# Designing Record Systems

Martin Sulzmann*
Yale University, Department of Computer Science
P.O. Box 208285, New Haven, CT 06520-8285
sulzmann@cs.yale.edu

Research Report YALEU/DCS/RR-1128
April 1997

## Abstract

We explore the design space for type systems with polymorphic records. We design record systems for extension, concatenation and removal of fields. Furthermore, we design a record system where field labels become first class values. That means, we can now quantify over field lables and pass them around as arguments. All designed record systems enjoy type inference with principal types. Especially, we can combine any features into a new record system retaining type inference with principal types.

We also point out some problems which are present in previous record systems. Our designed record systems can be seen as the proper logical formulation of previous approaches with even more expressive power.

We base our design on the HM(X) framework. HM(X) is a general framework for Hindley/Milner type systems that are parameterized in the constraint domain X. HM(X) enables us to design record systems in a systematic way retaining type inference with principal types. That means, designing record systems becomes construction of constraint systems which model record systems.

## 1 Introduction

Type systems for records have become a playing field for type theorists [HP91, Car84, CM89, EST95, Rém95a, Jon92, Oho95, Wan88, Wan89, Rém95b, Rém89]. One of the main motivations for record systems is that they can be used to encode object calculi [Wan89, Rém95a] or module systems [Apo93, Jon96]. Also, they are useful for data type declarations and in database programming [OB88].

But there are a couple of challenging problems to overcome. The type system should support polymorphic records and additional operations like extensions of records. At least it should be possible to give a type checking algorithm. In the best case we want to have a type inference algorithm which computes principal types. On the practical side it should be possible to give an efficient compilation method.

Type systems with records based on subtyping [HP91, Car84, CM89, EST95, Rém95a] have problems to support record concatenation and a compilation calculus. Also, it seems that such type systems do not provide good wrappers for object–oriented languages [BPF97]. The concept of row variables [Wan88, Wan89, Rém95b, Rém89] has also some limitations. Ohori [Oho95] introduced kinds for record types that can be seen as predicates. He could provide an efficient compilation calculus but his system lacks features like addition or removal of field labels. The approaches of Rémy [Rém92], Kennedy [Ken96] and Gaster and Jones [GJ96] are similar in spirit to ours. Rémy and Kennedy extend the Hindley/Milner type system with a sorted equational theory. We argue that our constraint system is more general and we conjecture that their systems are not able to handle field labels as first class values. The approach of Gaster et al is based on qualified types [Jon92]. They also present a full variety of record systems with similar expressive power. In the latter, we will see examples where one can see where our approach has advantages. In general, we will discuss some problems which are present in previous approaches.

We use the HM(X) framework to design record systems. The HM(X) framework was introduced in [SOW97]. A detailed description can be found in [SOW]. HM(X) is a Hindley/Milner type system parameterized in the constraint domain X. Under the assumption that X has the principal constraint property, a generic type inference algorithm can be given that computes principal types.

*Supported by Yale University Fellowship

The idea behind HM(X) is that whenever we need a new type system we do not have to event new typing rules and a type inference algorithm. We simply provide an instance of the constraint domain X which captures the desired properties that we want to model. If this instance satisfies the principal constraint property we get a type inference algorithm which computes principal types for free.

The main contribution of this paper is that we present a new methodology for designing record systems. Based on HM(X) we can do this in a systematic way retaining type inference with principal types. We systematically extend the expressive power of the constraint domain. At the end we get an instance of HM(X) which is able to deal with record and variant types, extension, concatenation of polymorphic records, removal of field labels and that is one of the novelties of our approach, field labels become now first class values. Furthermore, we point out some problems which are present in previous approaches and argue that HM(X) can be seen as the proper logical foundation of these previous approaches with even more expressive power.

## 2 Overview

We discuss an instance HM($\mathcal{R}$) of the HM(X) framework that deals with record types. We base our record system on Ohori's calculus [Oho95]. Record types are denoted by $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ where $l_i$ stands for a field label and $\tau_i$ for the associated type. We introduce a constraint system $\mathcal{R}$ where we can express constraints on record types. For instance, the constraint $(\alpha ::< l : \tau >)$ states that $\alpha$ is a record which contains at least a field with label $l$ and type $\tau$. On constraints $(\alpha ::< l : \tau >)$ we put conditions like

**R1** $\vdash^e (\{l_1 : \tau_1, \ldots l_n : \tau_n\} ::< l_i : \tau_i >)$
where $l_1, \ldots, l_n$ are distinct and $i \in \{1, \ldots, n\}$

**R2** $(\alpha ::< l : \tau_1 >) \wedge (\alpha ::< l : \tau_2 >) \vdash^e (\tau_1 = \tau_2)$

where $\vdash^e$ is the entailment relation between constraints in $\mathcal{R}$ and $(=)$ is the equality predicate. By condition **R2** we forbid overloading of field labels. In HM($\mathcal{R}$) we are able to handle polymorphic records. We can give a type to the selector function $.l$ that selects field label $l$ from a given record. We can express this by

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

The selector function $.l$ can be seen as a primitive construct in an initial type environment $\Gamma_0$. We introduce some more basic primitive constructs in later sections. When we apply this selector function to a given record we have to find an instance of the above type scheme. The constraint $(\alpha ::< l : \beta >)$ ensures that the given record actually contains field label $l$.

Bounded type variables in type schemes can now be constrained by constraints of the form $(\alpha ::< l : \beta >)$. That means, in general we deal with type schemes of the form $\forall \bar{\alpha}.C \Rightarrow \tau$ where the possible instances of the bound type variables $\bar{\alpha}$ are constrained by $C$. The presence of constraints is reflected in the typing rules. We have a rule

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

that handles instantiation of constrained type variables. For instance, the term [1]

$$\{Address : string\}.Name$$

is rejected by the type system because there is no valid instance for the constraint $(\alpha ::< l : \beta >)$. That means, type inference should report a type error.

The typing problem in HM(X) is reduced to constraint problems in X. To this purpose, X needs to be a rich constraint system to express all typing problems. In the type system we only admit a subset of constraints in X that are in so–called *solved form*. Type inference involves accumulation of constraint problems and normalizing constraints to solved forms. For example, when we consider the term

$$\{Address : string\}.Address$$

type inference in HM($\mathcal{R}$) generates the constraint problem

$$C = (\alpha ::< Address : \beta >) \wedge (\alpha = \{Address : string\})$$

The constraint $C$ is valid but not in solved form. Normalization of $C$ yields the constraint true with residual substitution

$$[\{Address : string\}/\alpha, string/\beta]$$

Normalization of a constraint should result in the best possible solved form in order to compute principal types. Under the condition that the constraint system X fulfills the principal constraint property we can give a generic type inference algorithm for HM(X) type systems that computes principal types. The principal constraint property states that normalizing a satisfiable constraint results in a so–called principal normal form. A principal normal form represents the best possible solved form of a satisfiable constraint. In case of HM($\mathcal{R}$),

[1] We switch the order of function application because record selection is usually written in postfix notation.

we have to show that $\mathcal{R}$ satisfies the principal constraint property.

In the latter, we discuss several extensions of HM($\mathcal{R}$). In HM($\mathcal{R}^e$) we discuss extensions of records. To this purpose we introduce constraints of the form $extend_l(\alpha, \beta, \gamma)$. Such constraints enable us to express record extension. The following primitive construct

$$extend_l : \forall \alpha, \beta, \gamma. extend_l(\alpha, \beta, \gamma) \Rightarrow \alpha \to \beta \to \gamma$$

handles extension of records with field label $l$. We have not specified the exact behavior of the constraint $extend_l(\alpha, \beta, \gamma)$. There are several choices. Do we allow to override an already existing field label or can we only extend a record with a non–existing field labelΓ The exact behavior is reflected in the specific constraint system which we consider. For the moment, we assume that we can only extend a record with a non–existing field label. Because of one of the nice properties of the HM(X) framework we can express removal of field labels in terms of $extend_l$. The primitive construct

$$remove_l : \forall \alpha, \gamma. \exists \beta. extend_l(\alpha, \beta, \gamma) \Rightarrow \gamma \to \alpha$$

handles removal of field labels. The operator $\exists \beta$ is called projection operator and is a construct of our constraint system. The projection operator corresponds to existential quantification if the constraint system models a boolean algebra. The constraint $\exists \beta. extend_l(\alpha, \beta, \gamma)$ expresses that record $\gamma$ contains a field with label $l$ which is not present in $\alpha$. The type of the field label $l$ is not of interest in this context. Therefore, $\beta$ is bound by the projection operator and hidden from outside. Additionally, it would also be possible to type $remove_l$ with

$$remove_l : \forall \alpha, \beta, \gamma. extend_l(\alpha, \beta, \gamma) \Rightarrow \gamma \to \alpha$$

For instance, in the theory of qualified types [Jon92] we can not type $remove_l$ with the above type. The reason is that type variable $\beta$ does only appear in the constraint and therefore $\beta$ is an ambiguous type variable. The HM(X) framework introduces a projection operator which enables us to hide such type variables. That means, $remove_l$ is typable in HM(X) but not in qualified types.

Another extension HM($\mathcal{R}^c$) uses constraints of the form $concat(\alpha, \beta, \gamma)$ to model record concatenation. We will see that in some sense HM($\mathcal{R}^c$) subsumes HM($\mathcal{R}^e$). Variant types are considered in HM($\mathcal{R}^v$). We now have constraints of the form $(\alpha ::\prec l_1 : \tau_1, \ldots, l_n : \tau_n \succ)$. Such constraints denote that $\alpha$ is a variant type which contains a tagged value with type $\tau_i$ and label $l_i$ where $i \in \{1, \ldots, n\}$. We also sketch how recursive types could be incorporated. It is also possible to choose any of the described applications and combine them in a new application which retains all previous properties.

In the previous record systems we treated field labels always as fixed constants. We introduce an application HM($\mathcal{R}_p$) where field labels become first class values. For example, field labels can now be passed around as arguments. This eliminates the need for primitive constructs $extend_l$ for each field label $l$. One construct *extend* with an additional parameter for field labels is then sufficient.

We always use the same methodology to model some desired features. We introduce some appropriate primitive constraints like $(\alpha ::< l : \beta >)$ and $extend_l(\alpha, \beta, \gamma)$. Then we give a constraint system that describes the desired behavior. Such a treatment can already be found in previous work. For instance, Jones [Jon92] uses row variables and predicates of the form $(r \backslash l)$ to express that row $r$ does not contain a field with label $l$. Rémy [Rém89] expresses by $r\ has\ l$ that $r$ contains a field with label $l$. A predicate $r_1 \# r_2$ is introduced by Harper and Pierce [HP91] to model symmetric concatenation of records $r_1$ and $r_2$. More sophisticated sorted equational theories can be found in [Rém92, Ken96] to model record calculi.

We point out some problems which are present in previous approaches. The main advantage of our framework is that we can bind type variables in constraints by the projection operator $\exists \bar{\alpha}$. Such a treatment can not be found in previous work. This is one of the main reasons of problems in previous approaches. We argue that HM(X) can be seen as the proper logical foundation of these approaches. Furthermore, we discuss how to reuse previous approaches and how to compile them into the HM(X) framework.

The rest of this report is organized as follows. In Section 3 we review basic definitions and properties of the HM(X) framework. Section 4 describes a record application. Further extensions like extensible records are discussed in Section 5. Section 6 presents an application where field labels become first class values. Related work is discussed in Section 7. Section 8 concludes.

## 3 The HM(X) framework

Here, you can find an overview of the HM(X) framework. For a detailed description we refer to [SOW]. First, we introduce our notion of constraint systems. Then we describe the HM(X) type system. Finally, we consider type inference for HM(X) type systems.

### 3.1 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 cylindric constraint system. A cylindric constraint system introduces a projection operator $\exists\alpha$. In the case where constraint systems are boolean algebras, projection corresponds to existential quantification. Furthermore, we introduce the new notion of a term constraint system. A term constraint system enables us to express constraint problems which arise during type inference.

**Definition 1** *(Simple Constraint System)*
*A* simple constraint system *is a structure* $(\Omega, \vdash^e)$ *where* $\Omega$ *is a non–empty set of tokens or (primitive) constraints* [2]*. The relation* $\vdash^e \subseteq p\Omega \times \Omega$ *is a decidable entailment relation where* $p\Omega$ *is the set of finite subsets of* $\Omega$*. 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.*

**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* $fv(C) = \{\alpha \mid \exists\alpha.C \not=^e C\}$*.*

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

We now define satisfiability of a constraint.

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

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

**Definition 5** *(Types) A* type *is a member of* $\mathcal{T}$*=Term*$(\Sigma)$ *where Term*$(\Sigma)$ *is the term algebra* $\mathcal{T}$ *built up from a signature* $\Sigma =$ *(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 $\phi$ *is an idempotent mapping from a set of variables Var to the term algebra Term*$(\Sigma)$*. Let id be the identity substitution.*

**Definition 7** *(Term Constraint System) A* term constraint system $\mathcal{TCS}_{\mathcal{T}} = (\Omega, \vdash^e, Var, \{\exists\alpha \mid \alpha \in Var\})$ *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* $\tau, \tau'$ *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.*

### 3.2 The type system

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** | $\tau$ | $::\supseteq$ | $\alpha \mid\ \tau \to \tau \mid T\bar{\tau}$ |
| **Type schemes** | $\sigma$ | $::=$ | $\tau \mid \forall \alpha.C \Rightarrow \sigma$ |

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 $\alpha$. 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 $\Gamma_0$.

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. Typing judgments are of the form $C, \Gamma \vdash e : \sigma$ where $C$ is a constraint in $\mathcal{S}$. The most interesting rules are the ($\forall$ Intro) rule and the ($\forall$ Elim) rule. By rule ($\forall$ Intro) we quantify some type variables. Rule (Equ) becomes important if there are type constructors in the term algebra $\mathcal{T}$ with non–trivial equality relations.

### 3.3 Type inference

For HM(X) type systems we can give now a generic type inference algorithm, see Figure 2.

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 that constraint $D$ is normalized. Normalizing means computing a substitution $\psi$ and a residual constraint $C$ in $X$ such that $\psi C$ entails $D$ where $\psi C$ 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.

**Definition 8** *(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 $\phi$,$\psi$ be substitutions. Then $(C, \psi)$ is a* normal form *of $(D, \phi)$ iff $\phi \leq \psi$, $C \vdash^e \psi D$ and $\psi C = C$*

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

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

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

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

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

Figure 2: Type inference

Given $(D,\phi)$ we can define a function *normalize* by:

$normalize(D,\phi)$
$= \quad (C,\psi)$ if $(C,\psi)$ principal normal form of $(D,\phi)$
$= \quad fail$ otherwise

We use the convention that when we say a principal normal form of $(D,\phi)$ exists we know how to compute the principal normal form of $(D,\phi)$. We now extend the property of having a principal normal form to constraint systems.

**Definition 9** *(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 $\phi$, either $(D,\phi)$ does not have a normal form or $(D,\phi)$ 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. We can conclude that a constraint system X which enjoys the principal constraint property comes with a computable function *normalize*.

**Theorem 10** *Given a HM(X) type system which satisfies the principal constraint property. Then the type inference algorithm computes principal types.*

## 4 Polymorphic records

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 $\Gamma_0$ 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 $\Sigma$. 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 $\pi$ 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 $\tau$. 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 $\tau$ is a type and $k$ is a kind. Technically, this means we add $(\tau :: k)$ to the set $\Omega$ 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* $\Omega$ *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 add now primitive constructs to the initial type environment $\Gamma_0$ 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 $\Gamma_0$. $l_1\_\ldots\_l_n$ 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 $\Gamma_0$. The first corresponds to record selection, the latter to record update.

The question arises how HM($\mathcal{R}$) is related to $\mathcal{O}$. We postpone a discussion of the relationship between HM($\mathcal{R}$) and $\mathcal{O}$ to Section 7.

We now consider type inference for HM($\mathcal{R}$). That means we have to show that $\mathcal{R}$ fulfills the principal constraint property. Our constraint system differs in two respects from the one used in $\mathcal{O}$. We have an equality predicate (=) and a projection operator $\exists\bar{\alpha}$. 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 projection operator. It holds that

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

where $\beta$ 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 1** *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 $\alpha$ 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 2** *Given a field label $l$ and types $\tau, \tau'$. 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 $\phi$ of the equality predicates (=) in $Cl(D)$. It remains to check whether this most general unifier $\phi$ 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 3** *Given a projection free constraint $D \in \mathcal{R}$ and a substitution $\phi$. Then $(D, \phi)$ 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. Before we proceed, we state the following lemma which we will use in the next two lemmas.

**Lemma 4** *Given a constraint $C \in \mathcal{R}$ such that $\exists\alpha.C \in \mathcal{S}$ and $C \notin \mathcal{S}$. Then there is a $\tau$ such that $\exists\alpha.C =^e [\tau/\alpha]C$.*

**Proof:** The only reason why $C$ is not in solved form is that there is an unresolved unification problem $(\tau' = \tau'')$ in $C$. But because $\exists\alpha.C$ is in solved form we know that $\vdash^e \exists\alpha.(\tau' = \tau'')$. Then we can conclude there is a $\tau$ such that $[\tau/\alpha](\tau' = \tau'') =^e \exists\alpha.(\tau' = \tau'')$. This gives us $\exists\alpha.C =^e [\tau/\alpha]C$. ■

**Lemma 5** *Let $D \in \mathcal{R}$ and $\phi$ be a substitution where $\alpha \notin codom(\phi) \cup dom(\phi)$. If $(C, \psi) = normalize(D, \phi)$ then $(\exists\alpha.C, \psi_{\setminus\{\alpha\}}) = normalize(\exists\alpha.D, \phi)$.*

**Proof:** Given the principal normal form $(C, \psi)$ of $(D, \phi)$. We assume w.l.o.g. that $\alpha \notin codom(\psi)$. It holds that $[\tau/\alpha]D \vdash^e \exists\alpha.D$ where $\alpha \notin fv(\tau)$. We can conclude that $\exists\alpha.C \vdash^e \psi_{\setminus\{\alpha\}}\exists\alpha.D$. We get that $(\exists\alpha.C, \psi_{\setminus\{\alpha\}})$ is a normal form of $(\exists\alpha.D, \phi)$.

Assume now $(C', \psi')$ is another normal form of $(\exists\alpha.D, \phi)$ and w.l.o.g. $\alpha \notin codom(\psi')$. Then we now that $C' \vdash^e \psi'(\exists\alpha.D) =^e \exists\alpha.\psi' D$. We distinguish the following two cases:

*Case $\psi' D \in \mathcal{S}$:* Then $(\psi' D, \psi')$ is a normal form of $(D, \phi)$. Because $(C, \psi)$ is principal we get that $\psi' D \vdash^e \psi' C$ and $\psi \leq \psi'$. We can derive that

$$C' \vdash^e \exists\alpha.\psi' D \vdash^e \exists\alpha.\psi' C \vdash^e \psi'(\exists\alpha.C)$$

which shows that $(\exists\alpha.C, \psi_{\setminus\{\alpha\}})$ is principal.

*Case $\psi' D \notin \mathcal{S}$:* Because $\exists\alpha.\psi' D \in \mathcal{S}$ there must be an unresolved unification problem in $\psi' D$. That means, there is a $\tau$ such that $\exists\alpha.\psi' D =^e [\tau/\alpha](\psi' D)$. Note, it holds that $[\tau/\alpha] \circ \psi' = \psi' \circ [\tau/\alpha]$. We get that $C \vdash^e \psi' \circ [\tau/\alpha]D$ because $(C, \psi)$ is principal we can follow that

$$C' \vdash^e (\psi' \circ [\tau/\alpha])C \qquad \psi \leq \psi' \circ [\tau/\alpha]$$

But then we get that

$$C \vdash^e \psi'\exists\alpha.C \qquad \psi_{\setminus\{\alpha\}} \leq \psi'$$

which also shows that $(\exists\alpha.C, \psi_{\setminus\{\alpha\}})$ is principal. ■

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

**Lemma 6** *Given a substitution $\phi$ where $\alpha \notin codom(\phi) \cup dom(\phi)$ and a constraint $D \in \mathcal{R}$. Then $(D, \phi)$ has a normal form if $(\exists\alpha.D, \phi)$ has a normal form.*

**Proof:** We assume that $(C, \psi)$ is a normal form of $(\exists\alpha.D, \phi)$. That means,

$$C \vdash^e \psi\exists\alpha.D \qquad \phi \leq \psi$$

Also, we know that $\exists\alpha.\psi D =^e \psi\exists\alpha.D$. We distinguish the following two cases:

*Case $\psi D \in \mathcal{S}$ :* Then it is easy to see that $(\psi D, \psi)$ is a normal form of $(D, \phi)$.

*Case $\psi D \notin \mathcal{S}$ :* Then we know there is a $\tau$ such that $\exists\alpha.\psi D =^e [\tau/\alpha] \circ \psi D$. In this case $(C, [\tau/\alpha] \circ \psi)$ is a normal form of $(D, \phi)$. ■

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

**Theorem 11** *The constraint system $\mathcal{R}$ has the principal constraint property.*

**Proof:** Assume the contrary. That means we have a tuple $(D, \phi)$ which has a normal form but no principal normal form. With Lemma 1 we know that $D =^e \exists \bar{\alpha}.D'$ where $D'$ is a projection free constraint. Then we apply Lemma 6 and get that the normal form of $(D', \phi)$ exists. From Lemma 3 we know that $(D', \phi)$ does have a principal normal form. With Lemma 5 we can lift the principal normal form and get that the principal normal form of $(D, \phi)$ 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 gave 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. A detailed discussion of the relationship between HM($\mathcal{R}$) and $\mathcal{O}$ is postponed to Section 7.

## 5 Further extensions

The record calculus of Ohori lacks some important features, e.g. we do not have extensible records. We show now how to extend HM($\mathcal{R}$) to get HM($\mathcal{R}^e$) where we additionally have record extensions.

We start with the HM($\mathcal{R}$) system. We give an extension of $\mathcal{R}$ that is able to deal with record extension. We call it $\mathcal{R}^e$. We add for each label $l$ primitive constraints of the form $extend_l(\tau_1, \tau_2, \tau_3)$ to the set of primitive constraints $\Omega$. $\mathcal{R}^e$ has to fulfill the following additional rules:

**R6** $extend_l(\alpha, \beta, \gamma) \vdash^e (\gamma ::< l : \beta >)$
**R7** $extend_l(\alpha, \beta, \gamma) \wedge (\alpha ::< l' : \tau >) \vdash^e (\gamma ::< l' : \tau >)$
**R8** $extend_l(\alpha, \beta, \gamma) \wedge (\gamma ::< l' : \tau >) \vdash^e (\alpha ::< l' : \tau >)$ where $l \neq l'$
**R9** $extend_l(\tau_1 \to \tau_1', \tau_2, \tau_3) \vdash^e$ false
**R10** $extend_l(\tau_1, \tau_2, \tau_3 \to \tau_3') \vdash^e$ false
**R11** $extend_l(\alpha, \beta, \gamma) \wedge (\alpha ::< l : \tau >) \vdash^e$ false
**R12** $extend_l(\alpha, \beta, \alpha) \vdash^e$ false
**R13** $\vdash^e extend_l(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}, \tau, \{l_1 : \tau_1, \ldots, l_n : \tau_n, l : \tau\})$ where $l_i \neq l$
**E7** $\exists \alpha, \gamma.extend_l(\alpha, \beta, \gamma) =^e$ true

**Remark 6** *Conditions* **R6** – **R12** *give a characterization of extending record $\alpha$ with field label $l$ and value $\beta$. The resulting record is $\gamma$. This is expressed through the predicate $extend_l(\alpha, \beta, \gamma)$. Note, we can only extend a record with a field label $l$ if this field label is not already present in the record. Without condition* **R11** *we could extend a record with an already existing field label but for this special case we already have the primitive $modify_l$. The projection operator is extended to $extend_l$ in condition* **E7**.

*Jones [Jon92] uses a predicate $(\alpha \backslash l)$ to express that record $\alpha$ does not contain a field label $l$. We can define $(\alpha \backslash l)$ as an abbreviation for $\exists \beta, \gamma.extend_l(\alpha, \beta, \gamma)$.*

In the set $\mathcal{S}$ of constraints are now additionally constraints of the form $extend_l(\tau_1, \tau_2, \tau_3)$. Then the set $\mathcal{S}$ of constraints in solved form is defined as in Section 4.

We now have to show that $\mathcal{R}^e$ has the principal constraint property. We proceed in a similar way as for $\mathcal{R}$. First, we give an algorithm which computes the principal normal form of projection free constraints.

We assume now that we have a projection free constraint $D$. We consider $D$ as a set of primitive constraints. Then the closure $Cl(D)$ of a constraint $D$ 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 $(\tau_1 = \tau_2) \in Cl(D)$
4. If $extend_l(\tau_1, \tau_2, \tau_3) \in Cl(D)$ then $(\tau_3 ::< l : \tau_2 >) \in Cl(D)$
5. If $extend_l(\tau_1, \tau_2, \tau_3), (\tau_1 ::< l' : \tau >) \in Cl(D)$ then $(\tau_3 ::< l' : \tau >) \in Cl(D)$
6. If $extend_l(\tau_1, \tau_2, \tau_3), (\tau_3 ::< l' : \tau >) \in Cl(D)$ and $l \neq l'$ then $Cl(D) \in (\tau_1 ::< l' : \tau >)$

In this case we can restate Lemma 2.

**Lemma 7** *Given a field label $l$ and types $\tau, \tau'$. 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 proceed as before and we can state the following lemma.

**Lemma 8** *Given a projection free constraint $D \in \mathcal{R}$ and a substitution $\phi$. Then $(D, \phi)$ does have a principal normal form or no normal form exits.*

We can make now the following observations. Lemmas 1, 5, 6 also hold in $\mathcal{R}^e$. Then it is straightforward to show that $\mathcal{R}^e$ also satisfies the principal constraint property.

**Theorem 12** *The constraint system $\mathcal{R}^e$ has the principal constraint property.*

**Proof:** Same argumentation as in Theorem 11. ■

We need now primitive constructs in the initial type environment $\Gamma_0$ for record extension. For each field label $l$ we add

$$extend_l : \forall\alpha,\beta,\gamma.extend_l(\alpha,\beta,\gamma) \Rightarrow \alpha \to \beta \to \gamma$$

to the initial type environment $\Gamma_0$. The construct

$$remove_l : \forall\alpha,\gamma.\exists\beta.extend_l(\alpha,\beta,\gamma) \Rightarrow \gamma \to \alpha$$

handles removal of the field label $l$.

In the appendix we consider further extensions for variants (HM($\mathcal{R}^v$)), concatenation (HM($\mathcal{R}^c$)). It is straightforward to show that we can take any of these extensions and combine them in a new instance HM($\mathcal{R}^x$) where $x$ is a term in the regular grammar $[e|c|v]$. We also conjecture that it is possible to design HM($\mathcal{R}^r$) which supports recursive types. We give a sketch of HM($\mathcal{R}^r$) in the appendix.

## 6 Polymorphic field labels

One of the shortcomings of HM($\mathcal{R}$) and its extensions is that we need primitive constructs for each field label $l$. Field labels are treated as constants. We now introduce an application HM($\mathcal{R}_p$) where field labels become first class values. That means, we can pass field labels as arguments to functions and can manipulate them in other ways. Especially, it is now possible to quantify over field labels.

Technically, we add now primitive constraints ($\tau :: label$) to the set $\Omega$ of primitive constraints where ($:: label$) is an unary predicate. The constraint ($l :: label$) denotes now that $l$ is a field label. We restate now the conditions for $\mathcal{R}$. Before, we introduce two abbreviations. The term $distinct(l_1,\ldots,l_n)$ is a short–hand for the constraint

$$\bigwedge_{i\neq j, i,j\in\{1,\ldots,n\}} (l_i \neq l_j)$$

The constraint ($\neq$) can be expressed in terms of ($=$) and is defined by

$$\vdash^e (\tau \neq \tau') \qquad \text{iff} \qquad \not\vdash^e (\tau = \tau')$$

The term $label(l_1,\ldots,l_n)$ is a short–hand for the constraint

$$(l_1 :: label) \wedge \ldots \wedge (l_n :: label)$$

The constraint system $\mathcal{R}_p$ is now defined as follows:

**R1** $distinct(l_1,\ldots,l_n) \wedge label(l_1,\ldots,l_n) \vdash^e (\{l_1:\tau_1,\ldots l_n:\tau_n\} ::< l_i:\tau_i >)$ where $i \in \{1,\ldots,n\}$
**R2** $(\alpha ::< l:\tau >) \vdash^e (l :: label)$
**R3** $(\alpha ::< l:\tau_1 >) \wedge (\alpha ::< l:\tau_2 >) \vdash^e (\tau_1 = \tau_2)$
**R4** $(\{\ldots, l:\tau_1, \ldots\} ::< l:\tau_2 >) \vdash^e (\tau_1 = \tau_2)$
**R5** $(\{l_1:\tau_1,\ldots,l_n:\tau_n\} ::< l:\tau >) \wedge (l \neq l_1) \wedge \ldots \wedge (l \neq l_n) \vdash^e$ false
**R6** $(\tau_1 \to \tau_2 :: label) \vdash^e$ false
**R7** $(\alpha :: label) \wedge (\alpha ::< l:\tau >) \vdash^e$ false
**R8** $(\tau_1 \to \tau_2 ::< l:\tau_3 >) \vdash^e$ false
**E5** $\exists\alpha.(\alpha :: label) =^e$ true
**E6** $\exists\alpha.(\alpha :: k) =^e$ true where $\alpha \notin fv(k)$
**E7** $\exists\alpha.(\alpha :: k) =^e$ false where $\alpha \in fv(k)$
**F1** false $\vdash^e P$ for $P \in \Omega$
**F2** $\exists\alpha.$false $=^e$ false

The set $\mathcal{S}$ of constraints in solved form contains now constraints of the form ($\alpha :: \_$) and ($\{l_1:\tau_1,\ldots,l_n:\tau_n\} :: l:\tau$) where $l_1,\ldots,l_n,l$ are distinct variables and $n >= 2$. Note, field labels are now represented by variables. In HM($\mathcal{R}$) a constraint of the form ($\{\_\} :: \_$) is either true or false. But in HM($\mathcal{R}_p$) a constraint of the form ($\{\_\} :: \_$) can be satisfiable or not. It is only possible to resolve records which contain only one field because it holds that $(\{l:\tau\} ::< l':\tau' >) \vdash^e (l = l')$. This fact is also reflected in the following definition of the closure of a constraint. The rest of the definition of $\mathcal{S}$ is along the lines as in Section 4.

Then, we have to show that $\mathcal{R}_p$ satisfies the principal constraint property. We do this along the lines as in Section 4. We simply have to adapt the definition of the closure $Cl(D)$ of a projection free constraint $D$. Now, the closure $Cl(D)$ is the smallest constraint which satisfies:

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 $(\tau_1 = \tau_2) \in Cl(D)$
4. If $(\alpha ::< l:\tau >) \in Cl(D)$ then $(l :: label) \in Cl(D)$
5. If $(\alpha = \{l:\tau\}), (\alpha ::< l':\tau' >) \in Cl(D)$ then $(l = l') \in Cl(D)$

We also have to adapt Lemma 2. In this case we have the following lemma.

**Lemma 9** *Given a field label $l$ and types $\tau,\tau'$. If there are no $l_1,\ldots,l_n$ such that $label(l_1,\ldots,l_n) \wedge distinct(l_1,\ldots,l_n) \vdash^e (\tau ::< l:\tau' >)$ 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* $D \vdash^e (\tau = \tau')$.

We can proceed exactly in the same way as in Section 5. We get the following theorem.

**Theorem 13** *The constraint system* $\mathcal{R}_p$ *satisfies the principal constraint property.*

We have now the following primitives in initial type environment $\Gamma_0$:

$$
\begin{array}{lll}
\mathit{formation}_n & : & \forall \alpha, \beta_1, \ldots, \beta_n, l_1, \ldots, l_n. \\
& & \mathit{distinct}(l_1, \ldots, l_n) \wedge \mathit{label}(l_1, \ldots, l_n) \wedge \\
& & \Rightarrow \\
& & \beta_1 \to \ldots \to \beta_n \to l_1 \to \ldots \to l_n \\
& & \to \{l_1 : \beta_1, \ldots, l_n : \beta_n\} \\
. & : & \forall \alpha, \beta, l. \\
& & (\alpha ::< l : \beta >) \wedge (l :: \mathit{label}) \Rightarrow \\
& & \alpha \to l \to \beta \\
\mathit{modify} & : & \forall \alpha, \beta, l. \\
& & (\alpha ::< l : \beta >) \wedge (l :: \mathit{label}) \\
& & \Rightarrow \\
& & \alpha \to l \to \beta \to \alpha
\end{array}
$$

The primitive constructs are now polymorphic in the field labels. That means, we do not need anymore for each field label $l$ a special primitive construct. It is also possible to add recursive types, variants, extension and concatenation of records to HM($\mathcal{R}_p$). That means, we get extensions HM($\mathcal{R}_p^v$), HM($\mathcal{R}_p^e$) and HM($\mathcal{R}_p^c$). For instance, instead of a set of predicates $\mathit{extend}_l$ we simply have now one predicate *extend* with one additional parameter which represents the new field label. It is straightforward to adapt the rules. For space reasons we omit a detailed description.

The question arises what expressive power gives us HM($\mathcal{R}_p$). For example, it is now possible to characterize all records which contain at least one field:

$$f : \forall \alpha. \exists \beta, l. (\alpha ::< l : \beta >) \wedge (l :: \mathit{label}) \Rightarrow \alpha.$$

The term $f$ represents all records which contain at least one field. We leave further investigations on the expressive power of HM($\mathcal{R}_p$) to future work.

## 7 Related work

In this section we discuss the relationship of the designed instances HM($\mathcal{R}^i$) to previous work. We point out some problems which are present in previous work. We restrict our attention to the work of Ohori [Oho95] and Gaster et al [GJ96]. Then, we show that under some sufficient conditions it is possible to reuse already existing approaches and compile them into the HM(X) framework.

### Ohori

We consider Ohori's calculus $\mathcal{O}$. We have based the design of HM($\mathcal{R}$) on this calculus. 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. 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 which assigns each type variable $\alpha$ its kind $k$. He writes $\mathcal{K}\{\alpha \mapsto k\}$ for the disjoint extension of $\mathcal{K}$ with a new type variable $\alpha$ with kind $k$. We can now write the following program in $\mathcal{O}$ where we assume that we have a function

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

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 1**

```
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 $\beta$. We can not quantify $\beta$ at the outermost level otherwise we would get a conflict with the conditions put on a kind assignment. Furthermore, $\beta$ should be a record type but there is no such restriction put on $\beta$. Actually, this program is non–sensical. We can type f but we can not apply f to an argument. The reason is that $\beta$ 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 otherwise $\mathcal{O}$ would not be sound. Here, one can see an example why in $\mathcal{O}$ every type variable has a kind wheres in HM($\mathcal{R}$) we only need to give kinds to records.

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

**Example 2**

f: $\forall\alpha.\exists\beta.(\alpha ::< l_1 : \beta >) \wedge (\beta ::< l_2 : \tau >) \Rightarrow \alpha \to \mathsf{Int}$
f x =
  let g: $\forall\beta.(\alpha ::< l_1 : \beta >) \wedge (\beta ::< l_2 : \tau >) \Rightarrow \beta \to \mathsf{Int}$
    g = $\lambda$ y. (x.$l_1$, (x.$l_1$).$l_2$, eq y (x.$l_1$))
  in 1

Especially, we get that HM($\mathcal{R}$) models $\mathcal{O}$ faithfully.

**Theorem 14** *(Faithfull) Every program typable in $\mathcal{O}$ is typable in HM($\mathcal{R}$).*

**Proof:** A proof can be found in the appendix. ■

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 $\mathcal{O}$.

### Gaster et al

We use $\mathcal{G}$ as an abbreviation for Gaster et al's record calculus. They base their record system on row variables. A row variable $r$ can be seen as a collection of fields. A predicate $(r\backslash l)$ expresses the absence of field label $l$ in row $r$. A record type is denoted by $Rec\, r$ where $Rec$ is a record constructor and $r$ is a row. Record extension is written as $Rec\{l : \tau | r\}$ where the operator $(\_|\_)$ extends a row with a new field. Because overloading of field labels is forbidden in $\mathcal{G}$ row $r$ has to fulfill the predicate $(r\backslash l)$.

In $\mathcal{G}$ typing judgments are written as $C, \Gamma \vdash e : \sigma$ where $C$ contains predicates of the form $(r\backslash l)$. In essence, the rule for quantifier introduction in $\mathcal{G}$ can be read as:

$$(\forall \text{ Intro-}\mathcal{G}) \quad \frac{C \wedge D, \Gamma \vdash e : \sigma \qquad \alpha \notin fv(\Gamma) \cup fv(C)}{C, \Gamma \vdash e : \forall\alpha.D \Rightarrow \sigma}$$

When we now type the program in Example 1 we get in an intermediate step that $y$ has type $Rec\ \{l_2 : \tau | r'\}$ and $x$ has type $Rec\ \{l_1 : Rec\ \{l_2 : \tau | r'\} | r\}$ where the rows $r$ and $r'$ have to fulfill the predicates $(r\backslash l_1)$ and $(r'\backslash l_2)$. In this case, we can not quantify over row $r'$ because $r'$ appears in the type of $x$. That means, rule ($\forall$ Intro-$\mathcal{G}$) is overly restrictive. We omit a discussion about the exact relationship between HM($\mathcal{R}$) (or HM($\mathcal{R}^i$)) and $\mathcal{G}$. But it should be straightforward to adapt $\mathcal{G}$ to get an instance HM($\mathcal{G}$). In the next section we discuss such a treatment in a more general context.

All problematic cases were constructed around polymorphic records which contain other polymorphic records. When we restrict our attention to polymorphic records which do not depend on other polymorphic records none of the problematic cases will apply. Let us discuss this in more detail. We consider Ohori's calculus $\mathcal{O}$. We assume that we have a record constraint $(\alpha ::< l : \tau >)$ where $\tau$ does not contain any polymorphic records. We use the HM(X) framework to argue that when we quantify over $\alpha$ it causes no problem to rule out the constraint $(\alpha ::< l : \tau >)$. The reason is that projection of such constraints is trivial. Because of the conditions put on $(\alpha ::< l : \tau >)$ we get that $\vdash^e \exists\alpha.(\alpha ::< l : \tau >)$. In this case, our ($\forall$ Intro) rule reduces to ($\forall$ Intro-$\mathcal{O}$). We get that if programs use only records of this special kind then HM($\mathcal{R}$) models $\mathcal{O}$ full.

**Theorem 15** *(Full and Faithfull) Given a program $p$ where polymorphic records do not depend on other polymorphic records. Then $p$ is typable in $\mathcal{O}$ iff $p$ is typable in HM($\mathcal{R}$).*

**Proof:** A discussion can be found in the Appendix F. ■

We have only considered $\mathcal{O}$. A similar argument would apply to $\mathcal{G}$. We can conclude that the HM(X) framework can be seen as the proper logical foundation for $\mathcal{O}$ and $\mathcal{G}$. This becomes important if we want to deal with cyclic dependencies such as recursive records or mutual recursive records. We argue that such features can not be incorporated into $\mathcal{O}$ and $\mathcal{G}$ without any serious restrictions whereas the HM(X) framework provides the proper logical foundation with even more expressive power. A more detailed discussion of the expressive power of HM($\mathcal{R}$) can be found in Appendix F.

### 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 16** *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 1*
- *(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 $\tau$ such that, $\exists\alpha.D =^e [\tau/\alpha]D$*

**Theorem 17** *(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.*

**Proof:** Because of the lifting condition we know that Lemmas 5 and 6 hold. Then we can perform the same proof steps as in Theorem 11. ■

In the last section we discussed the relationship between the original system and the lifted HM(X) instance. Namely, we showed that HM($\mathcal{R}$) models $\mathcal{O}$ faithfull and has even more expressive power. This might be interesting to examine in the general case. That means, are there any general conditions which describe whether the lifted constraint system has more expressive power than the original oneΓ We will pursue this topic in future work.

## 8 Conclusion

We have discussed several instances of the HM(X) framework. Namely, HM($\mathcal{R}^e$) (record extension), HM($\mathcal{R}^c$) (record concatenation) and HM($\mathcal{R}^v$) (variants). A language designer for record calculi can now choose some desired features and combine them in a new instance HM($\mathcal{R}^x$) where $x$ is a term in the regular grammar $[e|c|v]$. We also sketched the possiblitiy of HM($\mathcal{R}^r$) which might support recursive types. As one of the novelties of this work we introduced HM($\mathcal{R}_p$) where field labels are now first class values. All extensions can be extended with polymorphic field labels. We get instantiations HM($\mathcal{R}_p^x$) where $x$ is a term in the regular grammar $[e|c|v]$.

One motivation of this work is that we want to provide a full variety of record systems under one common core. In this context it might be worthwhile to consider some further applications. But this depends on the specific need for which we want to develop an application. The HM(X) framework enables us to concentrate on the design of constraint systems when designing a new type system. We have seen that we are now more concerned with the design of constraint systems which enjoy the principal constraint property. In case of constraint systems for records we could rely on the approach of Ohori. We adapted his constraint system and his normalization procedure (kinded unification). The main difference between his approach and ours is that we have a projection operator. In order to establish the principal constraint property we reduced normalization of constraints to the problem of normalization of projection free constraints. That means, we could rely on an already existing procedure (kinded unification) for normalization of projection free constraints. In general we could state the lifting condition under which we can lift already existing approaches to ours.

We have seen that HM(X) can be seen as the proper logical foundation of already existing approaches. We pointed out several problems which are present in the approaches of Ohori and Gaster et al. All these problems can be solved in HM(X). This is due to the projection operator which allows us to formulate a logically pleasing and pragmatically useful rule for quantifier introduction. We showed that we can faithfull model both approaches. Furthermore, we could show that our designed record systems have more expressive power than the approaches of Ohori and Gaster et al.

We did not discuss a compilation calculus. It should be straightforward to adapt the approach of Ohori. Regarding a practical implementation we think this is an important issue for future research.


## Acknowledgements

I would like to thank Stefan Monnier and Alastair Reid for helpful discussions about record systems and reading draft versions of this paper. Especially, I am grateful to Martin Odersky for pointing out Ohori's work, guiding me to new insights and helping in preparing this paper.


## References


[AK95] Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. *Acta Informatica*, 1995. To appear.

[Apo93] María Virginia Aponte. Extending record typing to type parametric modules with sharing. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 465–478. ACM Press, January 1993.

[BH97] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *Typed Lambda Calculi*

*and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '97, April 1997, Nancy, France, Proceedings*, April 1997.

[BPF97] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good "match" for object-oriented languages. In *FOOL4: 4th. Int. Workshop on Foundations of Object-oriented programming Languages*, January 1997.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, pages 51–67. Springer-Verlag, June 1984. Lecture Notes in Computer Science 173.

[CM89] Luca Cardelli and John C. Mitchell. Operations on records. In M. Main, A. Melton, M. Mislove, and David Schmidt, editors, *Proceedings Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA*, pages 22–52. Springer-Verlag, March/April 1989. Lecture Notes in Computer Science 442.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. January 1982.

[EST95] 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.

[GJ96] Benedict R. Gaster and Mark P. Jones. A Polymporphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, Nottingham NG7 2RD, UK, March 1996.

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

[HP91] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 131–142. ACM Press, January 1991.

[Jon92] Mark P. Jones. *Qualified Types: Theory and Practice.* D.phil. thesis, Oxford University, September 1992.

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structure. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 68–78, January 1996.

[Ken96] Andrew J. Kennedy. Type inference and equational theories. Technical Report LIX/RR/96/09, LIX, Ecole Polytechnique, 91128 Palaiseau Cedex, France, September 1996.

[OB88] Atshushi Ohori and Peter Buneman. Type inference in a database programming language. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 174–183, July 1988.

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

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. 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.

[Rém95a] Didier Rémy. A case study of typechecking with constrained types: Typing record concatenation. Presented at the workshop on *advances in types for computer science* at the Newton Institute, Cambridge, UK. Avaliable electronically at `http://pauillac.inria.fr/~remy`, August 1995.

[Rém95b] Didier Rémy. Refined subtyping and row variables for record types. Presented at the workshop on *advances in types for computer science* at the Newton Institute, Cambridge, UK. Avaliable electronically at `html://pauillac.inria.fr/~remy`, August 1995.

[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.

[SOW] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems.* invited submission; in preparation.

[SOW97] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *FOOL4: 4th. Int. Workshop on Foundations of Object-oriented programming Languages*, January 1997.

[Wan88] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, page 132, 1988.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 92–97, June 1989.

## A Variants

Additionally, we now want to deal with variants. We want to extend HM($\mathcal{R}$) with variants. We call the resulting system HM($\mathcal{R}^v$). We proceed similar in style as in Section 4. Variant types are denoted by $[l_1 : \tau_1, \ldots, l_n : \tau_n]$. A variant $[l = e]$ tagged with label $l$ can have types $[l_1 : \tau_1, \ldots, l_i : \tau_i, l : \tau, l_{i+1} : \tau_{i+1}, \ldots, l_n : \tau_n]$ where $\tau$ is the type of $e$. Therefore, we extend the term algebra $\mathcal{T}$ by adding constructors

$$l : \tau \to [l_1 : \tau_1, \ldots, l_i : \tau_i, l : \tau, l_{i+1} : \tau_{i+1}, \ldots, l_n : \tau_n]$$

to the signature $\Sigma$. We assume that the order of the variant fields does not matter. That means, we additionally require

**D8** $\vdash^e ([l_1 : \tau_1, \ldots, l_n : \tau_n] = [l_{\pi(1)} : \tau_{\pi(1)}, \ldots, l_{\pi(n)} : \tau_{\pi(n)}])$
where $\pi$ is a permutation on $\{1, \ldots, n\}$

We also introduce kinds for variant types. $\prec l_1 : \tau_1, \ldots, l_n : \tau_n \succ$ is a variant kind denoting a variant which contains a tagged value with type $\tau_i$ and label $l_i$. We assume that the order of the components of a variant kind does not matter. As usual we add constraints $(\tau :: k)$ where $\tau$ is a type and $k$ is a variant kind to the set $\Omega$ of primitive constraints.

**V1** $\vdash^e ([l_1 : \tau_1, \ldots, l_n : \tau_n] ::\prec l_1 : \tau_1, \ldots, l_{n+m} : \tau_{n+m} \succ)$
**V2** $(\tau ::\prec \ldots, l : \tau_1, \ldots \succ) \wedge (\tau ::\prec \ldots, l : \tau_2, \ldots \succ) \vdash^e (\tau_1 = \tau_2)$
**V3** $([\ldots, l : \tau_1, \ldots] ::\prec \ldots, l : \tau_2, \ldots \succ) \vdash^e (\tau_1 = \tau_2)$
**V4** $([\ldots, l : \tau, \ldots] ::\prec l_1 : \tau_1, \ldots, l_n : \tau_n \succ) \vdash^e \mathsf{false}$ where $l \neq l_i$
**V5** $(\tau_1 \to \tau_2 :: \tau_3) \vdash^e \mathsf{false}$
**V6** $(\alpha ::\prec l_1 : \tau_1, \ldots, l_n : \tau_n \succ) \wedge (\alpha ::< l : \tau >) \vdash^e \mathsf{false}$

The rest of the treatment is along the lines as in Section 4. It is straightforward to show that the resulting system enjoys also the principal constraint property.

## B Concatenation

We start with the term constraint system $\mathcal{R}$. We add primitive predicates $concat(\tau_1, \tau_2, \tau_3)$ to the set $\Omega$ of primitive predicates. We call the resulting system HM($\mathcal{R}^c$). We put the following conditions on $\mathcal{R}^c$:

**R6** $concat(\alpha, \beta, \gamma) \wedge (\alpha ::< l : \tau >) \vdash^e (\gamma ::< l : \tau >)$
**R7** $concat(\alpha, \beta, \gamma) \wedge (\beta ::< l : \tau >) \vdash^e (\gamma ::< l : \tau >)$
**R8** $\vdash^e concat(\{l_1 : \tau_1, \ldots, l_i : \tau_i\}, \{l_{i+1} : \tau_{i+1}, \ldots, l_n : \tau_n\}, \{l_1 : \tau_1, \ldots, l_n : \tau_n\})$
**R9** $concat(\alpha, \beta, \gamma) \wedge (\alpha ::< l : \tau >) \wedge (\beta ::< l : \tau >) \vdash^e \mathsf{false}$
**R10** $concat(\tau_1 \to \tau_1', \tau_2, \tau_3) \vdash^e \mathsf{false}$
**R11** $concat(\tau_1, \tau_2 \to \tau_2', \tau_3) \vdash^e \mathsf{false}$
**R12** $concat(\tau_1, \tau_2, \tau_3 \to \tau_3') \vdash^e \mathsf{false}$
**E7** $\exists\alpha, \gamma.concat(\alpha, \beta, \gamma) =^e \mathsf{true}$
**E8** $\exists\beta, \gamma.concat(\alpha, \beta, \gamma) =^e \mathsf{true}$

**Remark 7** *By condition* **R9** *we only allow concatenation of records with disjoint field labels. This choice is arbitrary. It would also be possible to model different behaviors of concatenation of records.*

It is straightforward to show that the resulting system satisfies the principal constraint property. We now add the following construct to the initial type environment $\Gamma_0$

$$concat : \forall\alpha, \beta, \gamma.concat(\alpha, \beta, \gamma) \Rightarrow \alpha \to \beta \to \gamma$$

which handles concatenation of records.

Furthermore, it is now possible to define record extension in terms of concatenation. We add primitive constructs

$$ext_l : \forall\alpha, \beta, \gamma.concat(\alpha, \{l : \beta\}, \gamma) \Rightarrow \alpha \to \beta \to \gamma$$

to the initial type environment. In order to define removal of a field label we need to put one additional

condition on $\mathcal{R}^c$:

**R13** $concat(\alpha, \{l:\beta\}, \gamma) \wedge (\gamma ::< l':\tau >) \vdash^e (\alpha ::< l':\tau >)$ where $l \neq l'$

Note, this condition models the same behavior as condition **R8** in Section 5. We can now add primitive constructs

$$rmv_l : \forall\alpha,\gamma.\exists\beta.concat(\alpha,\{l:\beta\},\gamma) \Rightarrow \gamma \to \alpha$$

to the initial type environment $\Gamma_0$. Primitive constructs $ext_l$ and $rmv_l$ have the same behavior as the primitive constructs $extend_l$ and $remove_l$ defined in Section 5.

## C Recursive types

We sketch how recursive types could be incorporated. Starting from HM($\mathcal{R}$) we discuss now an extension HM($\mathcal{R}^r$) which handles recursive types. Because we want to allow recursive records we omit condition **E6**. We add the recursion operator $\mu\alpha$ to the term algebra $\mathcal{T}$. Following the lines of [AK95] we put the following additional rules on $\mathcal{R}^r$:

**D8** $\vdash^e (\mu\alpha.\tau = [\mu\alpha.\tau/\alpha]\tau)$

**D9** $$\frac{\vdash^e (\tau_1 = [\tau_1/\alpha]\tau) \qquad \tau \text{ contractive in } \alpha}{\vdash^e (\mu\alpha.\tau = \tau_1)}$$

As usual, we put define the set $\mathcal{S}$ of solved forms as the greatest set which satisfies condition **S1 - S5**. It remains to establish the principal constraint property for HM($\mathcal{R}^r$). In this case we now have to extend unification over finite trees to unification over regular trees. We conjecture that this is possible following the line as described in [AK95]. We think this is an important issue. Also, it might be worthwile to consider a more recent approach [BH97] which uses a coinductive axiomatization of recursive type equality. We leave further investigations on this topic for future work.

## D Proof of Theorem 14 (Faithfull)

We now show that HM($\mathcal{R}$) models $\mathcal{O}$ faithfully. We prove that we can transform every typing judgment

$$\mathcal{K}, \Gamma \vdash^o e : \sigma$$

into a typing judgment

$$(\mathcal{K})^+, (\Gamma)^+ \vdash e : (\sigma)^+$$

where $\vdash^o$ is the derivation in $\mathcal{O}$ and $()^+$ stands for an appropriate transformation function.

We have already mentioned that $\mathcal{K}$ represents a kind assignment. In $\mathcal{O}$ every type variable is attached with a kind. A record type variable $\alpha$ can have kind $< l_1 : \tau_1, \ldots, l_n : \tau_n >$ and all other type variables have kind $U$.

We give now an inductive definition of the function $()^+$ on kind assignments $\mathcal{K}$. The function $()^+$ is a mapping from a kind assignment to a constraint in solved form.

$$\begin{array}{l}(\{\alpha \mapsto < l_1 : \tau_1, \ldots, l_n : \tau >\})^+ \\ = \bigwedge_{i\in\{1,\ldots,n\}} (\alpha ::< l_i : \tau_i >) \\ (\{\alpha \mapsto U\})^+ \\ = \mathsf{true} \\ (\mathcal{K}\{\alpha \mapsto k\})^+ \\ = (\mathcal{K})^+ \wedge (\{\alpha \mapsto k\})^+\end{array}$$

The function $()^+$ can be extended to type schemes $\sigma$ and type environment $\Gamma$ in a straightforward manner. We now give a more precise formulation of Theorem 14.

**Lemma 10** *Given a kind assignment $\mathcal{K}$, a type environment $\Gamma$, a term $e$ and a type scheme $\sigma$ such that $\mathcal{K}, \Gamma \vdash^o e : \sigma$. Then $(\mathcal{K})^+, (\Gamma)^+ \vdash e : (\sigma)^+$.*

**Proof:** We use induction over the derivation $\vdash^o$. We have already pointed out that there is essentially only one difference between the derivations $\vdash^o$ and $\vdash$. This difference lies in different rules for quantifier introduction. Therefore, we consider only one case. The other cases are similar. In $\mathcal{O}$ we have the following rule for quantifier introduction:

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

Applying the induction hypothesis we get

$$(\mathcal{K}\{\alpha \mapsto k\})^+, (\Gamma)^+ \vdash e : (\sigma)^+$$

We can now apply the ($\forall$ Intro) rule [3] and get

$$\exists\alpha.(\mathcal{K}\{\alpha \mapsto k\})^+, (\Gamma)^+ \vdash e : \forall\alpha.(\{\alpha \mapsto k\})^+ \Rightarrow (\sigma)^+$$

Because of the construction of a kind assignment we can deduce that

$$(\exists\alpha.(\mathcal{K})^+) \wedge (\exists\alpha.(\{\alpha \mapsto k\})^+) =^e \exists\alpha.(\mathcal{K}\{\alpha \mapsto k\})^+$$

Furthermore, we know that $\exists\alpha.(\{\alpha \mapsto k\})^+ =^e \mathsf{true}$. Then we get that

$$\begin{array}{lll}\exists\alpha.(\mathcal{K})^+ & =^e & \exists\alpha.((\mathcal{K})^+ \wedge \exists\alpha.(\{\alpha \mapsto k\})^+) \\ & =^e & (\exists\alpha.(\mathcal{K})^+) \wedge (\exists\alpha.(\{\alpha \mapsto k\})^+) \\ & =^e & \exists\alpha.(\mathcal{K}\{\alpha \mapsto k\})^+\end{array}$$

[3] The ($\forall$ Intro) in HM(X) is in one point a little bit more restrictive than the one in $\mathcal{O}$. We are only allowed to quantify over types and not type schemes. Actually, it would be possible to restate the ($\forall$ Intro) in such a way that we can now also quantify over type schemes.

We get that

$$(\mathcal{K})^+ \vdash^e \exists\alpha.(\mathcal{K}\{\alpha \mapsto k\})^+$$

and because $\vdash$ is closed under strengthening the constraint we get

$$(\mathcal{K})^+, (\Gamma)^+ \vdash e : \forall\alpha.(\{\alpha \mapsto k\})^+ \Rightarrow (\sigma)^+$$

which concludes the induction step. ∎

## E Proof of Theorem 15 (Full and Faithfull)

We rely on the notation introduced in the previous section. There, we have introduced a function $()^-$ which transforms a kind assignment into a solved constraint in $\mathcal{R}$. We give now a function $()^-$ which transforms a (restricted) solved constraint into a kind assigment.

We only consider constraints $C \in \mathcal{S}$ which contain only constraints of the form $(\alpha :: \_)$. In the latter we refer to such constraints as *simple* constraints. As usual $()^-$ is defined on the structure of $C$. We use a set $M$ to keep track of type variables. We assume $M$ which is initially empty.

$(D \wedge (\alpha ::< l_1 : \tau_1 >) \wedge \ldots \wedge (\alpha ::< l_n : \tau_n >))^-$
$= (D)^-\{\alpha \mapsto< l_1 : \tau_1, \ldots, l_n : \tau_n >\}$
where in $D$ there are no constraints of the form $(\alpha :: \_)$ and we set $M = M \cup \{\alpha\}$
$(\mathsf{true})^-$
$= \{\alpha_1 \mapsto U, \ldots \alpha_n \mapsto U\}$
where $\alpha_i \notin M$ and $\alpha_i \in Var$

It is straightforward to prove the next lemma.

**Lemma 11** *Given a simple constrant $C$ and a kind assignment $\mathcal{K}$. Then it holds that*

$$((C)^-)^+ = C \qquad ((\mathcal{K})^+)^- = \mathcal{K}$$

Then, we can now state the following lemma.

**Lemma 12** *Given a typing judgment $C, \Gamma \vdash^o e : \sigma$ such that all constraints are simple. Then $(C)^-, (\Gamma)^- \vdash e : (\sigma)^-$.*

**Proof:** Straightforward inductive proof. ∎

Theorem 15 follows now immediately as a corollary from Lemmas 10 and 12. A categorical proof of Theorem 15 can be found in the next section.

## F Expressive power of HM($\mathcal{R}$)

We give now a categorical proof that HM($\mathcal{R}$) has more expressive power than $\mathcal{O}$. We rely on the notation which we introduced in the two previous sections.

We want to consider a type system as a category. A typing judgment represents a valid derivation in $\mathcal{TS}$. A premise and a conclusion stands for a collection of typing judgments.

**Lemma and Definition 1** *Given a type system $\mathcal{TS}$. We consider $\mathcal{TS}$ as a category where objects are a collection of typing jugdments and arrows represent the derivation of a premise to a conclusion with respect to the typing rules in $\mathcal{TS}$.*

We are now ready to define when two type systems have the same expressive power.

**Definition 18** *Given two type systems $\mathcal{TS}$ and $\mathcal{TS}'$. We say $\mathcal{TS}$ and $\mathcal{TS}'$ have the same* expressive power *iff there is a functor $F : \mathcal{TS} \to \mathcal{TS}'$ which has a left adjoint functor $G : \mathcal{TS}' \to \mathcal{TS}$.*

*We write $\mathcal{TS} \cong \mathcal{TS}'$ in such a case.*

**Lemma 13** *$\cong$ is an equivalence relation.*

First, we show that HM($\mathcal{R}^-$) $\cong \mathcal{O}$ were $\mathcal{R}^-$ is the restriction of $\mathcal{R}$ to simple constraints. It is straightforward to consider $()^+$ and $()^-$ as functors. Then we can state the following lemma.

**Lemma 14** *Given $X \in HM(\mathcal{R}^-)$ and $Y \in \mathcal{O}$. Then $X \to (Y)^+$ iff $(X)^- \to Y$*

**Proof:** Use Lemmas 10, 12 and 11. ∎

Based on this lemma it is straightforward to prove that $((()^-, ()^+)$ form an adjunction. We can summarize this observation in the following theorem.

**Theorem 19** *$HM(\mathcal{R}^-) \cong \mathcal{O}$.*

We are now ready to state our main theorem.

**Theorem 20** *$HM(\mathcal{R})$ has more expressive power than $\mathcal{O}$.*

**Proof:** We show that HM($\mathcal{R}$) $\ncong$ HM($\mathcal{R}^-$). Then it follows immediately that HM($\mathcal{R}$) $\ncong \mathcal{O}$.

Assume HM($\mathcal{R}^-$) $\cong$ HM($\mathcal{R}$) W.l.o.g. we can assume that we have an adjunction $(Id, G)$. That means, given $X \in$ HM($\mathcal{R}$), HM($\mathcal{R}^-$) and $X \to Y$ there must be $G(X) \to Y$. $X \to Y$ represents a derivation in HM($\mathcal{R}$). It is not difficult to construct a derivation $X \to Y$ such that there is no corresponding derivation $G(X) \to Y$ in HM($\mathcal{R}^-$), i.e. take Example 1. ∎