

A Compilation Method for ML-Style Polymorphic Record Calculi*

Atsushi Ohori

Oki Electric Industry, Kansai Laboratory
Crystal Tower, 1-2-27 Shiromi
Chuo-ku, Osaka 540 Japan
email: ohori@kansai.oki.co.jp

Abstract

Polymorphic record calculi have recently attracted much attention as a typed foundation for object-oriented programming. This is based on the fact that a function that selects a field l of a record can be given a polymorphic type that enables it to be applied to various records containing a field l . Recent studies have established techniques to develop an ML-style type inference algorithm for such a polymorphic type system. There seems to be, however, no established method to compile an ML-style polymorphic record calculus into efficient code. The purpose of this paper is to present one such method. We define a polymorphic record calculus as an extension of Damas and Milner's proof system for ML. For this calculus, we define an implementation calculus where records are represented as arrays of (references to) values and field selection is performed by direct indexing. To represent polymorphic field selection, the implementation calculus contains an abstraction mechanism over indexes. We then develop an algorithm to translate the polymorphic record calculus into the implementation calculus by refining a type inference algorithm; it simultaneously computes a principal type scheme in the polymorphic record calculus and a correct implementation term in the implementation calculus. The type inference is shown to be sound and complete in the sense of Damas-Milner's algorithm for ML. Moreover, the polymorphic type system is shown to be sound with respect to an operational semantics of the translated terms in the implementation calculus.

*Appeared in **Proc. ACM POPL Symposium**, pages 154–165, 1992 (a few minor errors corrected.)

1 Introduction

The investigation of a polymorphic type discipline for labeled records was initiated by Cardelli [Car88], who defined a typed functional calculus with records and showed that certain aspects of *method inheritance* can be represented in a static type system. This is based on the observation that a subtype relation can be used to capture the polymorphic nature of functions involving field selection such as

$$\begin{aligned} & \text{function } \text{wealthy}(x : \{Name : \text{string}, Salary : \text{int}\}) \\ & = x.Salary > 100000 \end{aligned}$$

where $x.Salary$ selects the *Salary* field from a record. This function can be applied to all the subtypes of $\{Name : \text{string}, Salary : \text{int}\}$, i.e. those record types that may contain more fields. [CW85] extended this calculus to a second-order type system by combining the subtyping and the type system of the second-order lambda calculus [Gir71, Rey74]. A similar idea was presented in [Mit84]. More powerful second-order calculi for records were proposed in [CM89, HP91].

Wand [Wan87, Wan88] observed that the above form of method inheritance can properly be represented in an ML style polymorphic type system when it is extended to records (and variants). This idea was further developed in a number of type inference systems [Sta88, JM88, OB88, Rém89, Wan89, Rém90] that include a combination of type inference and data abstraction with multiple inheritance [OB89]. In these type systems, a most general polymorphic type scheme is *inferred* for any typable untyped term containing operations on records. By appropriate instantiation of the inferred type scheme, an untyped term can safely be used as values of various types. This approach not only captures the polymorphic nature of functions on records but also integrates a record calculus and ML-style type inference, which relieves the programmer from complicated type declarations required in explicit second-order calculi. We therefore hope that this approach will provide a basis for designing practical programming languages for object-oriented programming and other data intensive applications, such

as database programming [AB87], for which records are an essential data structure.

To implement a practical programming language embodying such a polymorphic type inference system, we need to develop a method to compile (typable) untyped polymorphic expressions involving operations on records into efficient code. In this paper, we attempt to provide one such method. As a first step, we only consider field selection and field modification (update) as operations on records. They are the basis of all record calculi so far proposed, and are also the operations commonly found in conventional programming languages. We hope that the method presented here can be extended to various other operations on records such as those to extend a record with an additional field [Wan87, JM88, Rém89, CM89, Rém90], various forms of record concatenation [Wan89, HP91, Rém91] and join [OB88]. We will comment on this issue in Section 6.

For a language with a *simple* type system, compiling field selection into efficient code is a standard practice that is routinely carried out by a compiler. This is due to the fact that exact type information on parameters is available when compiling a function operating on records. In a polymorphic type system, however, generating efficient code for field selection is far from trivial. Since types of actual parameters may differ, the position (or offset) of a field within a record passed as a parameter cannot be statically computed when compiling the body of a function operating on records. One naive approach is to implement directly the intended semantics of field selection and field modification by dynamically searching for the required label in a record represented as an association list of labels and values. An obvious drawback to such an approach is inefficiency in run-time execution. Since field selection is a basic operation that is frequently invoked, such a method is unacceptable for serious practical applications. Another approach might be to predetermine the offsets of all the possible labels and to represent a record as a potentially very large structure with many empty slots. A recent work by Cardelli [Car91] used this approach to represent records in a pure calculus of subtyping. While this approach is useful for studying formal properties of record polymorphism, it is unrealistic in practice.

For a polymorphic record calculus to become a basis of practical programming languages, we must develop a compilation method that achieves both compactness in representation of records and efficiency in execution of field selection and field modification. Connor, Deale, Morrisoan and Brown [CDMB89] considered this problem in the context of an explicitly typed language with subtyping and suggested a solution. However, they did not establish a systematic method to deal with arbitrary expressions, nor did they consider

a type inference system. To the author's knowledge, there has been no proposal that establishes a method to compile an ML-style implicitly typed polymorphic language with records into efficient code. The purpose of this paper is to present such a compilation method and to establish that the compilation achieves the intended operational behavior of a polymorphic record calculus. This requires a formulation of type inference for records with ML's *let* binding and a precise definition of "efficient implementation" for a record calculus.

Our strategy is to translate a polymorphic record calculus into an *implementation calculus*, where a record is represented as an array of (references to) values and field selection and field modification are performed by direct indexing. To deal with polymorphic field selection, the implementation calculus contains *index variables* and *index abstraction*. For example, from the untyped term of the form

$$\begin{array}{l} \text{let } \textit{wealthy} = \lambda x. x \cdot \textit{Salary} > 100000 \\ \text{in } \dots (\textit{wealthy } R) \dots \text{end} \end{array}$$

the translation algorithm produces the following implementation code:

$$\begin{array}{l} \text{let } \textit{wealthy} = \lambda I \lambda x. x[I] > 100000 \\ \text{in } \dots ((\textit{wealthy } \mathcal{I}) R) \dots \text{end} \end{array}$$

where I is an index variable, $\lambda I. M$ is index abstraction, $x[I]$ is an index expression and $(\textit{wealthy } \mathcal{I})$ is *index application* with an appropriate index value \mathcal{I} . This method was suggested in [CDMB89]. The major technical contribution of the present paper is to establish an inference algorithm that always constructs a correct implementation term for any type correct raw term of a polymorphic record calculus and to prove that the polymorphic type discipline is sound with respect to the translation. We achieve this by refining a polymorphic type inference algorithm for a record calculus.

In Section 2 we define a polymorphic record calculus $\lambda^{\textit{let}, \bullet}$ with field selection and field modification operations. Here we refine the mechanism of conditional type schemes that we proposed in [OB88] as *kinded type schemes*. This refinement allows us to extend a number of known formal properties of ML polymorphism to record structures. Examples include Damas and Milner's formal account [DM82] for *let* polymorphism and Mitchell and Harper's analysis [MH88] on ML polymorphism. Section 3 defines an implementation calculus $\lambda^{\textit{let}, []}$. In order to establish the soundness of the compilation algorithm, we present the calculus as a typed functional calculus and prove the soundness of the type system with respect to an operational semantics. In Section 4 we simultaneously develop a type inference algorithm for $\lambda^{\textit{let}, \bullet}$ and a compilation algorithm from $\lambda^{\textit{let}, \bullet}$ to $\lambda^{\textit{let}, []}$. We then es-

establish that the compilation algorithm preserves typings. This result, together with the soundness of the type system of $\lambda^{let,[]}$, establishes the soundness of the polymorphic type system of $\lambda^{let,\bullet}$ with respect to the semantics achieved by the compilation.

In our calculus, the polymorphic nature of field selection is obtained not by subtyping but by polymorphic instantiation of type schemes. Since the same problem arises in compiling a language with subtyping, it would be nice if we could also develop a similar compilation algorithm for record calculi based on subtyping. However, there exists some difficulty that seems to be inherent in calculi with subtyping. We will discuss this issue in Section 5.

2 The Record Calculus $\lambda^{let,\bullet}$

The set of raw terms (ranged over by e) of $\lambda^{let,\bullet}$ is given by the syntax:

$$e ::= c^\tau \mid x \mid \lambda x. e \mid e \mid e \mid \{l = e, \dots, l = e\} \mid e.l \mid \text{modify}(e, l, e) \mid \text{let } x = e \text{ in } e \text{ end}$$

where c^τ stands for constants of type τ . For the simplicity of presentation we assume that τ in c^τ is a ground type not containing any type variables. It is not hard to extend our formal development to constants having a polymorphic type. $\{l_1 = e_1, \dots, l_n = e_n\}$ is the syntax for labeled records where l_1, \dots, l_n are pairwise distinct labels and the order of their appearance is insignificant. $e.l$ is field selection and $\text{modify}(e_1, l, e_2)$ is field modification, which creates a new record from e_1 by changing the value of the l field to e_2 . $\text{let } x = e \text{ in } e \text{ end}$ is ML's *let* construct.

Following Damas and Milner's formal account [DM82] for let polymorphism, the set of types is divided into the set of *monotypes* and the set of *polytypes*. The set of monotypes (ranged over by τ) is given by the syntax:

$$\tau ::= t \mid b \mid \tau \rightarrow \tau \mid \{l : \tau, \dots, l : \tau\}$$

where t stands for type variables and b for base types. To represent polymorphic field selection, we need to refine type quantification in Damas-Milner polytypes. Instead of using row variables [Wan87], we use kinded type quantification of the form $\forall t :: k. \tau$, which denotes quantification over the subset of monotypes represented by the kind k . This is a refinement of *conditional type schemes* we proposed in [OB88], and is also similar to *bounded quantification* [CW85]. It should be noted, however, that we do not use any notion of subtyping. The set of polytypes (ranged over by σ) and the set of kinds (ranged over by k) are given as:

$$\sigma ::= \tau \mid \forall t :: k. \sigma \quad k ::= U \mid \langle l : \tau, \dots, l : \tau \rangle$$

U denotes the set of all monotypes and $\langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle$ is a record kind intuitively denoting the subset of monotypes that are record types containing the fields $l_1 : \tau_1, \dots, l_n : \tau_n$.

In our calculus, every type variable must be *kinded* by a *kind assignment* \mathcal{K} , which is a function from a finite set of type variables to kinds. A monotype τ has a kind k under a kind assignment \mathcal{K} , denoted by $\mathcal{K} \vdash \tau :: k$, if it is derivable by the following set of kinding rules:

$$\begin{aligned} \mathcal{K} \vdash \tau &:: U \quad \text{for all } \tau \\ \mathcal{K} \vdash t &:: \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle \\ &\text{if } t \in \text{dom}(\mathcal{K}), \mathcal{K}(t) = \langle l_1 : \tau_1, \dots, l_n : \tau_n, \dots \rangle \\ \mathcal{K} \vdash \{l_1 : \tau_1, \dots, l_n : \tau_n, \dots\} &:: \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle \end{aligned}$$

Let \mathcal{T} be a *type assignment*, which is a function from a finite set of variables to polytypes. Typing judgments in our calculus are formulae of the form $\mathcal{K}, \mathcal{T} \vdash e : \sigma$ such that all free type variables in \mathcal{T} and σ are contained in the domain of \mathcal{K} . For a function f , we write $f\{x \mapsto v\}$ for the function f' such that $\text{dom}(f') = \text{dom}(f) \cup \{x\}$, $f'(x) = v$ and $f'(y) = f(y)$ for all $y \in \text{dom}(f)$, $y \neq x$. We also write $\sigma[\tau/t]$ for the type obtained from σ by substituting all the free occurrences of t with τ . Figure 1 gives the set of typing rules for $\lambda^{let,\bullet}$. Note that kindings in rules DOT and MODIFY represent the exact conditions on record types. This allows us to integrate these rules with those for *let* construct (rules GEN and INST). Since *let* construct is the source of ML polymorphism, this formal treatment is essential to develop a systematic compilation method.

2.1 Properties of $\lambda^{let,\bullet}$

The above treatment of *let* binding in our proof system allows us to analyze various formal properties of *let* polymorphism with records. Indeed, most of the known results of ML polymorphism seems to extend to our calculus. Examples include Mitchell and Harper's analysis [MH88] of Damas and Milner's proof system for ML through a limited form of explicit calculus called XML. The explicit calculus corresponding to our calculus is defined by changing the term e in the conclusion of the rules ABS, GEN, INST and LET to $\lambda x : \tau_1. e_1$, $\lambda t :: k. e$, $e \tau$ and $\text{let } x : \sigma = e_1 \text{ in } e_2 \text{ end}$, respectively. Let E be an explicitly typed term and $\text{erase}(E)$ be the raw term obtained from E by erasing type specification, type abstraction and type application. We can show the following property, which is an extension of a result shown in [MH88].

Proposition 1 *If $\mathcal{K}, \mathcal{T} \vdash E : \sigma$ is a typing in the explicit calculus, then $\mathcal{K}, \mathcal{T} \vdash \text{erase}(E) : \sigma$ is a typing in $\lambda^{let,\bullet}$. Conversely, if $\mathcal{K}, \mathcal{T} \vdash e : \sigma$ is a typing in*

CONST	$\mathcal{K}, \mathcal{T} \vdash c^\tau : \tau$
VAR	$\mathcal{K}, \mathcal{T} \vdash x : \sigma \quad \text{if } x \in \text{dom}(\mathcal{T}), \mathcal{T}(x) = \sigma$
ABS	$\frac{\mathcal{K}, \mathcal{T}\{x \mapsto \tau_1\} \vdash e_1 : \tau_2}{\mathcal{K}, \mathcal{T} \vdash \lambda x. e_1 : \tau_1 \rightarrow \tau_2}$
APP	$\frac{\mathcal{K}, \mathcal{T} \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \mathcal{K}, \mathcal{T} \vdash e_2 : \tau_1}{\mathcal{K}, \mathcal{T} \vdash e_1 e_2 : \tau_2}$
RECORD	$\frac{\mathcal{K}, \mathcal{T} \vdash e_i : \tau_i \quad (1 \leq i \leq n)}{\mathcal{K}, \mathcal{T} \vdash \{l_1 = e_1, \dots, l_n = e_n\} : \{l : \tau_1, \dots, l_n : \tau_n\}}$
DOT	$\frac{\mathcal{K}, \mathcal{T} \vdash e : \tau_1 \quad \mathcal{K} \vdash \tau_1 :: \langle l : \tau_2 \rangle}{\mathcal{K}, \mathcal{T} \vdash e \cdot l : \tau_2}$
MODIFY	$\frac{\mathcal{K}, \mathcal{T} \vdash e_1 : \tau_1 \quad \mathcal{K}, \mathcal{T} \vdash e_2 : \tau_2 \quad \mathcal{K} \vdash \tau_1 :: \langle l : \tau_2 \rangle}{\mathcal{K}, \mathcal{T} \vdash \text{modify}(e_1, l, e_2) : \tau_1}$
GEN	$\frac{\mathcal{K}\{t \mapsto k\}, \mathcal{T} \vdash e : \sigma}{\mathcal{K}, \mathcal{T} \vdash e : \forall t :: k. \sigma} \quad t \text{ not free in } \mathcal{T}$
INST	$\frac{\mathcal{K}, \mathcal{T} \vdash e : \forall t :: k. \sigma \quad \mathcal{K} \vdash \tau :: k}{\mathcal{K}, \mathcal{T} \vdash e : \sigma[\tau/t]}$
LET	$\frac{\mathcal{K}, \mathcal{T} \vdash e_1 : \sigma \quad \mathcal{K}, \mathcal{T}\{x \mapsto \sigma\} \vdash e_2 : \tau}{\mathcal{K}, \mathcal{T} \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau}$

Figure 1: Typing rules of the polymorphic record calculus

$\lambda^{\text{let},*}$, then there is an explicitly typed term E such that $\text{erase}(E) = e$ and $\mathcal{K}, \mathcal{T} \vdash E : \sigma$ is a typing in the explicit calculus. \blacksquare

This result allows us to transfer various analyses presented in [MH88] to our calculus.

Rémy's recent work [Rém90] also provide a formal treatment of *let* and unbounded labeled records. It seems to be fairly obvious that other proposals for type inference systems of (unbounded) labeled records [Wan87, Sta88, JM88, Wan89] can deal with *let* binding by properly renaming types and row variables. It is, however, not immediately obvious that various formal properties underlying *let* polymorphism such as above can be extended to those proposals.

2.2 An alternative proof system of $\lambda^{\text{let},*}$

The existence of polytypes in the presentation of $\lambda^{\text{let},*}$, however, complicates the presentation of a type inference algorithm and a compilation algorithm we shall develop below. Fortunately there is a simpler proof system that is equivalent to $\lambda^{\text{let},*}$. This is based on simpler accounts for *let* polymorphism presented in [Oho89a, Mit90]. The simpler proof system is obtained by (1) restricting the set of types to be monotypes, (2) removing the rules GEN and INST, and (3)

replacing the rule LET with the following rule:

$$\text{LET}' \quad \frac{\mathcal{K}, \mathcal{T} \vdash e_1 : \tau_1 \quad \mathcal{K}, \mathcal{T} \vdash e_2[e_1/x] : \tau_2}{\mathcal{K}, \mathcal{T} \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau_2}$$

where τ_1 may be any monotype and the notation $e_2[e_1/x]$ denotes the term obtained from e_2 by substituting all the free occurrences of x in e_2 by e_1 with necessary bound variable renaming.

Since $e_2[e_1/x]$ in the above rule is not a subterm of $\text{let } x = e_1 \text{ in } e_2 \text{ end}$, some care must be taken in proving various properties of this proof system. In particular, we cannot use induction on the size of raw terms. We can, however, define a *complexity measure* of raw terms in such a way that, in any rule in this proof system, the complexity of the raw term in the conclusion is strictly greater than those of the raw terms in its premises. Based on this, we can prove properties of this proof system by usual case analysis in terms of the structure of raw terms. Here we omit a rather lengthy definition of the complexity measure of raw terms. One definition can be found in [Oho89b].

For closed raw terms, this proof system is equivalent to the original one.

Proposition 2 *Let e be a closed raw term. It has a typing in $\lambda^{\text{let},*}$ if and only if it has a typing in the simpler proof system. \blacksquare*

Another advantage of this simpler proof system is that

it has a simple denotational semantics as exploited in [Oho89a].

We use this alternative proof system in what follows.

3 The Implementation Calculus $\lambda^{let,[]}$

This section defines a calculus $\lambda^{let,[]}$ where field selection and field modification are executed efficiently. In order to establish the correctness of the compilation algorithm given in Section 4, we define $\lambda^{let,[]}$ as a typed functional calculus and show the soundness of the type system in terms of an operational semantics.

We assume that there is a linear order \leq on the set of labels and that a labeled record $\{l_1 = v_1, \dots, l_n = v_n\}$ is represented as an array of values such that the field $l_i = v_i$ is the j th entry of the array where j is the size of the set $\{l | l \in \{l_1, \dots, l_n\}, l \leq l_i\}$. Selecting the l_i field can then be performed by the simple indexing operation $\{l_1 = v_1, \dots, l_n = v_n\}[j]$. In an actual implementation, records may be presented by references (or “pointers”) to values and field selection is implemented by indexing followed by de-referencing. In examples below, we use lexicographical ordering on strings for \leq . For example, $Age \leq Name$ and the record $\{Name = \text{“Joe”}, Age = 21\}$ is represented by an array of two entries whose first entry is $Age = 21$ and whose second entry is $Name = \text{“Joe”}$. Selecting $Name$ field from this record is performed by the index expression $\{Name = \text{“Joe”}, Age = 21\}[2]$.

Since the required index values are not always available when compiling field selection, we introduce *index variables* (ranged over by I) and an abstraction mechanism over *indexes* (ranged over by \mathcal{I}). An index is either a natural number or an index variable. The set of raw terms (ranged over by M) of the implementation calculus is given as:

$$M ::= c^\tau | x | \lambda x. M | M M \\ | \{l = M, \dots, l = M\} | M[\mathcal{I}] | modify(M, \mathcal{I}, M) \\ | \lambda I. M | M \mathcal{I} | let x = M in M end$$

In order to assign a type to raw terms involving index variables, we introduce *index types* of the form $index(l, \tau)$ denoting the index value corresponding to field l in type τ . The set of types of the implementation calculus is given by the syntax:

$$\tau ::= t | b | \tau \rightarrow \tau | \{l : \tau, \dots, l : \tau\} | index(l, \tau) \Rightarrow \tau$$

where $index(l, \tau) \Rightarrow \tau$ is a type of terms that are abstracted over indexes. The set of kinds and the kinding rules are the same as before. In order for index types of the form $index(l, \tau)$ to be meaningful, we require τ to have an appropriate record kind: an index type $index(l, \tau)$ is *well formed under a kind assignment* \mathcal{K} if

$\mathcal{K} \vdash \tau :: \langle l, \tau' \rangle$ for some τ' . This condition guarantees that an index type denotes a position of a field in a record type. In particular, a well formed ground index type (under any kind assignment) must be of the form $index(l_i, \{l_1 : \tau_1, \dots, l_n : \tau_n\})$, $l_i \in \{l_1, \dots, l_n\}$ and it denotes the natural number equal to the size of the set $\{l | l \in \{l_1, \dots, l_n\}, l \leq l_i\}$.

Typings are presented relative to an *index assignment* (ranged over by \mathcal{L}), which is a function from a finite set of index variables to index types. An index assignment \mathcal{L} is *well formed under a kind assignment* \mathcal{K} if all the index types appearing in \mathcal{L} are well formed under \mathcal{K} . An index value \mathcal{I} has an *index type* $index(l, \tau)$ under an index assignment \mathcal{L} , denoted by $\mathcal{L} \vdash \mathcal{I} : index(l, \tau)$, if either $\mathcal{I} = I$, $\mathcal{L}(I) = index(l, \tau)$ or \mathcal{I} is the natural number denoted by $index(l, \tau)$. Typing judgments in $\lambda^{let,[]}$ are formulae of the form $\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M : \tau$ such that \mathcal{L} is well formed under \mathcal{K} . Figure 2 gives the typing rules of $\lambda^{let,[]}$.

3.1 Operational semantics and the soundness of the type system

To show the soundness of the type system, we define an operational semantics of $\lambda^{let,[]}$ in the style of [Tof88] by giving a set of rules of the form

$$E, L \vdash M \Longrightarrow v$$

where v is a *canonical value* defined below, E is a *variable environment*, which is a function from a finite set of variables to values, and L is an *index environment*, which is a function from a finite set of index variables to natural numbers. The set of canonical values is given by the syntax:

$$v ::= c^\tau | \{l_1 = v_1, \dots, l_n = v_n\} | fun(E, L, x, M) \\ | lab(E, L, I, M) | wrong$$

where $\{l_1 = v_1, \dots, l_n = v_n\}$ stands for canonical values for record expressions. For the purpose of pure evaluation, it is not necessary to keep record labels in canonical values of record expressions; however, they should be useful, for example, for a printing utility. $fun(E, L, x, M)$ stands for function closures, $lab(E, L, I, M)$ for closures corresponding to index abstraction and $wrong$ represents run-time error. Figure 3 gives the set of reduction rules. Note that although the above rules suggest associative lookup for lambda variables and index variables, standard techniques for compiling programming languages can be used to determine the actual address (e.g. the offset to an activation record) of each variable.

A value v has a *ground type* τ , denoted by $\models v : \tau$, if it is derivable from the following set of rules:

- $\models c^\tau : \tau$ for any c^τ .

CONST	$\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash c^\tau : \tau$
VAR	$\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash x : \tau \quad \text{if } x \in \text{dom}(\mathcal{T}), \mathcal{T}(x) = \tau$
ABS	$\frac{\mathcal{K}, \mathcal{T}\{x \mapsto \tau_1\}, \mathcal{L} \vdash M_1 : \tau_2}{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash \lambda x. M_1 : \tau_1 \rightarrow \tau_2}$
APP	$\frac{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M_1 : \tau_1 \rightarrow \tau_2 \quad \mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M_2 : \tau_1}{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M_1 M_2 : \tau_2}$
RECORD	$\frac{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M_i : \tau_i \quad (1 \leq i \leq n)}{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash \{l_1 = M_1, \dots, l_n = M_n\} : \{l : \tau_1, \dots, l_n : \tau_n\}}$
INDEX	$\frac{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M : \tau_1 \quad \mathcal{K} \vdash \tau_1 :: \langle l : \tau_2 \rangle \quad \mathcal{L} \vdash \mathcal{I} : \text{index}(l, \tau_1)}{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M[\mathcal{I}] : \tau_2}$
MODIFY	$\frac{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M_1 : \tau_1 \quad \mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M_2 : \tau_2 \quad \mathcal{K} \vdash \tau_1 :: \langle l : \tau_2 \rangle \quad \mathcal{L} \vdash \mathcal{I} : \text{index}(l, \tau_1)}{\mathcal{K}, \mathcal{T} \vdash \text{modify}(M_1, \mathcal{I}, M_2) : \tau_1}$
IABS	$\frac{\mathcal{K}, \mathcal{T}, \mathcal{L}\{I \mapsto \text{index}(l, \tau_1)\} \vdash M : \tau_2}{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash \lambda I. M : \text{index}(l, \tau_1) \Rightarrow \tau_2}$
IAPP	$\frac{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M : \text{index}(l, \tau_1) \Rightarrow \tau_2 \quad \mathcal{L} \vdash \mathcal{I} : \text{index}(l, \tau_1)}{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M \mathcal{I} : \tau_2}$
LET'	$\frac{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M_1 : \tau_1 \quad \mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M_2[M_1/x] : \tau_2}{\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash \text{let } x = M_1 \text{ in } M_2 \text{ end} : \tau_2}$

Figure 2: The typing rules of the implementation calculus

$E, L \vdash c^\tau \Longrightarrow c^\tau$
$E, L \vdash x \Longrightarrow v \quad \text{if } x \in \text{dom}(E) \text{ and } v = E(x)$
$E, L \vdash \lambda x. M \Longrightarrow \text{fun}(E, L, x, M)$
$\frac{E_1, L_1 \vdash M_1 \Longrightarrow \text{fun}(E_2, L_2, x, M_2) \quad E_1, L_1 \vdash M_3 \Longrightarrow v_1 \quad E_2\{x \mapsto v_1\}, L_2 \vdash M_2 \Longrightarrow v_2}{E_1, L_1 \vdash M_1 M_3 \Longrightarrow v_2}$
$\frac{E, L \vdash M_i \Longrightarrow v_i \quad (1 \leq i \leq n)}{E, L \vdash \{l_1 = M_1, \dots, l_n = M_n\} \Longrightarrow \{l_1 = v_1, \dots, l_n = v_n\}}$
$\frac{E, L \vdash M \Longrightarrow \{l_1 = v_1, \dots, l_n = v_n\}}{E, L \vdash M[\mathcal{I}] \Longrightarrow v_i} \quad \text{if } \mathcal{I} = i \text{ or } \mathcal{I} = I, L(I) = i \quad (1 \leq i \leq n)$
$\frac{E, L \vdash M_1 \Longrightarrow \{l_1 = v_1, \dots, l_n = v_n\} \quad E, L \vdash M_2 \Longrightarrow v}{E, L \vdash \text{modify}(M_1, \mathcal{I}, M_2) \Longrightarrow \{l_1 = v_1, \dots, l_i = v, \dots, l_n = v_n\}} \quad \text{if } \mathcal{I} = i \text{ or } \mathcal{I} = I, L(I) = i$
$E, L \vdash \lambda I. M \Longrightarrow \text{lab}(E, L, I, M)$
$\frac{E_1, L_1 \vdash M_1 \Longrightarrow \text{lab}(E_2, L_2, I, M_2) \quad E_2, L_2\{I \mapsto i\} \vdash M_2 \Longrightarrow v_2}{E_1, L_1 \vdash M_1 \mathcal{I} \Longrightarrow v_2} \quad \text{if } \mathcal{I} = i \text{ or } \mathcal{I} = I, L(I) = i$
$\frac{E_1, L_1 \vdash M_1 \Longrightarrow v_1 \quad E_1\{x \mapsto v_1\}, L_1 \vdash M_2 \Longrightarrow v_2}{E_1, L_1 \vdash \text{let } x = M_1 \text{ in } M_2 \text{ end} \Longrightarrow v_2}$

Cases yielding *wrong* are omitted.

Figure 3: Operational Semantics of the Implementation Calculus

- $\models \{l_1 = v_1, \dots, l_n = v_n\} : \{l_1 : \tau_1, \dots, l_n : \tau_n\}$
if $\models v_i : \tau_i$ for all $1 \leq i \leq n$.
- $\models fun(E, L, x, M) : \tau_1 \rightarrow \tau_2$
if $\forall v, v'$ if $\models v : \tau_1$ and $E\{x \mapsto v\}, L \vdash M \Longrightarrow v'$
then $\models v' : \tau_2$.
- $\models lab(E, L, I, M) : \alpha \Rightarrow \tau$
if $\forall v$ if $E, L\{I \mapsto i\} \vdash M \Longrightarrow v$ then $\models v : \tau$
where i is the natural number denoted by α .

In the last rule, since α is a well formed ground index type, it always denotes a natural number. A substitution S is a function from type variables to types. We write $[t_1 \mapsto \tau_1, \dots, t_n \mapsto \tau_n]$ for the substitution S such that $S(t_i) = \tau_i$ ($1 \leq i \leq n$), $S(t) = t$ ($t \notin \{t_1, \dots, t_n\}$). A substitution extends to monotypes and other structures containing monotypes. We identify a substitution and its extension. The composition $S_1 \circ S_2$ of two substitutions S_1 and S_2 is defined as $S_1 \circ S_2(t) = S_1(S_2(t))$. A substitution is *ground* if its range does not contain type variables. A ground substitution S *respects* a kind assignment \mathcal{K} if, for all $t \in dom(\mathcal{K})$, $\emptyset \vdash S(t) :: S(\mathcal{K}(t))$ is a derivable kinding. A variable environment E *respects* a ground type assignment \mathcal{T} if $dom(\mathcal{T}) = dom(E)$ and for all $x \in dom(E)$, $\models E(x) : \mathcal{T}(x)$. An index environment *respects* a ground index assignment \mathcal{L} if $dom(\mathcal{L}) = dom(L)$ and for all $I \in dom(\mathcal{L})$, $L(I)$ is the natural number denoted by $\mathcal{L}(I)$. We then have the following soundness theorem for $\lambda^{let, []}$.

Theorem 1 *If $\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M : \tau$, then for any ground substitution S , if S respects \mathcal{K} , E respects $S(\mathcal{T})$, L respects $S(\mathcal{L})$ and $E, L \vdash M \Longrightarrow v$, then $\models v : \tau$. ■*

Proof: It is easily checked that typings in $\lambda^{let, []}$ are preserved by kind respecting ground substitutions, i.e. if $\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M : \tau$ is a typing in $\lambda^{let, []}$ and S is a ground substitution that respects \mathcal{K} , then $\emptyset, S(\mathcal{T}), S(\mathcal{L}) \vdash M : S(\tau)$ is also a typing. It is then enough to prove the property:

if $\emptyset, \mathcal{T}, \mathcal{L} \vdash M : \tau$ is a *ground* typing, E respects \mathcal{T} , L respects \mathcal{L} and $E, L \vdash M \Longrightarrow v$, then $\models v : \tau$

To prove this property by induction, we use the operational semantics obtained from the one defined in Figure 3 by replacing the last rule for *let* expressions with the following rule:

$$\frac{E_1, L_1 \vdash M_1 \Longrightarrow v_1 \quad E_1, L_1 \vdash M_2[M_1/x] \Longrightarrow v_2}{E_1, L_1 \vdash let\ x = M_1\ in\ M_2\ end \Longrightarrow v_2}$$

It can be shown that the resulting operational semantics is equivalent to the original one. The desired property is then proved by induction on the complexity of raw terms we mentioned earlier. Proof proceeds by

cases. The case for LET follows from the induction hypothesis. Cases other than $M[Z]$, $\lambda I. M$ and $M \mathcal{I}$ are similar to the corresponding proof in [Tof88]. Here we only give the cases for $\lambda I. M$ and leave the cases for $M[Z]$ and $M \mathcal{I}$ to the reader.

Suppose $\emptyset, \mathcal{T}, \mathcal{L} \vdash \lambda I. M : index(l, \tau_1) \Rightarrow \tau_2$, E respects \mathcal{T} , L respects \mathcal{L} and $E, L \vdash \lambda I. M \Longrightarrow lab(E, L, I, M)$. By the definitions of typing rules, we must have $\emptyset, \mathcal{T}, \mathcal{L}\{I \mapsto index(l, \tau_1)\} \vdash M : \tau_2$. Since $index(l, \tau_1)$ is a well formed ground index type, this denotes a natural number. Let this number be i . Then $L\{I \mapsto i\}$ respects $\mathcal{L}\{I \mapsto index(l, \tau_1)\}$. By induction hypothesis, for any v , if $E, L\{I \mapsto i\} \vdash M \Longrightarrow v$, then $\models v : \tau_2$. By the definition of the typing rule for $lab(E, L, I, M)$, we have $\models lab(E, L, I, M) : index(l, \tau_1) \Rightarrow \tau_2$. ■

Later we use this result to establish the soundness of our compilation.

4 Type Inference and Compilation

In order to compile $\lambda^{let, \cdot}$ into $\lambda^{let, []}$, we need to maintain information about type instantiations and to insert appropriate code for index abstraction and index application. We use the technique of type inference to achieve this. For this purpose, we first refine a unification algorithm to *kinded unification*.

4.1 Kinded unification

A *kinded substitution* is a pair consisting of a kind assignment and a substitution. A kinded substitution (\mathcal{K}_1, S) *respects* a kind assignment \mathcal{K}_2 if, for all $t \in dom(\mathcal{K}_2)$, $\mathcal{K}_1 \vdash S(t) :: S(\mathcal{K}_2(t))$ is a derivable kinding. A kinded substitution (\mathcal{K}_1, S_1) is *more general* than (\mathcal{K}_2, S_2) if $S_2 = S_3 \circ S_1$ for some S_3 such that (\mathcal{K}_2, S_3) respects \mathcal{K}_1 . A *kinded set of equations* is a pair consisting of a kind assignment and a set of pairs of types. A kinded substitution (\mathcal{K}_1, S) is a *unifier* of a kinded set of equations (\mathcal{K}_2, E) if it respects \mathcal{K}_2 and $S(\tau_1) = S(\tau_2)$ for all $(\tau_1, \tau_2) \in E$.

Theorem 2 *There is an algorithm \mathcal{U} which, given any kinded set of equations, computes a most general unifier if one exists and reports failure otherwise.*

Proof: We define the algorithm \mathcal{U} in the style of [GS89] by a set of transformation rules on triples (\mathcal{K}, E, S) consisting of a kind assignment \mathcal{K} , a set E of type equations and a set S of “solved” type equations of the form (t, τ) such that $t \notin FTV(\tau)$. Let F range over functions from a finite set of labels to types. We write $\{F\}$ and $\langle F \rangle$ to denote the record type identified by F and the record kind identified by F , respectively. Figure 4 gives the set of transformation rules. Let

U-I	$(\mathcal{K}, E \cup \{(\tau, \tau)\}, S) \Longrightarrow (\mathcal{K}, E, S)$
U-II	$(\mathcal{K} \cup \{t \mapsto U\}, E \cup \{(t, \tau)\}, S) \Longrightarrow ([t \mapsto \tau](\mathcal{K}), [t \mapsto \tau](E), \{(t, \tau)\} \cup [t \mapsto \tau](S))$ if $t \notin FTV(\tau)$
U-III	$(\mathcal{K} \cup \{t_1 \mapsto \langle F_1 \rangle, t_2 \mapsto \langle F_2 \rangle\}, E \cup \{(t_1, t_2)\}, S) \Longrightarrow$ $([t_1 \mapsto t_2](\mathcal{K} \cup \{t_2 \mapsto \langle F \rangle\}), [t_1 \mapsto t_2](E \cup \{(F_1(l), F_2(l)) \mid l \in \text{dom}(F_1) \cap \text{dom}(F_2)\}),$ $\{(t_1, t_2)\} \cup [t_1 \mapsto t_2](S))$ where $F = \{(l, \tau_l) \mid l \in \text{dom}(F_1) \cup \text{dom}(F_2), \tau_l = F_1(l) \text{ if } l \in \text{dom}(F_1) \text{ otherwise } \tau_l = F_2(l)\}$ if t_1 does not appear in F_2 and t_2 does not appear in F_1
U-IV	$(\mathcal{K} \cup \{t_1 \mapsto \langle F_1 \rangle\}, E \cup \{(t_1, \{F_2\})\}, S) \Longrightarrow$ $([t_1 \mapsto \{F_2\}](\mathcal{K}), [t_1 \mapsto \{F_2\}](E \cup \{(F_1(l), F_2(l)) \mid l \in \text{dom}(F_1) \cap \text{dom}(F_2)\}), \{(t_1, \{F_2\})\} \cup [t_1 \mapsto \{F_2\}](S))$ if $\text{dom}(F_1) \subseteq \text{dom}(F_2)$ and $t \notin FTV(\{F_2\})$
U-V	$(\mathcal{K}, E \cup \{(\tau_1^1 \rightarrow \tau_1^2, \tau_2^1 \rightarrow \tau_2^2)\}, S) \Longrightarrow (\mathcal{K}, E \cup \{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\}, S)$
U-VI	$(\mathcal{K}, E \cup \{(\{F_1\}, \{F_2\})\}, S) \Longrightarrow (\mathcal{K}, E \cup \{(F_1(l), F_2(l)) \mid l \in \text{dom}(F_1)\}, S)$ if $\text{dom}(F_1) = \text{dom}(F_2)$

Figure 4: Transformation Rules for Unification

(\mathcal{K}, E) be a given kinded set of equations. The algorithm \mathcal{U} first transforms $(\mathcal{K}, E, \emptyset)$ to (\mathcal{K}', E', S') until no more rules can apply. It then returns (\mathcal{K}', S') if E' is empty; otherwise it reports failure. The correctness of the algorithm is proved by showing that each transformation rule preserves the following property:

If $(\mathcal{K}_1, E_1, S_1) \Longrightarrow (\mathcal{K}_2, E_2, S_2)$ then a kinded substitution is a unifier of $(\mathcal{K}_1, E_1 \cup S_1)$ if and only if it is a unifier of $(\mathcal{K}_2, E_2 \cup S_2)$.

Cases other than U-III and U-IV are same as in [GS89]. Here we show the case of the rule U-III. The rule U-IV is simpler.

Suppose (K, σ) is a unifier of $(\mathcal{K} \cup \{t_1 \mapsto \langle F_1 \rangle, t_2 \mapsto \langle F_2 \rangle\}, E \cup \{(t_1, t_2)\}, S)$. Then $K \vdash \sigma(t_1) :: \sigma(\langle F_1 \rangle)$, $K \vdash \sigma(t_2) :: \sigma(\langle F_2 \rangle)$, and $\sigma(t_1) = \sigma(t_2)$. By the definition of kindings, we have $\sigma(F_1(l)) = \sigma(F_2(l))$ for all $l \in \text{dom}(F_1) \cap \text{dom}(F_2)$. Therefore $K \vdash \sigma(t_2) :: \sigma(\langle F \rangle)$. Since $\sigma(t_1) = \sigma(t_2)$, $\sigma([t_1 \mapsto t_2](E)) = \sigma(E)$ and $\sigma([t_1 \mapsto t_2](S)) = \sigma(S)$. Thus (K, σ) is also a unifier of $(\mathcal{K} \cup \{t_2 \mapsto \langle F \rangle\}, [t_1 \mapsto t_2](E \cup \{(F_1(l), F_2(l)) \mid l \in \text{dom}(F_1) \cap \text{dom}(F_2)\}), \{(t_1, t_2)\} \cup [t_1 \mapsto t_2](S))$. Conversely, suppose (K, σ) is a unifier of $(\mathcal{K} \cup \{t_2 \mapsto \langle F \rangle\}, [t_1 \mapsto t_2](E \cup \{(F_1(l), F_2(l)) \mid l \in \text{dom}(F_1) \cap \text{dom}(F_2)\}), \{(t_1, t_2)\} \cup [t_1 \mapsto t_2](S))$. Then $K \vdash \sigma(t_2) :: \sigma(\langle F \rangle)$, $\sigma([t_1 \mapsto t_2](F_1(l))) = \sigma([t_1 \mapsto t_2](F_2(l)))$ for all $l \in \text{dom}(F_1) \cap \text{dom}(F_2)$, and $\sigma(t_1) = \sigma(t_2)$. By the definition of kindings, this implies that $K \vdash \sigma(t_1) :: \sigma(\langle F_1 \rangle)$, $K \vdash \sigma(t_2) :: \sigma(\langle F_2 \rangle)$. Since $\sigma(t_1) = \sigma(t_2)$, $\sigma([t_1 \mapsto t_2](E)) = \sigma(E)$ and $\sigma([t_1 \mapsto t_2](S)) = \sigma(S)$. Thus (K, σ) is also a uni-

fier of $(\mathcal{K} \cup \{t_1 \mapsto \langle F_1 \rangle, t_2 \mapsto \langle F_2 \rangle\}, E \cup \{(t_1, t_2)\}, S)$.

The termination can be proved by showing that each transformation rule decreases the termination measure of the lexicographical pair consisting of the number of type variables in E and the total size of E .

If the transformation terminates with $(\mathcal{K}, \emptyset, S)$, then it is obvious that (\mathcal{K}, S) is a most general unifier of $(\mathcal{K}, \emptyset, S)$. It is also easily checked that if the transformation terminates with non empty E then E has no unifier. ■

4.2 The algorithm for compilation and type inference

We now give an algorithm that simultaneously computes a principal type scheme and a compiled implementation term.

A typing $\mathcal{K}_1, \mathcal{T}_1 \vdash e : \tau_1$ is *more general than* $\mathcal{K}_2, \mathcal{T}_2 \vdash e : \tau_2$ if $\text{dom}(\mathcal{T}_1) \subseteq \text{dom}(\mathcal{T}_2)$, and there is a substitution S such that the kinded substitution (\mathcal{K}_2, S) respects \mathcal{K}_1 , $\mathcal{T}_2(t) = S(\mathcal{T}_1(t))$ for all $t \in \text{dom}(\mathcal{T}_1)$, and $\tau_2 = S(\tau_1)$. A typing $\mathcal{K}, \mathcal{T} \vdash e : \tau$ is *principal* if it is more general than all the derivable typings for e . We then have the following theorem.

Theorem 3 *There is an algorithm \mathcal{C} which takes a raw term and returns either $(\mathcal{K}, \mathcal{T}, \mathcal{L}, M, \tau)$ or failure such that if it returns $(\mathcal{K}, \mathcal{T}, \mathcal{L}, M, \tau)$, then $\mathcal{K}, \mathcal{T} \vdash e : \tau$ is a principal typing in $\lambda^{\text{let}, \bullet}$ and $\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M : \tau$ is a typing in $\lambda^{\text{let}, \bullet}$; otherwise e has no typing in $\lambda^{\text{let}, \bullet}$.*

Proof: We follow [Mit90] and present \mathcal{C} as

$$\mathcal{C}(e) = \text{Comp}(e, \emptyset)$$

where Comp is an algorithm which takes a raw term and an environment A which maps a finite set of variables to tuples of the form $(\mathcal{K}, \mathcal{T}, \tau)$. The purpose of the environment A is to maintain a principal typing of let bound variables.

A complete definition of Comp is given in Figure 5.

Since the treatment of let bound variables through an auxiliary parameter A to Comp is the same as the proof of the corresponding theorem in [Mit90], the correctness of the treatment of let expressions is shown similarly. It is therefore enough to show that (1) the cases other than let bound variables and let expressions preserve the principal typing property of e , and (2) each case yields a provable typing of M in $\lambda^{\text{let}, []}$.

Here we sketch below the proof of the property (1) for the case of $e.l$. The case for $\text{modify}(e_1, l, e_2)$ is similar and all the other cases are essentially the same as in [Mit90].

By the definition of typing rules of $\lambda^{\text{let}, \bullet}$, $e_1.l$ has a typing of the form $\mathcal{K}, \mathcal{T} \vdash e_1.l : \tau$ iff e_1 has a typing of the form $\mathcal{K}, \mathcal{T} \vdash e_1 : \tau'$ and $\mathcal{K} \vdash \tau' :: \langle l : \tau \rangle$. By induction hypothesis, e_1 has a typing $\mathcal{K}, \mathcal{T} \vdash e_1 : \tau'$ iff $(\mathcal{K}_1, \mathcal{T}_1, \mathcal{L}_1, M_1, \tau_1) = \text{Comp}(e_1, A)$ and $\mathcal{K}_1, \mathcal{T}_1 \vdash e : \tau_1$ is a more general than $\mathcal{K}, \mathcal{T} \vdash e_1 : \tau'$. Under the fact that $\mathcal{K} \vdash \tau' :: \langle l : \tau \rangle$, the later proposition of the previous sentence implies that $(\mathcal{K}_1 \cup \{t_1 \mapsto U, t_2 \mapsto \langle l : t_1 \rangle\}, \{\tau_1, t_2\})$ (t_1, t_2 fresh) has a unifier. The desired property then follows from that of \mathcal{U} .

The property of (2) can be shown by cases. The cases of let and let bound variables can be shown by using the following two properties of $\lambda^{\text{let}, []}$: (1) $\mathcal{K}, \mathcal{T}, \mathcal{L}\{I \mapsto \text{index}(l, \tau_1)\} \vdash M : \tau_2$ is a derivable typing in $\lambda^{\text{let}, []}$ iff so is $\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash \lambda I. M : \text{index}(l, \tau_1) \Rightarrow \tau_2$, and (2) $\mathcal{K}, \mathcal{T}, \mathcal{L}\{I \mapsto \text{index}(l, \tau_1)\} \vdash M I : \tau_2$ is a derivable typing in $\lambda^{\text{let}, []}$ iff so is $\mathcal{K}, \mathcal{T}, \mathcal{L} \vdash M : \text{index}(l, \tau_1) \Rightarrow \tau_2$. Other cases can be show by simple inductive reasoning. \blacksquare

Since we have shown the soundness of the type system of $\lambda^{\text{let}, []}$, the above theorem implies the following soundness property of the type system of $\lambda^{\text{let}, \bullet}$ with respect to the evaluation through the compilation.

Corollary 1 *If $\mathcal{C}(e) = (\mathcal{K}, \mathcal{T}, \mathcal{L}, M, \tau)$, E is a value environment respecting \mathcal{T} and L is an index environment respecting \mathcal{L} , and $E, L \vdash M \Rightarrow v$, then $\models v : \tau$. \blacksquare*

We end this section by showing some examples of the compilation.

$$\begin{aligned} \mathcal{C}(\lambda x. x.l) \\ = (\{t_1 \mapsto U, t_2 \mapsto \langle l : t_1 \rangle\}, \emptyset, \{I \mapsto \text{index}(l, t_2)\}, \\ \lambda x. x[I], t_2 \rightarrow t_1) \end{aligned}$$

$$\begin{aligned} \mathcal{C}(\text{let name} = \lambda x. x.\text{Name in} \\ \text{ name } \{ \text{Name} = \text{"Joe"}, \text{Age} = 21 \} \\ \text{end}) \\ = (\emptyset, \emptyset, \emptyset, \text{let name} = \lambda I. \lambda x. x[I] \text{ in} \\ \text{ (name 2) } \{ \text{Name} = \text{"Joe"}, \text{Age} = 21 \} \\ \text{end, string}) \end{aligned}$$

The following shows an examples of compiling an ML-style type inference session. We write $\text{let } x = e$; for $\text{let } x = e \text{ in } x \text{ end}$ and the subsequent expressions of the form e' ; are regarded as shorthand for $\text{let } x = e \text{ in } e'$. For the following type inference session:

$$\begin{aligned} (* \text{ to move a point-shape object horizontally *)} \\ \text{let inc}_X = \lambda p. \text{modify}(p, X, p.X + 1); \\ \rightarrow : \forall t :: \langle X : \text{int} \rangle. t \rightarrow t \end{aligned}$$

$$\begin{aligned} \text{let salary} = \lambda x. x.\text{Salary}; \\ \rightarrow : \forall t_1 :: U. \forall t_2 :: \langle \text{Salary} : t_1 \rangle. t_2 \rightarrow t_1 \end{aligned}$$

$$\begin{aligned} \text{salary } \{ \text{Name} = \text{"Susan"}, \text{Age} = 21, \\ \text{Salary} = 34000 \}; \\ \rightarrow : \text{int} \end{aligned}$$

$$\begin{aligned} \text{let wealthy} = \lambda x. (\text{salary } x) > 100000; \\ \rightarrow : \forall t :: \langle \text{Salary} : \text{int} \rangle. t \rightarrow \text{bool} \end{aligned}$$

$$\begin{aligned} \text{wealthy } \{ \text{Name} = \text{"Susan"}, \text{Age} = 21, \\ \text{Salary} = 34000 \}; \\ \rightarrow : \text{bool} \end{aligned}$$

the compilation algorithm produces the following code:

$$\begin{aligned} \text{let inc}_X = \lambda I \lambda p. \text{modify}(p, I, p[I] + 1); \\ \rightarrow : \forall t :: \langle X : \text{int} \rangle. \text{index}(X, t) \Rightarrow t \rightarrow t \end{aligned}$$

$$\begin{aligned} \text{let salary} = \lambda I \lambda x. x[I]; \\ \rightarrow : \forall t_1 :: U \forall t_2 :: \langle \text{Salary} : t_1 \rangle. \\ \text{index}(\text{Salary}, t_2) \Rightarrow t_2 \rightarrow t_1 \end{aligned}$$

$$\begin{aligned} (\text{salary } 3) \{ \text{Name} = \text{"Susan"}, \text{Age} = 21, \\ \text{Salary} = 34000 \}; \\ \rightarrow : \text{int} \end{aligned}$$

$$\begin{aligned} \text{let wealthy} = \lambda I. \lambda x. ((\text{salary } I) x) > 100000; \\ \rightarrow : \forall t :: \langle \text{Salary} : \text{int} \rangle. \\ \text{index}(\text{Salary}, t) \Rightarrow t \rightarrow \text{bool} \end{aligned}$$

$$\begin{aligned} (\text{wealthy } 3) \{ \text{Name} = \text{"Susan"}, \text{Age} = 21, \\ \text{Salary} = 34000 \}; \\ \rightarrow : \text{bool} \end{aligned}$$

5 Calculi with Subtyping

One might want to develop a similar compilation algorithm for a polymorphic record calculus with subtyping. As we mentioned in the introduction, however,

$Comp(x, A) = \text{if } x \in \text{dom}(A) \text{ then}$
 let $(\mathcal{K}_1, \mathcal{T}_1, \tau_1) = A(x)$ (with all type variables renamed with fresh names)
 $(\alpha_1 \Rightarrow \dots \Rightarrow \alpha_n \Rightarrow \tau'_1) = \tau_1$ ($0 \leq n$ and τ'_1 does not contain \Rightarrow)
 $\mathcal{I}_i = (\text{if } \alpha_i \equiv \text{index}(l, t) \text{ then } I_i \text{ (fresh) else the integer denoted by } \alpha_i)$
 $\mathcal{L}_1 = \{I_i \mapsto \alpha_i \mid \text{for new } I_i \text{ introduced above}\}$,
 in $(\mathcal{K}_1, \mathcal{T}_1, \mathcal{L}_1, ((\dots(x \mathcal{I}_1) \dots) \mathcal{I}_n), \tau'_1)$
 else $(\{t \mapsto U\}, \{x \mapsto t\}, \emptyset, x, t)$

$Comp(c^\tau, A) = (\emptyset, \emptyset, \emptyset, c^\tau, \tau)$

$Comp(\lambda x. e_1, A) = \text{let } (\mathcal{K}_1, \mathcal{T}_1, \mathcal{L}_1, M_1, \tau_1) = Comp(e_1, A)$
 in if $x \in \text{dom}(\mathcal{T}_1)$ then $(\mathcal{K}_1, \mathcal{T}_1 \uparrow^{\text{dom}(\mathcal{T}_1) \setminus \{x\}}, \mathcal{L}_1, \lambda x. M_1, \mathcal{T}_1(x) \rightarrow \tau_1)$
 else $(\mathcal{K}_1 \{t \mapsto U\}, \mathcal{T}_1, \mathcal{L}_1, \lambda x. M_1, t \rightarrow \tau_1)$ (t fresh)

$Comp(e_1 e_2, A) =$
 let $(\mathcal{K}_1, \mathcal{T}_1, \mathcal{L}_1, M_1, \tau_1) = Comp(e_1, A)$
 $(\mathcal{K}_2, \mathcal{T}_2, \mathcal{L}_2, M_2, \tau_2) = Comp(e_2, A)$
 $(\mathcal{K}_3, S) = \mathcal{U}(\mathcal{K}_1 \cup \mathcal{K}_2 \cup \{t \mapsto U\}, \{(\mathcal{T}_1(x), \mathcal{T}_2(x)) \mid x \in \text{dom}(\mathcal{T}_1) \cap \text{dom}(\mathcal{T}_2)\} \cup \{(\tau_2 \rightarrow t, \tau_1)\})$ (t fresh)
 in $(\mathcal{K}_3, S(\mathcal{T}_1 \cup \mathcal{T}_2), S(\mathcal{L}_1 \cup \mathcal{L}_2), M_1 M_2, S(t))$

$Comp(\{l_1 = e_1, \dots, l_n = e_n\}, A) =$
 let $(\mathcal{K}_i, \mathcal{T}_i, \mathcal{L}_i, M_i, \tau_i) = Comp(e_i, A)$ ($1 \leq i \leq n$)
 $\mathcal{T}'_i = \mathcal{T}_i \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$
 where $\{x_1, \dots, x_n\} = (\bigcup_{1 \leq j \leq n} \text{dom}(\mathcal{T}_j)) \setminus \text{dom}(\mathcal{T}_i)$ and $\{t_1, \dots, t_n\}$ are all fresh
 $(\mathcal{K}, S) = \mathcal{U}(\mathcal{K}_1 \cup \dots \cup \mathcal{K}_n \cup \{t_i \mapsto U \mid 1 \leq i \leq n\}, \{(\mathcal{T}'_i(x), \mathcal{T}'_{i+1}(x)) \mid x \in \text{dom}(\mathcal{T}'_i), 1 \leq i \leq n-1\})$
 in $(\mathcal{K}, S(\mathcal{T}_1), S(\mathcal{L}_1 \cup \dots \cup \mathcal{L}_n), \{l_1 = M_1, \dots, l_n = M_n\}, S(\{l_1 : \tau_1, \dots, l_n : \tau_n\}))$

$Comp(e_1 \cdot l, A) = \text{let } (\mathcal{K}_1, \mathcal{T}_1, \mathcal{L}_1, M_1, \tau_1) = Comp(e_1, A)$
 $(\mathcal{K}, S) = \mathcal{U}(\mathcal{K}_1 \cup \{t_1 \mapsto U, t_2 \mapsto \langle l : t_1 \rangle\}, \{(\tau_1, t_2)\})$ (t_1, t_2 fresh)
 in if $S(t_2) = \{l_1 : \tau_1^1, \dots, l_n : \tau_n^1\}$ then
 $(\mathcal{K}, S(\mathcal{T}_1), S(\mathcal{L}_1), M[j], S(t_1))$ where $j = \text{Size}(\{l \mid l \in \{l_1, \dots, l_n\}, l \leq l_i\})$
 else $(\mathcal{K}, S(\mathcal{T}_1), S(\mathcal{L}_1) \cup \{I \mapsto \text{index}(l, S(t_2))\}, M[I], S(t_1))$

$Comp(\text{modify}(e_1, l, e_2), A) =$
 let $(\mathcal{K}_1, \mathcal{T}_1, \mathcal{L}_1, M_1, \tau_1) = Comp(e_1, A)$
 $(\mathcal{K}_2, \mathcal{T}_2, \mathcal{L}_2, M_2, \tau_2) = Comp(e_2, A)$
 $(\mathcal{K}_3, S) = \mathcal{U}(\mathcal{K}_1 \cup \mathcal{K}_2 \cup \{t_1 \mapsto U, t_2 \mapsto \langle l : t_1 \rangle\},$
 $\{(\mathcal{T}_1(x), \mathcal{T}_2(x)) \mid x \in \text{dom}(\mathcal{T}_1) \cap \text{dom}(\mathcal{T}_2)\} \cup \{(\tau_2, t_1), (\tau_1, t_2)\})$ (t_1, t_2 fresh)
 in if $S(t_2) = \{l_1 : \tau_1^1, \dots, l_n : \tau_n^1\}$ then
 $(\mathcal{K}_3, S(\mathcal{T}_1 \cup \mathcal{T}_2), S(\mathcal{L}_1 \cup \mathcal{L}_2), \text{modify}(M_1, j, M_2), S(t_2))$ where $j = \text{Size}(\{l \mid l \in \{l_1, \dots, l_n\}, l \leq l_i\})$
 else $(\mathcal{K}_3, S(\mathcal{T}_1 \cup \mathcal{T}_2), S(\mathcal{L}_1 \cup \mathcal{L}_2) \cup \{I \mapsto \text{index}(l, S(t_2))\}, \text{modify}(M_1, I, M_2), S(t_2))$

$Comp(\text{let } x = e_1 \text{ in } e_2 \text{ end}, A) =$
 let $(\mathcal{K}_1, \mathcal{T}_1, \mathcal{L}_1, M_1, \tau_1) = Comp(e_1, A)$
 $\{I_1 \mapsto \alpha_1, \dots, I_n \mapsto \alpha_n, I'_1 \mapsto \alpha'_1, \dots, I'_m \mapsto \alpha'_m\} = \mathcal{L}_1$ (α_i not ground, α'_i ground)
 $A' = A \{x \mapsto (\mathcal{K}_1, \mathcal{T}_1, \alpha_1 \Rightarrow \dots \Rightarrow \alpha_n \Rightarrow \tau_1)\}$
 $(\mathcal{K}_2, \mathcal{T}_2, \mathcal{L}_2, M_2, \tau_2) = Comp(e_2, A')$
 $(\mathcal{K}, S) = \mathcal{U}(\mathcal{K}_1 \cup \mathcal{K}_2, \{(\mathcal{T}_1(x), \mathcal{T}_2(x)) \mid x \in \text{dom}(\mathcal{T}_1) \cap \text{dom}(\mathcal{T}_2)\})$
 M'_1 be the term obtained from M_1 by replacing occurrences of I'_i with the natural number denoted by α'_i
 in $(\mathcal{K}, S(\mathcal{T}_1 \cup \mathcal{T}_2), S(\mathcal{L}_2), \text{let } x = \lambda I_1. \dots \lambda I_n. M_1 \text{ in } M_2 \text{ end}, S(\tau_2))$

Figure 5: The Compilation Algorithm

some difficulty seems to be inherent in any record calculus containing the familiar subsumption rule of the form:

$$\text{SUB} \quad \frac{\vdash e : \tau_1 \quad \tau_1 \leq \tau_2}{\vdash e : \tau_2}$$

To see the difficulty, consider the expression:

$$e \equiv \text{if } e_1 \text{ then } \{A = \text{"abc"}, B = \text{true}\} \\ \text{else } \{B = \text{true}, C = \text{"abc"}\}$$

With the existence of the subsumption rule, this expression has the type $\{B : \text{bool}\}$. However, the actual set of labels of the value denoted by e depends on the value denoted by e_1 and therefore the offset of the label B can not be statically determined. It is therefore impossible to compile an expression such as $(\lambda x. x \cdot B) e$ into the implementation calculus. A similar problem arises in combination of bulk data types such as list or set types. This observation shows that in a record calculus with the subsumption rule, it is in general impossible to statically compute offsets of labels. This property is related to *loss of type information* – the phenomenon first observed in [CW85]. With the subsumption rule, a typing judgment of the form $\vdash e : \{l_1 : \tau_1, \dots, l_n : \tau_n\}$ no longer implies that e denotes a record value having the exact type $\{l_1 : \tau_1, \dots, l_n : \tau_n\}$.

As we have discussed in [BTBO89], this property is also problematic in dealing with operations such as equality and database join, which require the exact type information of their parameters. It remains to be investigated whether there is some interesting subset of a polymorphic record calculus with subtyping that does not suffer from this problem and allow efficient compilation. Another possibility of overcoming this difficulty would be to enrich a calculus with a new form of judgments for *exact typing*.

6 Conclusions

We have established a compilation method for a polymorphic type inference system with labeled records. The method always yields code that deals with field selection and field modification by a few machine instructions. The polymorphic type discipline for records is shown to be sound with respect to the compilation.

The presented algorithm can readily be incorporated in existing ML-style polymorphic programming languages with records. For example, Standard ML [MTH90] contains records and monomorphic field selection. By refining its type system to incorporate kinded abstraction and incorporating its compiler with our compilation algorithm, the language can be refined to support polymorphic field selection and its efficient execution without altering its language syntax.

As we mentioned in the introduction, one important extension to the work presented here is to include more powerful operations on records such as concatenation and join. To represent these operations, the polymorphic type system must be extended. One approach is to introduce constraints on instantiation of type variables of the form

$$t = \tau_1 * \tau_2$$

which denotes the requirement that the instantiation of t is restricted to those $S(t)$ such that $S(t)$ is a record type equal to $S(\tau_1) * S(\tau_2)$, where $*$ is an appropriate operator on record types representing join or concatenation. Typing mechanism incorporating this idea have been developed in [OB88, Wan89] for type inference systems and in [CM89, HP91] for second-order type systems. However, eliminating time consuming run-time scan of the fields of two records seems to require a substantial extension of the implementation calculus and the compilation strategy.

References

- [AB87] M.P. Atkinson and O.P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 1987.
- [BTBO89] V. Breazu-Tannen, P. Buneman, and A. Ohori. Can object-oriented databases be statically typed? In *Proc. 2nd International Workshop on Database Programming Languages*, pages 226 – 237, 1989. Morgan Kaufmann Publishers.
- [Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. (Special issue devoted to Symp. on Semantics of Data Types, 1984).
- [Car91] L. Cardelli. Extensible records in a pure calculus of subtyping. Technical report, DEC Systems Research Center, 1991.
- [CDMB89] R. Connor, A. Dearle, R. Morrison, and F. Brown. An object addressing mechanism for statically typed languages with multiple inheritance. In *Proc. ACM OOP-SLA Conference*, pages 279–285, 1989.
- [CM89] L. Cardelli and J. Mitchell. Operations on records. In *Proc. Mathematical Foundation of Programming Semantics, Lecture Notes in Computer Science 442*, pages 22–52, 1989.

- [CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.
- [DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
- [Gir71] J.-Y. Girard. Une extension de l’interprétation de gödel à l’analyse, et son application à l’élimination des coupures dans l’analyse et théorie des types. In *Second Scandinavian Logic Symposium*. North-Holland, 1971.
- [GS89] J. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67(2):203–260, 1989.
- [HP91] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proc. ACM Symp. on Principles of Programming Languages*, 1991.
- [JM88] L. A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Conference on LISP and Functional Programming*, pages 198–211, 1988.
- [MH88] J. C. Mitchell and R. Harper. The essence of ML. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 28–46, 1988.
- [Mit84] J.C. Mitchell. Coercion and type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 175–185, 1984.
- [Mit90] J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. MIT Press/Elsevier, 1990.
- [MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.
- [OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conference on LISP and Functional Programming*, pages 174–183, 1988.
- [OB89] A. Ohori and P. Buneman. Static type inference for parametric classes. In *Proc. ACM OOPSLA Conference*, pages 445–456, 1989.
- [Oho89a] A. Ohori. A simple semantics for ML polymorphism. In *Proc. ACM/IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 281–292, 1989.
- [Oho89b] A. Ohori. *A Study of Types, Semantics and Languages for Databases and Object-oriented Programming*. PhD thesis, University of Pennsylvania, 1989.
- [Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In *ACM Conference on Principles of Programming Languages*, pages 242–249, 1989.
- [Rém90] D. Rémy. Typechecking records in a natural extension of ML. Technical report, INRIA–Rocquencourt, Le Chesnay Cedex, France, 1990.
- [Rém91] D. Rémy. Typing record concatenation for free. Technical report, INRIA–Rocquencourt, Le Chesnay Cedex, France., 1991.
- [Rey74] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425. Springer-Verlag, 1974.
- [Sta88] R. Stansifer. Type inference with subtypes. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 88–97, 1988.
- [Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Department of Computer Science, University of Edinburgh, 1988.
- [Wan87] M. Wand. Complete type inference for simple objects. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987.
- [Wan88] M. Wand. Corrigendum : Complete type inference for simple object. In *Proc. IEEE Symposium on Logic in Computer Science*, 1988.
- [Wan89] M. Wand. Type inference for records concatenation and simple objects. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 92–97, 1989.