Type Inference with Constrained Types

Martin Sulzmann*
Yale University
Department of Computer Science
New Haven, CT 06520-8285
sulzmann@cs.yale.edu

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

Abstract

In this paper we present a general framework HM(X) for Hindley/Milner style type systems with constraints. We give a generic type inference algorithm for HM(X). Under sufficient conditions on X, type inference will always compute the principal type of a term. We give an extension of our HM(X) systems which deals with subtyping. In particular, the type inference algorithm for subtypes computes principal types. Simplification of the constraints inferred by the type inference algorithm is discussed in general for HM(X) and in the special case of subtyping.

1 Introduction

We study an extension of the Hindley/Milner [Mil78] system with constraints. Cardelli/Wegner [CW85] gave an early survey about general research directions. Reynolds [Rey85] and Mitchell [Mit84] are foundational papers that develop basic concepts of constraints and subtyping. There are examples for extensions of the Hindley/Milner system with records [Oho95, Rem89], overloading [Jon92, Kae92, HHJW96, NP93, KC92, OWW95], and subtyping [CCH+89, BSvG95, AW93, EST95b]. Extensions of Hindley/Milner with constraints are also increasingly popular in program analysis [DHM95, TJ92]. Palsberg [Pal95] gave an efficient inference algorithm for a calculus of objects. The main feature of his system is that he does not use the Hindley/Milner approach to type inference. It remains to be seen how his approach is related to ours.

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.

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:

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

where C and D are constraints over the type variables in the type context Γ and the type scheme σ.

Second, projection is an important source of opportunities for simplifying constraints [Jon95, Pot96, EST95a]. In our framework, *simplifying* means changing the syntactic representation of a constraint without changing its denotation. For example, the subtyping constraint

$$\exists\beta.(\alpha \leq \beta) \wedge (\beta \leq \gamma)$$

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.

*Supported by a Yale University Fellowship.

Two of the main strengths of the Hindley/Milner system are the existence of a principal types theorem and a type inference algorithm. One of the key ideas of our paper is to 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.

Additionally, we give an extension of our HM(X) framework that deals with subtypes. The concept of subtyping is of great importance in many record and object calculi. We enrich our constraint system with a subtype predicate between types τ and τ', denoted by $(\tau \leq \tau')$. In our type system we additionally have the subsumption rule. We provide an extension of our generic type inference algorithm that computes principal types. Previous approaches towards type inference with subtypes [AW93, EST95b] lack this property.

In case of subtyping, simplifying constraints [Jon95, Pot96, EST95a] becomes an important issue. Type inference simply accumulats all constraint problems and ends up in a large constraint set. One reason is that unification–based approaches to type inference with subtypes do not seem to apply. We discuss simplification in general for our HM(X) framework and in the special case of subtypes.

The rest of this paper is structured as follows: We give a characterization of constraint systems in Section 2. A general framework HM(X) for Hindley/Milner style type systems with constraints is presented in Section 3. In section 4 we discuss how to extend our approach with subtyping. Simplification is discussed in Section 5. Section 6 concludes.

2 Constraint Systems

We present a characterization 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]. *The relation* $\vdash^e \subseteq p\Omega \times \Omega$ *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.*

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

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* $P \in D$. *Furthermore, we define* $C =^e D$ *iff* $C \vdash^e D$ *and* $D \vdash^e C$. *The term* $\vdash^e C$ *is an abbreviation for* $\emptyset \vdash^e C$ *and* $\mathsf{true} = \{P \mid \emptyset \vdash^e P\}$ *represents the true element.*

By the following lemma we can identify the minimal constraint $\emptyset$ as an representation for true.

Lemma 1 *Given a simple constraint system constraint* $(\Omega, \vdash^e)$. *Then* $\mathsf{true} =^e \emptyset$.

Remark 1 *For simplicity, we omit set notation for constraints. We connect constraints by* $\wedge$ *instead of the union operator* $\cup$. *Also, we omit to enclose simple constraints* P *in opening and closing braces. That means,* $P \wedge Q$ *is an abbreviation for* $\{P\} \cup \{Q\}$.

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

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

E1 $C \vdash^e \exists\alpha.C$
E2 $C \vdash^e D$ *implies* $\exists\alpha.C \vdash^e \exists\alpha.D$
E3 $\exists\alpha.(C \wedge \exists\alpha.D) =^e \exists\alpha.C \wedge \exists\alpha.D$
E4 $\exists\alpha.\exists\beta =^e \exists\beta.\exists\alpha.C$

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

Definition 3 *(Free Variables) Let* C *be a constraint. Then* $fv(C) = \{\alpha \mid \exists\alpha.C \neq^e C\}$.

We now define satisfiability of a constraint.

Definition 4 *(Satisfiability) Let* C *be a constraint. Then* C *is* satisfiable *iff* $\vdash^e \exists fv(C).C$.

The next lemma states that the projection operator does not influence the satisfiability of a constraint.

Lemma 2 *Let* C *be a constraint. Then* C *is satisfiable iff* $\exists\alpha.C$ *is satisfiable.*

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

Definition 5 *(Types) A* type *is a member of* $\mathcal{T} = Term(\Sigma)$ *where* $Term(\Sigma)$ *is the term algebra* $\mathcal{T}$ *build up from a signature* $\Sigma = (Var, \rightarrow)$*. Var is a set of variables and* $\rightarrow$ *is the function constructor of arity 2.*

Definition 6 *(Substitutions) A* substitution ϕ *is an idempotent mapping from a set of variables Var to the term algebra* $Term(\Sigma)$*. Let id be the identity substitution.*

Definition 7 *(Term Constraint System) A* term constraint system $\mathbf{D}_{\mathcal{T}} = (\Omega, \vdash^e, Var, \{\exists\alpha \mid \alpha \in Var\})$ *over a term algebra* $\mathcal{T}$ *is a cylindric constraint system with tokens of the form*

$$p(\tau_1, \ldots, \tau_n) \qquad (\tau_i \in \mathcal{T})$$

such that the following holds:

- *For each pair of types* τ, τ' *there is an equality predicate* $(\tau = \tau')$ *in* $\mathbf{D}_{\mathcal{T}}$*, which satisfies:*

  D1 $\vdash^e (\alpha = \alpha)$
  D2 $(\alpha = \beta) \vdash^e (\beta = \alpha)$
  D3 $(\alpha = \beta) \wedge (\beta = \gamma) \vdash^e (\alpha = \gamma)$
  D4 $(\alpha = \beta) \wedge \exists\alpha.C \wedge (\alpha = \beta) \vdash^e C$
  D5 $(\alpha_1 = \beta_1) \wedge (\alpha_2 = \beta_2) \vdash^e$
  $(\alpha_1 \rightarrow \alpha_n = \beta_1 \rightarrow \beta_n)$

- *For each predicate P,*

  D6 $[\tau/\alpha]P =^e \exists\alpha.P \wedge (\alpha = \tau)$
  where $\alpha \notin fv(\tau)$

Remark 2 *Conditions* **D1** – **D4** *are the conditions imposed on a cylindric constraint system with diagonal elements, which is usually taken as the foundation of constraint programming languages.* **D4** *says that equals can be substituted for equals; it is in effect the Leibniz principle.* **D5** *extends equality to function types.* **D6** *connects the syntactic operation of a substitution over predicates with the semantic concepts of projection and equality. Substitution is extended to arbitrary constraints in the canonical way:*

$$[\tau/\alpha](P_1 \wedge \ldots \wedge P_n) = [\tau/\alpha]P_1 \wedge \ldots \wedge [\tau/\alpha]P_n.$$

The intention of a term constraint system is to express unification problems through the equality predicate $(=)$ with respect to a cylindric constraint system **C** over a term algebra $\mathcal{T}$.

Some basic lemmas follow.

Lemma 3 *(Normal Form) Let C be a constraint and* $\phi = [\bar{\tau}/\bar{\alpha}]$ *be a substitution. Then* $\phi C =^e \exists\bar{\alpha}.C \wedge (\alpha_1 = \tau_1) \wedge \ldots \wedge (\alpha_n = \tau_n)$*.*

Lemma 4 *(Substitution) Let* C, D *be constraints and* ϕ *be a substitution such that* $C \vdash^e D$*. Then* $\phi C \vdash^e \phi D$*.*

Then it is an easy observation how to extend conditions **D1** – **D5** to arbitrary types.

Lemma 5 *(Congruence)* $(=)$ *is a congruence.*

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

This sections describes a general extension HM(X) of the Hindley/Milner type system with a constraint system X over a term algebra $\mathcal{T}$. We require that X be a cylindric constraint sytem. Our development is similar to the original presentation [DM82].

3.1 The Type System

Values	v	$::=$	$x \mid \lambda x.e$
Expressions	e	$::=$	$v \mid ee \mid \mathsf{let}\ e = x\ \mathsf{in}\ e$
Types	τ	$::=$	$\alpha \mid \tau \rightarrow \tau$
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 in X which restricts the types that can be substituted for the type variable α.

The typing rules can be found in Figure 1. The most interesting rules are the ($\forall$ Intro) rule and the ($\forall$ Elim) rule. By rule ($\forall$ Intro) we quantify some type variables. We often use vector notation for type variables in type schemes. 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 $\exists\alpha_1.\ldots\exists\alpha_n.D$.

Definition 8 *(Typing Judgements) Let C be a constraint in X,* Γ *a type context and* σ *a type scheme. Then* $C, \Gamma \vdash e : \sigma$ *is a* typing judgement *iff* $C, \Gamma \vdash e : \sigma$ *can be derived by application of the typing rules.*

Example 1 *(Hindley/Milner) The Hindley/Milner system is an instance of our type system. Let* $\mathbf{1} = (\emptyset, \vdash^e, Var, \{\exists\alpha \mid \alpha \in Var\})$ *be the miminal cylindric constraint system. Then the Hindley/Milner system is equal to HM(***1***). All constraints are considered to be* `true`*.*

We now discuss type soundness of our type system. We use a syntactic approach to type soundness [WF91]. We have the following two reductions:

$$\begin{array}{lll} (\beta) & (\lambda x.e)v & \longrightarrow [v/x]e \\ (\mathrm{LET}) & \mathsf{let}\ x = v\ \mathsf{in}\ e & \longrightarrow [v/x]e \end{array}$$

Then, we can state the following lemma.

$$
\begin{array}{ll}
\text{(VAR)} & C, \Gamma \vdash x : \sigma \quad (x : \sigma \in \Gamma) \\[2ex]
\text{(ABS)} & \dfrac{C, \Gamma_x.x : \tau \vdash e : \tau'}{C, \Gamma_x \vdash \lambda x.e : \tau \rightarrow \tau'} \\[3ex]
\text{(APP)} & \dfrac{C, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad C, \Gamma \vdash e_2 : \tau_1}{C, \Gamma \vdash e_1 e_2 : \tau_2} \\[3ex]
\text{(LET)} & \dfrac{C, \Gamma_x \vdash e : \sigma \qquad C, \Gamma_x.x : \sigma \vdash e' : \tau'}{C, \Gamma_x \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau'} \\[3ex]
(\forall\ \text{Intro}) & \dfrac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C \wedge \exists \bar{\alpha}.D, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau} \\[3ex]
(\forall\ \text{Elim}) & \dfrac{C, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau' \qquad C \vdash^e [\bar{\tau}/\bar{\alpha}]D}{C, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau'}
\end{array}
$$

Figure 1: Logical type system

Lemma 6 *(Subject Reduction Property) Let $C, \Gamma \vdash e : \sigma$ be typing judgement and $e \longrightarrow e'$. Then $C, \Gamma \vdash e' : \sigma$.*

We now extend the relation $\longrightarrow$ to evaluation contexts E. We define

$$E[e] \longmapsto E[e'] \quad \text{iff} \quad e \longrightarrow e'$$

where

$$E ::= [\,] \mid Ee \mid vE \mid \mathsf{let}\ x = E\ \mathsf{in}\ e.$$

Let $\overset{*}{\longmapsto}$ be the reflexive and transitive closure of $\longmapsto$. The partial function *eval* evaluates an expression e:

$$eval(e) = v \quad \text{iff} \quad e \overset{*}{\longmapsto} v$$

We write $e \Uparrow$ if e diverges. Now, we have everything at hand to state syntactic soundness of our type system.

Theorem 1 *(Syntactic Soundness) Let $C, \Gamma \vdash e : \sigma$ be a typing judgement. Then $e \Uparrow$ or $eval(e) = v$.*

In the syntactic approach to type soundness we do not consider whether the constraints on the left hand side of typing judgements are satisfiable or not. The question arises how our type system deals with unsatisfiable constraints. We can state that our typing rules are sound.

Lemma 7 *The constraint of a conclusion of a typing rule is satisfiable iff the constraints of the premises are satisfiable.*

That means whenever we derive a typing judgement $C, \Gamma \vdash e : \sigma$ where C is consistent, all constraints on the left hand side of typing judgements of the derivation tree are satisfiable.

3.2 Type Inference

Now, we consider the type inference problem. The following definition is a generalization of the ($\forall$ Intro) rule.

Definition 9 *Let C be a constraint, Γ be a type context, σ be a type scheme and $\bar{\alpha} = (fv(\sigma) \cup fv(C)) \backslash fv(\Gamma)$. Then $gen(C, \Gamma, \sigma) = (\exists \bar{\alpha}.C, \forall \bar{\alpha}.C \Rightarrow \sigma)$.*

The type inference algorithm can be found in Figure 2. We treat a type τ as $\forall \alpha.\mathsf{true} \Rightarrow \tau$ where $\alpha \notin fv(\tau)$ to avoid a special treatment of types in the (VAR) rule. The algorithm is formulated as a deduction system over clauses of the form $\psi, C, \Gamma \vdash^W e : \tau$ with type context Γ, expression e as input values and substitution ψ, constraint C, type τ as output values. For each syntactic construct of expressions e we have one clause.

Type inference involes *normalization* of constraints. In each rule except for the (ABS) rule a normalization step is performed. Remember, on the left hand side of typing judgements we only admit constraints in the cylindric constraint system X whereas type inference generally results in constraint problems in a term constraint system. That means we have to normalize a constraint in a term constraint system attached with a substitution ϕ. We want to get a constraint C in X and a substitution ψ.

$$\text{(VAR)} \quad \frac{x : (\forall \bar{\alpha}.D \Rightarrow \tau) \in \Gamma \qquad \bar{\beta} \text{ new} \quad (C,\psi) = normalize(D, [\bar{\beta}/\bar{\alpha}])}{\psi_{|fv(\Gamma)}, C, \Gamma \vdash^W x : \psi\tau}$$

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

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

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

Figure 2: Type inference

We give an axiomatic description of normalization in the next section. For instance, in the Hindley/Milner type system normalization of

$$(int \to bool = int \to \alpha)$$

yields (C, ψ) where $C =^e \mathsf{true}$ and $\psi = [bool/\alpha]$. Then, we introduce an instance relation to compare the results of the type inference algorithm. Finally, we state the main results: soundness and completeness of type inference.

3.2.1 Normalization

In this section we study the relationship between cylindric and term constraint systems. Below we give an axiomatic description of normalization. First, we introduce some preliminary definitions.

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 10 *Let ϕ, ϕ' and ψ be substitutions. Then $\psi \leq_U^{\phi'} \phi$ iff $(\phi' \circ \psi)_{|U} = \phi_{|U}$.*

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.

We consider now the task of normalization. Normalization means computation of a *principal normal form* of a constraint D from a term constraint system and a substitution ϕ.

Definition 11 *(Principal Normal Form) Let $\mathbf{C}$ be a cylindric constraint system, $\mathcal{T}$ be a term algebra and let $\mathbf{D}_{\mathcal{T}}$ be a term constraint system. Let $C \in \mathbf{C}$ and $D \in \mathbf{D}_{\mathcal{T}}$ be constraints and let ϕ,ψ be substitutions. Then (C, ψ) is a* normal form *of (D, ϕ) iff $\phi \leq \psi$, $C \vdash^e \psi D$ and $\psi C = C$*

(C, ψ) is principal *if for all normal forms (C', ψ') of (D, ϕ) we have that $\psi \leq \psi'$ and $C' \vdash^e \psi' C$.*

Principal normal forms are unique up to variable renaming:

Lemma 8 *(Uniqueness) Let (C, ψ) and (C', ψ') be principal normal forms of (D, ϕ). Then there is a vari-*

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

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

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

Figure 3: Instance rules

able renaming ϕ' such that $C' = \phi' C$ and $\psi' = \phi' \circ \psi$.

That means computation of the principal norm form of (D, ϕ) is well–defined. Given (D, ϕ) we can define a function *normalize* by:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{normalize}(D, \phi)} \\ = \ (C, \psi) & \text{if } (C, \psi) \text{ principal normal form of } (D, \phi) \\ = \ \mathit{fail} & \text{otherwise} \end{array}$$

Definition 12 *(Principal Constraint Property) Given a term algebra $\mathcal{T}$. A cylindric constraint system $\mathbf{C}$ has the* principal constraint property *if there exists a term constraint system $\mathbf{D}_{\mathcal{T}}$ such that $\mathbf{C} \subseteq \mathbf{D}_{\mathcal{T}}$ and for every constraint $D \in \mathbf{D}_{\mathcal{T}}$ and substitution ϕ, either (D, ϕ) does not have a normal form or (D, ϕ) has a principal normal form.*

We also say that a HM(X) type system has the principal constraint property if X has the principal constraint property.

In the case of the Hindley/Milner type system (or HM($\mathbf{1}$) in terms of our notation), normalization means simply computation of the most general unifier. We have in this case:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{normalize}((\tau_1 = \tau_2), \phi)} \\ = \ (\mathsf{true}, \mathsf{mgu}(\phi\tau_1, \phi\tau_2) \circ \phi) & \text{if } \phi\tau_1, \phi\tau_2 \text{ are unifiable} \\ = \ \mathit{fail} & \text{otherwise} \end{array}$$

This is easy to see. Assume first that $\phi\tau_1$ and $\phi\tau_2$ are unifiable and let $\psi = \mathsf{mgu}(\phi\tau_1, \phi\tau_2)$. Then

$$\mathsf{true} \vdash^e \ (\psi \circ \phi)(\tau_1 = \tau_2)$$

and $(\mathsf{true}, \psi \circ \phi)$ is minimal because ψ is a most general unifier of $(\phi\tau_1, \phi\tau_2)$. On the other hand, if there is no solution to $(\tau_1 = \tau_2)$, no unifier exists at all. That means, normalization fails.

In this case, the constraint system $\mathbf{1}$ itself is not expressive enough to express failure of normalization. The reason is all constraints in $\mathbf{1}$ are satisfiable. The question arise, what happens if the principal normal form of an unsatisfiable constraint exists. We can state the following lemma.

Lemma 9 *Given a term algebra $\mathcal{T}$. Let $\mathbf{C}$ be a cylindric constraint system having the principal constraint property. Let $D \in \mathbf{D}_{\mathcal{T}}$ and ϕ be a substitution such that ϕD is unsatisfiable. If (C, ψ) is the principal normal form of (D, ϕ) then C is unsatisfiable.*

3.2.2 The Instance Relation

In order to state the main 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 $C \vdash^i \ \sigma \preceq \sigma'$, see Figure 3. It is possible to enforce that all $(\forall \preceq)$ steps are performed before $(\preceq \forall)$ steps. We simply have the following new relation $\vdash^{i_2}$. All instance rules are the same execpt for the $(\forall \preceq)$ rule:

$$(\forall \preceq_2) \quad \frac{C \vdash^{i_2} \ [\tau/\alpha]\sigma \preceq \tau' \qquad C \vdash^e \ [\tau/\alpha]D}{C \vdash^{i_2} \ \forall \alpha . D \Rightarrow \sigma \preceq \tau'}.$$

The next lemma states that $\vdash^i$ and $\vdash^{i_2}$ are equivalent.

Lemma 10 *(Equivalence) Let C be a constraint and σ, σ' be type schemes. Then $C \vdash^i \ \sigma \preceq \sigma'$ iff $C \vdash^{i_2} \ \sigma \preceq \ \sigma'$.*

Also, the (Ref-σ), (Cut) and (Trans) rule are derivable.

Lemma 11 *(Ref-σ) Let C be a constraint and σ be a type scheme. Then we can derive $C \vdash^i \ \sigma \preceq \sigma$.*

Lemma 12 *(Cut) Let C, D be constraints and σ, σ' be type schemes such that $D \vdash^e \ C$ and $C \vdash^i \ \sigma \preceq \sigma'$. Then $D \vdash^i \ \sigma \preceq \sigma'$ holds.*

Lemma 13 *(Trans) Let C be a constraint and σ_1, σ_2, σ_3 be type schemes such that $C \vdash^i \ \sigma_1 \preceq \sigma_2$ and $C \vdash^i \ \sigma_2 \preceq \sigma_3$. Then we get $C \vdash^i \ \sigma_1 \preceq \sigma_3$.*

Now, we introduce a relation between so–called typing configurations (C, σ, ψ).

Definition 13 *(Typing Configurations) Let C be a constraint, σ be a type scheme and ψ be a substitution such that $\psi C = C$ and $\psi\sigma = \sigma$. Then (C, σ, ψ) is a* typing configuration*.*

Let (C, σ, ψ), (C', σ', ψ') be typing configurations and Γ be a type context. Then (C, σ, ψ) is more general *than (C', σ', ψ') with respect to Γ iff $\psi \leq^{\phi'}_{fv(\Gamma)} \phi$, $C' \vdash^e \phi' C$ and $C' \vdash^i \phi'\sigma \preceq \sigma'$.*

We write $(C, \sigma, \psi) \preceq_\Gamma (C', \sigma', \phi)$.

The next lemma expresses the relationship between typing jugdements and configurations.

Lemma 14 *Let (C, σ, ψ), (C', σ', ψ') be typing configurations where $(C, \sigma, \psi) \preceq_\Gamma (C', \sigma', \psi')$, Γ be a type context and $C, \psi\Gamma \vdash e : \sigma$ be a typing judgement. Then we can derive the typing judgement $C', \phi\Gamma \vdash e : \sigma'$.*

3.2.3 Main Results

We now present our main results. We assume that we have a HM(X) type system which has the principal constraint property.

Theorem 2 *(Soundness) Given a term e and a type context Γ. If $\psi, C, \Gamma \vdash^W e : \tau$ then $C, \psi\Gamma \vdash e : \tau$, $\psi C = C$ and $\psi\tau = \tau$.*

Now, we present our completeness result. Informally speaking, we want to have the following. Given a derivation $C', \phi\Gamma \vdash e : \sigma'$, our type inference algorithm should report a constraint that is at least as small as C' and a type that is at least as general as σ'. First, we require that every type scheme in the type context $\phi\Gamma$ has a type instance with respect to the constraint C'. This is reflected by the following definition.

Definition 14 *Let C' be a constraint and Γ' be a type context. Then Γ' is* satisfiable *in C' iff for every $x : \sigma \in \Gamma'$ there is a τ such that $C' \vdash^i \sigma \preceq \tau$*

It now follows the completeness result.

Theorem 3 *(Completeness) Let $C', \phi\Gamma \vdash e : \sigma'$ be a typing judgement and $\phi\Gamma$ is satisfiable in C'. Then $\psi, C, \Gamma \vdash^W e : \tau$ and $(C_o, \sigma_o, \psi) \preceq_\Gamma (C', \sigma', \phi)$ where $(C_o, \sigma_o) = gen(C, \psi\Gamma, \tau)$.*

Let us briefly comment the completeness result. We assume that we have a typing configuration (C', σ', ϕ) such that $C', \phi\Gamma \vdash e : \sigma'$. Then if we generalize the results of the type inference algorithm, we get a more general typing configuration.

The following corollary states that our type inference algorithm computes principal types.

Corollary 1 *(Principal Types) Let $C', \Gamma \vdash e : \sigma'$ be a typing judgement, $fv(\Gamma) = \emptyset$ and Γ is satisfiable in C'. Then $id, C, \Gamma \vdash^W e : \tau$ and $(C_o, \sigma_o, id) \preceq_\Gamma (C', \sigma', id)$ where $(C_o, \sigma_o) = gen(C, \Gamma, \tau)$.*

4 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 1. First, we must introduce a constraint system that is able to express subtyping.

Definition 15 *(Subtype Constraint System) A* subtype constraint system $\mathbf{SC}_\mathcal{T}$ *over a term algebra $\mathcal{T}$ is a term constraint system with the following additional properties. For each pair of types τ and τ' there is a subtype predicate $(\tau \leq \tau')$ in $\mathbf{SC}_\mathcal{T}$, which satisfies:*

S1 $(\alpha = \alpha') =^e (\alpha \leq \alpha') \wedge (\alpha' \leq \alpha)$

S2 $$\frac{D \vdash^e (\alpha'_1 \leq \alpha_1) \quad D \vdash^e (\alpha_2 \leq \alpha'_2)}{D \vdash^e (\alpha_1 \rightarrow \alpha_2 \leq \alpha'_1 \rightarrow \alpha'_2)}$$

S3 $$\frac{D \vdash^e (\alpha_1 \leq \alpha_2) \quad D \vdash^e (\alpha_2 \leq \alpha_3)}{D \vdash^e (\alpha_1 \leq \alpha_3)}$$

Now, we introduce the subtype rules in Figure 4. As for $\vdash^i$, we get that the (Ref-σ), (Cut) and (Trans) rules hold. These lemmas are already stated for $\vdash^i$ in Lemmas 11, 12 and 13. Additionally, we have the ($\rightarrow$ Intro) rule.

Lemma 15 *($\rightarrow$ Intro) Let C be a constraint and τ_1, τ_2, τ_3 be types. If $C \vdash^s \tau'_1 \leq \tau_1$ and $C \vdash^s \tau_2 \leq \tau'_2$ then $C \vdash^s \tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2$.*

The type system can be found in Figure 4. 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 4 is the same except for the (APP) rule. Instead of the equality predicate $(=)$ we have now the subtype predicate $(\leq)$.

In this extension computation of the principal normal form is trivial because a subtype constraint system $\mathbf{SC}_\mathcal{T}$ is expressive enough to state all constraint problems which arise during type inference. That means, the principal constraint property holds trivially for a subtype constraint system $\mathbf{SC}_\mathcal{T}$. We get that

$$normalize(D, \psi) = (\psi D, \psi).$$

But then our type inference algorithm reports always the *id* substitution. That is true for the (VAR) rule.

Subtype rules :	(ENTAIL)	$\dfrac{C \vdash^e (\tau \leq \tau')}{C \vdash^s \tau \leq \tau'}$
	$(\leq \forall)$	$\dfrac{C \wedge D \vdash^s \sigma \leq \sigma' \quad \alpha \notin tv(\sigma) \cup tv(C)}{C \vdash^s \sigma \leq \forall \alpha . D \Rightarrow \sigma'}$
	$(\forall \leq)$	$\dfrac{C \vdash^s [\bar{\tau}/\bar{\alpha}]\sigma \leq \sigma' \quad C \vdash^e [\bar{\tau}/\bar{\alpha}]D}{C \vdash^s \forall \bar{\alpha}.D \Rightarrow \sigma \leq \sigma'}$
Logical subtype system :	(SUB)	$\dfrac{C, \Gamma \vdash e : \sigma \quad C \vdash^s \sigma \leq \tau}{C, \Gamma \vdash e : \tau}$
Subtype inference :	(APP)	$\dfrac{\begin{array}{c} \psi_1, C_1, \Gamma \vdash^W e_1 : \tau_1 \quad \psi_2, C_2, \Gamma \vdash^W e_2 : \tau_2 \\ \psi' = \psi_1 \sqcup \psi_2 \\ D = C_1 \wedge C_1 \wedge (\tau_1 \leq \tau_2 \rightarrow \alpha) \quad \alpha \text{ new} \\ (C, \psi) = normalize(D, \psi') \end{array}}{\psi_{\mid fv(\Gamma)}, C, \Gamma \vdash^W e_1 e_2 : \psi\alpha}$

Figure 4: Extensions for Subtyping

And whenever we have to compute a normalization we know that $normalize(D, id) = (D, id)$. Therefore, the other inference rules report also the id substitution. Consider the following example.

Example 2

$$id, (\alpha \leq \beta \rightarrow \gamma) \wedge (\alpha \leq \gamma \rightarrow \delta), \emptyset$$
$$\vdash^W$$
$$\lambda f.\lambda x.f(fx) : \alpha \rightarrow \beta \rightarrow \delta.$$

We get the same main results as already stated in Section 3. We have not exploited the treatment of records. The reason is we only admit the function constructor $\rightarrow$ in the term algebra $\mathcal{T}$. It is straightforward to extend the term algebra $\mathcal{T}$ with further type constructors T. The encoding of records is postponed to future work.

Related Work:
Our extension is most closely related to the one of the Hopkins Object Group [EST95b]. But instead of simply duplicating the constraint in the ($\forall$ Intro) rule we split it up into two versions, one existentially quantified the other universally quantified. Consider the constraint

$$C = (\alpha \leq \beta) \wedge (\beta \leq \gamma).$$

If we quantify the type variable β we get the constraint $\exists\beta.C$ which can safely be simplified to $(\alpha \leq \gamma)$ since the denotation is the same for both constraints. The type inference algorithm of the Hopkins Object Group keeps the constraint C. That means their type inference algorithm ends up in a large constraint set whereas our framework gives a semantic basis for simplifications by the projection operator. But there is also another reason why simplification becomes an important issue. We have already mentioned that the subtype constraint system is expressive enough to state all constraint problems which arise during type inference. Then type inference simply accumulats all constraint problems and also in our case we end up in a large constraint set. That means, the reported constraints are very verbose and hard to interpret. In the next section we discuss simplification in more detail.

5 Simplification

A current trend in research for type inference with constrained types is to simplify constraints, see [Pot96, EST95a]. The problem is that the reported constraints are to precise. It is very difficult for the programmer to interpret the constraints. During type inference many intermediate type variables are introduced. We want to hide such type variables and make them invisible to the programmer. By the projection operator $\exists\bar{\alpha}$ we can bind type variables $\bar{\alpha}$ in a constraint. The following lemma expresses under which circumstances we can bind type variables by the projection operator, that means making them invisible to the programmer.

Lemma 16 *Let $C, \Gamma \vdash e : \sigma$ be a typing judgement and $\bar{\alpha} \notin fv(\Gamma) \cup fv(\sigma)$. Then $\exists\bar{\alpha}.C, \Gamma \vdash e : \sigma$.*

Based on the above lemma we are able to state the following simplification rule:

$$\text{(SIMP)} \quad \frac{C, \Gamma \vdash e : \sigma \qquad \bar{\alpha} \notin fv(\Gamma) \cup fv(\sigma)}{\exists\bar{\alpha}.C, \Gamma \vdash e : \sigma}$$

Pottier [Pot96] and the Hopkins Object Group [EST95a] developped independently methods to remove *unreachable* constraints. The (SIMP) rule gives a semantic justification for doing so.

They also examine some other simplification techniques:

- Replacing a variable with its bound:
  For instance, if a type variable occurs only positively in a type, we can replace this type variable with its lower bound.
- Merging of variables:
  We consider now subtyping. If we have a typing judgement $C, \Gamma \vdash e : \sigma$ and the cyclic constraint $(\alpha \leq \alpha') \wedge (\alpha' \leq \alpha)$ is entailed by C and furthermore α' is not free in Γ then we can substitute α' with α.

All of these techniques involve computation of a substitution. This substitution is applied to the given typing judgement. We can state the following substitution rule which follows immediately from Lemma 14:

$$\text{(SUBST)} \quad \frac{\begin{array}{c} C, \phi\Gamma \vdash e : \sigma \qquad \phi \leq^{\phi'} \psi \\ \phi'_{|fv(\Gamma)} = id \quad C \vdash^e \psi C \quad C \vdash^i \psi\sigma \leq \sigma \end{array}}{\psi C, \phi\Gamma \vdash e : \psi\sigma}$$

We need the conditions

$$\phi'_{|fv(\Gamma)} = id \qquad C \vdash^e \psi C \qquad C \vdash^i \psi\sigma \leq \sigma$$

to ensure that application of the (SUBST) rule is complete. That means the typing judgement of the conclusion should be as least as general as the typing judgement of the premise.

We believe that our framework supports a semantic basis for simplifications. Our semantic definition of constraint systems allows us to change the syntactic representation of a constraint without changing its denotation. Also, we sketched how to integrate already existing simplification techniques [Pot96, EST95a].

6 Conclusion

We have presented a general framework HM(X) for Hindley/Milner style type systems with constraints. Constraint systems are introduced in terms of cylindric algebras. We have introduced a new formulation of the ($\forall$ Intro) rule. Also, if the constraint domain X has the principal normal form property we get the principal type property. 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. We have outlined this procedure for a language extension with subtyping. The embedding of object oriented features such as overloading and records will be presented in future work.

Acknowledgements

We would like to thank the anonymous referees for their helpful suggestions.

References

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 31–41, New York, June 1993. ACM Press.

[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.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[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. January 1982.

[EST95a] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA*, 1995.

[EST95b] 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.* D.phil. thesis, Oxford University, September 1992.

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

[Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping, and recursive types. pages 193–204, June 1992.

[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 Kauffman, 1987.

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

[Mit84] John C. Mitchell. Coercion and type inference. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pages 175–185, 1984.

[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] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pages 135–1469, June 1995.

[Pal95] Jens Palsberg. Efficient inference of object types. 123(2):198–209, December 1995.

[Pot96] Francois Pottier. Simplifying subtyping constraints. In *International Conference on Functional Programming*, page 122 to 133, May 1996.

[Rem89] D. Remy. Typechecking records and variants in a natural extension of ML. pages 77–88. ACM, January 1989.

[Rey85] John C. Reynolds. Three approaches to type structure. In *Proceedings TAPSOFT/CAAP 1985*, pages 97–138. Springer-Verlag, 1985. Lecture Notes in Computer Science 185.

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

[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.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report Rice COMP TR91-160, Rice University, Houston, Texas, 1991.