Type Inference with Constrained Types

Article *in* Theory and Practice of Object Systems · January 1999

DOI: 10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4 · Source: DBLP

3 authors, including:

Martin Sulzmann
Karlsruhe University of Applied Sciences
112 PUBLICATIONS **1,341** CITATIONS

Type Inference with Constrained Types

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

Martin Odersky
University of South Australia
School of Computer and Information Science
The Levels, South Australia 5095
odersky@cis.unisa.edu.au

Martin Wehr
University of Edinburgh
Laboratory for Foundations of Computer Science
wehr@dcs.ed.ac.uk

Research Report YALEU/DCS/RR-1129, April 1997

Abstract

In this paper we present a general framework HM(X) for Hindley/Milner style type systems with constraints. HM(X) stays in the tradition of the Hindley/Milner type system. Type systems in HM(X) are sound under a standard untyped compositional semantics that guarantees the slogan 'well–typed programs can not go wrong'. Furthermore, we can 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. An instance of our framework which deals with polymorphic records is discussed. Also, we give a subtyping extension SHM(X) of our HM(X) system. In particular, the type inference algorithm for subtypes computes principal types. Simplification of the constraints inferred by the type inference algorithm is discussed for HM(X) and SHM(X).

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, Rém89], overloading [Jon92, Kae92, VHJW96, NP93, CHO92, OWW95], and subtyping [CCH+89, BSvG95, AW93, EST95b]. A general extension of the Hindley/Milner with qualified types can be found in [Jon92]. We discuss Jones approach in more detail in the section about related work. Rémy [Rém92] extends the Hindley/Milner type system with a sorted equational theory. His approach is very close in spirit to ours. But he restricts his attention to equality constraints whereas our approach gives a foundation for more general constraint systems. 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 sub-

*Supported by a Yale University Fellowship.

set 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.

Two of the main strengths of the Hindley/Milner system are a type soundness result and the existence of a type inference algorithm that computes principal types. HM(X) stays in the tradition of the Hindley/Milner type system. Type systems in HM(X) are sound under a standard untyped compositional semantics provided the underlying constraint system X is sound. This result can be summarized in the slogan 'well–typed programs can not go wrong'. 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.

To this purpose, we represent the typing problem itself as a constraint. Generally, the constraint system X needs to be rich system to express all constraint problems. On the left hand side of the turnstyle and in type schemes we only admit a subset of constraints in X that are in so–called *solved form*. The task of type inference is then to split a typing problem into a substitution and a residual constraint in solved form. This we call *constraint normalization*. We require that normalization always yields a "best" solution, if there is a solution at all. For those constraint systems meeting this condition, we present a generic type inference algorithm that will always yield the principal type of a term.

Our work generalizes Milner's results to systems with non–trivial constraints and makes it thus possible to experiment with new constraint domains without having to invent yet another type inference algorithm and without having to repeat the often tedious proofs of soundness and completeness of type inference.

We discuss an instance of our HM(X) framework that deals with polymorphic records. Namely, we show how to encode the record calculus of Ohori [Oho95] in terms of our system.

Furthermore, we give an extension SHM(X) 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 SHM(X) we additionally have the subsumption rule. We provide an extension of our generic type inference algorithm that computes principal types for SHM(X) type systems. 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 accumulates 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 for HM(X) and SHM(X).

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. Section 4 establishes conditions on the constraint system so that type inference is feasible and a principal types theorem holds. An instance of our framework with polymorphic records is considered in Section 5. In Section 6 we discuss a general condition under which we get instances of our HM(X) framework for free. We discuss an extension SHM(X) of HM(X) with subtypes in section 7. Simplification is discussed in Section 8 The relationship to previous approaches is considered in Section 9. Section 10 concludes.

2 Constraint systems

We present a characterization of constraint systems along the lines of Henkin [HMT71] and Saraswat [Sar93]. We restate the definitions of a simple and a cylindric constraint system. Furthermore, we introduce the new no-

tion of a term constraint system.

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. We also refer to such constraints as predicates. 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.*

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 a 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* $\mathcal{CCS} = (\Omega, \vdash^e, Var, \{\exists\alpha \mid \alpha \in Var\})$ *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** $\exists\alpha.\exists\beta.C =^e \exists\beta.\exists\alpha.C$

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(*Σ*) where Term(*Σ*) is the term algebra* $\mathcal{T}$ *built up from a signature* Σ *= (Var, Cons). Var is a set of variables and Cons is a set of type constructors containing at least the function constructor* $\to$ *of arity 2.*

A type context $t[]$ *is a type with a hole and is defined by*

$$t[] ::= [] \mid \tau \mid t[] \to t[] \mid Tt[] \ldots t[]$$

where $\tau \in \mathcal{T}$ *and* T *stands for a type constructor in Cons besides* $\to$*.*

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

Definition 7 (Term Constraint System) *A* term constraint system $\mathcal{TCS}_\mathcal{T} = (\Omega, \vdash^e, Var, \{\exists\alpha \mid \alpha \in Var\})$ *over a term algebra* $\mathcal{T}$ *is a cylindric constraint system with predicates 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* $\mathcal{TCS}_\mathcal{T}$*, 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** $(\tau = \tau') \vdash^e (t[\tau] = t[\tau'])$ *where* $t[]$ *is a type context*
- *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** *states that (=) is a congruence.* **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:*

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

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

$$\text{(Equ)} \qquad \frac{C, \Gamma \vdash e : \tau \qquad \vdash^e (\tau = \tau')}{C, \Gamma \vdash e : \tau'}$$

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

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

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

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

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

Figure 1: Logical type system

The intention of a term constraint system $\mathcal{TCS}_\mathcal{T}$ is to express unification problems through the equality predicate (=). A term constraint system $\mathcal{TCS}_\mathcal{T}$ is a very powerful constraint system. In the latter we distinguish a specific subset of constraints in $\mathcal{TCS}_\mathcal{T}$. We denote this set by $\mathcal{S}$. We say the constraints in $\mathcal{S}$ are in *solved form*. We put in general the following conditions on $\mathcal{S}$:

- **S1** $\mathcal{S} \subseteq \mathcal{TCS}_\mathcal{T}$
- **S2** Each $C \in \mathcal{S}$ is satisfiable
- **S3** If $C \in \mathcal{S}$ and $C \vdash^e (\tau = \tau')$ then $\vdash^e (\tau = \tau')$
- **S4** If $C \in \mathcal{S}$ then $\exists \alpha.C \in \mathcal{S}$

Remark 3 *Condition* **S3** *prohibits equality predicates in $\mathcal{S}$. Equality predicate should be resolved in $\mathcal{S}$ by a kind of unification. Condition* **S4** *enforces that $\mathcal{S}$ is closed under projection. For a special instance of a term constraint system $\mathcal{TCS}_\mathcal{T}$ we might put some further restrictions on the set $\mathcal{S}$ of constraints in solved form.*

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*

$$\phi C \vdash^e \phi D$$

3 The HM(X) framework

This section describes a general extension HM(X) of the Hindley/Milner type system with a term constraint system X over a term algebra $\mathcal{T}$. We denote the set of solved constraints in X by $\mathcal{S}$. In our type system we only admit constraints in $\mathcal{S}$. Our development is similar to the original presentation [DM82]. We work with the following syntactic domains.

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

This generalizes the formulation in [DM82] in two respects. First, types are now members of an arbitrary term algebra, hence there might be other constructors besides $\to$, with possibly non–trivial equality relations. Second, type schemes $\forall \alpha.C \Rightarrow \sigma$ now include a constraint component $C \in \mathcal{S}$, which restricts the types that can be substituted for the type variable α. On the other hand, the language of terms is exactly as in [DM82]. That is, we assume that any language constructs that make use of type constraints are expressible as predefined values, whose names and types are recorded in the initial type environment Γ_0.

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 $\exists\alpha_1.\ldots\exists\alpha_n.D$.

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 $\exists\alpha_1.\ldots\exists\alpha_n.D$. Rule (Equ) becomes important if there are type constructors in the term algebra $\mathcal{T}$ with non–trivial equality relations. The other rules are a straightforward extension of the Hindley/Milner rules.

Definition 8 (Typing Judgments) *Let C be a constraint in $\mathcal{S}$, Γ a type environment and σ a type scheme. Then $C, \Gamma \vdash e : \sigma$ is a* typing judgment *iff $C, \Gamma \vdash e : \sigma$ can be derived by application of the typing rules.*

Example 1 *The Hindley/Milner system is an instance of our type system. Let*

$$1 = (\Omega, \vdash^e, Var, \{\exists\alpha \mid \alpha \in Var\})$$

be the minimal term constraint system. Ω contains only primitive constraints of the form $(\tau = \tau')$ where τ and τ' are types. We define the set $\mathcal{S}$ of constraints in solved form as $\{\mathsf{true}\}$. Then the Hindley/Milner system is equivalent to HM(1).

Now, we give a type soundness theorem based on an ideal semantics [MPS86]. We show that our type system is sound, provided the underlying constraint system is sound. We say a constraint system is *sound* if every satisfiable constraint has a monotype solution:

Definition 9 *A* monotype *is a type τ with $fv(\tau) = \emptyset$.*

We let μ range over monotypes.

Definition 10 *A constraint system $\mathcal{C}$ is* sound *if for all type variables α, constraints $C \in \mathcal{S}$, if $\vdash^e \exists\alpha.C$ then there is a monotype μ such that $\vdash^e \exists\alpha.(\alpha = \mu) \wedge C$.*

The soundness proof is based on an ideal semantics of types which is a direct extension of the semantics in [Mil78].

The meaning of a term is a value in the CPO $\mathcal{V}$, where $\mathcal{V}$ contains at all continuous functions from $\mathcal{V}$ to $\mathcal{V}$ and an error element $\mathbf{W}$, usually pronounced "wrong". Depending on the concrete type system used, $\mathcal{V}$ might contain other elements as well. That is, we only assume that $\mathcal{V}$ satisifes the inequation $\mathcal{V} \supseteq \mathbf{W}_\perp + \mathcal{V} \to \mathcal{V}$.

The meaning function on terms is the same as in the original semantics of Hindley/Milner terms. That is, we assume that any language constructs that make use of type contraints are expressible as predefined values, whose names and types are recorded in the initial type environment.

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

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

$$[\![e\,e']\!]\eta = \mathsf{if}\ [\![e]\!]\eta \in \mathcal{V} \to \mathcal{V} \wedge [\![e']\!]\eta \neq \mathbf{W}\ \mathsf{then}\ ([\![e]\!]\eta)\,([\![e']\!]\eta)\ \mathsf{else}\ \mathbf{W}$$

$$[\![\mathsf{let}\ x = e\ \mathsf{in}\ e']\!]\eta = [\![e']\!]\eta[x := [\![e]\!]\eta]$$

We will show in the following that the meaning of a well-typed program is always different from "wrong".

As a first step, we give a meaning to types. Following [Mil78], we let types denote ideals, i.e. downward-closed and limit-closed subsets of $\mathcal{V}$. The meaning function $[\![\cdot]\!]$ takes closed types and type schemes to an ideals. On function types and type schemes it is defined as follows:

$$\begin{array}{lcl} [\![\mu_1 \to \mu_2]\!] & = & \{f \in \mathcal{V} \to \mathcal{V} \mid v \in [\![\mu_1]\!] \Rightarrow f\,v \in [\![\mu_2]\!]\}. \\ [\![\forall\bar{\alpha}.C \Rightarrow \tau]\!] & = & \bigcap\{[\![[\bar{\mu}/\bar{\alpha}]\tau]\!] \mid \vdash^e [\bar{\mu}/\bar{\alpha}]C\} \end{array}$$

Lemma 5 *Let σ be a closed type scheme. Then $[\![\sigma]\!]$ is an ideal.*

Proof: A straightforward induction on the structure of μ. ∎

Unlike in the treatment of [Mil78], a denotation of a type scheme in HM(X) can contain the error element $\mathbf{W}$. This is the case when the constraint C in a type scheme $\forall\bar{\alpha}.C \Rightarrow \tau$ is unsatifiable. In this case, the denotation of $\forall\bar{\alpha}.C \Rightarrow \tau$ is an empty intersection, that is, all of $\mathcal{V}$. However, we will show that none of the type schemes arising in a type derivation of a judgment $C, \Gamma \vdash e : \sigma$ with a satisfiable final constraint C and a consistent type environment Γ can contain the error element $\mathbf{W}$.

Definition 11 *A closed type environment Γ is* consistent *if for all $x : \sigma \in \Gamma$, $\mathbf{W} \notin [\![\sigma]\!]$. An arbitrary type environment Γ is consistent if $\phi\Gamma$ is consistent for all type variable substitutions ϕ such that $\phi\Gamma$ is closed.*

Definition 12 *A variable environment η* models *a closed typing environment Γ, written $\eta \models \Gamma$, if for all $x : \sigma \in \Gamma$, $\eta(x) \in [\![\sigma]\!]$.*

Theorem 13 (Type Soundness) *Let $C, \Gamma \vdash e : \sigma$ be a valid typing judgment in* HM($\mathcal{C}$)*, where $\mathcal{C}$ is a sound constraint system. Let ϕ be a substitution such that $\phi\Gamma$ and $\phi\sigma$ are closed and such that $\vdash^e \phi C$. Let η be a variable environment such that $\eta \models \phi\Gamma$. Then*

(1) $[\![e]\!]\eta \in [\![\phi\sigma]\!]$
(2) *if Γ is consistent then $\mathbf{W} \notin [\![\phi\sigma]\!]$.*

Proof: A structural induction on typing derivations. There are two interesting cases.

Case (Var) Then the last step of the derivation is:

$$C, \Gamma \vdash x : \sigma \qquad (x : \sigma \in \Gamma)$$

Therefore $x : \phi\sigma \in \phi\Gamma$. Since $\eta \models \phi\Gamma$, $[\![x]\!]\eta = \eta(x) \in [\![\phi\sigma]\!]$. Furthermore, since Γ is consistent, $\mathbf{W} \notin [\![\phi\sigma]\!]$.

Case ($\forall$ Intro) Then the last step of the derivation is:

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

Let ϕ be such that $\phi\Gamma$ and $\phi(\forall\bar{\alpha}.D \Rightarrow \tau)$ are closed and such that $\vdash^e \phi(C \wedge \exists\bar{\alpha}.D)$. Let $\bar{\mu}$ be an arbitrary vector of monotypes such that

$$\vdash^e \exists\bar{\alpha}.(\bar{\alpha} = \bar{\mu}) \wedge \phi D$$

Since $\mathcal{C}$ is sound there is at least one such vector $\bar{\mu}$. Let $\phi' = [\bar{\mu}/\bar{\alpha}] \circ \phi$. Then since $\bar{\alpha} \notin fv(C)$, $\phi'(C \wedge D) = \phi C \wedge \phi' D$, which expands to $\phi C \wedge \exists\bar{\alpha}.(\bar{\mu} = \bar{\alpha}) \wedge \phi D$. By assumption this constraint is valid. Furthermore, $\phi'\Gamma$ and $\phi'\tau$ are both closed. By the induction hypothesis, $[\![e]\!]\eta \in [\![\phi'\tau]\!]$. Since $\bar{\mu}$ was arbitrary such that $\vdash^e [\bar{\mu}/\bar{\alpha}](\phi D)$,

$$\begin{array}{rcl} [\![e]\!]\eta & \in & \bigcap\{[\![[\bar{\mu}/\bar{\alpha}](\phi\tau)]\!] \mid \vdash^e [\bar{\mu}/\bar{\alpha}](\phi D)\} \\ & = & [\![\phi(\forall\bar{\alpha}.D \Rightarrow \tau)]\!]. \end{array}$$

This gives (1). Assume now that Γ is consistent. Again by the induction hypothesis, $\mathbf{W} \notin [\![[\bar{\mu}/\bar{\alpha}](\phi\tau)]\!]$, for any $\bar{\mu}$ such that $\vdash^e [\bar{\mu}/\bar{\alpha}](\phi D)$. Since, furthermore, there is at least one such $\bar{\mu}$, $\mathbf{W} \notin [\![\phi(\forall\bar{\alpha}.D \Rightarrow \tau)]\!]$. ■

As a corollary, we get that Milner's slogan "well types programs do not go wrong" carries over to sound constraint extensions.

Corollary 1 *Let $\mathcal{C}$ be a sound constraint system. Let* $\mathsf{true}, \Gamma \vdash e : \sigma$ *be a valid closed typing judgment in* HM($\mathcal{C}$). *If $\eta \models \Gamma$ then $[\![e]\!]\eta \neq \mathbf{W}$.*

Proof: Immediate from (1) and (2) of Theorem 13. ■

4 Type inference

Type inference in HM(X) is performed in two steps. First, a typing problem is translated into a constraint D in the term constraint system X. Then the constraint D is normalized. Normalizing means computing a substitution ψ and a residual constraint C in X such that ψC entails D where ψC is a constraint in the set $\mathcal{S}$ of constraints in solved form. To ensure that a typing problem has a most general solution, we require that constraints in X have most general normalizers.

4.1 Normalization

In this section we study normalization of constraints in a term constraint system. 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 14 *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. Actually, a typing problem is translated in a constraint in the term constraint system X attached with a substitution. Normalization means then computation of a *principal normal form* of a constraint D from a term constraint system and a substitution ϕ.

Definition 15 (Principal Normal Form) *Let X be a term constraint system over a term algebra $\mathcal{T}$ and $\mathcal{S}$ be the set of solved constraints in X. Let $C \in \mathcal{S}$ and $D \in X$ 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 6 (Uniqueness) *Let (C, ψ) and (C', ψ') be principal normal forms of (D, ϕ). Then there is a variable renaming ϕ' such that $C' = \phi' C$ and $\psi' = \phi' \circ \psi$.*

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

$$\begin{array}{lll} \multicolumn{3}{l}{normalize(D, \phi)} \\ = & (C, \psi) & \text{if } (C, \psi) \text{ principal normal form of } (D, \phi) \\ = & fail & \text{otherwise} \end{array}$$

$$\text{(Ref)} \qquad \frac{\vdash^e (\tau = \tau')}{\vdash^i \tau \preceq \tau}$$

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

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

Figure 2: Instance rules

In the latter, we use the convention that when we say the principal normal form of (D, ϕ) exists we imply that there is a procedure which computes the principal normal form of (D, ϕ).

Actually, a constraint problem consists of a constraint attached with a substitution. Sometimes, it is more convenient to think of a constraint problem simply as a constraint. Both views are consistent if we want to compute the principal normal form of a constraint problem.

Lemma 7 *Given a constraint $D \in X$ and a substitution ϕ. Then $(C, \psi) = normalize(\phi D, id)$ iff $(C, \psi \circ \phi) = normalize(D, \phi)$ where $C \in \mathcal{S}$ and ψ a substitution.*

We now extend the property of having a principal normal form to constraint systems.

Definition 16 (Principal Constraint Property) *Given a term constraint system X over a term algebra $\mathcal{T}$ and a set of solved constraints $\mathcal{S}$ in X. The term constraint system X has the* principal constraint property *if for every constraint $D \in X$ and substitution ϕ, either (D, ϕ) does not have a normal form or (D, ϕ) has a principal normal form.*

We also say that the HM(X) type system has the principal constraint property if X has the principal constraint property. Furthermore, we can conclude that every term constraint system X which enjoys the principal constraint property defines a computable function *normalize*.

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

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

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.

4.2 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 2. 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 except for the $(\forall \preceq)$ rule:

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

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

Lemma 8 (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), (Weaken) and (Trans) rule are derivable.

Lemma 9 *The following inference rules are derivable from those in Figure 2.*

$$\textit{(Ref-}\sigma\textit{)} \qquad C \vdash^i \sigma \preceq \sigma$$

$$\textit{(Weaken)} \qquad \frac{D \vdash^e C \qquad C \vdash^i \sigma \preceq \sigma'}{D \vdash^i \sigma \preceq \sigma'}$$

$$\textit{(Trans)} \qquad \frac{C \vdash^i \sigma_1 \preceq \sigma_2 \qquad C \vdash^i \sigma_2 \preceq \sigma_3}{C \vdash^i \sigma_1 \preceq \sigma_3}$$

The output of a typing problem is a triple consisting of a type scheme , a substitution, and a constraint C. We call such a triple a typing configuration.

Definition 17 *Given a constraint C and a type environment Γ. Then a substitution ϕ is called* consistent *with respect to C and Γ iff $\phi C \in \mathcal{S}$ and for each $x : \forall\bar{\beta}.D \Rightarrow \tau \in \Gamma$ the constraint ϕD is in $\mathcal{S}$ where we assume that there are no name clashes between ϕ and $\bar{\beta}$.*

W.l.o.g. we can assume that each type scheme σ is of the form $\forall\bar{\beta}.D \Rightarrow \tau$.

Definition 18 (Typing Configurations) *Given a type environment Γ, a constraint $C \in \mathcal{S}$, a type scheme σ and a substitution ψ such that $\psi C = C$, $\psi\sigma = \sigma$ and ψ is consistent in Γ. Then (C, σ, ψ) is a* typing configuration *with respect to Γ. For simplicity, we omit sometimes the phrase "with respect to Γ".*

Given typing configurations (C, σ, ψ), (C', σ', ψ') and a type environment Γ. 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 judgments and configurations.

Lemma 10 *Let (C, σ, ψ) and (C', σ', ψ') be typing configurations where $(C, \sigma, \psi) \preceq_\Gamma (C', \sigma', \psi')$, Γ be a type environment and $C, \psi\Gamma \vdash e : \sigma$ be a typing judgment. Then we can derive the typing judgment $C', \phi\Gamma \vdash e : \sigma'$.*

4.3 Type inference algorithm

We now give a generic type inference algorithm for a HM(X) type system. The algorithm can be found in Figure 3. The following definition is a generalization of the ($\forall$ Intro) rule.

Definition 19 *Let C be a constraint, Γ be a type environment, σ be a type scheme and $\bar{\alpha} = (fv(\sigma) \cup fv(C)) \setminus fv(\Gamma)$. Then $gen(C, \Gamma, \sigma) = (\exists\bar{\alpha}.C, \forall\bar{\alpha}.C \Rightarrow \sigma)$.*

In the algorithm 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 environment 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. In an operational reading of the type inference algorithm, it constructs a bottom–up derivation of $\vdash^W$ clauses.

Type inference in HM(X) is performed in two steps. First, the results of the typing problems of the premises are combined. This step is trivial in case of the (Abs) rule. The (Var) rule represents the base case. An initial result is generated. In case of the (App) a unification problem is added to the combined constraints. In all cases, we get a tuple consisting of a constraint and a substitution. A normalization step is performed which results in a tuple consisting of a constraint in solved form and a substitution. This normalization is not necessary in case of the (Abs) rule. As a result, the algorithm reports a triple consisting of a constraint, a type and a substitution.

We can state that the result triple of the type inference algorithm $\vdash^W$ always forms a typing configuration.

Lemma 11 *Given a type environment Γ and a term e. If $\psi, C, \Gamma \vdash^W e : \tau$ then (C, τ, ψ) is a typing configuration.*

Furthermore, this typing configuration always represents a vaild typing of the given term under the given type environment.

Theorem 20 (Soundness) *Given a term e and a type environment Γ. If $\psi, C, \Gamma \vdash^W e : \tau$ then $C, \psi\Gamma \vdash e : \tau$, $\psi C = C$ and $\psi\tau = \tau$.*

To obtain a completeness result for type inference, we assume that we have an HM(X) type system which fulfills the principal constraint property. Furthermore, we consider only those typing judgments $C, \Gamma \vdash e : \sigma$ where the type environment and the constraint on the left hand side of the turnstyle are realizable, i.e. have a type instance.

Definition 21 *Let C' be a constraint and Γ' be a type environment. Then Γ' is* realizable *in C' iff for every $x : \sigma \in \Gamma'$ there is a τ such that $C' \vdash^i \sigma \preceq \tau$.*

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'.

Theorem 22 (Completeness) *Let $C', \phi\Gamma \vdash e : \sigma'$ be a typing judgment and $\phi\Gamma$ is realizable in C'. Then*

$$\psi, C, \Gamma \vdash^W e : \tau$$

for some substitution ψ, constraint C, type τ, such that

$$\begin{array}{rcl} gen(C, \psi\Gamma, \tau) & = & (C_o, \sigma_o) \\ (C_o, \sigma_o, \psi) & \preceq_\Gamma & (C', \sigma', \phi) \end{array}$$

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

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

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

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

Figure 3: Type inference

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.

A sketch of the proofs of soundness and completeness of type inference can be found in the appendix. For a more detailed discussion we refer to [Sul97b].

The completeness theorem can be simplified for top–level programs to the following corollary, which states that our type inference algorithm computes principal types.

Corollary 2 (Principal Types) *Let* $\mathsf{true}, \Gamma \vdash e : \sigma$ *be a closed typing judgment such that* Γ *is realizable in* true. *Then* $\phi, C, \Gamma \vdash^W e : \tau$ *for some substitution* ϕ, *constraint* C, *such that*

$$\begin{array}{rcl}\mathit{gen}(C, \phi\Gamma, \tau) & = & (\mathsf{true}, \sigma_o) \\ \vdash^i \sigma_o & \preceq & \sigma\end{array}$$

Our type inference algorithm interleaves constraint generation and normalization. Each inference rule combines the constraint problems of the premises and performs then a normalization step. This treatment is due to the (Let) rule. We only admit constraints in solved form in a type scheme. Therefore, we have to perform normalization before we do generalization of a type. But it is possible to generate all constraint problems first and perform normalization only right before we reach a (Let) clause or the root clause. The following lemma states that we can delay normalization and perform normalization in any order and we always get the same result.

Lemma 12 *Given constraints* D_1, D_2 *in* X *and substitutions* ϕ_1, ϕ_2. *Then the principal normal form of* $(D_1 \wedge D_2, \phi_1 \sqcup \phi_2)$ *exists iff the principal normal form of* $(C_1 \wedge D_2, \psi_1 \sqcup \phi_2)$ *exists with* (C_1, ψ_1) *principal normal form of* (D_1, ϕ_1) *iff the principal normal form of* $(C_1 \wedge C_2, \psi_1 \sqcup \psi_2)$ *exists with* (C_2, ψ_2) *principal normal form of* (D_2, ϕ_2) . *Especially,*

$$\begin{array}{l}\mathit{normalize}(D_1 \wedge D_2, \phi_1 \sqcup \phi_2) = \\ \mathit{normalize}(C_1 \wedge D_2, \psi_1 \sqcup \phi_2) = \\ \mathit{normalize}(C_1 \wedge C_2, \psi_1 \sqcup \psi_2)\end{array}$$

where

$$\begin{array}{rcl}\mathit{normalize}(D_1, \phi_1) & = & (C_1, \psi_1) \\ \mathit{normalize}(D_2, \phi_2) & = & (C_2, \psi_2)\end{array}$$

Hence, if our type inference algorithm only applies clauses (Var), (Equ), (Abs) and (App) it is equivalent to accumulate all typing problems first and perform normalization at the end. This can be proven by a simple

induction on the number of inference steps and with the help of Lemma 12.

5 A record application HM($\mathcal{R}$)

Following ideas of Ohori [Oho95] we give an instance of our HM(X) system which deals with polymorphic records. First, we give an instance $\mathcal{R}$ of a term constraint system $\mathcal{TCS}_{\mathcal{T}}$. It must now be able to deal with constraints on records. We also add primitive operations to the initial type environment Γ_0 that deal with record selection and record update. Furthermore, we discuss the relationship of HM($\mathcal{R}$) to Ohori's calculus (In the latter we use $\mathcal{O}$ as an abbreviation for Ohori's calculus). Finally, we show that $\mathcal{R}$ enjoys the principal constraint property. That means, we get a type inference algorithm for HM($\mathcal{R}$) which computes principal types.

In addition to types and function types we have now record types denoted by $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ where l_i is an element of a enumerable set of record labels. We extend the term algebra $\mathcal{T}$ by adding constructors

$$l_1\_\ldots\_l_n : \tau_1 \to \ldots \to \tau_n \to \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

to the signature Σ. We assume that the order of the fields does not matter. Additionally, we require

D7 $\vdash^e (\{l_1 : \tau_1, \ldots, l_n : \tau_n\} = \{l_{\pi(1)} : \tau_{\pi(1)}, \ldots, l_{\pi(n)} : \tau_{\pi(n)}\})$
where π is a permutation on $\{1, \ldots, n\}$

Note, here you can see an example for a type constructor with non–trivial equality relation.

In [Oho95] kinded types are introduced. $< l : \tau >$ is a record kind intuitively denoting all records which contain at least a field label l with value τ. Constraints on record types are now expressed by kinded types. That means in our constraint language we have now constraints of the form $(\tau :: k)$ where τ is a type and k is a kind. Technically, this means we add $(\tau :: k)$ to the set Ω of primitive constraints where $(::)$ is a primitive predicate of arity 2. Also, we require for the term constraint system $\mathcal{R}$ the following additional rules:

R1 $\vdash^e (\{l_1 : \tau_1, \ldots l_n : \tau_n\} ::< l_i : \tau_i >)$
where $l_1, \ldots, l_n$ are distinct
R2 $(\tau ::< l : \tau_1 >) \wedge (\tau ::< l : \tau_2 >) \vdash^e (\tau_1 = \tau_2)$
R3 $(\{\ldots, l : \tau_1, \ldots\} ::< l : \tau_2 >) \vdash^e (\tau_1 = \tau_2)$
R4 $(\{l_1 : \tau_1, \ldots, l_n : \tau_n\} ::< l : \tau >) \vdash^e \mathsf{false}$
where l is distinct from $l_1, \ldots, l_n$
R5 $(\tau_1 \to \tau_2 ::< l : \tau_3 >) \vdash^e \mathsf{false}$
E5 $\exists\alpha.(\alpha :: k) =^e \mathsf{true}$
where $\alpha \notin fv(k)$
E6 $\exists\alpha.(\alpha :: k) =^e \mathsf{false}$
where $\alpha \in fv(k)$
F1 $\mathsf{false} \vdash^e P$ for $P \in \Omega$
F2 $\exists\alpha.\mathsf{false} =^e \mathsf{false}$

Remark 4 *Conditions* **R1** – **R5** *are a straightforward extension to constraints on records. Conditions* **E5** – **E6** *define how projection operates on record constraints.* **E5** *and* **E6** *express the fact that no recursive records are allowed. A token* false *is used. It represents the false element of the constraint system. An axiomatization is given in* **F1** *and* **F2**. *The token* false *is assumed to be contained in the set* Ω *of primitive constraints.*

It is also important to point out that we only have width subtyping because we do not have the subsumption rule in the HM(X) type system. Furthermore, we also forbid overloading of field labels. This is along the lines as in [Oho95].

It remains to define the set $\mathcal{S}$ of constraints in solved form. First, we require that all constraints $(::)$ in $\mathcal{S}$ are of the form $(\alpha :: \_)$.

We put additionally the following condition on $\mathcal{S}$:

S5 If $C \in \mathcal{S}$ then there is an ordering $<$ on the type variables in C such that for all predicates $(\alpha ::< l : \tau >)$ and $\beta \in fv(\tau)$ with $C \vdash^e (\alpha ::< l : \tau >)$ we have that $\alpha < \beta$

Then we define $\mathcal{S}$ as the greatest set of constraints which fulfills conditions **S1** – **S5**.

Remark 5 *Condition* **S5** *simply states that no recursive records are allowed in HM($\mathcal{R}$). Recursive records are also not allowed in [Oho95].*

We now add primitive constructs to the initial type environment Γ_0 that deal with record formation, selection and update. For every constructor $l_1\_\ldots\_l_n$ we have a datatype constructor

$$l_1\_\ldots\_l_n : \tau_1 \to \ldots \to \tau_n \to \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

in the initial type environment Γ_0. $l_1\_\ldots\_l_n$ allows us formation of records $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$. For each field label l we add

$$.l : \forall\alpha, \beta.(\alpha ::< l : \beta >) \Rightarrow \alpha \to \beta$$

and

$$modify_l : \forall\alpha,\beta.(\alpha ::< l : \beta >) \Rightarrow \alpha \to \beta \to \alpha$$

to the initial type environment Γ_0. The first corresponds to record selection, the latter to record update.

The question arises how HM($\mathcal{R}$) is related to $\mathcal{O}$. In HM($\mathcal{R}$) we have the same basic operations as in $\mathcal{O}$. But our constraint system differs in two respects from the one used in $\mathcal{O}$. We have an equality predicate (=) and a projection operator $\exists\bar{\alpha}$. The equality predicate becomes important when we consider type inference which we will do in the latter. We have already pointed out that the projection operator allows us to formulate a logically pleasing ($\forall$ Intro) rule. Essentially, the only difference between HM($\mathcal{R}$) and $\mathcal{O}$ lies in the introduction of quantified type variables. In $\mathcal{O}$ we have the following ($\forall$ Intro) rule:

$$(\forall \text{ Intro-}\mathcal{O}) \quad \frac{\mathcal{K}\{\alpha \mapsto k\}, \Gamma \vdash e : \sigma \qquad \alpha \notin fv(\Gamma)}{\mathcal{K}, \Gamma \vdash e : \forall\alpha.(\alpha :: k) \Rightarrow \sigma}$$

Ohori uses instead of a constraint on the left hand side of a typing judgment a so-called *kind assignment* $\mathcal{K}$ which can be considered as a function from type variables to kinds. He writes $\mathcal{K}\{\alpha \mapsto k\}$ for the disjoint extension of $\mathcal{K}$ with a new type variable α with kind k. We can now write the following program in $\mathcal{O}$ where we assume that we have a funciton

$$\mathsf{eq} : \forall\alpha.\alpha \to \alpha \to \mathsf{Bool}$$

in an initial type environment and an operator $(\_,\_,\_)$ for formation of triples. We use a Haskell-style notation, adding type annotations for illustration purposes.

Example 2

```
f: ∀α.(α ::< l1 : β >) ⇒ α → Int
f x =
    let g: ∀β.(β ::< l2 : τ >) ⇒ β → Int
        g = λ y. (x.l1, (x.l1).l2, eq y (x.l1))
    in 1
```

The type of f looks strange. The function f is closed but the type of f contains the free type variable β. We can not quantify β at the outermost level otherwise we would get a conflict with the conditions put on a kind assignment. Furthermore, β should be a record type but there is no such restriction put on β. Actually, this program is non–sensical. We can type f but we can not apply f to an argument. The reason is that β does not appear anymore in the kind assignment $\mathcal{K}$. And this means we can not find an instance for the constraint in f's type.

In HM($\mathcal{R}$) we can type f in the following way.

Example 3

```
f: ∀α.∃β.(α ::< l1 : β >) ∧ (β ::< l2 : τ >) ⇒ α → Int
f x =
  let g: ∀β.(α ::< l1 : β >) ∧ (β ::< l2 : τ >) ⇒ β → Int
      g = λ y. (x.l1, (x.l1).l2, eq y (x.l1))
  in 1
```

That means, HM($\mathcal{R}$) can be seen as the proper logical formulation of $\mathcal{O}$. Especially, we get that HM($\mathcal{R}$) models $\mathcal{O}$ faithful.

Theorem 23 *Every program typable in $\mathcal{O}$ is also typable in HM($\mathcal{R}$).*

But we can type programs in HM($\mathcal{R}$) which are not typable in $\mathcal{O}$. In the above example we can apply f to an appropriate argument which is not possible in $\mathcal{O}$. In Section 9 we discuss some more examples for different formulations of the ($\forall$ Intro) rule and problems which then arise.

We now consider type inference for HM($\mathcal{R}$). That means we have to show that $\mathcal{R}$ fulfills the principal constraint property. We have already mentioned that our constraint system differs from the one used in $\mathcal{O}$. We have an equality predicate (=) and a projection operator $\exists\bar{\alpha}$. We do the following steps to prove that the principal constraint property holds for $\mathcal{R}$. We show that it is always possible to split a constraint in a projection free subpart. Then we give a procedure which computes the principal normal form of projection free constraints, or no normal form exists at all. Finally, we show that it is sufficient to compute principal normal forms of projection free constraints.

We now want to split a constraint into a projection free subpart. The idea is that we can always rename type variables which are bound by the projcetion operator. It holds that

$$\exists\alpha.C \;=^e\; \exists\beta.[\beta/\alpha]C$$

where β is a new type variable. That means, w.l.o.g. there are no name clashes between two projected constraints

$$(\exists\alpha.C) \wedge (\exists\beta.D)$$

Then we can use condition **E3** to shift all projection operators to the outermost level. We can summarize this observation in the following lemma.

Lemma 13 *Let $C \in \mathcal{R}$. Then there exists a projection free constraint D such that $C \;=^e\; \exists\bar{\alpha}.D$.*

We assume now that we have a projection free constraint D which contains only primitive predicates of the form (=) and (::). Furthermore, we can assume

that all predicates (::) are of the form $(\alpha :: k)$ because we know that

$$(\tau :: k) =^e \exists\alpha.(\alpha = \tau) \wedge (\alpha :: k)$$

where α is a new type variable. The closure $Cl(D)$ of D is the smallest constraint which fulfills the following conditions:

1. $D \subseteq Cl(D)$
2. If $(\alpha = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}) \in Cl(D)$
   then $(\alpha ::< l_1 : \tau_1 >), \ldots, (\alpha ::< l_n : \tau_n >) \in Cl(D)$
3. If $(\alpha ::< l : \tau_1 >), (\alpha ::< l : \tau_2 >) \in Cl(D)$
   then $(\tau_1 = \tau_2) \in Cl(D)$

From a semantic view point we have not done anything because $Cl(D) =^e D$. We simply have changed the syntactic representation of D. The intention of building the closure of D is simply to generate all predicates $(\tau ::< l : \tau' >)$ which might cause any inconsistencies. From that we can generate all unification problems $(\tau = \tau')$ which have to be resolved. The following lemma states that we really have generated all such predicates.

Lemma 14 *Given a field label l and types τ, τ'. If $\not\vdash^e (\tau ::< l : \tau' >)$ then $(\tau ::< l : \tau' >) \in Cl(D)$ iff $D \vdash^e (\tau ::< l : \tau' >)$. Furthermore, if $\not\vdash^e (\tau = \tau')$ then $(\tau = \tau') \in Cl(D)$ iff $D \vdash^e (\tau = \tau')$.*

Then we can apply unification over Herbrand terms [Rob65]. to resolve all equality predicates (=) in $Cl(D)$. Actually, that is not the whole truth. Remember, we have a record type constructor with non–trivial equality relation in the term algebra $\mathcal{T}$. That means, when we perform unification we have to take into account that records are equal modulo up to the order of the fields. We get a most general unifier ϕ of the equality predicates (=) in $Cl(D)$. It remains to check whether this most general unifier ϕ is consistent with $Cl(D)$. That means, we check whether there are any inconsistencies in $\phi Cl(D)$. If not, $(\phi Cl(D), \phi)$ represents the principal normal form of (D, id). We can summarize this observation in the following lemma.

Lemma 15 *Given a projection free constraint $D \in \mathcal{R}$ and a substitution ϕ. Then (D, ϕ) does have a principal normal form or no normal form exists.*

We have given now a procedure to compute the principal normal form of projection free constraints. The next lemma gives us a procedure to lift principal normal forms of constraints to arbitrary constraints. It states that whenever we can compute the principal normal form of a constraint D then we get the principal normal form of the constraint $\exists\alpha.D$ for free.

Lemma 16 *Let $D \in \mathcal{R}$ and ϕ be a substitution where $\alpha \notin codom(\phi) \cup dom(\phi)$. If $(C, \psi) = normalize(D, \phi)$ then $(\exists\alpha.C, \psi_{\setminus\{\alpha\}}) = normalize(\exists\alpha.D, \phi)$.*

The next lemma states that a normal form of a constraint exists if a normal form of the projected constraint exist.

Lemma 17 *Given a substitution ϕ where $\alpha \notin codom(\phi) \cup dom(\phi)$ and a constraint $D \in \mathcal{R}$. Then (D, ϕ) has a normal form if $(\exists\alpha.D, \phi)$ has a normal form.*

We have now everything at hand to prove that $\mathcal{R}$ has the principal constraint property.

Theorem 24 *The constraint system $\mathcal{R}$ has the principal constraint property.*

Proof: Assume the contrary. That means we have a tuple (D, ϕ) which has a normal form but no principal normal form. With Lemma 13 we know that $D =^e \exists\bar{\alpha}.D'$ where D' is a projection free constraint. Then we apply Lemma 17 and get that the normal form of (D', ϕ) exists. From Lemma 15 we know that (D', ϕ) does have a principal normal form. With Lemma 16 we can lift the principal normal form and get that the principal normal form of (D, ϕ) exists. But this is a contradiction to our assumption. We get that $\mathcal{R}$ has the principal constraint property. ■

That means, we know that the constraint system $\mathcal{R}$ has the principal constraint property and we have a decidable procedure at hand to compute the principal normal form.

We have given an instance HM($\mathcal{R}$) of our HM(X) system which deals with records. We extended the term algebra $\mathcal{T}$ and had to provide a constraint system $\mathcal{R}$ which is able to deal with records. We showed that $\mathcal{R}$ satisfies the principal constraint property.

We omit the discussion of further extensions like record extension or recursive records. In [Sul97a] we discuss an instance of our HM(X) framework that deals with recursive records, extension and concatenation of records. We believe that this instance provides a good wrapper for object calculi.

6 Getting a HM(X) instance for free

Whenever we want to design a new instance of our HM(X) framework we have to give a term constraint system which enjoys the principal constraint property. Very often we can rely on already existing approaches which also use a kind of constraint system and a procedure to normalize constraints. For example Ohori[Oho95] introduced a constraint system where types were associated with kinds. His normalization procedure is a

form of kinded unification. The main difference between such approaches and ours is that we have a projection operator. It would be a nice property if we could *lift* already existing constraint systems and normalization procedures to get an instance of our HM(X) system. For instance, in case of HM($\mathcal{R}$) we adapted an already existing constraint system. We added rules how projection operates on constraints. Then, we gave a procedure which computes the principal normal form of projection free constraint. Finally, we showed how to lift this procedure to arbitrary constraints.

It is possible to generalize this result for a specific class of constraint systems.

Definition 25 *Given a term constraint system $\mathcal{TCS}_\mathcal{T}$. We say that $\mathcal{TCS}_\mathcal{T}$ satisfies the* lifting *conditions iff the following conditions hold:*

- *(L1) Every $D \in \mathcal{TCS}_\mathcal{T}$ has a normal form as described in Lemma 13*
- *(L2) Every projection free constraint $D \in \mathcal{TCS}_\mathcal{T}$ has a principal normal form*
- *(L3) If $\exists\alpha.D \in \mathcal{S}$ and $D \notin \mathcal{S}$ then there exists a τ such that, $\exists\alpha.D =^e [\tau/\alpha]D$*

Theorem 26 (Lifting) *Let $\mathcal{TCS}_\mathcal{T}$ be a term constraint system, such that the lifting condition holds Then $\mathcal{TCS}_\mathcal{T}$ has the principal constraint property.*

7 An extension SHM(X) with subtypes

We now consider an extension SHM(X) of our HM(X) framework that deals with subtypes. We enrich the constraint system X with subtype constraints and we add the subsumption rule to the logical type system. First, we introduce a constraint system that is able to express subtyping.

Definition 27 (Subtype Constraint System) *A* subtype constraint system $\mathcal{SC}$ *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 $\mathcal{SC}$, which satisfies:*

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

S2 $$\frac{D \vdash^e (\alpha_1' \leq \alpha_1) \quad D \vdash^e (\alpha_2 \leq \alpha_2')}{D \vdash^e (\alpha_1 \to \alpha_2 \leq \alpha_1' \to \alpha_2')}$$

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

The set of solved constraints $\mathcal{S}$ in $\mathcal{SC}$ is equal to the set of satisfiable constraints in $\mathcal{SC}$.

Now, we introduce an extension SHM(X) of the Hindley/Milner system with a subtype constraint system X over a term algebra $\mathcal{T}$. The subsumption rule is added to the logical type system in Figure 1:

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

The relation $\vdash^s$ obeys the following rules:

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

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

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

As for $\vdash^i$, we get that the (Refl-σ), (Weaken) and (Trans) rules hold. That means, Lemma 9 holds also in this case. Additionally, we have the ($\to$ Intro) rule.

Lemma 18 ($\to$ 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 $C \vdash^s \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$.*

It is interesting to point out that rules ($\forall$ Elim) and (Equ) are now entailed in the (Sub) rule. We can leave them out from the typing rules for SHM(X) type systems.

In this case, computation of the principal normal form represents simply a satisfiability test. We get that:

$$\begin{array}{lll} & \mathit{normalize}(D, \phi) & \\ = & (\phi D, \phi) & \text{if } \phi D \text{ satisfiable} \\ = & \mathit{fail} & \text{otherwise} \end{array}$$

We can show that for all subtype constraint systems X the test whether a constraint is satisfiable or not is decidable. We adopt the approach by Aiken/Wimmers [AW93]. They use an ideal model [MPS86] to show that each type variable of a satisfiable constraint can be assigned to an ideal in the ideal model. It is decidable to test whether each type variable in a constraint has a solution in the ideal model. It is important to note that we are not interested in the *best* solution. We only need a decidable satisfiability test. In case of simplification, finding the best solutions becomes an important issue. We can conclude that the principal constraint property holds for SHM(X) type systems.

The type inference algorithm for SHM(X) type systems is the same except for the (App) rule. Instead

of the equality predicate (=) we have now the subtype predicate ($\leq$):

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

We get the same results as already stated in Section 4. Furthermore, we get that type inference always reports the *id* substitution. That is true for the (Var) rule. And whenever we have to compute a normalization we know that $\mathit{normalize}(D, id) = (D, id)$. Therefore, the other inference rules report also the *id* substitution. Consider the following example.

Example 4

$$\begin{array}{c} id, (\alpha \leq \beta \to \gamma) \wedge (\alpha \leq \gamma \to \delta), \emptyset \\ \vdash^W \\ \lambda f.\lambda x.f(fx) : \alpha \to \beta \to \delta. \end{array}$$

Related Work:
SHM(X) is most closely related to the approach 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.

8 Simplification

A current trend in research for type inference with constrained types is to simplify constraints, see [Pot96, EST95a]. Simplification deals with the problem that type inference introduces many intermediate variables which make constraints large and hard to understand for the programmer. Simplification hides these intermediate variables. This corresponds exactly to the role of projection in our framework.

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 19 *Let $C, \Gamma \vdash e : \sigma$ be a typing judgment 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] developed 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 judgment $C, \Gamma \vdash e : \sigma$ 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 judgment. We can state the following substitution rule which follows immediately from Lemma 10:

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

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 judgment of the conclusion should be as least as general as the typing judgment of the premise.

Our framework thus 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. Existing simplification techniques [Pot96, EST95a] can be integrated into our framework as was sketched above.

9 Related work

All constrained type systems we study extend the type judgements $\Gamma \vdash e : \sigma$ of the Hindley/Milner system

with a constraint hypothesis on the left hand side of the turnstyle, written $C, \Gamma \vdash e : \sigma$. 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:

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

The rule is a refinement of the corresponding rule in the Hindley/Milner system. It says that, when instantiating a type scheme $\forall \bar{\alpha}.D \Rightarrow \tau'$, the only valid instances are those instances $[\bar{\tau}/\bar{\alpha}]\tau'$ which satisfy the constraint part D of the type scheme.

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 4 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. Even though these details matter for each particular type system, we have to abstract here from them in order to concentrate on general principles. 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 parametrized 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 5

```
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 5 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 5 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 5.

Example 6

```
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.

Nevertheless, it might be possible to integrate Jones' approach into our HM(X) framework, thus giving it a semantic basis independent of dictionary passing. The essential idea is that we have to restrict ourselves to constraint systems in which projections of solved constraints are trivial, i.e $\vdash^e \exists \alpha.C$, for all constraints $C \in \mathcal{S}$, type variables $\alpha \in fv(C)$. The term constraint system $\mathcal{TCS}_\mathcal{T}$ on the other hand, can have non–trivial projections. In this case, our rule (∀ Intro) simplifies to (∀ Intro-1).

Note, that trivial projections correspond well to Haskell's "open world" assumption, which says that the range of possible instance types for an overloaded operation is not fixed in advance. Therefore, we can never rule out that a given constraint which still has free variables might have a solution. A formalization of this principle using a "bottom type" [OWW95] makes it possible to define a compositional semantics for Haskell–style overloading.

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

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

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

Strong satisfiability check[Smi91]:
$$\frac{C \wedge D, \Gamma \vdash e : \tau \qquad C \vdash [\bar{\tau}/\bar{\alpha}]D \qquad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau} \quad (\forall \text{ Intro-3})$$

Duplication[EST95b]:
$$\frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C \wedge D, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow \tau} \quad (\forall \text{ Intro-4})$$

Figure 4: Versions of the quantifier introduction rule

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 7

```
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

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

On the other hand, if we substitute the actual parameter true in f's definition, we get again Example 6 which is not typable under the system with (∀ Intro-2). Hence, the system with (∀ 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. In a later version, they use rule (∀ Intro-4) instead.

Where Aiken and Wimmers require only a weak form of satisfiability for traded constraints, G. Smith requires a strong one [Smi91]. In rule (∀ 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, (∀ Intro-3) rule 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 two instances:

```
Int ≤ {less: Int → Bool}
Char ≤ {less: Char → Bool}
```

Now consider the following program:

Example 8

```
let
    f: ∀β.β → Int
    f x =
        let g y = y.less(x)
        in 1
in 1
```

When typing the definition of g, Smith's system requires a solution of the constraint $\tau \leq \mathsf{Less}; \tau$, where τ is y's type. Two solutions exist: $\tau = \mathsf{Int}$ or $\tau = \mathsf{Char}$, there is no best type for y that improves on both solutions.

The system of the Hopkins Objects Group [EST95b] differs from the previous three systems in that in rule (∀ 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 (∀ 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. HM(X) can be seen as the proper logical formulation of their more algorithmically–formulated type system. Furthermore, instead of dealing exclusively with subtype constraints, we admit arbitrary constraint systems. Finally, we have

already seen that our use of projection gives a semantic justification to the simplification techniques they study [EST95a].

10 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 given an instance of our HM(X) framework that deals with polymorphic records. Based on the record system of Ohori, we have shown how to encode his system in terms of our HM(X) framework. The critical part was to lift existence and computation of kinded unification to arbitrary constraints. By the Lifting Theorem 26 it is not difficult to encode also further systems. Furthermore, we discussed an extension SHM(X) of HM(X) where we additionally have subtype constraints. We added the subsumption rule to our logical type system. We could show that the principal constraint property holds in general for SHMX(X) type systems. That means, type inference for SHM(X) type systems always computes principal types.

Our HM(X) framework provides a foundation to experiment with new constraint domains in a systematic way. It remains a topic for the future to consider further applications.

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 Schuet, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language (extended abstract). In *Procceding of ECOOP*, pages 27–51. Springer Verlag, 1995. LNCS 952.

[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.

[CHO92] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proc. of Lisp and F.P.*, pages 170–191. ACM Press, June 1992.

[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 binding-time analysis in polynomial time. In *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] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA*, 1995.

[EST95b] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursivly constrainted types and its application to object oriented programming. In *Electronic Notes in Theoretical Computer Science*, volume 1, 1995.

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

[JM94] Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *Journal of Logic 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 *FPCA '95: Conference on Functional Programming Languages and Computer Architecture.* ACM Press, 1995.

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

[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. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 409–418. ACM Press, January 1993.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM TOPLAS*, 6(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. *Information and Computation*, 123(2):198–209, 1995.

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

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. pages 77–88. ACM, January 1989.

[Rém92] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institute National de Recherche en Informatique et en Automatique, 1992.

[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.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

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

[Smi91] Geoffrey S. Smith. *Polymorphic type inference for languages with overloading and subtyping*. PhD thesis, Cornell University, Ithaca, NY, August 1991.

[Sul97a] Martin Sulzmann. Designing Record Systems. Research Report YALEU/DCS/RR-1128, Yale University, Department of Computer Science, April 1997.

[Sul97b] Martin Sulzmann. Proofs of Soundness and Completeness of Type Inference for HM(X). Research Report YALEU/DCS/RR-1102, Yale University, Department of Computer Science, February 1997.

[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.

[VHJW96] Cordelia V.Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Waderler. Type classes in haskell. *ACM TOPLAS*, 18(2):109–138, March 1996.

A Proof of Theorem 20 (Soundness)

One thing about which we have to be aware is when we apply a substitution to a typing judgment

$$C, \Gamma \vdash e : \sigma$$

Because we only allow constraints in solved form (constraints in $\mathcal{S}$) in our type system, we have to be sure that a substitution does not cause any inconsistencies. In the type inference algorithm normalization ensures that there are no inconsistencies.

The following two lemmas can both be proven by a straightforward induction on the derivation $\vdash$.

Lemma 20 *Given $C, \Gamma \vdash e : \sigma$ and a substitution ϕ such that ϕ is consistent with respect to C and Γ. Then $\phi C, \phi\Gamma \vdash e : \phi\sigma$.*

Lemma 21 *Given $C, \Gamma \vdash e : \sigma$ and a constraint $D \in \mathcal{S}$ such that $D \vdash^e C$. Then $D, \Gamma \vdash e : \sigma$.*

We restate Theorem 20 in the following lemma.

Lemma 22 (Soundness of $\vdash^W$) *Given a type environment Γ and a term e. If $\psi, C, \Gamma \vdash^W e : \tau$ then $C, \psi\Gamma \vdash e : \tau$, $\psi C = C$ and $\psi\tau = \tau$.*

Proof: We apply induction on the derivation $\vdash^W$. We only consider one case. The other cases can be proven in a similar style.

Case (App)

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

We apply the induction hypothesis to the left and right premise and get

$$C_1, \psi_1\Gamma \vdash e_1 : \tau_1 \quad \psi_1 C_1 = C_1 \quad \psi_1\tau_1 = \tau_1$$

and

$$C_2, \psi_2\Gamma \vdash e_2 : \tau_2 \quad \psi_2 C_2 = C_2 \quad \psi_2\tau_2 = \tau_2$$

With Lemma 21 we can conclude that

$$C, \psi_1\Gamma \vdash e_1 : \tau_1 \quad C, \psi_2\Gamma \vdash e_2 : \tau_2$$

W.l.o.g. we can assume that all identifier in Γ are contained in e_1 and e_2 and not more. This fact and normalization ensures that ψ is consistent in C and Γ. Then we can apply Lemma 20 and obtain

$$C, \psi\Gamma \vdash e_1 : \psi\tau_1 \quad C, \psi\Gamma \vdash e_2 : \psi\tau_2$$

We also now that $\vdash^e (\psi\tau_2 = \psi\tau_1 \to \psi(\alpha))$ because the constraint $(\tau_1 = \tau_2 \to \alpha)$ has been resolved by normalization. Hence, we can first apply the (Equ) rule and then the (App) rule and get

$$C, \psi\Gamma \vdash e_1 e_2 : \psi(\alpha)$$

■

B Proof of Theorem 22 (Completeness)

In order to prove completeness we have to do a little more work. The idea is to introduce two intermediate derivations. But the last derivation looks much more like a type inference algorithm. We then show that all derivations have the same expressive power.

First, we introduce some basic lemmas. We state them without proof. A detailed discussion can be found in [Sul97b]. The first lemma states that we can lift entailment between two constraints to the generalized constraints.

Lemma 23 *Given type environments Γ, Γ', constraints $C, \tilde{C}$, types τ, τ' and substitutions ϕ, ϕ', ψ such that $\vdash^i \phi\Gamma \preceq \Gamma'$, $C \vdash^e \phi'\tilde{C}$ and $\psi \leq^{\phi'}_{fv(\Gamma)} \phi$. Then $C_o \vdash^e \phi'\tilde{C}_o$ where $C_o = gen_2(C, \Gamma, \tau')$ and $\tilde{C}_o = gen_2(\tilde{C}, \psi\Gamma, \tau)$.*

The next lemma is similar to the previous one. It is simply formulated for types.

Lemma 24 *Given type environments Γ, Γ', constraints $C, \tilde{C}$, types τ, τ' and substitutions ϕ, ϕ', ψ such that $\vdash^i \phi\Gamma \preceq \Gamma'$, $C \vdash^e \phi'\tilde{C}$, $C \vdash^i \phi'\tau \preceq \tau'$ and $\psi \leq^{\phi'}_{fv(\Gamma)} \phi$. Then $\vdash^i \phi'\tilde{\sigma}_o \preceq \sigma_o$ where $\sigma_o = gen_1(C, \Gamma, \tau')$ and $\tilde{\sigma}_o = gen_1(\tilde{C}, \psi\Gamma, \tau)$.*

By the next Lemma we lift some properties about constraints and substitution to the same constraints but extended substitution.

Lemma 25 *Given constraints C_1, C' and substitutions $\psi, \psi_1, \ldots, \psi_n, \phi, \phi_1$ such that $\psi_1 C_1 = C_1$, $C' \vdash^e \phi'_1 C_1$, $\psi = \psi_1 \sqcup \ldots \sqcup \psi_n$, $\psi \leq^{\phi'}_{fv(\Gamma)} \phi$, $\psi_1 \leq^{\phi'_1}_{fv(\Gamma)} \phi$. Then $C' \vdash^e (\phi' \circ \psi)C_1$.*

The next Lemma is similar to the previous one but it is stated for the $\vdash^i$ relation.

Lemma 26 *Given a constraint C', type schemes $\tilde{\sigma}, \sigma''$ and substitutions $\psi, \psi_1, \ldots, \psi_n, \phi, \phi_1$ such that $\psi_1\tilde{\sigma} = \tilde{\sigma}$, $C' \vdash^i \phi'_1\tilde{\sigma} \preceq \sigma''$, $\psi = \psi_1 \sqcup \ldots \sqcup \psi_n$, $\psi \leq^{\phi'}_{fv(\Gamma)} \phi$, $\psi_1 \leq^{\phi'_1}_{fv(\Gamma)} \phi$. Then $C' \vdash^i (\phi' \circ \psi)\tilde{\sigma} \preceq \sigma''$.*

Now, we introduce the intermediate derivations. We introduce a derivation $\vdash^2$ which is based on derivation $\vdash$ in figure 1. Instead of rule ($\forall$ Elim) we have the following new (Inst) rule in $\vdash^2$:

$$\text{(Inst)} \quad C, \Gamma \vdash^2 x : \tau \quad (x : \sigma \in \Gamma \quad C \vdash^i \sigma \preceq \tau)$$

All other rules stay unchanged. The idea of derivation $\vdash^2$ is simply to enforce ($\forall$ Elim) steps as early as possible.

Next, we consider a syntax directed derivation $\vdash^s$. We also want to get rid of the ($\forall$ Intro) rule. This rule

is combined with the (Let) rule and the rest of the rules remain unchanged:

$$\text{(Var)} \quad C, \Gamma \vdash^s x : \tau \quad (x : \sigma \in \Gamma \quad C \vdash^i \sigma \preceq \tau)$$

$$\text{(Abs)} \quad \frac{C, \Gamma_x.x : \tau \vdash^s e : \tau'}{C, \Gamma_x \vdash^s \lambda x.e : \tau \to \tau'}$$

$$\text{(App)} \quad \frac{C, \Gamma \vdash^s e_1 : \tau_1 \to \tau_2 \quad C, \Gamma \vdash^s e_2 : \tau_1}{C, \Gamma \vdash^s e_1 e_2 : \tau_2}$$

$$\text{(Let)} \quad \frac{\begin{array}{c} C, \Gamma_x \vdash^s e : \tau \quad (C', \sigma) = gen(C, \Gamma_x, \tau) \\ C'', \Gamma_x.x : \sigma \vdash^s e' : \tau' \end{array}}{C' \wedge C'', \Gamma_x \vdash^s \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau'}$$

Note, in the (Let) we implicitely state that the constraint $C' \wedge C''$ is in solved form. Remember that the set of constraints of solved forms is not necessarily closed under $\wedge$ in general. That means, when we apply the (Let) we always have to ensure that $C' \wedge C''$ is in solved form.

How these derivations are connected is stated in the next lemmas. The first two lemmas can both be proven by a straightforward induction on the derivation relation.

Lemma 27 (Equivalence of $\vdash$ and $\vdash^2$) *Given a type environment Γ, a constraint C, a term e and a type scheme σ. Then $C, \Gamma \vdash e : \sigma$ iff $C, \Gamma \vdash^2 e : \sigma$.*

Lemma 28 (Soundness of $\vdash^s$) *Given $C, \Gamma \vdash^s e : \tau$. Then $C, \Gamma \vdash e : \tau$.*

We now show that $\vdash^s$ is complete with respect to $\vdash^2$ and $\vdash^W$ is complete with respect to $\vdash^2$. In order to prove it we have to strengthen the assumption about the given type environment. This is due to the (Let) rule where the two premises use different type environments. Therefore, we introduce the following definition.

Definition 28 *Let C be a constraint and Γ and Γ' be type environments such that $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ and $\Gamma' = \{x_1 : \sigma'_1, \ldots, x_n : \sigma'_n\}$. Then $C \vdash^i \Gamma' \preceq \Gamma$ iff $C \vdash^i \sigma'_i \preceq \sigma_i \quad \forall i : i \in \{1, \ldots, n\}$.*

Lemma 29 (Completeness of $\vdash^s$) *Given $C', \Gamma' \vdash^2 e : \sigma'$, $C' \vdash^i \Gamma \preceq \Gamma'$ and Γ' is realizable in C'. Then*

(a) $\sigma' = \tau$: $\quad C, \Gamma \vdash^s e : \tau \quad C' \vdash^e C$

(b) otherwise : $\quad C, \Gamma \vdash^s e : \tau \quad (\sigma_o, C_o) = gen(C, \Gamma, \tau)$
$C' \vdash^e C_o \quad C' \vdash^i \sigma_o \preceq \sigma$

Proof:

We use induction on the derivation $\vdash^s$. Due to space limitation we only show one case.

Case (Let) We have the following situation:

$$\frac{C', \Gamma'_x \vdash^2 e : \sigma \quad C', \Gamma'_x.x : \sigma \vdash^2 e' : \tau'}{C', \Gamma'_x \vdash^2 \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau'}$$

First, we consider the case if σ is a type τ. We apply the induction hypothesis to left premise and get $C_1, \Gamma_x \vdash^s e : \tau$ and $C' \vdash^e C_1$. We set $(\sigma_o, C_o) = gen(C_1, \Gamma_x, \tau)$. It is an easy observation that $C' \vdash^i \sigma_o \preceq \tau$ holds. Now, we apply the induction hypothesis to the right premise. This yields $C_2, \Gamma_x.x : \sigma_o \vdash^s e' : \tau'$ and $C' \vdash^e C_2$. We know that $C' \vdash^e C_o \wedge C_2$ which ensures that $C_o \wedge C_2$ is in solved form. We can apply the (Let) rule and obtain $C_o \wedge C_2, \Gamma_x \vdash^s \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau'$.

Now, let us consider the case if σ is a type scheme. Application of the induction hypothesis to the left premise yields:

$$\begin{array}{c} C_1, \Gamma_x \vdash^s e : \tau \quad (\sigma_o, C_o) = gen(C_1, \Gamma_x, \tau) \\ C' \vdash^i \sigma_o \preceq \sigma \quad C' \vdash^e C_o. \end{array}$$

We can conclude that $\Gamma'.x : \sigma$ is realizable in C' because $C', \Gamma' \vdash^{i_2} e : \sigma$. Then we can apply the induction hypothesis to the right and get

$$C_2, \Gamma_x.x : \sigma_o \vdash^s e : \tau' \quad C' \vdash^e C_2.$$

We can conclude that $C' \vdash^e C_o \wedge C_2$ which ensures that $C_o \wedge C_2$ is in solved form. We can apply the (Let) rule and get

$$C_o \wedge C_2, \Gamma_x \vdash^s \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau'$$

■

Lemma 30 (Completeness of $\vdash^W$) *Given $C', \Gamma' \vdash^s e : \tau'$ and $\vdash^i \phi\Gamma \preceq \Gamma'$. Then*

$$\psi, C, \Gamma \vdash^W e : \tau$$

for some substitution ψ, constraint C and type τ such that,

$$\psi \leq^{\phi'}_{fv(\Gamma)} \phi \quad C' \vdash^e \phi' C \quad \vdash^i \phi'\sigma \preceq \sigma'$$

where $\sigma' = gen_1(C', \Gamma', \tau')$ and $\sigma = gen_1(C, \Gamma, \tau)$.

Proof: We use induction on the derivation $\vdash^s$. Due to space limitation we only show one case.

Case (App) We have the following situation:

$$\frac{C', \Gamma' \vdash^s e_1 : \tau'_1 \to \tau'_2 \quad C', \Gamma' \vdash^s e_2 : \tau'_1}{C', \Gamma' \vdash^s e_1 e_2 : \tau'_1}$$

Application of the induction hypothesis yields

$$\begin{array}{c}\psi_1, C_1, \Gamma \vdash^W e_1 : \tau_1 \quad \psi_1 \leq^{\phi_1'}_{fv(\Gamma)} \phi \\ C' \vdash^e \phi_1' C_1 \quad \vdash^i \phi_1' \sigma_1 \preceq \sigma_1' \\ \sigma_1' = gen_1(C', \Gamma', \tau_1' \to \tau_2') \quad \sigma_1 = gen_1(C_1, \psi_1\Gamma, \tau_1)\end{array} \tag{1}$$

and

$$\begin{array}{c}\psi_2, C_2, \Gamma \vdash^W e_2 : \tau_2 \quad \psi_2 \leq^{\phi_2'}_{fv(\Gamma)} \phi \\ C' \vdash^e \phi_2' C_2 \quad \vdash^i \phi_2' \sigma_2 \preceq \sigma_2' \\ \sigma_2' = gen_1(C', \Gamma', \tau_1') \quad \sigma_2 = gen_1(C_2, \psi_2\Gamma, \tau_2)\end{array}$$

W.l.o.g. we assume $\sigma_i = \forall\bar{\beta}.C_i \Rightarrow \tau_i$. We want to rename all bound type variables, we define a renaming substition $\pi = [\bar{\gamma}/\bar{\beta}]$. We now deal with the type scheme $\forall\bar{\gamma}.\pi C_i \Rightarrow \pi\tau_i$. From 1 we know that

$$\vdash^i \phi_1'\sigma_1 \preceq \sigma_1' \qquad \vdash^i \phi_2'\sigma_2 \preceq \sigma_2'$$

We have already convinced us in Lemma 8 that we can use $\vdash^{i_2}$ instead. Then it is not difficult to follow that both above statement must have been derived from

$$\begin{array}{c}C' \vdash^e (\delta_1 \circ \phi_1' \circ \pi)C_1 \quad C' \vdash^i (\delta_1 \circ \phi_1' \circ \pi)\tau_1 \preceq \tau_1' \to \tau_2' \\ C' \vdash^e \delta_2 \circ \phi_2' \circ \pi C_2 \quad C' \vdash^i (\delta_2 \circ \phi_2' \circ \pi)\tau_2 \preceq \tau_1'\end{array}$$

where δ_i's are solution to the bound type variables. We set $\psi' = \psi_1 \sqcup \psi_2$ then we get that $\psi' \leq^{\phi'}_{fv(\Gamma)} \phi$. We define $\delta_3 = [\tau_2'/\alpha]$ and $\delta = \delta_1 \sqcup \delta_2 \sqcup \delta_3$. We set $D = C_1 \wedge C_2 \wedge (\tau_1 = \tau_2 \to \alpha)$. We now apply Lemmas 25 and 26. In this case we additionally deal with substitutions $\delta, \delta_1, \delta_2, \delta_3$. Simply think that ϕ' is now attached with δ. The same holds for ϕ_1' and δ_1 and also ϕ_2' and δ_2. This does not cause any trouble because δ_i's operate only on bound type variables. We get that

$$\begin{array}{c}C' \vdash^e (\delta \circ \phi' \circ \psi' \circ \pi)C_1 \\ C' \vdash^e (\delta \circ \phi' \circ \psi' \circ \pi)C_2 \\ C' \vdash^e (\delta \circ \phi' \circ \psi' \circ \pi)\tau_1 \preceq \tau_1' \to \tau_2' \\ C' \vdash^e (\delta \circ \phi' \circ \psi' \circ \pi)\tau_2 \preceq \tau_1'.\end{array}$$

We can conclude that

$$C' \vdash^e (\delta \circ \phi' \circ \psi' \circ \pi)\tau_1 \preceq (\delta \circ \phi' \circ \psi' \circ \pi)\tau_2 \to \tau_2'.$$

By definition it holds that $(\delta \circ \phi' \circ \psi' \circ \pi)(\alpha) = \tau_2'$ and then we get that

$$C' \vdash^e (\delta \circ \phi' \circ \psi' \circ \pi)D.$$

But that means $(C', \delta \circ \phi' \circ \psi')$ is a normal form of $(\pi D, \delta \circ \phi' \circ \psi')$. Because HM(X) satisfies the principal constraint property we know that there exist a principal normal form of $(\pi D, \delta \circ \phi' \circ \psi')$. That means we get $normalize(D, \delta \circ \phi' \circ \psi') = (C, \psi)$ where

$$\psi \leq^\rho \delta \circ \phi' \circ \psi' \circ \pi \quad C' \vdash^e \rho C. \tag{2}$$

Then we can deduce that

$$(\rho \circ \psi)_{|fv(\Gamma)} = (\delta \circ \phi' \circ \psi')_{|fv(\Gamma)} = \phi$$

Hence, we get that

$$\psi \leq^\rho_{fv(\Gamma)} \phi$$

It remains to show that $\vdash^i \rho\sigma_o \preceq \sigma_o'$ where

$$\sigma_o' = gen_1(C', \Gamma', \tau_2') \text{ and } \sigma_o = gen_1(C, \psi\Gamma, \psi(\alpha)).$$

We have already seen that $(\delta \circ \phi' \circ \psi' \circ \pi)(\alpha) = \tau_2'$. Because of the choice of the substitution π it holds also that $(\delta \circ \phi' \circ \psi')(\alpha) = \tau_2'$. And from 2 we get $(\rho \circ \psi)(\alpha) = \tau_2'$. That means, we know that

$$C' \vdash^e \rho C \quad C' \vdash^i (\rho \circ \psi)(\alpha) \preceq \tau_2'$$

Now we can apply Lemma 24 and get $\vdash^i \rho\sigma_o \preceq \sigma_o'$ as desired. Finally, we apply the (App) rule and get $\psi_{|fv(\Gamma)}, C, \Gamma \vdash^W e_1 e_2 : \psi(\alpha)$.

■

Before we prove the main theorem we state the following corollary which is based on Lemma 29.

Corollary 3 *Given $C', \Gamma' \vdash^2 e : \sigma'$, $C' \vdash^i \Gamma \preceq \Gamma'$ and Γ' is realizable in C'. Then*

$$C, \Gamma \vdash^s e : \tau$$

such that

$$C' \vdash^e C_o \qquad C' \vdash^i \sigma_o \preceq \sigma$$

where $(\sigma_o, C_o) = gen(C, \Gamma, \tau)$.

Now we have everything at hand to prove completeness of type inference.

Theorem 29 *Given $C', \Gamma' \vdash e : \sigma'$, $C' \vdash^i \phi\Gamma \preceq \Gamma'$ and Γ' is realizable in C'. Then*

$$\psi, C, \Gamma \vdash^W e : \tau$$

for some substitution ψ, constraint C and type τ such that,

$$\psi \leq^{\phi'}_{fv(\Gamma)} \phi \quad C' \vdash^e \phi' C_o \quad C' \vdash^i \phi'\sigma_o \preceq \sigma'$$

where $(\sigma_o, C_o) = gen(C, \psi\Gamma, \tau)$.

Proof: First, we apply Lemma 27 in order to get a derivation in $\vdash^2$. Then, we can apply Corollary 3 and get

$$\begin{array}{c}C, \phi\Gamma \vdash^s e : \tau \\ (\sigma_o, C_o) = gen(C, \phi\Gamma, \tau) \\ C' \vdash^e C_o \quad C' \vdash^i \sigma_o \preceq \sigma'\end{array} \tag{3}$$

After that we apply Lemma 30 (completeness of $\vdash^W$) and get

$$\begin{gathered}\psi, \tilde{C}, \Gamma \vdash^W e : \tilde{\tau} \quad \psi \leq^{\phi'}_{fv(\Gamma)} \phi \\ (\tilde{\sigma}_o, \tilde{C}_o) = gen(\tilde{C}, \psi\Gamma, \tilde{\tau}) \\ C \vdash^e \phi'\tilde{C} \quad \vdash^i \phi'\tilde{\sigma}_o \preceq \sigma_o.\end{gathered}$$

It remains to show

1. $C' \vdash^i \phi'\tilde{\sigma}_o \preceq \sigma'$
2. $C' \vdash^e \phi'\tilde{C}_o$.

We know that $C \vdash^e \phi'\tilde{C}$ with Lemma 23 we get that $C_o \vdash^e \phi'\tilde{C}_o$. From 3 we know that $C' \vdash^e C_o$ and because $\vdash^e$ is transitive we get that $C' \vdash^e \phi'\tilde{C}_o$. Also, we know that $C' \vdash^i \sigma_o \preceq \sigma'$ and because $\vdash^i$ is transitive and closed under strengthening the constraint we get $C' \vdash^i \phi'\tilde{\sigma}_o \preceq \sigma'$. ■

Then we get Theorem 22 as a corollary from Theorem 29.