An ML-style Record Calculus with Extensible Records*

Sandra Alves
DCC-FCUP & CRACS
University of Porto, Porto, Portugal
sandra@fc.up.pt

Miguel Ramos
DCC-FCUP & LIACC
University of Porto, Porto, Portugal
jmiguelsramos@gmail.com

In this work, we develop a polymorphic record calculus with extensible records. Extensible records are records that can have new fields added to them, or preexisting fields removed from them. We also develop a static type system for this calculus and a sound and complete type inference algorithm. Most ML-style polymorphic record calculi that support extensible records are based on row variables. We present an alternative construction based on the polymorphic record calculus developed by Ohori. Ohori based his polymorphic record calculus on the idea of kind restrictions. This allowed him to express polymorphic operations on records such as field selection and modification. With the addition of extensible types, we were able to extend Ohori's original calculus with other powerful operations on records such as field addition and removal.

1 Introduction

A record is a basic data structure that provides a flexible way to aggregate data and which is present in several programming and specification languages. Record polymorphism has been studied using approaches based on subtyping [7, 5], kinded quantification [17, 18] and, most commonly, on the mechanism of row variables [23, 13, 21, 24], among others. Row variables range over finite sets of field types, which are constructed by extension starting from the empty row $\{||\}$. With the aim of developing a sound polymorphic programming language that supported labelled records and variants, while providing efficient compilation, Ohori [17] followed an approach based on the notion of a kind. In Ohori's system, variables ranging over record types, are annotated with a specification that represents the fields the record is expected to contain. This refines ML-type quantification to what is called kinded quantification, of the form $\forall \alpha :: \kappa.\sigma$, where type variable α is constrained to range only over the set of types denoted by kind κ. A kind κ is either the universal kind $\mathscr{U}$, denoting the set of all types, or a record kind of the form $\{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}$, denoting the set of all record types that contain fields $l_1, \ldots, l_n$, with types $\tau_1, \ldots, \tau_n$, respectively.

The type inference algorithm in [17] provides a sound extension of ML's let-polymorphism [8], which allows for a polymorphic treatment of record-based operations such as field selection and modification, but with the limitation of lacking support for extensible records. This limitation is often accepted in practical implementations of languages with record types, in a trade for efficiency, or due to the difficulty in guaranteeing the correctness of types for more flexible operations.

Nevertheless, record types, and in particular polymorphic record types, are most relevant not only in the context of ML-style programming languages, but also due to its relevance to areas were data aggregation and manipulation is a key feature. The style of record polymorphism developed by Ohori was recently explored in the context of event processing, through the development of a domain specific higher-order functional language for events [1], with a typing system that was both a restriction and an

---

extension of Ohori's polymorphic record calculus. Although this domain specific language proved to be adequate to deal with the notion of generic events, with the potential of providing a formal semantics to Complex Event Processing (CEP) systems, the lack of support for extensibility was once more a limitation, since the ability to extend a record with a new field or remove an existing field from a record is often useful in the context of CEP.

In this paper we address this limitation and develop a polymorphic record calculus with extensible records, that is, records that can have new fields added to them, or preexisting fields removed from them. To that end, we refine the notion of record kind, such that, a kind is of the form

$$\{\{l_1 : \tau_1, \ldots, l_n : \tau_n \parallel l'_1 : \tau'_1, \ldots, l'_m : \tau'_m\}\}$$

and denotes the set of all record types that contain the fields before $\parallel$ and do not contain the fields after $\parallel$. This system allows us to represent polymorphic versions of various types of record operations, such as field selection and modification, just as before, but is also powerful enough to represent field extension (the operation that adds a new field to a record) and field removal (the operation that removes a preexisting field from a record). The notion of record types is also extended to accommodate the notion of extensible record types. We refine the notion of kinded restrictions, using the refined notion of record kind, so that we can impose conditions on the extension and removal operations: a record can only be extended with a field that it does not already contain, and one can only remove existing fields from records. We extend Ohori's calculus with two new operations on records: $M \setminus\setminus l$ removes field l from term M, provided that M is a record with that field, and $\text{extend}(M_1, l, M_2)$ extends term M_1 with a field labelled l with value M_2, provided that M_1 is a record not containing l. These restrictions are imposed by the type system. We present a sound and complete ML-style type inference algorithm for our extended calculus.

The main contributions of this paper are:

- The design definition of an ML-style record calculus with operations for field extension and field removal.

- An ML-style type system for this calculus based on the notion of extensible types.

- A sound and complete type inference algorithm, that extends the ML-style record calculus in [17].

Overview   In Section 2 we define our ML-style record calculus with extensible records. In Section 3 we define a type system for our calculus and in Section 4 we present a type inference algorithm, which is proved to be sound and complete. We discuss related work in Section 5 and we finally conclude and discuss further work in Section 6.

2   An ML-style Record Calculus with Extensible Records

In this section we introduce an ML-style record calculus with extensible records. Our set of terms follows the one used by Ohori in [17], except for the exclusion of variants and the addition of two new terms: one for adding a new field to a record; and another for removing a preexisting field from a record. We assume some familiarity with the λ-calculus (see [2] for a detailed reference).

2.1 Terms

We start by formally defining the set of terms. In the following, let k range over a countable set of constants, $x, y, z, \ldots$ range over a countable set of variables and $l, l_1, \ldots$ range over a countable set $\mathscr{L}$ of labels. Additionally we assume a finite set of base types $\mathbb{B}$, ranged by b.

Definition 2.1. *The set of terms is given by the following grammar:*

$$
\begin{aligned}
M \quad ::= \quad & x \mid k^b \mid \lambda x.M \mid MM \mid let\ x = M\ in\ M \\
& \{l = M, \ldots, l = M\} \mid M.l \mid modify(M, l, M) \mid M \setminus\!\setminus l \mid extend(M, l, M)
\end{aligned}
$$

2.2 Types and Kinds

We now define the set of types and kinds. Following Damas and Milner's presentation of ML [8], we divide the set of types into monotypes (ranged over by τ) and polytypes (ranged over by σ). Monotypes can be base types (represented by b, which is obtained from a given set of base types), extensible types (ranged over by χ), and arrow types (of the form $\tau \to \tau$). Polytypes can be monotypes, or quantified types of the form $\forall \alpha :: \kappa.\sigma$, where κ is a kind restriction and α is a type variable quantified over the set of types denoted by κ. Extensible types can be type variables (represented by α, which is obtained from a given countably infinite set of type variables), record types (of the form $\{l : \tau, \ldots, l : \tau\}$, where l is obtained from a given set of labels), type extensions (of the form $\chi + \{l : \tau\}$), or type contractions (of the form $\chi - \{l : \tau\}$). Type extensions of the form $\chi + \{l : \tau\}$ are the type of records of type χ that are extended with a new field with label l and type τ and type contractions of the form $\chi - \{l : \tau\}$ are the type of records of type χ that have a preexisting field with label l and type τ removed.

Definition 2.2. *The set of types σ and kinds κ are specified by the following grammar:*

$$
\begin{aligned}
\sigma &::= \tau \mid \forall \alpha :: \kappa.\sigma \\
\tau &::= b \mid \chi \mid \tau \to \tau \\
\chi &::= \alpha \mid \{l : \tau, \ldots, l : \tau\} \mid \chi + \{l : \tau\} \mid \chi - \{l : \tau\} \\
\kappa &::= \mathscr{U} \mid \{\{l : \sigma, \ldots, l : \sigma \,\|\, l : \sigma, \ldots, l : \sigma\}\}
\end{aligned}
$$

All empty records are typed with the same type: the empty record type $\{\}$. Note that a type variable kinded with the universal kind can be instantiated with any type, while a type variable kinded with the empty kind restriction $\{\{\|\}\}$ can only be instantiated with a record type. Also, note that extensible types are defined recursively and can either have type variables or records types as base cases. To a type that appears as the base case of an extensible type we will call the base type of the extensible type (or just base type, if the context makes this clear). Also, if χ is an extensible type, then $\mathscr{B}(\chi)$ is its base type.

Labels that appear in types and kinds must always be pairwise distinct and each label can only be assigned one type during its existence. Also, the order in which labels occur is insignificant. Finally, if two extensible types have the same base type and have the same kind restrictions, we will consider them equal. This will made precise in Section 4, when we introduce a type reduction mechanism for extensible types.

Example 2.3. *The following extensible types are equal:*

$$
\alpha + \{l_1 : \tau_1\} - \{l_2 : \tau_2\} \equiv \alpha - \{l_2 : \tau_2\} + \{l_1 : \tau_1\} \tag{1}
$$

$$
\alpha + \{l_1 : \tau_1\} - \{l_2 : \tau_2\} - \{l_1 : \tau_1\} \equiv \alpha + \{l_1 : \tau_1\} - \{l_1 : \tau_1\} - \{l_2 : \tau_2\} \tag{2}
$$

(1) Clearly, despite the order of their type extensions and contractions, both types have the same base type and tell us that records with type α must not have the field $\{l_1 : \tau_1\}$ and must have the field $\{l_2 : \tau_2\}$.

(2) Here, we know that both types are equal because, despite the order of their type extensions and contractions, both have the same base type and tell us that records with type α must have the field $\{l_2 : \tau_2\}$ and must not have the field $\{l_1 : \tau_1\}$.

The two following extensible types are not equal:

$$\alpha_1 + \{l_1 : \tau_1\} - \{l_2 : \tau_2\} \not\equiv \alpha_2 - \{l_1 : \tau_1\} + \{l_2 : \tau_2\} \tag{3}$$

$$\alpha_1 - \{l_1 : \tau_1\} + \{l_2 : \tau_2\} + \{l_1 : \tau_1\} \not\equiv \alpha_2 + \{l_1 : \tau_1\} - \{l_1 : \tau_1\} + \{l_2 : \tau_2\} \tag{4}$$

(3) Clearly, these two types are not equal, and not just because they have syntactically different base types, but because the type on the left tells us that α_1 must not have the field $\{l_1 : \tau_1\}$ and must have the field $\{l_2 : \tau_2\}$, while the type on the right tells us that α_2 must have the field $\{l_1 : \tau_1\}$ and must not have the field $\{l_2 : \tau_2\}$.

(4) Here, we also have two types that are not equal, because they must have two syntactically different base types. The reason is that the type on the left tells us that its base type must have the field $\{l_1 : \tau_1\}$ and the type on the right tells us that its base type must not have that same field.

Note that we can only ignore the order of type extensions and contractions with <u>different labels</u> since changing the ordering between a type extensions and type contraction of a type will render it "ill" formed, as can be seen in Example 2.3. This property is enforced by the kinding rules in Definition 2.8.

Definition 2.4. *The set of free type variables of a type σ or a kind κ are denoted by $FTV(\sigma)$ and $FTV(\kappa)$, respectively. For second-order types, it is defined as $FTV(\forall \alpha :: \kappa.\sigma) = FTV(\kappa) \cup (FTV(\sigma) \setminus \{\alpha\})$. FTV for other types and kinds are defined as expected.*

We say that a type σ is closed if $FTV(\sigma) = \emptyset$. We assume that all bounded type variables are distinct and different from any free type variables, and that this property is preserved by substitution through α-equivalence. The type construct $\forall \alpha :: \kappa.\sigma$ binds the type variable α in σ, but not in κ.

Definition 2.5. *We write $\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$ for the kind assignment that binds α_i to κ_i, $(1 \leq i \leq n)$ and $\emptyset$ for the empty kind assignment. We will also write $K\{\alpha :: \kappa\}$ for $K \cup \{\alpha :: \kappa\}$ provided that K is well formed, $\alpha \notin dom(K)$, and $FTV(\kappa) \subseteq dom(K)$. Note that $K\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$ implies that $\alpha_i \notin FTV(\kappa_j)$ for any $1 \leq j < i \leq n$.*

Any type variables that appear in K must also be properly kinded by K itself.

Definition 2.6. *A kind assignment K is well formed if for all $\alpha \in dom(K), FTV(K(\alpha)) \subseteq dom(K)$, where $dom(f)$ denotes the domain of a function f.*

From now on, we assume that every kind assignment is well formed.

Definition 2.7. *A type σ is well formed under a kind assignment K if $FTV(\sigma) \subseteq dom(K)$.*

Definition 2.7 is naturally extended to other syntactic constructs containing types except substitutions (see Definition 2.12).

2.3 Kind Restrictions

Definition 2.8. *Type τ has kind restriction κ, that we denote as $K \Vdash \tau :: \kappa$, if it can be derived from the following kinding rules:*

$$i) \ K \Vdash \tau :: \mathcal{U} \text{ for any } \tau \text{ well formed under } K$$

$$ii) \ K \Vdash \{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l, \ldots\} :: \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \| l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\}$$
$$\text{if } \{l_1^l, \ldots, l_n^l, \ldots\} \cap \{l_1^r, \ldots, l_m^r\} = \emptyset,$$
$$\text{both } \{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l, \ldots\} \text{ and } \tau_i^r \ (1 \leq i \leq m) \text{ are well formed under } K$$

$$iii) \ K \Vdash \alpha :: \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \| l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\}$$
$$\text{if } K(\alpha) = \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l, \ldots \| l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r, \ldots\}\}$$

$$iv) \ K \Vdash \chi + \{l : \tau\} :: \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l, [l : \tau] \| l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\}$$
$$\text{if } K \Vdash \chi :: \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \| l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r, l : \tau\}\}$$

$$v) \ K \Vdash \chi - \{l : \tau\} :: \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \| l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r, [l : \tau]\}\}$$
$$\text{if } K \Vdash \chi :: \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l, l : \tau \| l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\}$$

where $[l : \tau]$ means that the inclusion of $l : \tau$ in its respective kind is optional.

Note that if $K \Vdash \tau : \kappa$, then both κ and τ are well formed under K.

Example 2.9. *Consider the type $\tau = \{l_1 : \tau_1\} + \{l_2 : \tau_2\} - \{l_1 : \tau_1\}$ and the empty kind assignment $K = \emptyset$. These are some possible derivations of kind restrictions for τ and its subtypes:*

$$\emptyset \Vdash \{l_1 : \tau_1\} :: \{\{\|\}\}$$
$$\emptyset \Vdash \{l_1 : \tau_1\} :: \{\{l_1 : \tau_1 \|\}\}$$
$$\emptyset \Vdash \{l_1 : \tau_1\} + \{l_2 : \tau_2\} :: \{\{l_2 : \tau_2 \|\}\}$$
$$\emptyset \Vdash \{l_1 : \tau_1\} + \{l_2 : \tau_2\} :: \{\{\| l_3 : \tau_3\}\}$$
$$\emptyset \Vdash \{l_1 : \tau_1\} + \{l_2 : \tau_2\} - \{l_1 : \tau_1\} :: \{\{l_2 : \tau_2 \| l_1 : \tau_1\}\}$$

Note that we cannot derive any kind restrictions for τ where $\{l_1 : \tau_1\}$ appears on the left of the $\|$.

Proposition 2.10. *The kinding rules ensure the two following properties:*

1. *If $K \Vdash \tau :: \{\{l : \tau', \ldots \| \ldots\}\}$, then $K \not\Vdash \tau :: \{\{\ldots \| l : \tau', \ldots\}\}$;*

2. *If $K \Vdash \tau :: \{\{\ldots \| l : \tau', \ldots\}\}$, then $K \not\Vdash \tau :: \{\{l : \tau', \ldots \| \ldots\}\}$.*

2.4 Kinded Substitutions

Definition 2.11. *A type substitution is a function from a finite set of type variables to types. We write $[\sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n]$ for the substitution that maps each α_i to σ_i. A substitution S is extended to the set of all type variables by letting $S(\alpha) = \alpha$ for all $\alpha \notin dom(S)$, where $dom(f)$ denotes the domain of a function f, and is extended uniquely to polytypes, record types, function types and extensible types as*

follows:

$$S(\forall \alpha :: \kappa.\sigma) = \forall \alpha :: S(\kappa).S(\sigma)$$
$$S(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}) = \{l_1 : S(\tau_1), \ldots, l_n : S(\tau_n)\})$$
$$S(\tau_1 \rightarrow \tau_2) = S(\tau_1) \rightarrow S(\tau_2)$$
$$S(\chi - \{l : \tau\}) = S(\chi) - \{l : S(\tau)\}$$
$$S(\chi + \{l : \tau\}) = S(\chi) + \{l : S(\tau)\}$$

Definition 2.12. *A substitution S is well formed under a kind assignment K if for any $\alpha \in dom(S)$, $S(\alpha)$ is well formed under K.*

Since all type variables are kinded by a kind assignment, the conventional notion of substitution (see Definition 2.11) must be refined by also taking into consideration kind constraints.

Definition 2.13. *A kinded substitution is a pair (K, S) of a kind assignment K and a substitution S that is well formed under K. A kinded substitution (K, S) is ground if $K = \emptyset$. We will write S for a ground kinded substitution $(\emptyset, S)$.*

Definition 2.14. *A kinded substitution (K_1, S) respects a kind assignment K_2 if for any $\alpha \in dom(K_2)$, $K_1 \Vdash S(\alpha) :: S(K_2(\alpha))$.*

This notion specifies the condition under which a substitution can be applied, i.e., if (K_1, S) respects K then it can be applied to a type σ kinded by K, yielding a type $S(\sigma)$ kinded by K_1.

Lemma 2.15. *If $K \Vdash \tau :: \kappa$ and a kinded substitution (K_1, S) respects K, then $K_1 \Vdash S(\tau) :: S(\kappa)$.*

Corollary 2.16. *If (K_1, S_1) respects K and (K_2, S_2) respects K_1, then $(K_2, S_2 \circ S_1)$ respects K.*

Proof. Since (K_1, S_1) respects K, we know that $\forall \alpha \in dom(K), K_1 \Vdash S_1(\alpha) :: S_1(K(\alpha))$. Since (K_2, S_2) respects K_1, we also know that $\forall \alpha \in dom(K_1), K_2 \Vdash S_2(\alpha) :: S_2(K_1(\alpha))$. But, since we know that (K_2, S_2) respects K_1, we have that $K_2 \Vdash S_2 \circ S_1(\alpha) :: S_2 \circ S_1(K(\alpha))$. Therefore, $\forall \alpha \in dom(K), K_2 \Vdash S_2 \circ S_1(\alpha) :: S_2 \circ S_1(K(\alpha))$ and $(K_2, S_2 \circ S_1)$ respects K. $\qquad\square$

A kind assignment is a constraint on possible substitutions of type variables. Furthermore, since types may depend on type variables other than their own free type variables, we need to take into consideration kind assignments and extend the notion of free type variables of a type.

Definition 2.17. *For a type σ well formed under K, the set of essentially-free type variables of σ under K, denoted $EFTV(K, \sigma)$, is the smallest set satisfying:*

(i) $FTV(\sigma) \subseteq EFTV(K, \sigma)$.

(ii) *If $\alpha \in EFTV(K, \sigma)$, then $FTV(K(\alpha)) \subseteq EFTV(K, \sigma)$.*

3   Type System

Before going into the typing rules, we first need to define what are generic instance of monotypes and closures of polytypes. To define the closure of a type, we first need to define what is a type assignment.

Definition 3.1. *We write $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ for the type assignment that binds x_i to σ_i, $(1 \leq i \leq n)$ and $\emptyset$ for the empty type assignment. We will also write $\Gamma\{x : \sigma\}$ for $\Gamma \cup \{x : \sigma\}$ provided that $x \notin dom(\Gamma)$.*

Definition 3.2. *We say that a type assignment Γ is well formed under a kind assignment K, if $\forall x \in dom(\Gamma)$, $\Gamma(x)$ is well formed under K.*

Definition 3.3. *Let Γ and τ be well formed under K. The closure of τ under Γ, K, denoted by $Cls(K,\Gamma,\tau)$, is a pair $(K', \forall \alpha_1 :: \kappa_1 \cdots \forall \alpha_n :: \kappa_n.\tau)$ such that $K'\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\} = K$ and $\{\alpha_1, \ldots, \alpha_n\} = EFTV(K,\tau) \setminus EFTV(K,\Gamma)$.*

Definition 3.4. *Let σ_1 be a polytype well formed under K. We say that σ_2 is a generic instance of σ_1 under K, written $K \Vdash \sigma_1 \geq \sigma_2$, if $\sigma_1 = \forall \alpha_1^1 :: \kappa_1^1 \cdots \alpha_n^1 :: \kappa_n^1.\tau_1$, $\sigma_2 = \forall \alpha_1^2 :: \kappa_1^2 \cdots \alpha_1^2 :: \kappa_m^2.\tau_2$, and there is a substitution S such that $dom(S) = \{\alpha_1^1, \ldots, \alpha_n^1\}$, $(K\{\alpha_1^2 :: \kappa_1^2, \ldots, \alpha_m^2 :: \kappa_m^2\}, S)$ respects $K\{\alpha_1^1 :: \kappa_1^1, \ldots, \alpha_n^1 :: \kappa_n^1\}$ and $\tau_2 = S(\tau_1)$.*

Lemma 3.5. *If $K \Vdash \sigma_1 \geq \sigma_2$ and $K \Vdash \sigma_2 \geq \sigma_3$, then $K \Vdash \sigma_1 \geq \sigma_3$.*

Proof. Without loss of generality, let us assume that $\sigma_1 = \forall \alpha_1^1 :: \kappa_1^1 \cdots \alpha_n^1 :: \kappa_n^1.\tau_1$, $\sigma_2 = \forall \alpha_1^2 :: \kappa_1^2 \cdots \alpha_m^2 :: \kappa_m^1.\tau_2$, and $\sigma_3 = \forall \alpha_1^3 :: \kappa_1^3 \cdots \alpha_k^3 :: \kappa_k^3.\tau_3$. Since $K \Vdash \sigma_1 \geq \sigma_2$, we know that there exists a substitution S_1 such that $dom(S_1) = \{\alpha_1^1, \ldots, \alpha_n^1\}$, $(K\{\alpha_1^2 :: \kappa_1^2, \ldots, \alpha_m^2 :: \kappa_m^2\}, S_1)$ respects $K\{\alpha_1^1 :: \kappa_1^1 :: \kappa_n^1\}$ and $\tau_2 = S_1(\tau_1)$. Since $K \Vdash \sigma_2 \geq \sigma_3$, we know that there exists a substitution S_2 such that $dom(S_2) = \{\alpha_1^2, \ldots, \alpha_m^2\}$, $(K\{\alpha_1^3 :: \kappa_1^3, \ldots, \alpha_k^3 :: \kappa_k^3\}, S_2)$ respects $K\{\alpha_1^2 :: \kappa_n^2 :: \kappa_m^2\}$ and $\tau_3 = S_2(\tau_2)$. To show that $K \Vdash \sigma_1 \geq \sigma_3$, we just need to find a substitution S_3, such that $dom(S_3) = \{\alpha_1^1, \ldots, \alpha_n^1\}$, $(K\{\alpha_1^3 :: \kappa_1^3, \ldots, \alpha_k^3 :: \kappa_k^3\}, S_3)$ respects $K\{\alpha_1^1 :: \kappa_n^1 :: \kappa_n^1\}$ and $\tau_3 = S_3(\tau_1)$. If we choose $S_3 = S_2 \circ S_1$, then $dom(S_3) = dom(S_1) = \{\alpha_1^1, \ldots, \alpha_n^1\}$, by Corollary 2.16, $(K\{\alpha_1^3 :: \kappa_1^3, \ldots, \alpha_k^3 :: \kappa_k^3\}, S_2 \circ S_1)$ respects K, and $\tau_3 = S_3(\tau_1) = S_2 \circ S_1(\tau_1) = S_2(S_1(\tau_1)) = S_2(\tau_2)$. $\square$

The type assignment system is given in Figure 1. We use $K, \Gamma \vdash M : \sigma$ to denote that term M has type σ given the type and kind assignments Γ and K, respectively.

Example 3.6. *Let $K = \{\alpha_1 :: \{\{\| \ l : \alpha_2\}\}, \alpha_2 :: \mathscr{U}\}$ and $\Gamma = \{x : \alpha_1, y : \alpha_2\}$. Then we can construct the following type derivation for $extend(x,l,y).l$:*

$$\Delta = \cfrac{\cfrac{K \Vdash \Gamma(x) \geq \alpha_1}{K, \Gamma \vdash x : \alpha_1} \ (Var) \quad \cfrac{K \Vdash \Gamma(y) \geq \alpha_2}{K, \Gamma \vdash y : \alpha_2} \ (Var) \quad K \Vdash \alpha_1 :: \{\{\| \ l : \alpha_2\}\} \quad \alpha_1 \notin FTV(\alpha_2)}{K, \Gamma \vdash extend(x,l,y) : \alpha_1 + \{l : \alpha_2\}} \ (Ext)$$

$$\cfrac{\Delta \quad K \Vdash \alpha_1 + \{l : \alpha_2\} :: \{\{l : \alpha_2 \ \|\}\}}{K, \Gamma \vdash extend(x,l,y).l : \alpha_2} \ (Sel)$$

The following lemma allows us to strengthen the type assignment.

Lemma 3.7. *If $K, \Gamma\{x : \sigma_1\} \vdash M : \tau$ and $K \Vdash \sigma_2 \geq \sigma_1$, then $K, \Gamma\{x : \sigma_2\} \vdash M : \tau$.*

The following lemma shows that typings are closed under kind-respecting kinded substitutions.

Lemma 3.8. *If $K_1, \Gamma \vdash M : \sigma$ and (K_2, S) respects K_1, then $K_2, S(\Gamma) \vdash M : S(\sigma)$.*

4 Type Inference

If we add a field to a record and then immediately remove it, we end up with a record with the same set of fields and the same kind restrictions, but with a different type. The same thing happens if we remove a preexisting field from a record and then immediately add it again. This means that extensible types can have different forms, but represent the same set of record types and records. This induces a set of identities on extensible types that can be interpreted as rewrite rules and used to define a type reduction system.

$$\frac{\Gamma \text{ is well formed under } K \qquad K \Vdash \Gamma(x) \geq \tau}{K,\Gamma \vdash x : \tau} \text{ (Var)}$$

$$\frac{\Gamma \text{ is well formed under } K}{K,\Gamma \vdash k^b : b} \text{ (Const)}$$

$$\frac{K,\Gamma\{x : \tau_1\} \vdash M : \tau_2}{K,\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2} \text{ (Abs)}$$

$$\frac{K,\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad K,\Gamma \vdash M_2 : \tau_1}{K,\Gamma \vdash M_1\, M_2 : \tau_2} \text{ (App)}$$

$$\frac{K,\Gamma \vdash M_1 : \sigma \qquad K,\Gamma\{x : \sigma\} \vdash M_2 : \tau}{K,\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \tau} \text{ (Let)}$$

$$\frac{K,\Gamma \vdash M_i : \tau_i \ (1 \leq i \leq n)}{K,\Gamma \vdash \{l_1 = M_1, \ldots, l_n = M_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \text{ (Rec)}$$

$$\frac{K,\Gamma \vdash M : \tau_1 \qquad K \Vdash \tau_1 :: \{\{l : \tau_2 \,\|\}\}}{K,\Gamma \vdash M.l : \tau_2} \text{ (Sel)}$$

$$\frac{K,\Gamma \vdash M_1 : \tau_1 \qquad K,\Gamma \vdash M_2 : \tau_2 \qquad K \Vdash \tau_1 :: \{\{l : \tau_2 \,\|\}\}}{K,\Gamma \vdash \text{modify}(M_1, l, M_2) : \tau_1} \text{ (Modif)}$$

$$\frac{K,\Gamma \vdash M : \tau \qquad Cls(K,\Gamma,\tau) = (K',\sigma)}{K',\Gamma \vdash M : \sigma} \text{ (Gen)}$$

$$\frac{K,\Gamma \vdash M : \tau_1 \qquad K \Vdash \tau_1 :: \{\{l : \tau_2 \,\|\}\}}{K,\Gamma \vdash M \setminus\!\setminus l : \tau_1 - \{l : \tau_2\}} \text{ (Contr)}$$

$$\frac{K,\Gamma \vdash M_1 : \tau_1 \qquad K,\Gamma \vdash M_2 : \tau_2 \qquad K \Vdash \tau_1 :: \{\{\| \, l : \tau_2\}\} \qquad \mathscr{B}(\tau_1) \notin FTV(\tau_2)}{K,\Gamma \vdash \text{extend}(M_1, l, M_2) : \tau_1 + \{l : \tau_2\}} \text{ (Ext)}$$

Figure 1: Typing rules for ML-style record calculus with extensible records

Definition 4.1. *Let χ be well formed under some kind assignment K. The rewrite rules for the type reduction system depend on the form of χ and are the following:*

i) $\{l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n\} - \{l_i : \tau_i\} \pm \{l' : \tau'\} \cdots \Rightarrow_\chi \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \pm \{l' : \tau'\} \cdots$

ii) $\{l_1 : \tau_1, \ldots, l_n : \tau_n\} + \{l : \tau\} \pm \{l' : \tau'\} \cdots \Rightarrow_\chi \{l_1 : \tau_1, \ldots, l_n : \tau_n, l : \tau\} \pm \{l' : \tau'\} \cdots$

iii) $\alpha \pm_1 \{l_1 : \tau_1\} \cdots -_i \{l : \tau\} \cdots +_j \{l : \tau\} \cdots \Rightarrow_\chi$
  $\alpha \pm_1 \cdots \pm_{i-1} \{l_{i-1} : \tau_{i+1}\} \pm_{i+1} \{l_{i+1} : \tau_{i+1}\} \cdots \pm_{j-1} \{l_{j-1} : \tau_{j-1}\} \pm_{j+1} \{l_{j+1} : \tau_{j+1}\} \cdots$

iv) $\alpha \pm_1 \{l_1 : \tau_1\} \cdots +_i \{l : \tau\} \cdots -_j \{l : \tau\} \cdots \Rightarrow_\chi$
  $\alpha \pm_1 \cdots \pm_{i-1} \{l_{i-1} : \tau_{i-1}\} \pm_{i+1} \{l_{i+1} : \tau_{i+1}\} \cdots \pm_{j-1} \{l_{j-1} : \tau_{j-1}\} \pm_{j+1} \{l_{j+1} : \tau_{j+1}\} \cdots$

To ensure that $\Rightarrow_\chi$ is confluent without considering the order of type extensions and/or contractions, we are going to assume that, in rules iii) and iv), i and j are the positions of the two first occurrences of those same-labelled type contraction and extension, in the case of rule iii), or type extension and contraction, in the case of rule ix).

　　We can show that the type reduction system is convergent, i.e. that normal forms are unique. This fact in conjunction with the fact that each reduction step preserves the set of kind restrictions of a type (see Proposition 4.4), allows us to replace any type with its normal form. Having types in their normal forms allows us to to identify types that correspond to records that are constructed from a particular record by adding and removing fields in different orders but that end up having the same set of fields.

Proposition 4.2. $\Rightarrow_\chi$ *is convergent.*

　　It is obvious that $\Rightarrow_\chi$ is terminating, since the number of type extensions and type contractions diminishes with each reduction step. Also, note that $\Rightarrow_\chi$ is confluent by construction.

Definition 4.3. *Let $\Rightarrow_\chi^*$ be the reflexive and transitive closure of $\Rightarrow_\chi$. (i) We say that σ_2 is a normal form of σ_1 if, and only if, $\sigma_1 \Rightarrow_\chi^* \sigma_2$ and there is no σ_3 such that $\sigma_2 \Rightarrow_\chi \sigma_3$. (ii) We write $\overline{\sigma_1}$ for the (unique) normal form of σ_1. (iii) We write $\sigma_1 \downarrow \sigma_2$ if $\overline{\sigma_1} = \overline{\sigma_2}$.*

　　Now we show that each reduction step $\Rightarrow_\chi$ preserves kind restrictions.

Proposition 4.4. *Every rewrite rule $\Rightarrow_\chi$, transforms a type χ_1 into a χ_2, such that $K \Vdash \chi_1 :: \kappa$ if, and only if, $K \Vdash \chi_2 :: \kappa$.*

　　Every field that appears in an extensible type that is a subtype of a type in normal form occurs exactly once.

Proposition 4.5. *If χ is in normal form, then every label that appears in it occurs exactly once.*

　　Note that if a type is in normal form, then every extensible type that appears as its subtype will have a type variable base type and each field that appears in it will occur exactly once.

Proposition 4.6. *Let S be a substitution and τ be a type. Then $\overline{S(\overline{\tau})} = \overline{S(\tau)}$.*

Proof. Let $\overline{\tau} = \tau$. Then there exists an extensible type χ in τ that can be normalized to its normal form $\overline{\chi}$. Let $\alpha \in dom(S)$ appear in χ, but not in $\overline{\chi}$. then α only appears in type operations of χ that were removed by normalization. Clearly, $S(\alpha)$ will appear in $S(\chi)$, but not in $S(\overline{\chi})$. But then, since α only appeared in type operations of χ that were removed by normalization, $S(\alpha)$ will only appear in those same type operations. Therefore, $S(\alpha)$ will not appear in $\overline{S(\chi)}$. Thus $S(\alpha)$ will neither appear in $\overline{S(\overline{\chi})}$ nor in $\overline{S(\chi)}$. Let $\alpha \in dom(S)$ appear in χ and in $\overline{\chi}$. Then α must not appear in any type operations of χ that can be removed by normalization. Then $S(\alpha)$ will appear both in $S(\chi)$ and $S(\overline{\chi})$:

- If $S(\overline{\chi})$ is in normal form, then $S(\alpha)$ is in normal form and will appear in both $\overline{S(\chi)}$ and $\overline{S(\overline{\chi})}$;

- If $S(\overline{\chi})$ is not in normal form, then either $S(\alpha)$ is not in normal form, $S(\alpha)$ is in normal form, but type operations that can be cancelled out by preexisting type operations in $\overline{\chi}$ were added to $\overline{\chi}$, or both. In any case, these operations will not appear in $\overline{S(\overline{\chi})}$ or $\overline{S(\chi)}$.

This means we can conclude that type type operations that appear in $\overline{S(\overline{\chi})}$ and $\overline{S(\chi)}$ are precisely the same, *i.e.* $\overline{S(\overline{\chi})} = \overline{S(\chi)}$.   □

To account for normal forms, we need to redefine what it means for two substitutions to be equal.

Definition 4.7. *Let S_1 and S_2 be two substitutions. We say that $S_1 = S_2$ if, for any type τ, $S_1(\tau) \downarrow S_2(\tau)$.*

Note that if two substitutions are syntactically equal, then they are also equal in the sense of Definition 4.7.

Proposition 4.8. *Let S_1, S_2, S_1', and S_2' be substitutions, such that $S_1 = S_2$ and $S_1' = S_2'$. Then $S_1 \circ S_1' = S_2 \circ S_2'$.*

Proof. Since $S_1 = S_2$ and $S_1' = S_2'$, we know that $S_1(\tau) = S_2(\tau)$ and $S_1'(\tau) = S_2'(\tau)$ for any type τ. But then $\overline{S_1'(\tau)} = \overline{S_2'(\tau)}$. Therefore, $S_1(\overline{S_1'(\tau)}) = S_2(\overline{S_2'(\tau)})$ and, by Proposition 4.6, $S_1(S_1'(\tau)) = S_2(S_2'(\tau))$. Thus, $S_1 \circ S_1' = S_2 \circ S_2'$.   □

4.1   Kinded Unification

We are going to extend the kinded unification algorithm used by Ohori in [17] with rules for unifying extensible types with type variables and other extensible types.

Definition 4.9. *A kinded set of equations is a pair (K, E) consisting of a kind assignment K and a set E of pairs of types such that E is well formed under K.*

Definition 4.10. *A substitution S satisfies E if $S(\tau_1) \downarrow S(\tau_2)$, for all $(\tau_1, \tau_2) \in E$.*

Definition 4.11. *A kinded substitution (K_1, S) is a unifier of a kinded set of equations (K, E) if it respects K and if S satisfies E.*

Definition 4.12. *(K_1, S) is a most general unifier of (K_2, E) if it is a unifier of (K_2, E) and if for any unifier (K_3, S_2) of (K_2, E), there is some substitution S_3 such that (K_3, S_3) respects K_1 and $S_2 = S_3 \circ S$.*

The unification algorithm $\mathcal{U}$ is defined by transformation. Each rule transforms a 4-tuple of the form (E, K, S, SK) consisting of a set E of type equations, a kind assignment K, a substitution S, and a (not necessarily well-formed) kind assignment SK. E keeps the set of equations to be unified; K specifies kind constraints to be verified; S records "solved" equations in the form of a substitution; and SK records "solved" kind constraints that have already been verified for S.

In specifying rules, we treat K, SK, and S as sets of pairs. We also use the following notations. Let F range over functions from a finite set of labels to types. We write $\{F\}$ and $\{\{F^l \parallel F^r\}\}$ to denote the record type identified by F and the record kind identified by F^l and F^r, respectively. For two functions F_1 and F_2 we write $F_1 + F_2$ for the function F such that $dom(F) = dom(F_1) \cup dom(F_2)$ and, for $l \in dom(F)$, if $l \in dom(F_1)$, then $F(l) = F_1(l)$; otherwise, $F(l) = F_2(l)$. For those same two functions, we write $F_1 - F_2$ for the function F such that $dom(F) = dom(F_1) \setminus dom(F_2)$ and, for $l \in dom(F)$, $F(l) = F_1(l)$. For an extensible type χ, we write $F_{e(\chi)}$ and $F_{c(\chi)}$ for the functions that represent the set of type extensions and contractions of χ, respectively.

Example 4.13. *Let $\chi = \alpha + \{l_1 : \tau_1\} - \{l_2 : \tau_2\} + \{l_3 : \tau_3\}$. Then $F_{e(\chi)}$ is the function (of domain $\{l_1, l_3\}$) that sends l_1 to τ_1 and l_3 to τ_3 and $F_{c(\chi)}$ is the function (of domain $\{l_2\}$) that sends l_2 to τ_2.*

Definition 4.14. *Let (K,E) be a given kinded set of equations. Algorithm $\mathcal{U}$ first transforms the system $(E,K,\emptyset,\emptyset)$ into (E',K',S,SK) until no more rules can be applied. It then returns the pair (K',S), if $E' = \emptyset$; otherwise, it fails. Its rules can be found in Figure 2.*

Since the constructions present in rule *vii)* from Definition 4.14 are somewhat intricate, we will state the following facts in hope that they will help the reader better understand this rule. Let α be type variable and χ an extensible type, both well formed under some kind assignment K, such that $\mathscr{B}(\chi) = \alpha_\chi$, $K(\alpha) = \{\{F_1^l \parallel F_1^r\}\}$, $K(\alpha_\chi) = \{\{F_2^l \parallel F_2^r\}\}$. Also, let us assume that $dom(F_1^l) \cap dom(F_{c(\overline{\chi})}) = \emptyset$, $dom(F_1^r) \cap dom(F_2^r + (F_2^l - F_{c(\overline{\chi})})) = \emptyset$ and $\alpha \notin FTV(\chi)$:

- Any field that appears in F_2^l was either introduced by the (Sel), (Modif) or (Contr) rule. This means that the set of the labels that appear in type contractions of χ is $dom(F_{c(\overline{\chi})}) \subseteq dom(F_2^l)$.

- Any field that appears on F_2^r was introduced by the (Ext) rule. This means that the set of the labels that appear in type extensions of χ is $dom(F_2^r) \subseteq dom(F_{e(\overline{\chi})})$.

- The set of fields that are guaranteed to appear in a record typed with χ is represented by the function $F_2^r + (F_2^l - F_{c(\overline{\chi})})$ and the set of fields that are guaranteed to *not* appear in a record typed with χ is represented by the function $F_{c(\overline{\chi})}$.

- The set of fields that are guaranteed to exist by $K(\alpha)$, but not by χ, is represented by the function $F_1^l - (F_2^r + (F_2^l - F_{c(\overline{\chi})})))$ and the set of fields that are guaranteed *not* to exist by $K(\alpha)$, but not by χ, is represented by the function $F_1^r - F_{c(\overline{\chi})}$.

Note that we use $\overline{\chi}$ instead of χ in $F_{c(\overline{\chi})}$ so that constructing this function is more straightforward. That being said, to keep proofs simpler, we do not normalize extensible types during unification.

Theorem 4.15. *Algorithm $\mathcal{U}$ takes any kinded set of equations and computes a most general unifier, if one exists; otherwise it fails.*

Example 4.16. *Let $K = \{\alpha :: \{\{\parallel l : \alpha''\}\}, \alpha' :: \{\{l : \alpha'' \parallel\}\}, \alpha'' :: \mathcal{U}\}$. Then $\mathcal{U}(K, \{(\alpha + \{l : \alpha''\} - \{l : \alpha''\}, \alpha' - \{l : \alpha''\})\}) = (\{\alpha :: \{\{l : \alpha' \parallel\}\}\}, \{(\alpha', \alpha + \{l : \alpha''\})\})$:*

$$(\{(\alpha + \{l : \alpha''\} - \{l : \alpha''\}, \alpha' - \{l : \alpha''\})\}, K, \emptyset, \emptyset)$$
$$\overset{viii)}{\Rightarrow} (\{(\alpha + \{l : \alpha''\}, \alpha')\}, K, \emptyset, \emptyset)$$
$$\overset{vii)}{\Rightarrow} (\emptyset, \{\alpha :: \{\{\parallel l : \alpha''\}\}, \alpha'' :: \mathcal{U}\}, \{(\alpha', \alpha + \{l : \alpha''\})\}, \{(\alpha', \{\{l : \alpha'' \parallel\}\})\})$$

Note that the first transformation is given by rule viii) and not by rule ix). This is the case because the latter rule can only be applied whenever two extensible types have no matching type extensions or contractions.

It is true that our unification algorithm introduces an overhead associated with the search of matching type extensions and contraction (see rules *viii)* and *ix)*) when compared with Ohori's original algorithm. That being said, we believe that with some care during the algorithm's implementation this overhead should be negligible.

4.2 Type Inference

Using the kinded unification, we extend Ohori's type inference algorithm to the cases of record field extension and record field removal. The type inference algorithm, $WK(K,\Gamma,M)$, is defined in Figure 3. Given a kind assignment K, a type assignment Γ, and a term M, the Type Inference Algorithm $WK(K,\Gamma,M)$ returns a tuple (K',S,σ). It is implicitly assumed that the algorithm fails if unification or any of the recursive calls fail.

$i)$ $(E \cup \{(\tau_1, \tau_2)\}, K, S, SK) \Rightarrow (E, K, S, SK)$ if $\tau_1 \downarrow \tau_2$

$ii)$ $(E \cup \{(\alpha, \tau)\}, K \cup \{(\alpha, \mathscr{U})\}, S, SK) \Rightarrow ([\tau/\alpha]E, [\tau/\alpha]K, [\tau/\alpha](S) \cup \{(\alpha, \tau)\}, [\tau/\alpha](SK) \cup \{(\alpha, \mathscr{U})\})$
   if $\alpha \notin FTV(\tau)$

$iii)$ $(E \cup \{(\alpha_1, \alpha_2)\}, K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}, S, SK) \Rightarrow$
   $([\alpha_2/\alpha_1](E \cup \{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\}),$
   $[\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1](\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\},$
   $[\alpha_2/\alpha_1](S) \cup \{(\alpha_1, \alpha_2)\}, [\alpha_2/\alpha_1](SK) \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\})\})$
   if $dom(F_1^l) \cap dom(F_2^r) = \emptyset$ and $dom(F_1^r) \cap dom(F_2^l) = \emptyset$

$iv)$ $(E \cup \{(\alpha, \{F_2\})\}, K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}, S, SK) \Rightarrow$
   $([\{F_2\}/\alpha](E \cup \{(F_1^l(l), F_2(l)) \mid l \in dom(F_1^l)\}),$
   $[\{F_2\}/\alpha](K), [\{F_2\}/\alpha](S) \cup \{(\alpha, \{F_2\})\}, [\{F_2\}/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\})$
   if $dom(F_1^l) \subseteq dom(F_2), dom(F_1^r) \cap dom(F_2) = \emptyset$, and $\alpha \notin FTV(\{F_2\})$

$v)$ $(E \cup \{(\{F_1\}, \{F_2\})\}, K, S, SK) \Rightarrow (E \cup \{(F_1(l), F_2(l)) \mid l \in dom(F_1)\}, K, S, SK)$
   if $dom(F_1) = dom(F_2)$

$vi)$ $(E \cup \{\tau_1^1 \rightarrow \tau_1^2, \tau_2^1 \rightarrow \tau_2^2\}, K, S, SK) \Rightarrow (E \cup \{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\}, K, S, SK)$

$vii)$ $(E \cup \{(\alpha, \chi)\}, K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (\mathscr{B}(\chi), \{\{F_2^l \parallel F_2^r\}\})\}, S, SK) \Rightarrow$
   $([\chi/\alpha](E \cup \{(F_1^l(l), (F_2^r + (F_2^l - F_{c(\overline{\chi})}))(l)) \mid l \in dom(F_1^l) \cap dom(F_2^r + (F_2^l - F_{c(\overline{\chi})}))\}$
   $\cup \{(F_1^r(l), F_{c(\overline{\chi})}(l)) \mid l \in dom(F_1^r) \cap dom(F_{c(\overline{\chi})})\}),$
   $[\chi/\alpha](K) \cup \{(\mathscr{B}(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\overline{\chi})}))) \parallel F_2^r + (F_1^r - F_{c(\overline{\chi})})\}\}))\},$
   $[\chi/\alpha](S) \cup \{(\alpha, \chi)\}, [\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\})$
   if $dom(F_1^l) \cap dom(F_{c(\overline{\chi})}) = \emptyset, dom(F_1^r) \cap dom(F_2^r + (F_2^l - F_{c(\overline{\chi})})) = \emptyset$, and $\alpha \notin FTV(\chi)$

$viii)$ $(E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_i^1 \{l_i^1 : \tau_i^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\},$
   $\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_j^2 \{l_j^2 : \tau_j^2\} \cdots \pm_m^2 \{l_m^1 : \tau_m^2\})\}, K, S, SK) \Rightarrow$
   $(E \cup \{(\tau_i^1, \tau_j^2), (\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_{i-1}^1 \{l_{i-1}^1 : \tau_{i-1}^1\} \pm_{i+1}^1 \cdots \pm_n^1 \{l_n^1 : \tau_n^1\},$
   $\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_{j-1}^2 \{l_{j-1}^2 : \tau_{j-1}^2\} \pm_{j+1}^2 \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\}, K, S, SK)$
   if $(\pm_i^1 = \pm_j^2 \wedge l_i^1 = l_j^2), \forall i < k \leq n : l_k^1 \neq l_i^2$, and $\forall j < r \leq m : l_r^2 \neq l_j^2$

$ix)$ $(E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\},$
   $K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}, S, SK) \Rightarrow$
   $(S_{ix)}(E \cup \{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\}),$
   $S_{ix)}(K) \cup \{(\alpha, S_{ix)}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\},$
   $S_{ix)}(S) \cup \{(\alpha_1, \alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}), (\alpha_2, \alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})\}$
   $S_{ix)}(SK) \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\})$
   where $S_{ix)} = [\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}/\alpha_1, \alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}/\alpha_2]$
   if $dom(F_1^l) \cap dom(F_2^r) = \emptyset, dom(F_1^r) \cap dom(F_1^l) = \emptyset, \alpha_1 \notin FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}),$
   $\alpha_2 \notin FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}), \forall 1 \leq i \leq n, 1 \leq j \leq m, l_i^1 \neq l_j^1$ and α is fresh

Figure 2: Kinded Unification

i) $WK(K,\Gamma,x) = $ if $x \notin dom(\Gamma)$ then *fail*

$\quad\quad\quad$ else let $\forall \alpha_1 :: \kappa_1 \cdots \forall \alpha_n :: \kappa_n.\tau = \Gamma(x),$

$\quad\quad\quad\quad\quad S = [\beta_1/\alpha_1,\ldots,\beta_n/\alpha_n]$ $(\beta_1,\ldots,\beta_n$ are fresh$)$

$\quad\quad\quad$ in $(K\{\beta_1 :: S(\kappa_1),\ldots,\beta_n :: S(\kappa_n)\}, id, \overline{S(\tau)})$

ii) $WK(K,\Gamma,\lambda x.M) = $ let $(K_1,S_1,\tau_1) = WK(K\{\alpha :: \mathscr{U}\},\Gamma\{x : \alpha\},M)$ $(\alpha$ fresh$)$

$\quad\quad\quad$ in $(K_1,S_1,\overline{S_1(\alpha)} \to \tau_1)$

iii) $WK(K,\Gamma,M_1\ M_2) = $ let $(K_1,S_1,\tau_1) = WK(K,\Gamma,M_1)$

$\quad\quad\quad\quad\quad (K_2,S_2,\tau_2) = WK(K_1,S_1(\Gamma),M_2)$

$\quad\quad\quad\quad\quad (K_3,S_3) = \mathscr{U}(K_2\{\alpha :: \mathscr{U}\}, \{(S_2(\tau_1),\tau_2 \to \alpha)\})$ $(\alpha$ fresh$)$

$\quad\quad\quad$ in $(K_3,S_3 \circ S_2 \circ S_1, \overline{S_3(\alpha)})$

iv) $WK(K,\Gamma,\text{let } x = M_1 \text{ in } M_2) = $ let $(K_1,S_1,\tau_1) = WK(K,\Gamma,M_1)$

$\quad\quad\quad\quad\quad (K_1',\sigma) = Cls(K_1,S_1(\Gamma),\tau_1)$

$\quad\quad\quad\quad\quad (K_2,S_2,\tau_2) = WK(K_1',(S_1(\Gamma))\{x : \sigma\},M_2)$

$\quad\quad\quad$ in $(K_2,S_2 \circ S_1,\tau_2)$

v) $WK(K,\Gamma,\{l_1 = M_1,\ldots,l_n = M_n\}) = $

$\quad\quad$ let $(K_1,S_1,\tau_1) = WK(K,\Gamma,M_1)$

$\quad\quad\quad\quad (K_i,S_i,\tau_i) = WK(K_{i-1},S_{i-1} \circ \cdots \circ S_1(\Gamma),M_i)$ $(2 \leq i \leq n)$

$\quad\quad$ in $(K_n,S_n \circ \cdots \circ S_2 \circ S_1, \{l_1 : \overline{S_n \circ \cdots \circ S_2(\tau_1)},\ldots,l_i : \overline{S_n \circ \cdots \circ S_{i+1}(\tau_i)},\ldots,l_n : \tau_n\})$

vi) $WK(K,\Gamma,M.l) = $ let $(K_1,S_1,\tau_1) = WK(K,\Gamma,M)$

$\quad\quad\quad\quad\quad (K_2,S_2) = \mathscr{U}(K_1\{\alpha_1 :: \mathscr{U},\alpha_2 :: \{\!\{l : \alpha_1 \,\|\}\!\}\}, \{(\alpha_2,\tau_1)\})$ $(\alpha_1,\alpha_2$ fresh$)$

$\quad\quad\quad$ in $(K_2,S_2 \circ S_1,\overline{S_2(\alpha_1)})$

vii) $WK(K,\Gamma,\text{modify}(M_1,l,M_2)) = $

$\quad\quad$ let $(K_1,S_1,\tau_1) = WK(K,\Gamma,M_1)$

$\quad\quad\quad\quad (K_2,S_2,\tau_2) = WK(K_1,S_1(\Gamma),M_2)$

$\quad\quad\quad\quad (K_3,S_3) = \mathscr{U}(K_2\{\alpha_1 :: \mathscr{U},\alpha_2 :: \{\!\{l : \alpha_1 \,\|\}\!\}\}, \{(\alpha_1,\tau_2),(\alpha_2,S_2(\tau_1))\})$ $(\alpha_1,\alpha_2$ fresh$)$

$\quad\quad$ in $(K_3,S_3 \circ S_2 \circ S_1,\overline{S_3(\alpha_2)})$

viii) $WK(K,\Gamma,M \setminus\!\!\setminus l) = $ let $(K_1,S_1,\tau_1) = WK(K,\Gamma,M)$

$\quad\quad\quad\quad\quad (K_2,S_2) = \mathscr{U}(K_1\{\alpha_1 :: \mathscr{U},\alpha_2 :: \{\!\{l : \alpha_1 \,\|\}\!\}\}, \{(\alpha_2,\tau_1)\})$ $(\alpha_1,\alpha_2$ fresh$)$

$\quad\quad\quad$ in $(K_2,S_2 \circ S_1,\overline{S_2(\alpha_2 - \{l : \alpha_1\})})$

ix) $WK(K,\Gamma,\text{extend}(M_1,l,M_2)) = $

$\quad\quad$ let $(K_1,S_1,\tau_1) = WK(K,\Gamma,M_1)$

$\quad\quad\quad\quad (K_2,S_2,\tau_2) = WK(K_1,S_1(\Gamma),M_2)$

$\quad\quad$ in if $\mathscr{B}(\tau_1) \in FTV(\tau_2)$ then *fail*

$\quad\quad\quad$ else let $(K_3,S_3) = \mathscr{U}(K_2\{\alpha_1 :: \mathscr{U},\alpha_2 :: \{\!\{\| \, l : \alpha_1\}\!\}\}, \{(\alpha_1,\tau_2),(\alpha_2,S_2(\tau_1))\})$ $(\alpha_1,\alpha_2$ fresh$)$

$\quad\quad\quad\quad$ in $(K_3,S_3 \circ S_2 \circ S_1,\overline{S_3(\alpha_2 + \{l : \alpha_1\})})$

Figure 3: Type inference algorithm

Proposition 4.17. *If $WK(K,\Gamma,M) = (K',S,\tau)$, then τ is in normal form.*

Theorem 4.18. *The Type Inference Algorithm is sound and complete:*

- *(Soundness) If $WK(K,\Gamma,M) = (K',S,\tau)$, then (K',S) respects K, $K',S(\Gamma) \vdash M : \tau$.*

- *(Completeness)*
    - *If $WK(K,\Gamma,M) = (K',S,\tau)$ and $K_0,S_0(\Gamma) \vdash M : \tau_0$, for some (K_0,S_0) and τ_0 such that (K_0,S_0) respects K, then there is some S', such that (K_0,S') respects K', $S'(\tau) \downarrow \tau_0$, and $S_0(\Gamma) = S' \circ S(\Gamma)$;*
    - *If $WK(K,\Gamma,M) = fail$, then there is no (K_0,S_0) and τ_0 such that (K_0,S_0) respects K and $K_0,S_0(\Gamma) \vdash M : \tau_0$.*

Example 4.19. *Let $K = \{\alpha_1 :: \{\{\| l : \alpha_2\}\}, \alpha_2 :: \mathscr{U}\}$ and $\Gamma = \{x : \alpha_1, y : \alpha_2\}$. Then we can apply the type inference algorithm to $extend(x,l,y).l$ and get the following results:*

$$WK(K,\Gamma,extend(x,l,y).l) = (K, \{(\alpha_3,\alpha_2),(\alpha_4,\alpha_1),(\alpha_5,\alpha_2),(\alpha_6,\alpha_1 + \{l : \alpha_2\})\}, \alpha_2)$$
$$WK(K,\Gamma,extend(x,l,y)) = (K, \{(\alpha_3,\alpha_2),(\alpha_4,\alpha_1)\}, \alpha_1 + \{l : \alpha_2\})$$
$$WK(K,\Gamma,x) = (K, id, \alpha_1)$$
$$WK(K,\Gamma,y) = (K, id, \alpha_2)$$

5   Related Work

There are several alternative type systems in the literature that deal with polymorphic records with some form of extensibility. The most common approaches are based on subtyping [5, 13] or row variables [23, 11, 21, 24], but there are also others based on flags [20, 22], predicates [12, 9] and scope variables [16], to name a few.

The approaches using subtyping [4, 7, 13, 19] have been widely used to build polymorphic type systems with records, in particular for object-oriented programming languages. However, there are several issues that arise when combining record polymorphism with a strong mechanism of subtyping. In the the presence of a subtyping relation ($r_1 \leq r_2$), meaning that r_1 contains at least the fields in r_2, a selection operator ($\_.l$) can have a type $\forall \alpha. \forall \beta \leq \{l : \alpha\}.\beta \to \alpha$, meaning that a label l can be selected for a record with type β, if β is a subtype of $\{l : \alpha\}$. This leads to additional information on the remaining fields of β being lost, making it harder to define operations dealing with extensibility. This was overcome by moving to a second-order type system [5, 6], however the resulting type system relied on explicitly typed extensible records, yielding a system where type-checking and subtyping are decidable, but type-inference is not addressed. The existence of a subtyping relation also complicates compilation, (again) because information on the exact type of a record can be lost, leading to the need of incorporating some degree of dynamic typing at runtime.

Several approaches dealing with extensibility use Wand's notion of row variables [23], which are variables that range over sets of field types, allowing for incremental construction of records. However, unlike the approach followed in our paper, operations in [23] are unchecked, meaning that, when extending a row with a field l, one does not check if this introduces a new field or replaces an existing one, leading to programs for which a principal type does not exist. Flexible systems with extensible records have been constructed over the mechanism of row variables [20, 22], extended with the notion of flags, yielding a system with extensible records and principal types. Flags are used to provide information on which fields the record must have, and which it cannot have. However, despite the flexibility to define

various powerful operations on records, compilation is not dealt efficiently in the presence of flags, due in part to the ability to support some unchecked operations.

Harper and Pierce [12] have studied type systems for extensible records where presence and absence of fields is given by predicates on types, thus leading to a system with checked operations, but without dealing with type inference or compilation. The use of predicates was further developed by Jones in his general theory of qualified types [15, 14], where extensible records are presented has a special case. Building on that is the approach by Gaster and Jones [10, 9] that combines the notion of row variables with the notion of qualified types, and is perhaps the work that is more closely related to ours. In this approach, row extension is used to capture positive information about the fields, while predicates are used to capture negative information, thus avoiding duplicated labels. In our approach both negative and positive information is given by the kind restrictions, resulting in a type system where constraints on label addition and label removal are treated in a uniform way.

Building on the work of Wand, Rémy and Gaster and Jones [23, 22, 10], Leijen has developed a polymorphic type system with extensible records based on scoped labels [16]. In this approach, duplicate labels are allowed and retained, and an appropriate scoping mechanism is provided to prevent ambiguity and still allow for safe operations. This provides a notion of free extension where update and extension operations are clearly separated, yielding a system that is flexible from the user's point of view. Our approach does not allow for duplicated labels and uses the kinding restrictions to implement a strict notion of extensibility.

In addition to record extension, there are several systems that deal with other powerful record operations such as concatenation [12, 21, 24], or the natural join [3] (an operation largely used in database programming, where labeled records play an important role). We choose not to include these operation as they tend to complicate both the implementation and the typing analysis, and simply follow Ohori's approach [17], which efficiently supports the basic operations on records, extending it with two basic operations that support extensibility. Nevertheless, Rémy [21] has developed an encoding of record concatenation via record extension, thus proving that a system supporting checked record extension can also support some form of record concatenation.

Our decision to build our work on Ohori's record calculus was highly motivated by the existence of an efficient compilation for such a calculus. This was achieved by translating the polymorphic record calculus into an implementation calculus, in which records are represented as vectors whose elements are accessed by direct indexing based on a total order $\ll$ on the set of labels. Polymorphic functions containing polymorphic record operations are compiled through the insertion of appropriate index abstractions, indicated by the kinded quantifiers given by the type of the function. Ohori then shows that this algorithm preserves types and that the compilation calculus has the subject reduction property, thus showing that the compilation algorithm preserves the operational behaviour of the original polymorphic record calculus. We believe that an efficient compilation algorithm can also be defined for the calculus developed in this work because variables still range over complete record types and the negative information that was added to kinds only affects type inference, not compilation. For these reasons, it should be possible to extend the compilation method in [17] to our extensible records.

6 Conclusions and Future Work

We have presented an ML-style polymorphic record calculus with extensible records, developed a typing system based on the notion of kinded quantification and a sound and complete type inference algorithm, based on kinded unification. While records are a basic commodity in a variety of programming lan-

guages, regarding the use of more powerful operations on records, there is still a gap between the theory and the practice, with no consensus on what is the best approach. With this work, we hope to contribute to that discussion.

Ohori's main goal was to support the most common operations dealing with polymorphic records, while maintaining an efficient compilation method. As already stated in the related work, although we do not deal with compilation in this paper, this is something that we would like to address in future work.

References

[1] Sandra Alves, Maribel Fernández & Miguel Ramos (2020): EVL: A Typed Higher-order Functional Language for Events. In Cláudia Nalon & Giselle Reis, editors: *Proceedings of the 15th International Workshop on Logical and Semantic Frameworks with Applications, LSFA 2020, Online, September 15, 2020, Electronic Notes in Theoretical Computer Science* 351, Elsevier, pp. 3–23, doi:10.1016/j.entcs.2020.08.002.

[2] Hendrik Pieter Barendregt (1985): The lambda calculus - its syntax and semantics. *Studies in logic and the foundations of mathematics* 103, North-Holland.

[3] Peter Buneman & Atsushi Ohori (1996): Polymorphism and Type Inference in Database Programming. *ACM Trans. Database Syst.* 21(1), pp. 30–76, doi:10.1145/227604.227609.

[4] Luca Cardelli (1988): A Semantics of Multiple Inheritance. *Inf. Comput.* 76(2/3), pp. 138–164, doi:10.1016/0890-5401(88)90007-7.

[5] Luca Cardelli & John C. Mitchell (1990): Operations on records. In M. Main, A. Melton, M. Mislove & D. Schmidt, editors: *Mathematical Foundations of Programming Semantics*, Springer New York, New York, NY, pp. 22–52, doi:10.1145/62678.62700.

[6] Luca Cardelli & John C. Mitchell (1991): Operations on Records. *Math. Struct. Comput. Sci.* 1(1), pp. 3–48, doi:10.1017/S0960129500000049.

[7] Luca Cardelli & Peter Wegner (1985): On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17(4), pp. 471–522, doi:10.1145/6041.6042.

[8] Luís Damas & Robin Milner (1982): Principal Type-Schemes for Functional Programs. In Richard A. DeMillo, editor: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, ACM Press, pp. 207–212, doi:10.1145/582153.582176.

[9] Benedict R. Gaster (1998): Records, variants and qualified types. Ph.D. thesis, University of Nottingham, UK. Available at `http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.262959`.

[10] Benedict R. Gaster & Mark P. Jones (1996): A Polymorphic Type System for Extensible Records and Variants. Technical Report, University of Nottingham.

[11] Robert Harper & John C. Mitchell (1993): On the Type Structure of Standard ML. *ACM Trans. Program. Lang. Syst.* 15(2), pp. 211–252, doi:10.1145/169701.169696.

[12] Robert Harper & Benjamin C. Pierce (1991): A Record Calculus Based on Symmetric Concatenation. In David S. Wise, editor: *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, ACM Press, pp. 131–142, doi:10.1145/99583.99603.

[13] Lalita A. Jategaonkar & John C. Mitchell (1993): Type Inference with Extended Pattern Matching and Subtypes. *Fundam. Inf.* 19(1–2), p. 127–165, doi:10.3233/FI-1993-191-206.

[14] Mark P. Jones (1994): Qualified Types: Theory and Practice. Distinguished Dissertations in Computer Science, Cambridge University Press, doi:10.1017/CBO9780511663086.

[15] Mark P. Jones (1994): A Theory of Qualified Types. *Sci. Comput. Program.* 22(3), pp. 231–256, doi:10.1016/0167-6423(94)00005-0.

[16] Daan Leijen (2005): Extensible records with scoped labels. In Marko C. J. D. van Eekelen, editor: *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*, *Trends in Functional Programming* 6, Intellect, pp. 179–194.

[17] Atsushi Ohori (1995): A Polymorphic Record Calculus and Its Compilation. *ACM Trans. Program. Lang. Syst.* 17(6), pp. 844–895, doi:10.1145/218570.218572.

[18] Atsushi Ohori & Peter Buneman (1988): Type Inference in a Database Programming Language. In: *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, LFP 1988, July 25-27, 1988, Snowbird, Utah, USA.*, ACM, pp. 174–183, doi:10.1145/62678.62700.

[19] Benjamin C. Pierce & David N. Turner (1994): Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming* 4(2), p. 207–247, doi:10.1017/S0956796800001040.

[20] Didier Rémy (1989): Typechecking Records and Variants in a Natural Extension of ML. In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, ACM Press, pp. 77–88, doi:10.1145/75277.75284.

[21] Didier Rémy (1992): Typing Record Concatenation for Free. In Ravi Sethi, editor: *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, ACM Press, pp. 166–176, doi:10.1145/143165.143202.

[22] Didier Rémy (1994): Type Inference for Records in Natural Extension of ML, p. 67–95. MIT Press, Cambridge, MA, USA.

[23] Mitchell Wand (1987): Complete Type Inference for Simple Objects. In: *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*, IEEE Computer Society, pp. 37–44. Available at http://www.ccs.neu.edu/home/wand/papers/wand-lics-87.pdf.

[24] Mitchell Wand (1989): Type Inference for Record Concatenation and Multiple Inheritance. In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, IEEE Computer Society, pp. 92–97, doi:10.1109/LICS.1989.39162.