DRAFT

Type Systems for Records revisited

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

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. 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 [OSW]. 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.

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

[*]Supported by DARPA Grant F30602-96-2-0232.

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.

2 Overview

We discuss an instance HM(REC) 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 τ_i for the associated type. We introduce a constraint system REC where we can express constraints on record types. For instance, the constraint $(\alpha :: \{\!\{l : \tau\}\!\})$ states that α is a record which contains at least a field with label l and type τ. 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 REC and $(=)$ is the equality predicate. By condition **R2** we forbid overloading of field labels. In HM(REC) 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 Γ_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]

$$(\{\mathit{Address} : \texttt{string}\}.\mathit{Name})$$

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

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

$$(\{\mathit{Address} : \texttt{string}\}.\mathit{Address})$$

type inference in HM(REC) generates the constraint problem

$$C = (\alpha :: \{\!\{\mathit{Address} : \beta\}\!\}) \wedge (\alpha = \{\mathit{Address} : \texttt{string}\})$$

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

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

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(REC), we have to show that REC satisfies the principal constraint property.

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

$$\mathit{extend}_l : \forall \alpha, \beta, \gamma.\mathit{extend}_l(\alpha, \beta, \gamma) \Rightarrow \alpha \rightarrow \beta \rightarrow \gamma$$

handles extension of records with field label l. We have not specified the exact behavior of the constraint $\mathit{extend}_l(\alpha, \beta, \gamma)$. 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

$$\mathit{remove}_l : \forall \alpha, \gamma.\exists \beta.\mathit{extend}_l(\alpha, \beta, \gamma) \Rightarrow \gamma \rightarrow \alpha$$

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 $\exists\beta.\mathit{extend}_l(\alpha,\beta,\gamma)$ expresses that record γ contains a field with label l which is not present in α. The type of the field label l is not of interest in this context. Therefore, β is bound by the projection operator and hidden from outside.

Another extension HM(REC(c)) uses constraints of the form $\mathit{concat}(\alpha,\beta,\gamma)$ to model record concatenation. We will see that in some sense HM(REC(c)) subsumes HM(REC(e)). Furthermore, we consider also an extension that deals with recursive records.

In the previous record systems we treated field labels always as fixed constants. We introduce an application HM(REC(p)) where field labels become first class values. For instance, we can now write a function that takes a field label as an argument:

```
f l x = x.l
```

The function f takes a label and a record and selects that label from the given record. This is essentially the record selection operator extended to first class field labels. In a later section we will see how to type function f.

We always use the same methodology to model some desired features. We introduce some appropriate primitive constraints like $(\alpha :: \{\!\{l:\beta\}\!\})$ and $\mathit{extend}_l(\alpha,\beta,\gamma)$. 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 $r\ \mathit{has}\ l$ 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. It is easy to reuse these approaches and encode their systems in a terms of a constraint system. Then we obtain a new record system in terms of an HM(X) application.

We start in Section 3 with an overview of the HM(X) framework. Section 4 considers an HM(X) applications in style of Ohori's record system. Two extensions of this record system are considered in Section 5 and 6.

3 The HM(X) framework

We now review the basic components of the HM(X) framework introduced in [SOW97]. The interested reader can find a detailed development in [OSW].

Types are members of an arbitrary term algebra $\mathcal{T}$ where there might be other constructors besides $\rightarrow$.

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

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

$$\text{(App)}\qquad \frac{C,\Gamma \vdash e_1:\tau_1\rightarrow\tau_2 \qquad C,\Gamma \vdash e_2:\tau_1}{C,\Gamma \vdash e_1e_2:\tau_2}$$

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

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

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

Figure 1: Logical type system

Type schemes include a constraint component C which restricts the types that can be substituted for the type variable α.

$$\begin{array}{lll} \textbf{Types} & \tau & ::\supseteq\ \alpha \mid \tau\rightarrow\tau \\ \textbf{Type schemes} & \sigma & ::=\ \tau \mid \forall\alpha.C\Rightarrow\sigma \end{array}$$

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.

$$\begin{array}{lll} \textbf{Values} & v & ::=\ x \mid \lambda x.e \\ \textbf{Expressions} & e & ::=\ v \mid e\,e \mid \mathsf{let}\ x=e\ \mathsf{in}\ e \end{array}$$

Typing judgments are of the form $C,\Gamma \vdash e:\sigma$ where C is in X, Γ is a type environment and σ is a type scheme. Furthermore, we restrict the set of constraints C that can appear in type schemes and on the left hand side of the turnstile to so–called *solved forms*. The set of solved forms, denoted by $\mathcal{S}$, is always a subset of the satisfiable constraints in X. A constraint C is *satisfiable* iff $\vdash^e \exists \mathit{fv}(C).C$ where the entailment relation $\vdash^e$ between constraints is defined by the constraint system X.

The typing rules can be found in Figure 1. Most rules are straightforward extensions of the standard Hindley/Milner rules. The formulation of the ($\forall$ Elim) rule is similar to previous formulations. The only valid instances of a type scheme $\forall\alpha.D\Rightarrow\sigma$ are those that satisfy the constraint part of the type scheme. The term $C\vdash^e [\tau/\alpha]D$ states that the constraint C entails the constraint obtained by substituting type variable α with type τ in D. Substitutions on constraints is connected to projection of constraints. In constraint system X we require that $[\tau/\alpha]D =^e \exists\alpha.((\alpha = \tau) \wedge D)$ where $=$ is type equality and satisfies at least the conditions put on a congruence. It follows immediately that $[\tau/\alpha]D \vdash^e \exists\alpha.D$ holds for any type τ. We have already pointed out the novel formulation of the ($\forall$ Intro) rule. We will discuss the relationship to previous formulations in later sections.

We can also give a soundness proof for HM(X) type systems based on an untyped compositional semantics. Due to space limitations we omit to give a detailed description. The interested reader can find a detailed description in [OSW]. Milner's catch–phrase can now be restated for Hindley/Milner type systems extended with constraints: "well–typed programs can not go wrong".

The Hindley/Milner system is an instance of our type system framework. Take X to be the Herbrand constraint system HERBRAND over the algebra of types τ. HERBRAND consists only of primitive constraints of the form $(\tau = \tau')$ where τ and τ' are elements of a term algebra $\mathcal{T}$. Equality in HERBRAND is syntactic, $\mathcal{T}$ is a free algebra. Take the set of solved forms to be the set consisting only of true, which is represented by the empty token set. Then the only type schemes arising in proof trees of valid typing judgments are of the form $\forall\alpha.\{\} \Rightarrow \sigma$, which we equate with Hindley/Milner type schemes $\forall\alpha.\sigma$. It is easy to convince oneself that a judgment $\Gamma \vdash e : \sigma$ is derivable in Hindley/Milner if and only if $\{\}, \Gamma \vdash e : \sigma$ is derivable in HM(HERBRAND).

Further applications are discussed in the following sections.

4 Polymorphic records

Following ideas of Ohori [Oho95] we give an instance of our HM(X) system which deals with polymorphic records. Ohori's system, abbreviated REC in the following, has besides type variables and function types also record types denoted by $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, where l_i is an element of an enumerable set of record labels. We assume that there is an ordering relation between all field labels. All record fields are ordered with respect to this ordering relation. Because we have a fixed ordering of record fields we can apply Herbrand unification for solving equality constraints between records.

Type quantification in REC is kinded; in the type scheme $\forall\alpha.\alpha :: \kappa \Rightarrow \sigma$ the type variable α ranges only over kind κ. A kind is of the form $\{\!\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\!\}$; it comprises all records that contain at least fields $l_1, \ldots, l_n$ with types $\tau_1, \ldots, \tau_n$.

Instead of a constraint on the left hand side of a typing judgment, Ohori uses a *kind assignment* $\mathcal{K}$ which can be considered as a function which assigns each type variable α its kind k. He writes $\mathcal{K} \wedge (\alpha :: k)$ for the disjoint extension of $\mathcal{K}$ with a new type variable α with kind k.

Here's an example of a program typed in REC.

Example 1

```
f: $\forall\alpha,\beta.(\alpha :: \{\!\{l : \beta\}\!\}) \Rightarrow \alpha \to \mathsf{Int}$
f x =
    let g: $\beta \to \mathsf{Bool}$
        g = $\lambda$ y. eq y (x.$l$)
    in 1
```

We use a Haskell-style notation, with type scheme annotations added for illustration purposes. The program assumes that there is a function

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

in the initial type environment.

4.1 Type system

We now translate REC into the HM(X) framework. We add to the initial type environment Γ_0 primitive constructs that deal with record formation, selection and update. For every ordered sequence of record labels $l_1, \ldots, l_n$ we postulate an n-ary parameterized data type $R_{l_1\_\ldots\_l_n}$. The record type $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ is then represented as $R_{l_1\_\ldots\_l_n}\tau_1 \ldots \tau_n$. For simplicity we will keep using the record type notation as a synonym for the datatype notation. For every record datatype $R_{l_1\_\ldots\_l_n}$ we have in the initial environment a datatype constructor

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

Then, $l_1\_\ldots\_l_n\, e_1 \ldots e_n$ represents record formation $\{l_1 = e_1, \ldots, l_n = e_n\}$. For each field label l we add to the initial type environment Γ_0 the two functions

$$\begin{aligned}(\_.l) &: \forall\alpha,\beta.(\alpha :: \{\!\{l : \beta\}\!\}) \Rightarrow \alpha \to \beta \\ \mathit{modify}_l &: \forall\alpha,\beta.(\alpha :: \{\!\{l : \beta\}\!\}) \Rightarrow \alpha \to \beta \to \alpha\end{aligned}$$

The first function corresponds to record selection, the second to record update.

Kinded quantification in REC is modeled by primitive 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. We define REC as the smallest term constraint system that satisfies the following additional rules:

REC1 $\vdash^e$ $(\{l_1 : \tau_1, \ldots l_n : \tau_n\} :: \{\!\!\{l_i : \tau_i\}\!\!\})$
where $l_1, \ldots, l_n$ are distinct
REC2 $(\tau :: \{\!\!\{l : \tau_1\}\!\!\}) \wedge (\tau :: \{\!\!\{l : \tau_2\}\!\!\}) \vdash^e (\tau_1 = \tau_2)$

The following are derived rules:

REC3 $(\{\ldots, l : \tau_1, \ldots\} :: \{\!\!\{l : \tau_2\}\!\!\}) \vdash^e (\tau_1 = \tau_2)$
REC4 $\exists\alpha.(\alpha :: k) =^e$ true
where $\alpha \notin fv(k)$

Note that these conditions rule out recursive records, since our type algebra does not have recursive types. On the other hand, we do allow recursive constraints between type variables in REC. For instance, the constraint $(\alpha :: \{\!\!\{l : \alpha \to \alpha\}\!\!\})$ is well-formed. But that constraint is not satisfiable and therefore cannot appear as a solved form. Also ruled out (by conditions **REC2** and **REC3**) is overloading of field labels.

The set $\mathcal{S}$ of solved forms in HM(REC) consists of all satisfiable constraints of the form

$$C \quad ::= \quad \{\} \,|\, (\alpha :: \{\!\!\{l : \tau\}\!\!\}) \,|\, C \wedge C \,|\, \exists\bar{\alpha}.C$$

where we take the empty token set as a representation of true. Furthermore, we require that the constraints in $\mathcal{S}$ are in simplified form, i.e. $C \vdash^e (\tau = \tau')$ must imply $\vdash^e (\tau = \tau')$. For instance,

$$(\alpha :: \{\!\!\{l : \beta\}\!\!\}) \wedge (\alpha :: \{\!\!\{l : \gamma \to \gamma\}\!\!\})$$

is not in simplified form and is therefore excluded.

The type system HM(REC) is as given in Figure 1. As an example, here the annotated program from Example 1 re-formulated in HM(REC):

Example 2

```
f: ∀α.(∃β.(α :: {{l : β}})) ⇒ α → Int
f x =
  let g : ∀β.(α :: {{l : β}}) ⇒
          β → Bool
      g = λ y. eq y (x.l)
  in 1
```

In HM(REC) we quantify in the innermost let over type variable β, leaving just α to be quantified in the toplevel function f. This is not possible in REC, since α's kind depends on β. The question arises whether this makes HM(REC) a more permissive type system than REC. Specifically, are there examples where we can use function g polymorphicallyΓ The answer is no. Every instance of g has to satisfy the constraint $\exists\beta.(\alpha :: \{\!\!\{l_1 : \beta\}\!\!\})$. But α can only have one field entry with label l_1. Therefore, we can use g in the let-body only monomorphically. In general, we can observe that REC and HM(REC) type exactly the same programs, but the types are more precise in HM(REC).

Theorem 1 (Full and Faithful) *Every program typable in* REC *is typable in* HM(REC) *and vice versa.*

4.2 Type inference

We now consider type inference for HM(REC). Since REC is a regular constraint system, we can obtain type inference with principal types, provided it fulfills the principal constraint property. To show the principal constraint property for REC, we proceed in three steps. First, we show that it is always possible to formulate a constraint as a projection over a projection–free subpart. A constraint D is *projection–free* if D (considered as a set) contains only tokens of the form $(\alpha :: k)$ and $(\tau = \tau')$. Then we give a procedure which computes the principal normal form of projection–free constraints, or fails if no normal form exists. Finally, we show that it is sufficient to compute principal normal forms of projection–free constraints. This is achieved by a lifting method. Given an arbitrary constraint C we compute the principal normal form of the projection–free part. Then we lift this result to the projected part. We show that this lifting method is sound and complete. Detailed proofs can be found in the appendix.

In a first step we transform a constraint into a projection over 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 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 lift all projection operators to the outermost level using condition **E3** of a cylindric constraint system:

$$(\exists\alpha.C) \wedge (\exists\beta.D) \;=^e\; \exists\alpha.(\exists\beta.(C \wedge D))$$

We can summarize these observations in the following lemma.

Lemma 1 *Let $C \in$ REC. Then there exists a projection–free constraint D such that $C =^e \exists\bar{\alpha}.D$.*

In the next step we show how to compute principal normal forms for projection–free constraints. We assume that we have a projection–free constraint D which contains only primitive predicates of the form $(=)$ and $(::)$. W.l.o.g., we can assume that all predicates $(::)$ are of the form $(\alpha :: k)$. This can be achieved 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 only have changed the syntactic representation of D. The intention of building the closure of D is to generate all predicates $(\tau :: \{\!\{l : \tau'\}\!\})$ which might cause any inconsistencies. Given all such predicates 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 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 $D \vdash^e (\tau = \tau')$.*

We can apply unification over Herbrand terms [Rob65] to resolve all equality predicates $(=)$ in $Cl(D)$. We obtain 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 $Cl(D)$. This can be done by checking 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 $D \in$ REC and a substitution ϕ. Then (D, ϕ) has a principal normal form, which can be computed by the procedure described above, or else no normal form exists.*

It remains to lift this procedure to arbitrary constraints. First, we state some essential lemmas that are necessary to establish this lifting method. Then we apply this lifting method to state that REC satisfies the principal constraint property.

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 4 *Let D be a projection–free constraint in REC and ϕ be a substitution where $\bar{\alpha} \notin codom(\phi) \cup dom(\phi)$. If $(C, \psi) = normalize(D, \phi)$ then $(\exists\alpha.C, \psi_{\setminus\{\bar{\alpha}\}}) = normalize(\exists\bar{\alpha}.D, \phi)$.*

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

Lemma 5 *Given a substitution ϕ where $\bar{\alpha} \notin codom(\phi) \cup dom(\phi)$ and a projection–free constraint $D \in$ REC. Then (D, ϕ) has a normal form iff $(\exists\alpha.D, \phi)$ has a normal form.*

We have now everything at hand to prove that REC satisfies the principal constraint property. The proof of the theorem consists in describing a method how to lift computation of principal normal forms for projection–free constraints to arbitrary constraints. Details can be found in the appendix.

Theorem 2 *The constraint system REC satisfies the principal constraint property.*

Proof: Given an arbitrary constraint problem (D, ϕ) where $D =^e \exists\bar{\alpha}.D'$ such that D' is projection–free. We consider two cases.

First, assume (D, ϕ) has no normal form. Because of Lemma 5 we know that (D', ϕ) does not have a normal form either.

Now, assume (D, ϕ) does have a normal form. We apply Lemma 5 and find that the normal form of (D', ϕ) exists. By assumption we know how to normalize (D', ϕ). That means (D', ϕ) does have a principal normal form and we can compute its principal normal form. With Lemma 4 we can lift the principal normal form of the projection–free constraint problem and obtain the principal normal form of (D, ϕ) exists.

We can conclude that X has the principal constraint property.

■

Usually, the principal constraint property does not constitute a decidable *normalize* function. In case of REC we find that the proof of the above theorem is constructive. Hence, we obtain a decidable procedure that computes principal normal forms or reports failure.

5 Extensible records

The record calculus of Ohori lacks some important features, e.g. we do not have extensible records. We show now how to extend HM(REC) to obtain HM(REC(e)) where we additionally have record extensions.

Starting from REC we give an extension that is able to deal with record extension. We call it REC(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 Ω. REC(e) is the smallest term constraint system that fulfills the following rules:

REC1 $\vdash^e (\{l_1 : \tau_1, \ldots l_n : \tau_n\} :: \{\!\{l_i : \tau_i\}\!\})$
where $l_1, \ldots, l_n$ are distinct
REC2 $(\tau :: \{\!\{l : \tau_1\}\!\}) \wedge (\tau :: \{\!\{l : \tau_2\}\!\}) \vdash^e (\tau_1 = \tau_2)$
REC3 $\mathit{extend}_l(\alpha, \beta, \gamma) \vdash^e (\gamma :: \{\!\{l : \beta\}\!\})$
REC4 $\mathit{extend}_l(\alpha, \beta, \gamma) \wedge (\alpha :: \{\!\{l' : \tau\}\!\}) \vdash^e (\gamma :: \{\!\{l' : \tau\}\!\})$
REC5 $\mathit{extend}_l(\alpha, \beta, \gamma) \wedge (\gamma :: \{\!\{l' : \tau\}\!\}) \vdash^e (\alpha :: \{\!\{l' : \tau\}\!\})$ where $l \neq l'$
REC6 $\vdash^e \mathit{extend}_l(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}, \tau, \{l_1 : \tau_1, \ldots, l_n : \tau_n, l : \tau\})$
where $l_i \neq l$

Remark 1 *Condition* **REC6** *expresses that we can only extend a record with a field if this field is not already present in the record. Jones [Jon92] uses a predicate $(\alpha \backslash l)$ to express that record α does not contain a field label l. We can define $(\alpha \backslash l)$ as an abbreviation for $\exists \beta, \gamma.\mathit{extend}_l(\alpha, \beta, \gamma)$. Furthermore, we also find the derived constraint rules mentioned in the previous section.*

The set $\mathcal{S}$ of solved forms in HM(REC) consists of all satisfiable and simplified constraints of the form

$$\begin{array}{lll} C & ::= & \{\} \mid (\alpha :: \{\!\{l : \tau\}\!\}) \mid \mathit{extend}_l(\tau_1, \tau_2, \tau_3) \\ & & C \wedge C \mid \exists \bar{\alpha}.C \end{array}$$

where either τ_1 or τ_2 is a type variable.

We need now primitive constructs in the initial type environment Γ_0 for record extension. For each field label l we add

$$\mathit{extend}_l : \forall \alpha, \beta, \gamma.\mathit{extend}_l(\alpha, \beta, \gamma) \Rightarrow \alpha \to \beta \to \gamma$$

to the initial type environment Γ_0. The construct

$$\mathit{remove}_l : \forall \alpha, \gamma.\exists \beta.\mathit{extend}_l(\alpha, \beta, \gamma) \Rightarrow \gamma \to \alpha$$

handles removal of the field label l.

In the appendix we consider type inference for this application. Furthermore, we consider other extensions that deal with record concatenation and recursive records.

6 Polymorphic field labels

In the previous application, field lables were considered as constants. We extend the term algebra in REC. We distinguish now between labels and types:

$$\begin{array}{llll} \textbf{Labels} & l & ::\supseteq & l_\alpha \mid l_c \\ \textbf{Records} & r & ::\supseteq & \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\ \textbf{Types} & \tau & ::= & \alpha \mid \mathit{label}(l) \mid r \mid \tau \to \tau \end{array}$$

where r stands for a $\{l_1 : \tau_1, \ldots, l_n : \tau\}$ with constant field labels $l_1, \ldots, l_n$.

The predicate $(r :: \{\!\{l : \tau\}\!\})$ takes now a record r, a label l and a type τ as its arguments. The constraint systems REC(p) fulfills the following rules:

REC1 $\vdash^e (\{l_1 : \tau_1, \ldots l_n : \tau_n\} :: \{\!\{l_i : \tau_i\}\!\})$
where $l_1, \ldots, l_n$ are distinct
REC2 $(\tau :: \{\!\{l : \tau_1\}\!\}) \wedge (\tau :: \{\!\{l : \tau_2\}\!\}) \vdash^e (\tau_1 = \tau_2)$

We find the following derived rule:

REC3 $(\{l_1 : \tau_1, \ldots, l_n : \tau_n\} :: l : \tau) \vdash^e (l = l_i)$
for some $i \in \{1, \ldots, n\}$

All other previously derivable rules still hold.

The set $\mathcal{S}$ of solved forms in HM(REC(p)) consists of all satisfiable and simplified constraints of the form

$$\begin{array}{lll} C & ::= & \{\} \mid (\alpha :: \{\!\{l : \tau'\}\!\}) \mid \exists l.(\{\bar{l} : \bar{\tau}\} :: \{\!\{l : \tau'\}\!\}) \\ & \mid & C \wedge C \mid \exists \bar{\alpha}.C \end{array}$$

where we take the empty token set as a representation for true. Note, we find additonally $\exists l.(\{\bar{l} : \bar{\tau}\} :: \{\!\{l : \tau'\}\!\})$ as a constraint in solved form. This holds for the following reason. Consider the record $\{l_1 : \tau_1, l_2 : \tau_2\}$. Then the constraint $(\{l_1 : \tau_1, l_2 : \tau_2\} :: \{\!\{l : \alpha\}\!\})$ is not in solved form because for some $i = 1, 2$ we find that $(\{l_1 : \tau_1, l_2 : \tau_2\} :: \{\!\{l : \alpha\}\!\}) \vdash^e (l_i = l)$ and this contradicts that the constraint has to be in simplified form. But the constraint $\exists l.(\{l_1 : \tau_1, l_2 : \tau_2\} :: \{\!\{l : \alpha\}\!\})$ is in solved form because it holds that $\vdash^e \exists l_i.(l_i = l)$ for any $i \in \{1, 2\}$.

To give an idea of the expressiveness of REC(p) we show how to encode a kind of intersection type in REC(p). Consider the following type

$$\sigma = \forall \alpha.(\exists l.(\{l_1 : \tau_1, l_2 : \tau_2\} :: \{\!\{l : \alpha\}\!\})) \Rightarrow \alpha$$

where $_1$ and l_2 are some constant field labels. The only possible instances of σ are τ_1 or τ_1. That means, σ can be seen equivalent to the type $\tau_1 \wedge \tau_2$ in an intersection type discipline.

We have now the following primitives in the initial type environment Γ_0:

$$\begin{array}{lll} (\_.\_) & : & \forall \alpha, \beta, l. \\ & & (\alpha :: \{\!\{l : \beta\}\!\}) \Rightarrow \\ & & \alpha \to \mathit{label}(l) \to \beta \\ \\ \mathit{modify} & : & \forall \alpha, \beta, l. \\ & & (\alpha :: \{\!\{l : \beta\}\!\}) \Rightarrow \\ & & \alpha \to \mathit{label}(l) \to \beta \to \alpha \end{array}$$

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.

Application We present some applications of records with polymorphic field labels. We show how to type type–selective functions based on a record calculus with polymorphic field labels. For a type–selective function we do not have to specify the order of the arguments if the argument types are distinct. We can apply the arguments in any order and the function is able to select the right argument depeding on its type. A similiar idea can be used for label–selective function application [GAK94].

Assume we have a function f which takes as arguments types $\tau_1, \dots, \tau_n$ and returns τ. Furthermore, we assume the types $\tau_1, \dots, \tau_n$ are distinct. In this context, distinct means that not any two types τ_i, τ_j are unifiable. We define the record r by $\{l_1 : \tau_1, \dots, l_n : \tau_n\}$ where $l_1, \dots, l_n$ are some constant field labels. Then we can give f the following type.

$$\begin{aligned} f : \forall\bar{\beta}.\ \ & \exists l'_1, \dots, l'_n.(r :: \{\!\{l'_1 : \beta_1\}\!\}) \wedge \dots \wedge (r :: \{\!\{l'_n : \beta_n\}\!\}) \wedge \\ & \mathit{distinct}(l'_1, \dots, l'_n) \Rightarrow \beta_1 \to \dots \to \beta_n \to \tau \end{aligned}$$

We need the constraint $\mathit{distinct}(l'_1, \dots, l'_n)$ to avoid that the same argument can be applied twice. The constraint *distinct* takes as arguments only field labels. The rules are as follows:

$$\vdash^e \vdash^e \ \mathit{distinct}(l_1, l_2)$$
where l_1, l_2 are distinct, constant field labels

The term $\mathit{distinct}(l_1, \dots, l_n)$ is a short–hand for

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

We consider now a specific type–selective function. For instance, think of a function which takes as argument a list of arbitary type and a tree datatype and returns an integer. Both argument types are distinct. In a type–selective calculus we could type such a function as follows

$$\begin{aligned} f : \forall\alpha,\beta,\gamma,\delta.\ \ & \exists l'_1, l'_2 : (r :: \{\!\{l'_1 : \alpha\}\!\}) \wedge (r :: \{\!\{l'_2 : \beta\}\!\}) \wedge \\ & \mathit{distinct}(l'_1, l'_2) \Rightarrow \alpha \to \beta \to \mathsf{Int} \end{aligned}$$

where l_1, l_2 are constant field labels and $r = \{l_1 : [\gamma], l_2 : \mathsf{Tree}\,\delta\}$. Another example would be the well–known fold function. Usually, typed as follows

$$\mathit{fold} : (a \to b \to a) \to [b] \to a \to a$$

The arguments types $a \to b \to a$, $[b]$ and a are distinct. Therefore, it is also possible type this function in such a way where we do not have to specify the order of the arguments.

A similar idea can be used to consider a label–selective calculus. Such a calculus has already been introduced by Garrigue/Aït-Kaci [GAK94]. The basic idea here is to attach additionally labels to the arguments. That means essentially the arguments types form a record but we can perform curried function application. For instance, consider the exponentiation function where we attach labels to distinguish between the base and the exponent.

$$\mathit{exp}\ (\mathit{base}, x)\ (\mathit{exponent}, y) \ = \ y^x$$

In the record calculus with polymorphic field labels we could give *exp* the following type

$$\begin{aligned} \mathit{exp} : \forall\alpha,\beta,l_1,l_2.\ \ & (r :: \{\!\{l_1 : \alpha\}\!\}) \wedge (r :: \{\!\{l_2 : \beta\}\!\}) \wedge \\ & \mathit{distinct}(l_1, l_2) \Rightarrow (l_1, \alpha) \to (l_2, \beta) \to \mathsf{Int} \end{aligned}$$

where *base*,*exponent* are constant field labels and $r = \{\mathit{base} : \mathsf{Int}, \mathit{exponent} : \mathsf{Int}\}$.

We only sketched how a record calculus with polymorphic field lables could be a basis for a type– or label–selective calculus. We leave it to future research to investigate this topic more in detail.

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 constrained types and its application to object oriented programming. In *Electronic Notes in Theoretical Computer Science*, volume 1, 1995.

[GAK94] Jacques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective lambda-calculus. In *Principles of Programming Languages*, Portland, 94.

[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.

[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.

[OSW] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems.* to appear.

[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.

[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.

[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.

.1 Extensible records

We now have to show that REC(e) has the principal constraint property. We proceed in a similar way as for REC. 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 6 *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 proceed as before and we can state the following lemma.

Lemma 7 *Given a projection free constraint $D \in$ REC and a substitution ϕ. Then (D, ϕ) does have a principal normal form or no normal form exits.*

We can make now the following observations. Lemmas 1, 4, 5 also hold in REC(e). Then it is straightforward to show that REC(e) also satisfies the principal constraint property.

Theorem 3 *The constraint system REC(e) has the principal constraint property.*

Proof: Same argumentation as in Theorem 2. ■

A Concatenation

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

REC1 $\vdash^e (\{l_1 : \tau_1, \ldots l_n : \tau_n\} :: \{\!\{l_i : \tau_i\}\!\})$ where $l_1, \ldots, l_n$ are distinct
REC2 $(\tau :: \{\!\{l : \tau_1\}\!\}) \wedge (\tau :: \{\!\{l : \tau_2\}\!\}) \vdash^e (\tau_1 = \tau_2)$
REC3 $concat(\alpha, \beta, \gamma) \wedge (\alpha :: \{\!\{l : \tau\}\!\}) \vdash^e (\gamma :: \{\!\{l : \tau\}\!\})$
REC4 $concat(\alpha, \beta, \gamma) \wedge (\beta :: \{\!\{l : \tau\}\!\}) \vdash^e (\gamma :: \{\!\{l : \tau\}\!\})$
REC5 $\vdash^e concat(\{l_1 : \tau_1, \ldots, l_i : \tau_i\},$
$\{l_{i+1} : \tau_{i+1}, \ldots, l_n : \tau_n\},$
$\{l_1 : \tau_1, \ldots, l_n : \tau_n\})$

Remark 2 *By condition* **REC5** *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.*

The set $\mathcal{S}$ of solved forms in HM(REC) consists of all satisfiable and simplified constraints of the form

$$C \quad ::= \quad \{\} \mid (\alpha :: \{\!\{l : \tau\}\!\}) \mid concat(\tau_1, \tau_2, \tau_3) \quad C \wedge C \mid \exists \bar{\alpha}.C$$

where at least one of the types τ_1, τ_2, τ_3 is a type variable.

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 Γ_0

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

which handles concatenation of records.

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

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

to the initial type environment. In order to define removal of a field label we need to put one additional condition on REC(c):

REC6 $concat(\alpha, \{l : \beta\}, \gamma) \wedge (\gamma :: \{\!\{l' : \tau\}\!\}) \vdash^e (\alpha :: \{\!\{l' : \tau\}\!\})$ where $l \neq l'$

Note, this condition models the same behavior as condition **R6** in Section 5. We can now add primitive constructs

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

to the initial type environment Γ_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.

B Recursive types

We sketch how recursive types could be incorporated. Starting from HM(REC) we discuss now an extension HM(REC(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 REC(r):

$$\textbf{D8} \quad \vdash^e (\mu\alpha.\tau = [\mu\alpha.\tau/\alpha]\tau)$$

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

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(REC(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.