal{C}$ *is a structure* $(\Omega, \vdash^e, Var, \{\exists\alpha | \alpha \in Var\})$ *such that:*

- $(\Omega, \vdash^e)$ *is a simple constraint system,*
- *Var is an infinite set of variables,*
- *For each variable* $\alpha \in Var, \exists\alpha : p\Omega \rightarrow p\Omega$ *is an operation satisfying:*

E1 $C \vdash^e \exists\alpha.C$
E2 $C \vdash^e D \Rightarrow \exists\alpha.C \vdash^e \exists\alpha.D$
E3 $\exists\alpha.(C \cup \exists\alpha.D) =^e \exists\alpha.C \cup \exists\alpha.D$
E4 $\exists\alpha.\exists\beta.C =^e \exists\beta.\exists\alpha.C$

The next definition defines the free type variables $tv(C)$ of a constraint C.

Definition 3 (Free Variables) *Let* C *be a constraint.*

$$tv(C) := \{\alpha | \exists\alpha.C \neq^e C\}$$

We now introduce a much more expressive constraint system. We want to deal with types and substitutions.

Definition 4 (Types) *A* type *is a member of Term(Var,*$\rightarrow$*) where Term(Var,*$\rightarrow$*) is the term algebra build up from a set Var of variables and the function constructor* $\rightarrow$ *of arity 2.*

Definition 5 (Substitutions) *A* substitution ϕ *is an idempotent mapping from a set of variables Var to the term algebra Term(Var,*$\rightarrow$*). Let* **1** *be the identity substitution.*

[1] We also refer to such constraints as predicates.

$$\text{(VAR)} \qquad C, , \ \vdash x : \sigma \quad (x : \sigma \in ,)$$

$$\text{(ABS)} \qquad \frac{C, , {}_x.x : \tau \vdash e : \tau'}{C, , \ \vdash \lambda x.e : \tau \to \tau'}$$

$$\text{(APP)} \qquad \frac{C, , \ \vdash e_1 : \tau_1 \to \tau_2 \quad C, , \ \vdash e_2 : \tau_1}{C, , \ \vdash e_1 e_2 : \tau_2}$$

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

$$(\forall \text{ Intro}) \qquad \frac{C \cup D, , \ \vdash e : \tau \quad \bar{\alpha} \notin tv(C) \cup tv(,)}{C \cup \exists \bar{\alpha}.D, , \ \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau}$$

$$(\forall \text{ Elim}) \qquad \frac{C, , \ \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau \quad C \vdash^e [\bar{\tau}/\bar{\alpha}]D}{C, , \ \vdash e : [\bar{\tau}/\bar{\alpha}]\tau}$$

Figure 2: Logical type system

Definition 6 (Type Constraint System) *A* type constraint system $\mathcal{D}$ *is a cylindric constraint system having some additional properties. For types* τ, τ' *the token* $< \tau = \tau' >$ *is contained in the set of primitive constraints.*

F1 $\emptyset \vdash^e \{< \tau = \tau >\}$
F2 $\{< \tau = \tau' >\} \vdash^e \{< \tau' = \tau >\}$
F3 $\{< \alpha = \tau >, < \beta = \tau >\} \vdash^e \{< \alpha = \beta >\}$
F4 $\{< \alpha = \tau >\} \cup \exists \alpha.(C \cup \{< \alpha = \tau >\}) \vdash^e C$
F5 $\{< \tau_1 = \tau_1' >, < \tau_2 = \tau_2' >\} \vdash^e \{< \tau_1 \to \tau_2 = \tau_1' \to \tau_2' >\}$
F6 *for all predicates* P *holds:* $\{[\tau/\alpha]P\} =^e \exists \alpha.\{P, < \alpha = \tau >\}$ *where* $\alpha \notin tv(\tau)$

Remark 1 *Conditions* **F1** $-$**F3** *are quite obvious. Condition* **F4** *states that we can substitute the same with the same. It is in fact the Leibniz principle. Condition* **F5** *states that* $< = >$ *is a congruence. Condition* **F6** *defines how a substitution acts on a primitive constraint.*

Some basic lemmas follow. The first states that $< = >$ is transitive.

Lemma 1 *Let* C *be a constraint and* τ, τ', τ'' *be types.*

$$\textit{if} \quad C \vdash^e \{< \tau = \tau' >, < \tau' = \tau'' >\} \quad \textit{then} \quad C \vdash^e \{< \tau = \tau'' >\}$$

The next lemmas state some basic properties about how a substitution acts on a constraint.

Lemma 2 *Let* C *be a constraint and* $\phi = [\bar{\tau}/\bar{\alpha}]$ *a substitution.*

$$\phi C =^e \exists \alpha_1 \ldots \exists \alpha_n.(C \cup \{< \alpha_1 = \tau_1 > \wedge \ldots \wedge < \alpha_n = \tau_n >\})$$

Lemma 3 *Let* C *be a constraint and* ϕ_1, ϕ_2, ϕ_3 *be substitutions.*

$$(\phi_1 \circ (\phi_2 \circ \phi_3))C =^e ((\phi_1 \circ \phi_2) \circ \phi_3)C$$

Lemma 4 *Let* C, D *be constraints and* ϕ *be a substitution.*

$$\phi(C \cup D) =^e \phi C \cup \phi D$$

Lemma 5 *Let* C, D *be constraints and* ϕ *be a substitution.*

$$\textit{if} \quad C \vdash^e D \quad \textit{then} \quad \phi C \vdash^e \phi D$$

4 A General Framework HM(X) for Hindley/Milner Type Systems with Constraints

This sections describes an extension of the Hindley/Milner type system with constraints. The ingredients are very much the same as in the original presentation [DM82].

4.1 The Type System

Expressions	e	$::=$	$x \mid \lambda x.e \mid ee' \mid \mathsf{let}\ e = x\ \mathsf{in}\ e' \mid \ldots$
Types	τ	$::=$	$\alpha \mid \tau \to \tau' \mid \ldots$
Type schemes	σ	$::=$	$\tau \mid \forall \alpha.C \Rightarrow \sigma$

The only extension compared to Hindley/Milner types is in the definition of a quantified type scheme $\forall \alpha.C \Rightarrow \sigma$. Here, C is a constraint which restricts the types that can be substituted for the type variable α. Note, that in our logical type system we only deal with a cylindric constraint system $(\dot{\Omega}, \vdash^e, Var, \{\exists \alpha | \alpha \in Var\})$.

The type rules can be found in figure 2. They have the following structure. Intiutively, the statement $C, , \ \vdash e : \sigma$ says that from a set C of global constraints and type context $, $ one can derive term e with type σ. As

$$\text{(REF)} \qquad \emptyset \vdash^i \tau \preceq \tau$$

$$(\preceq \forall) \qquad \frac{C \cup D \vdash^i \sigma \preceq \sigma' \quad \alpha \notin tv(\sigma) \cup tv(C)}{C \vdash^i \sigma \preceq \forall \alpha.D \Rightarrow \sigma'}$$

$$(\forall \preceq) \qquad \frac{C \vdash^i [\bar{\tau}/\bar{\alpha}]\tau \preceq \tau' \quad C \vdash^e [\bar{\tau}/\bar{\alpha}]D}{C \vdash^i \forall \bar{\alpha}.D \Rightarrow \tau \preceq \tau'}$$

Figure 3: Instance rules

usual, the type context , contains statements of the form $x : \sigma$. Now, let us have a look at the type system. The core rules(VAR), (LET), (ABS) and (APP) are a straightforward generalization of the original Hindley/Milner rules. The most interesting rules are the ($\forall$ Intro) rule and the ($\forall$ Elim) rule. By rule ($\forall$ Intro) we quantify some type variables. The term $\forall \bar{\alpha}.D \Rightarrow \tau$ is an abbreviation for $\forall \alpha_1.\mathsf{true} \Rightarrow \ldots \forall \alpha_n.D \Rightarrow \tau$ and $\exists \bar{\alpha}.D$ for $\exists \alpha_1. \ldots \exists \alpha_n.D$. By rule ($\forall$ Elim) we build an instance of an type scheme. Of course, one can think of further extensions such as the treatment of records.

Example 5 (Hindley/Milner) *The Hindley/Milner system is an instance of our type system. We have the following minimal cylindric constraint system:*

$$(\{\mathsf{true}\}, \vdash^e, Var, \{\exists \alpha | \alpha \in Var\})$$

4.2 Type Inference

Now, we consider the type inference problem. Some technical details that are used in this section can be found in Appendix A. The type inference algorithm can be found in Figure 4. The algorithm takes as input a type context , and a term e and reports as output a substitution ϕ, a constraint C and a type τ.

The most interesting rules are the (APP) and (LET) rule. In order to get a conclusion the results of the premises are combined and then a *normalization* step is performed. Remember, we only want to deal with a cylindric constraint system $\mathcal{C}$. But, if the premises are combined we get a constraint in the type constraint system $\mathcal{D}$. Then we have to perform a normalization step which yields a constraint in $\mathcal{C}$. For example, normalization of $< \tau_1 = \tau_2 \to \alpha >$ in the (APP) rule computes a solution to the unification problem $(\tau_1, \tau_2 \to \alpha)$. We give an axiomatic description of such a normalization in the next section. Then, we introduce an instance relation to compare the results of the type inference algorithm. Finally, we state the results.

4.2.1 Normalized Constraints

In this section we study the relationship between constraint systems $\mathcal{C}$ and $\mathcal{D}$: We only want to deal with constraint system $\mathcal{C}$ but some constraints will be expressible only in $\mathcal{D}$. Thus we must perform a *normalization* step. Consider for example the constraint $\{< int \to bool = int \to \alpha >\}$. Normalization yields (C, ψ) where $C =^e 1$ and $\psi = [bool/\alpha]$. Below we give an axiomatic description of such a normalization.

Preliminaries: Assume U is a finite set of type variables that are of interest in the situation at hand. We need a handle to compare two substitutions.

Definition 7 *Let ϕ, ϕ' and ψ be substitutions.*

$$\psi \leq_U^{\phi'} \phi \quad \textit{:iff} \quad (\phi' \circ \psi)_{|U} = \phi_{|U}$$

We write $\psi \leq \phi$ if $\exists \phi' : \psi \leq^{\phi'} \phi$. Sometimes, we omit the set U.

Note that this makes the "more general" substitution the smaller element in the pre–order $\leq_U$. This choice, which reverses the usual convention in treatments of unification (e.g. [LMM87]), was made to stay in line with the semantic notion of type instances.

We make $\leq_U$ a partial order by identifying substitutions that are equal up to variable renaming, or equivalently, by defining $\psi =_U \phi$ iff $\psi \leq_U \phi$ and $\phi \leq_U \psi$. It follows from [LMM87] that $\leq_U$ is a complete lower semi–lattice where least upper bounds, if they exist, correspond to unifications and greatest lower bounds correspond to anti–unifications.

Definition 8 (Normalization) *Let D be a constraint from $\mathcal{D}$ and ϕ be a substitution. Then there is a* normalization *(C, ψ) where C is a constraint from $\mathcal{C}$ and ψ a substitution such that*

$$C \vdash^e \psi D$$
$$\phi \leq^{\phi'} \psi \quad \psi C = C \quad C \textit{ is minimal}$$

Minimal means, if there is another normalization (C', ψ') which satisfies the condition above then $\psi \leq^{\phi''} \psi'$ and $C' \vdash^e \phi'' C$.
The function normalize$(D, \phi) = (C, \psi)$ computes the normalization.

$$\text{(VAR)} \quad \mathbf{1}, C, , \ \vdash^W \ x : \tau \quad ((C, \tau) = \mathit{new\_inst}(\sigma) \quad x : \sigma \in ,)$$

$$\text{(ABS)} \quad \frac{\psi, C, , {}_x.x : \alpha \ \vdash^W \ e : \tau \quad \alpha \text{ new}}{\psi_{\setminus\{\alpha\}}, C, , \ \vdash^W \ \lambda x.e : \psi(\alpha) \to \tau}$$

$$\text{(APP)} \quad \frac{\begin{array}{c}\psi_1, C_1, , \ \vdash^W \ e_1 : \tau_1 \quad \psi_2, C_2, , \ \vdash^W \ e_2 : \tau_2 \\ \psi' = \psi_1 \sqcup \psi_2 \\ D = C_1 \cup C_1 \cup \{< \tau_1 = \tau_2 \to \alpha >\} \quad \alpha \text{ new} \\ (C, \psi) = \mathit{normalize}(D, \psi')\end{array}}{\psi_{|tv(\Gamma)}, C, , \ \vdash^W \ e_1 e_2 : \psi\alpha}$$

$$\text{(LET)} \quad \frac{\begin{array}{c}\psi_1, C_1, , {}_x \ \vdash^W \ e : \tau \quad (\sigma, C_2) = \mathit{gen}(C_1, \psi_1, , \tau) \\ \psi_2, C_3, , {}_x.x : \sigma \ \vdash^W \ e' : \tau' \\ \psi' = \psi_1 \sqcup \psi_2 \quad D = C_2 \cup C_3 \\ (C, \psi) = \mathit{normalize}(D, \psi')\end{array}}{\psi_{|tv(\Gamma_x)}, C, , {}_x \ \vdash^W \ \mathsf{let}\ x = e\ \mathsf{in}\ e' : \psi\tau'}$$

Figure 4: Type inference

The normalization algorithm for HM(X)with extension X have following outlook.

normalize(D, ϕ)
  $C := \emptyset$; *(* set of normalized constraints *)*
  $\psi := \phi$; *(* resulting substitution *)*
  while $D \neq \emptyset$ do
    take a $P \in D$; $D := D \setminus \{P\}$;
    case P of
      $< \tau = \tau' >$ $\Rightarrow$ $\psi' = \mathsf{mgu}(\tau, \tau')$;
        $D := \psi' D$; $\psi := \psi \sqcup \psi'$;

normalization extension : X

general token $P \in \mathcal{D} \Rightarrow$
  if *unsatisfyable* then fail
    else $D := D \cup \{$*tokens generated by structural reduction of token* $P\}$;
    mayby extend ψ;
normalized token $P \in \mathcal{C} \Rightarrow$
  if $\exists$*conflicting normalized token* $\in D$ then fail
    else $C := C \cup \{P\}$;
    mayby extend ψ;

  od
  return(C, ψ);

In case of the Hindely/Milner type system normalization means simply computation of the most general unifier. In Sections 5 and 6 we consider normalization extensions for extensible records, type classes and overloading.

4.2.2 The Instance Relation

In order to state the results we need a handle to compare two type schemes σ and σ' with respect to a constraint C. This relation is expressed by the term $C \vdash^i \sigma \preceq \sigma'$. The rules for $\vdash^i$ can be found in Figure 3. We introduce some basic lemmas.

The (CUT) and (TRANS) rules are derivable.

Lemma 6 *Let C and D be constraints and σ and σ' be type schemes.*

$$\begin{array}{c}\textit{if} \quad D \vdash^e C \quad C \vdash^i \sigma \preceq \sigma' \\ \textit{then} \\ D \vdash^i \sigma \preceq \sigma'\end{array}$$

Lemma 7 *Let C be a constraint and σ_1, σ_2 and σ_3 be type schemes.*

$$\begin{array}{c}\textit{if} \quad C \vdash^i \sigma_1 \preceq \sigma_2 \quad C \vdash^i \sigma_2 \preceq \sigma_3 \\ \textit{then} \\ C \vdash^i \sigma_1 \preceq \sigma_3\end{array}$$

We introduce the next definition in order to state the completeness result in a convenient manner.

Definition 9 *Let C be a constraint and , and , $'$ be type contexts.*

$$\begin{array}{c}C \vdash^i , \preceq , ' \\ \textit{:iff} \\ , = \{e_1 : \sigma_1, \ldots, e_n : \sigma_n\}, , ' = \{e_1 : \sigma_1', \ldots, e_n : \sigma_n'\} \textit{ and} \\ C \vdash^i \sigma_i' \preceq \sigma_i \quad \forall i : i \in \{1, \ldots, n\}\end{array}$$

4.2.3 Main Results

We now present the desired results.

Theorem 1 (Soundness)

$$\begin{array}{c}\textit{if} \quad \psi, C, , \ \vdash^W \ e : \tau \\ \textit{then} \\ C, \psi, \ \vdash \ e : \tau \\ \psi C = C \quad \psi\tau = \tau\end{array}$$

Theorem 2 (Completeness)

$$\begin{array}{c}\textit{if} \quad C', , ' \vdash \ e : \sigma' \quad C' \vdash^i \phi, \ \preceq , ' \\ \textit{then} \\ \psi, C, , \ \vdash^W \ e : \tau \\ \sigma_o = \mathit{gen}_1(C, \psi, , \tau) \\ \psi \leq^{\phi'}_{tv(\Gamma)} \phi \quad C' \vdash^i \phi' \sigma_o \preceq \sigma'\end{array}$$

Labels $\mathcal{L}$	l			
Expressions	e	$::=$	$\ldots \mid (e \mid l = e')$	record extension
			$\mid e.l$	selection
Record types	r	$::=$	$\alpha \mid <> \mid < r \mid l : \tau >$	
Types	τ	$::=$	$r \mid \tau \to \tau'$	
Token constructors for $\mathcal{D}$	Ω	$::=$	$r\ lacks\ l$	record r has no label l
			$\mid r\ has\ l : \tau$	record r has unique label $l : \tau$
			$\mid < \tau = \tau' >$	diagonal token
Normalized token for $\mathcal{C}$	Ω	$::=$	$\alpha\ lacks\ l$	
			$\mid \alpha\ has\ l : \tau$	

New logical rules

$$\text{(select)} \quad \frac{D,,\ \vdash\ e : r \qquad D \vdash^e r\ has\ l : \tau}{D,,\ \vdash\ e.l : \tau}$$

$$\text{(r-extend)} \quad \frac{D,,\ \vdash\ e : r \qquad D,,\ \vdash\ e' : \tau \qquad D \vdash^e r\ lacks\ l}{d,,\ \vdash\ (e \mid l = e') :< r \mid l : \tau >}$$

Figure 5: Ingredients for extensible records

Corollary 1 (Principal Types)

$$\begin{array}{c} \textit{if} \quad C',,' \vdash\ e : \sigma' \quad tv(,') = \emptyset \\ \textit{then} \\ \mathbf{1}, C,,' \vdash^W\ e : \tau \\ \sigma_o = gen_1(C,,\tau) \\ C' \vdash^i\ \sigma_o \preceq \sigma \end{array}$$

One might wonder why we do not need the notion of satisfiability for the completeness result. In fact, we do not need the notion of satisfiability for the theoretical approach. Of course, if we consider a specific type system we must check whether type inference reports a satisfiable constraint. First, we give a definition of satisfiability.

Definition 10 (Satisfiability) *Let C be a constraint.*

$$\begin{array}{c} C \textit{ is} \text{ satisfiable} \\ \textit{:iff} \\ \emptyset \vdash^e\ \exists \bar{\alpha}.C \quad \textit{where} \quad \bar{\alpha} = tv(C) \end{array}$$

Because of the following two lemmas it is straightforward to take into account the satisfiability of constraints. If we derive an unsatisfiable constraint in our logical type system, then the rest of the derivation always yields an unsatisfiable constraint, and the normalization of an unsatisfiable constraint always yields an unsatisfiable constraint.

Lemma 8 *If the constraints of the premises of a rule of the logical type system are satisfiable then the constraint of the conclusion is satisfiable.*

Lemma 9 *Let D be a constraint and ϕ be a substitution.*

$$\begin{array}{c} \textit{if } \phi D \textit{ is unsatisfiable} \\ \textit{then} \\ normalize(D, \phi) = (C, \psi) \textit{ where } C \textit{ is unsatisfiable.} \end{array}$$

Therefore it does not matter whether we test the satisfiability of constraints during or after type inference.

5 Extensible Records as constrained Type System

We walk through a program which is similar for every language extension X in HM(X). First the language is extended by record extension and record selection (see figure 5). Next the constraint system is extended to express the lack of a label or the existence of a label. We must extend the logical type system by rules to give records a type. We must define the operation of the constraint entailment relation on the new constraints which is done in figure 6. For inference we must define the notion of a satisfied constraint problem, leading to the notion of normalized constraint sets. Finally we extend the inference to work with records and extend the normalization algorithm.

The change relative to Hindley/Milner is the introduction of a new form of types the record types. The new logical rules explain that all record formation constructs can only apply on record typed expressions.

5.1 Type Inference for extensible Records

Type inference should result in a normalized constraint set C and a type τ, given a typotheses , and a program e with extensible records. The core of the inference system is unchanged (figure 4) the extensions deal with records (see figure 7). The constraint set C is normalized if its elements are normalized tokens and it is consistent. The set C is inconsistent iff there exists a pair of tokens $\alpha\ lacks\ l, \alpha\ has\ l : \tau \in C$ for a label l and a type variable α, otherwise its is called consistent. The last condition for normalized constraint set sets C is the type uniqueness for existing labels.

$$\begin{array}{c} \alpha\ has\ l : \tau \in C \implies \\ \forall \tau'. \neg \exists \alpha\ has\ l : \tau' \in C \setminus \{\alpha\ has\ l : \tau\} \end{array}$$

The extension of the normalization algorithm (figure 8 ensures these properties, or indicate a failure if the conditions are violated.

$$\text{(HAS)}\quad \frac{D \vdash^e r\ lacks\ l}{D \vdash^e\ <r\,|\,l:\tau>\ has\ l:\tau} \qquad D \vdash^e\ <>\ lacks\ l \qquad \text{(LACKS)}$$

$$\text{(HAS-ext)}\quad \frac{D \vdash^e r\ has\ l:\tau}{D \vdash^e\ <r\,|\,l':\tau>\ has\ l:\tau} \qquad \frac{D \vdash^e r\ lacks\ l}{D \vdash^e\ <r\,|\,l':\tau>\ lacks\ l}\quad \text{(LACKS-ext)}$$

Figure 6: Constraint entailment relation for extensible records

Inference extension : extensible records

$$(\text{select})^{We}\quad \frac{\phi, C, , \vdash^W e : r \qquad (C', \psi) = normalize(C \wedge \{r\ has\ l:\alpha\}, \phi) \qquad \alpha\ new}{\psi, C', , \ \vdash\ e.l : \alpha}$$

$$(\text{r-extend})^{We}\quad \frac{\phi, C, , \vdash^W e : r \qquad \phi' C', , \vdash^W e : \tau \qquad (C'', \phi'') = normalize(C \wedge C' \wedge \{r\ lacks\ l\}, \phi \sqcup \phi')}{\phi'', C'', , \vdash^W (e\,|\,l = e') :< r\,|\,l:\tau>}$$

Figure 7: Inference system for extensible records

Normalization extension : extensible records

$$\begin{array}{lcl}
r\ lacks\ l & \Rightarrow & \mathsf{case}\ r\ \mathsf{of} \\
 & & \quad <r'\,|\,l:\tau> \ \Rightarrow\ \mathsf{fail} \\
 & & \quad <r'\,|\,l':\tau'> \ \Rightarrow\ D := D \cup \{r'\ lacks\ l\} \\
r\ has\ l:\tau & \Rightarrow & \mathsf{case}\ r\ \mathsf{of} \\
 & & \quad <> \ \Rightarrow\ \mathsf{fail} \\
 & & \quad <r'\,|\,l:\tau'> \ \Rightarrow\ D := D \cup \{<\tau' = \tau>, r'\ lacks\ l\} \\
 & & \quad <r'\,|\,l':\tau'> \ \Rightarrow\ D := D \cup \{r'\ has\ l:\tau\} \\
\alpha\ lacks\ l & \Rightarrow & \mathsf{if}\ \alpha\ has\ l:\tau \in C\ \mathsf{then\ fail} \\
 & & \mathsf{else}\ C := C \cup \{\alpha\ lacks\ l\} \\
\alpha\ has\ l:\tau & \Rightarrow & \mathsf{if}\ \alpha\ lacks\ l \in C\ \mathsf{then\ fail} \\
 & & \mathsf{else\ for}\ all\ \alpha\ has\ l:\tau' \in D\ \mathsf{do} \\
 & & \quad D := D \cup \{<\tau = \tau'>\} \setminus \{\alpha\ has\ l:\tau'\}; \\
 & & \quad C := C \cup \{\alpha\ has\ l:\tau\}; \\
 & & \mathsf{od}
\end{array}$$

Figure 8: Normalization for extensible records

Type constructors	T		for example $\{\rightarrow, \mathsf{List}, \ldots\}$
Class names	C		for example $\{\mathsf{Eq}, \mathsf{Ord}, \ldots\}$
Sorts	S	$::=$	$\{C_1, \ldots, C_n\}$
Signature	Σ	$::=$	$\emptyset \mid \Sigma, T(S_1, \ldots, S_n)C$
Token constructor for $\mathcal{D}$	Ω	$::=$	$\tau : S$ a type belonging to a sort
Normalized token for $\mathcal{C}$	Ω	$::=$	$\alpha : S$ type variable restricted to belong to a sort
Types	τ	$::=$	$\alpha \mid \tau \rightarrow \tau' \mid T\overline{\tau}$
Type schemes	σ	$::=$	$\tau \mid \forall\alpha.\{\alpha : S\} \Rightarrow \sigma$
Declarations	d	$::=$	$\mathsf{class}\ C\ \mathsf{where}\ x : \forall\alpha.\{\alpha : C\} \Rightarrow \sigma$ class declaration
			$\mathsf{inst}\ T : (S_1, \ldots, S_n)C\ \mathsf{where}\ x = e$ instance declaration

Figure 9: Ingredients for type classes

$$\text{(taut)} \quad D \vdash^e \tau : S \qquad (\tau : S \in D) \qquad\qquad \frac{D \vdash^e \tau_1 : S_1 \ \ldots \ D \vdash^e \tau_n : S_n \qquad T(S_1 \ldots S_n)C}{D \vdash^e T\tau_1 \ldots \tau_n : C} \quad \text{(tconst)}$$

$$\text{(sort-I)} \quad \frac{D \vdash^e \tau : C_1 \ \ldots \ D \vdash^e \tau : C_n}{D \vdash^e \tau : \{C_1, \ldots, C_n\}} \qquad\qquad \frac{D \vdash^e \tau : \{C_1, \ldots, C_n\} \qquad i = 1 \ldots n}{D \vdash^e \tau : C_i} \quad \text{(sort-E)}$$

Figure 10: Constraint entailment relation for type classes

6 Type Classes

With the use of the proposed constraint type system it is simple to include type classes in a Hindley/Milner type system. We compare two approaches to realize type classes.

First we restate the inference mechanism proposed by Nipkow and Prehofer in [NP93]. A look at this algorithm will lead to the observation that the style of this type system is that of a constraint type system.

We then consider the approach of Jones [Jon92]. This example will shed a light on the two constraint systems we are using, the general system $\mathcal{D}$ and the normalized $\mathcal{C}$. Our analysis will classify this approach as one without the notion of normalizing.

The third example in this section is System O [OWW95]. It is a overloading type system which resolves some of the problem of type classes.

6.1 Nipkow/Prehofer Approach

Figure 9 lists the ingredients used to extend Hindley/Milner with type classes. A program in the extended language consists of a sequence of declarations d followed by a expression e of the core language. The result of the declaration will be a initialized signature Σ. A signature entry $T(S_1, \ldots, S_n)C$ tells us that there is an instance of class C for type $T\alpha_1 \ldots \alpha_n$. Moreover the type variable α_i can only be bound to a type τ iff there exists an instance $C'\tau$ for every $C' \in S_i$. The above lengthy condition is expressed formally by the entailment relation (see Figure 10) for the constraint system. In the figure we use letter D to denote a set of constraints $\tau : S$.

The normalization algorithm needs to know about the constraints on type constructor arguments. This access to the signature is done by

$$Dom(C, T) = \{S_1 \ldots S_n \mid T(S_1 \ldots S_n) \in \Sigma\}$$

The result of the access is either a failure, if there is no instance declaration for type constructor T in class C, or the result constrains the arguments of the type constructor. In the following you see the extension of the normalization algorithm as given in section 4.2.1

Normalization extension : type classes

$T\overline{\tau} : S \ \Rightarrow$ for *all* $C \in S$ do
  if *no* $T(S_1 \ldots S_n)C \in \Sigma$ then fail
  else $D := D \cup \{\overline{\tau}_1 : S_1, \ldots, \overline{\tau}_n : S_n\}$

$\alpha : S \ \Rightarrow$ if $\alpha : S' \in C$
  then $C := C \setminus \{\alpha : S'\} \cup \{\alpha : S \cup S'\}$
  else $C := C \cup \{\alpha : S\}$

The approach is slightly extended in taking account of subclasses. The authors assume an ordering $\preceq$ on the class names. With the restrictions of Haskell the function $Dom(T, C)$ still works with a maximality property leading again to the principal type property.

6.2 Jones Approach

The qualified types approach of Jones offers a improved expressibility compared to Nipkow/Prehofer, but lacks of a normalization. The main differences are as follow:

- (J) Constraint tokens $C\overline{\tau}$ can hold a sequence $\overline{\tau}$ of types as argument $\leftrightarrow$ instead of only one argument (N/P).
- (J) The instance relation $C_1\overline{\tau}^1 \ldots C_n\overline{\tau}^n \Rightarrow C\overline{\tau}$ is build with complex preconditions $\leftrightarrow$ instead of $S_1 \ldots S_n CT$ simple variable constraints $\alpha_i : S_i$ on the arguments of type constructor T (N/P).
- (J) The subclass relation $C_1\overline{\tau}^1 \ldots C_n\overline{\tau}^n \Rightarrow C\overline{\alpha}$ is build with complex preconditions $\leftrightarrow$ instead of a simple order $\preceq$ on the class names (N/P).
- **But** there is no general notion of normalization. In [Jon95] the idea of simplifying is proposed by examples, but no axiomatic investigation is done.

6.3 System O

The system O can deal with overloaded parametric polymorphic functions. Further it has an denotational semantics opposed to Haskell which can have only a translation semantics leading to complicated coherence arguments. Therefore the known ambiguity problem from Haskell does not appear in System O.

The idea is to separate the set of term variables $x \in \mathcal{V}$ into the set $u \in \mathcal{U}$ of unique variables and overloaded variables $o \in \mathcal{O}$. The types of an overloaded function must have form

$$\sigma_T \ = \ T \ \alpha_1 \ldots \alpha_n \rightarrow \tau \mid \forall\overline{\alpha}.C \Rightarrow \sigma_T$$

In a typotheses , multiple typings $o : \sigma_T$ can live, but every overloaded function o can have at most one instance for a type constructor T. When representing System O as a constraint type system, the two kinds of constraints are:

General token $o : \tau \rightarrow \tau' \in \mathcal{D}$**.** This kind of constraint states the usage of a overloaded function.

Normalized token $o : \alpha \rightarrow \tau \in \mathcal{C}$**.** This kind of constraint restricts the instantiation possibilities for type variable α.

$$\text{(ENTAIL)} \quad \frac{C \vdash^e \{< \tau \leq \tau' >\}}{C \vdash^s \tau \leq \tau'}$$

$$(\leq \forall) \quad \frac{C \cup D \vdash^s \sigma \leq \sigma' \quad \alpha \notin tv(\sigma) \cup tv(C)}{C \vdash^s \sigma \leq \forall \alpha.D \Rightarrow \sigma'}$$

$$(\forall \leq) \quad \frac{C \vdash^s [\bar{\tau}/\bar{\alpha}]\sigma \leq \tau' \quad C \vdash^e [\bar{\tau}/\bar{\alpha}]D}{C \vdash^s \forall \bar{\alpha}.D \Rightarrow \sigma \leq \tau'}$$

Figure 11: Subtype rules

$$\text{(SUB)} \quad \frac{C,, \vdash e : \sigma \quad C \vdash^e \sigma \leq \tau}{C,, \vdash e : \tau}$$

Figure 12: Logical subtype system

While type inference the appearence a of constraint $o : \tau \to \tau'$ in constraint set D indicates an application of overloaded functions o with argument of type τ and result of type τ'. So normalization must check wether such an instance is defined. A proof of termination and minimality of the result for the normalization algorithm can be found in [WO96].

Normalization extension : overloading

$o : T\overline{\tau} \to \tau' \quad \Rightarrow \quad$ if $o : \sigma_T \notin$, then fail
else let $\sigma_T \equiv \forall \overline{\alpha}.C' \Rightarrow T\overline{\alpha} \to \tau$
in $D := D \cup [\overline{\tau}/\overline{\alpha}]C' \cup \{< \tau' = [\overline{\tau}/\overline{\alpha}]\tau >\}$;

$o : \alpha \to \tau \quad \Rightarrow \quad$ for *all* $o : \alpha \to \tau' \in D$ do
$D := D \cup \{< \tau = \tau' >\} \setminus \{o : \alpha \to \tau'\}$;
$C := C \cup \{o : \alpha \to \tau\}$;

7 A Type System with Subtyping

We now consider a type system with subtyping. The subsumption rule is added to the logical type system in figure 2. Of course, we must introduce a constraint system that is able to express subtyping.

Definition 11 (Subtype Constraint System) *A subtype constraint system $\mathcal{SD}$ is a type constraint system with the following additional properties. For types τ and τ' the token $< \tau \leq \tau' >$ is contained in the set Ω of primitive constraints.*

S1 $\{< \tau = \tau' >\} =^e \{< \tau \leq \tau' > \wedge < \tau' \leq \tau >\}$

S2 $$\frac{D \vdash^e \{< \tau_2 \leq \tau_1 >\} \quad D \vdash^e \{< \tau_1 \leq \tau_2' >\}}{D \vdash^e \{< \tau_1 \to \tau_1 \leq \tau_2 \to \tau_2' >\}}$$

S3 $$\frac{D \vdash^e \{< \tau_1 \leq \tau_2 >\} \quad D \vdash^e \{< \tau_2 \leq \tau_3 >\}}{D \vdash^e \{< \tau_1 \leq \tau_3 >\}}$$

Now, we introduce the subtype rules in Figure 11.

As for $\vdash^i$, we get that the (CUT) and (TRANS) rules hold. These lemmas are already stated for $\vdash^i$ in Lemmas 6 and 7. Additionally, we have the ($\to$ Intro) rule.

Lemma 10 *Let C be a constraint and τ_1, τ_2, τ_3 be types.*

$$\text{if} \quad C \vdash^s \tau_1' \leq \tau_1 \quad C \vdash^s \tau_2 \leq \tau_2'$$
$$\text{then}$$
$$C \vdash^s \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$$

The type system can be found in Figure 12. There is just one additional rule compared to the previous type system, it is the (SUB) rule. Note that the ($\forall$ Elim) rule is entailed by the (SUB) rule so there is no need for it. Rules (VAR) – (LET) stay unchanged.

The type inference algorithm in Figure 13 is the same except for the (APP) rule. In this extension normalization is trivial if we only deal with subtype constraints.

Of course, we get the same results as already stated in section 4.

Examples:
We have already discussed the type systems of Aiken, Wimmers [AW93], G. Smith [Smi94] and Hopkins Objects Group [EST95]. Because of the differences in the ($\forall$ Intro) rules there is no one to one correspondence to our approach. As already mentioned, the type system of the Hopkins Objects Group can be seen as a special instance of our approach that deals with just subtyping constraints. In other words normalization only tests if the constraint is satisfiable. Type inference reports only a constraint and always the **1** substitution. Consider the following example.

Example 6

$$\mathbf{1}, \{< \alpha \leq \beta \to \gamma >, < \alpha \leq \gamma \to \delta >\}, \emptyset \vdash^W \lambda f.\lambda x.f(fx) : (\alpha \to \beta) \to \delta.$$

$$\text{(APP)}\quad \frac{\begin{array}{c}\psi_1, C_1, , \ \vdash^W \ e_1 : \tau_1 \quad \psi_2, C_2, , \ \vdash^W \ e_2 : \tau_2 \\ \psi' = \psi_1 \sqcup \psi_2 \\ D = C_1 \wedge C_1 \wedge \{ < \tau_1 \leq \tau_2 \rightarrow \alpha > \} \quad \alpha \text{ new} \\ (C, \psi) = normalize(D, \psi')\end{array}}{\psi_{|tv(\Gamma)}, C, , \ \vdash^W \ e_1 e_2 : \psi\alpha}$$

Figure 13: Subtype inference

8 Conclusion

We have presented a general framework HM(X) for Hindley/Milner style type systems with constraints. We have introduced a new formulation of the ($\forall$ Intro) rule. Also, if the constraint domain X satisfies some sufficient conditions we get the principal type property. Several extensions, such as the treatment of extensible records, type classes, overloading or subtypes have been examined. To design a full language or static analysis based on our approach, one must simply check that the conditions on the constraint system are met. If this is the case, one gets a type inference algorithm and the principal type property for free.

References

[AW93] A.Aiken and E.L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language (extended abstract). In *Proceeding of ECOOP*, pages 27–51, LNCS 952, 1995. Springer Verlag.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[DHM95] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype quantifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Proceedings of SAS*, pages 118–135. Springer Verlag, September 1995.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, January 1982.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursivly constrained types and its application to object oriented programming. In *Electronic Notes in Theoretical Computer Science*, volume 1, 1995.

[HHJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *TOPLAS*, 18(2):109–138, March 1996.

[HMT71] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras.* North-Holland Publishing Company, 1971.

[JM94] Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *Journal of Loagic Programming*, 19,20:503–581, 1994.

[Jon92] Mark P. Jones. *Qualified Types : Theory and Practice.* PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992.

[Jon95] Mark P. Jones. Simplifying and improving qualified types. In *Proc. FPCA'95 Conf. on Functional Programming Languages and Computer Architecture*, 1995.

[KC92] Martin Odersky Kung Chen, Paul Hudak. Parametric type classes. In *Proc. of Lisp and F.P.*, pages 170–181. ACM, 1992.

[LMM87] J. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kauffmann, 1987.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[MPS86] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[NP93] Tobias Nipkow and Christian Prehofer. Type checking type classes. *POPL*, pages 409–418, 1993.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *TOPLAS*, 17(6):805–843, November 1995.

[OWW95] M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *Proc. FPCA'95 Conf. on Functional Programming Languages and Computer Architecture*, 1995.

[Rem89] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 77–88. ACM, January 1989.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming.* Logic Programming Series, ACM Doctoral Dissertation Award Series. MIT Press, Cambridge, Massachusetts, 1993.

[Smi94] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[Sul96] Martin Sulzmann. Proof of the properties for constrained types. Technical report, University of Karlsruhe, 1996.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 162–173, Los Alamitos, California, June 1992. IEEE Computer Society Press.

[WO96] M. Wehr and M. Odersky. Proof of the principal type property for system O. Technical report, Universität Karlsruhe, Interner Bericht 16/96, June 1996.

A Technical Details for Type Reconstruction

Definition 12

$$\begin{array}{lcl} gen(C,,,\sigma) & = & (\forall \bar{\alpha}_n.C \Rightarrow \sigma, \exists \bar{\alpha}_n.C) \\ gen_1(C,,,\sigma) & = & \forall \bar{\alpha}_n.C \Rightarrow \sigma \end{array}$$

where $\bar{\alpha}_n = (tv(\sigma) \cup tv(C)) \backslash tv(,)$.

Also, we have to compute the *generic instance* of a type scheme σ.

Definition 13 *The* generic instance *of a type scheme* σ*, namely* $new\_inst(\sigma)$*, is defined by the following:*

$$\begin{array}{lcl} new\_inst(\sigma) & = & do\_inst(\sigma, 1) \\ do\_inst(\forall \alpha.C \Rightarrow \sigma, D) & = & do\_inst([\beta/\alpha]\sigma, [\beta/\alpha]C \wedge D) \\ do\_inst(\tau, C) & = & (\tau, C) \end{array}$$

where β *is a new type variable.*