A Polymorphic Record Calculus and Its Compilation

ATSUSHI OHORI
Kyoto University

The motivation of this work is to provide a type-theoretical basis for developing a practical polymorphic programming language with labeled records and labeled variants. Our goal is to establish both a polymorphic type discipline and an efficient compilation method for a calculus with those labeled data structures. We define a second-order, polymorphic record calculus as an extension of Girard-Reynolds polymorphic lambda calculus. We then develop an ML-style type inference algorithm for a predicative subset of the second-order record calculus. The soundness of the type system and the completeness of the type inference algorithm are shown. These results extend Milner's type inference algorithm, Damas and Milner's account of ML's let polymorphism, and Harper and Mitchell's analysis on XML To establish an efficient compilation method for the polymorphic record calculus, we first define an implementation calculus, where records are represented as vectors whose elements are accessed by direct indexing, and variants are represented as values tagged with a natural number indicating the position in the vector of functions in a switch statement. We then develop an algorithm to translate the polymorphic record calculus into the implementation calculus using type information obtained by the type inference algorithm. The correctness of the compilation algorithm is proved, that is, the compilation algorithm is shown to preserve both typing and the operational behavior of a program. Based on these results, Standard ML has been extended with labeled records, and its compiler has been implemented.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory, D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*

General Terms: Languages

Additional Key Words and Phrases: Compilation, polymorphism, record calculus, type inference, type theory

1. INTRODUCTION

Labeled records and labeled variants are widely used data structures and are essential building blocks in various data-intensive applications such as database programming. Despite their practical importance, however, existing polymorphic programming languages do not properly support these data structures. Standard ML

A preliminary summary of some of the results of this article appeared in Proceedings of ACM Symposium on Principles of Programming Languages, 1992, under the title "A compilation method for ML-style polymorphic record calculi."

This work was partly supported by the Japanese Ministry of Education under scientific research grant no. 06680319.

Author's address: Research Institute for Mathematical Sciences, Kyoto University, Sakyo-ku, Kyoto 606-01, JAPAN; email: ohori@kurims.kyoto-u.ac.jp

[Milner et al. 1990] contains labeled records and a form of labeled variants, but their allowable operations are restricted to monomorphic ones. For example, consider the following simple function *name* on records:

$$name \equiv \lambda x.x.\mathsf{Name}$$

where x.Name is our syntax for selecting the Name field from record x. This function is polymorphic in the sense that it can be applied to terms of any record type containing a *Name* field, such as $\{Name : string, Age : int\}$ or $\{Name : string, Office : int\}$. One way of writing this function in ML is

```
fun name(x) = #Name x
```

where #Name is an ML's syntax for $\lambda x.x.\mathsf{Name}$. This program is rejected by the current ML compiler, unless the programmer explicitly specifies the type of x that restricts the possible argument values to be those records that have a fixed set of labels, as in:

```
fun name(x:{Name:string, Age:int}) = #Name x
```

Unfortunately, writing such a type specification is not only cumbersome but also eliminates most of the flexibility of functions operating on records like *name* above.

An analogous situation exists for labeled variants. We use variant types (disjoint union types) when we want to treat values of different types uniformly. A significant advantage of labeled variants over simple disjoint union types is that they support flexible programming by designating the kind of a value by a symbolic label. For example, consider a variant value *payment* defined as:

$$payment \equiv \langle \mathsf{Pound}{=}100.0 \rangle$$

where value 100.0 is tagged with variant label Pound. *payment* can be treated as a value of various different variant types such as $\langle Pound : real, Dollar : real \rangle$ or $\langle Pound : real, Yen : int \rangle$ or any other variant types containing a $Pound : real$ field. Unfortunately, this form of flexible programming is unavailable in existing polymorphic languages since they restrict variant values to be monomorphic. In Standard ML, for example, a variant type may be defined as:

```
datatype PoundOrYen = Pound of real | Yen of int
```

But this definition ties the variant labels Pound and Yen to this particular type PoundOrYen. As a consequence, if a value such as *payment* is defined for this type, it cannot be used as a value of other variant types. In Standard ML programming, a commonly adopted ad hoc strategy used to get around this problem is to define a variant type containing all the possible components and to omit some of the cases when manipulating variant values. This approach, however, introduces runtime exceptions of "match failure," many of which are essentially type errors that should have been caught at compile time.

It is highly desirable to extend a polymorphic programming language to allow polymorphic manipulation of labeled records and labeled variants. In this article, we use the term *record polymorphism* to refer to the form of polymorphism required for polymorphic manipulation of both labeled records and labeled variants. Our goal is to provide a basis to develop a polymorphic programming language that supports record polymorphism. There are two technical challenges in achieving this goal. The first is the development of a static type system that can represent

record polymorphism. The second is the development of an efficient compilation method for polymorphic operations on records and variants. In this article, we provide solutions to the two problems. In the rest of this section, we shall explain the problems and outline the solutions presented in this article. Part of this article is based on a preliminary presentation of kinded abstraction and type-inference-based compilation [Ohori 1992].

1.1 Static Type System for Record Polymorphism

Record polymorphism is based on the property that labeled-field access is polymorphic and can therefore be applied to any labeled data structure containing the specified field. For example, the function *name*, which accesses the Name field in a record, and the value *payment*, which accesses the Pound branch in a case statement, have all the types of the forms:

$$name : \{Name : \tau, \cdots\} \rightarrow \tau,$$
$$payment : \langle Pound : real, \cdots \rangle$$

However, conventional polymorphic type systems cannot represent the set of all possible types of those shapes and therefore cannot represent the polymorphic nature of programs containing those terms.

Cardelli [1988] observed that this form of polymorphism can be captured by defining a *subtyping* relation and allowing a value to have all its supertypes. This approach also supports certain aspects of *method inheritance* and provides a type-theoretical basis for object-oriented programming. Cardelli and Wegner [1985] extended this approach to a second-order type system. Type inference systems with subtyping have also been developed [Fuh and Mishra 1988; Mitchell 1984; Stansifer 1988]. It is, however, not clear whether or not the mechanism for record polymorphism should be coupled with a strong mechanism of subtyping. In the presence of subtyping, a static type no longer represents the exact record structure of a runtime value. For example, a term

```
if true then {A = 1, B = true} else {B=false, C="Cat"}
```

has a type $\{B : bool\}$, but its runtime value would presumably be {A=1,B=true}. This property may be problematic when we want to deal with those operations, such as equality test, that depend on the exact structure of values. As we shall discuss in Section 6, subtyping also complicates implementation.

An alternative approach, initiated by Wand [1987; 1988], is to extend ML-style polymorphic typing directly to record polymorphism. This idea was further developed in a number of type inference systems [Jategaonkar and Mitchell 1993; Ohori and Buneman 1988; 1989; Rémy1989; 1992; 1994b; Wand 1989].

In these type systems, a most general polymorphic type scheme can be 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 a value of various different types. This approach not only captures the polymorphic nature of functions on records but also integrates record polymorphism and ML-style type inference, which relieves the programmer from writing complicated type declarations required in explicit second-order calculi.

Most of the proposed type inference systems have been based on the mechanism of *row variables* [Wand 1987], which are variables ranging over finite sets of field types.

fun move p = modify(p, X, p.X + 1) : $\forall t::\{\!\{X : int\}\!\}.t \to t$
move({X=10, Y=0, Color="Red"}) : $\{X : int, Y : int, Color : string\}$
fun transpose p = modify(modify(p, X, p.Y), Y, p.X)
: $\forall t_1::U.\forall t_2::\{\!\{X : t_1, Y : t_1\}\!\}.t_2 \to t_2$
transpose({X=1.0, Y=10.0, Direction={Speed=30.0, Theta=1.0}})
: $\{X : real, Y : real, Direction : \{Speed : real, Theta : real\}\}$
fun dist p = case p of ⟨Cartesian=λc.sqroot(square(c.X) + square(c.Y)), Polar=λp.R⟩
: $\forall t_1::\{\!\{X : real, Y : real\}\!\}.\forall t_2::\{\!\{R : real\}\!\}.\langle Cartesian : t_1, Polar : t_2\rangle \to real$
dist(⟨Cartesian={X=0.0, Y=10.0, Color="Green"}⟩) : $real$

Fig. 1. Example of programs with their inferred types.

Here, instead of using row variables, we base our development on the idea presented in Ohori and Buneman [1988] of placing restrictions on possible instantiations of type variables. We formalize this idea as a *kind system* of types and refine the ordinary type quantification to *kinded quantification* of the form $\forall t::k.\sigma$ where type variable t is constrained to range only over the set of types denoted by a kind k. This mechanism is analogous to *bounded quantification* [Cardelli and Wegner 1985]. A kind k is either the universal kind U denoting the set of all types, a record kind of the form $\{\!\{l_1 : \tau_1, \cdots, t_n : \tau_n\}\!\}$ denoting the set of all record types that contain the specified fields, or a variant kind of the form $\langle\!\langle l_1 : \tau_1, \cdots, t_n : \tau_n\rangle\!\rangle$ denoting the set of all variant types that contain the specified fields.

This mechanism allows us to represent polymorphic types of various record operations. For example, the function *name* and the value *payment* are given the following types:

$$name : \forall t_1::U.\forall t_2::\{\!\{Name : t_1\}\!\}.t_2 \to t_1$$
$$payment : \forall t::\langle\!\langle Pound : real\rangle\!\rangle.t$$

indicating that *name* is a function that takes a value of any record type containing a $Name : t_1$ field and returns a value of type t_1 where t_1 may be any type, and *payment* is a polymorphic value having any variant type containing a $Pound : real$ component. By this mechanism, these terms can be used polymorphically. In addition to labeled-field access, kinded abstraction can also be used to represent polymorphic record modification (update) operations modify(e_1,l,e_2), which creates a new record from e_1 by modifying the value of the l field to e_2, leaving all the other fields unchanged. The following typing shows the polymorphic nature of this construct.

$$\lambda x.\lambda y.\text{modify(y,l,x)} : \forall t_1::U.\forall t_2::\{\!\{l : t_1\}\!\}.t_1 \to t_2 \to t_2$$

Combination of these features allows flexible programming without sacrificing the advantage of static typing and the existence of an ML-style, complete, type inference algorithm. Figure 1 shows examples of typings involving labeled records and variants using ML-style polymorphic function declaration.

This form of programming is also possible in the other proposals for ML-style polymorphic record calculi based on row variables mentioned earlier. One advantage of our formulation is that it yields a uniform treatment for both explicitly typed calculi and ML-style type inference systems. Most of the type inference algo-

rithms are defined without second-order types and do not explicitly treat ML's let binding. Rémy [1994b] formally treats ML's let binding. However, its relationships to an explicitly typed second-order system have not been well investigated. Using kinded abstraction, we extend the second-order lambda calculus of Girard [1971] and Reynolds [1974] to record polymorphism and show that the extension preserves basic properties of the second-order lambda calculus. We then develop an ML-style type inference system for a predicative subset of the second-order lambda calculus. These results extend the type inference algorithm of Milner [1978], the type system for ML let polymorphism by Damas and Milner [1982], and the analysis on the relationship between ML polymorphism and a predicative second-order system by Harper and Mitchell [1993]. These connections will allow us to transfer various known results of polymorphic type discipline to record polymorphism.

It should be mentioned, however, that row variables appear to be better suited to represent various powerful operations on records. Among the type systems based on row variables, perhaps the most flexible is that of Rémy [1994b], which uses sorted equational theory on row variables. For polymorphic record field access, record modification and polymorphic variants, his type system provides polymorphic typings equivalent to ours. For example, the function *name* is given the following typing in his system:

$$\lambda x.x.\mathsf{Name} \ : \ (\rho^{\{Name\}}; Name : pre(t)) \rightarrow \ t$$

where $\rho^{\{Name\}}$ is a sorted row variable representing possible rows (finite sets of record fields) that do not contain a *Name* field, and $pre(t)$ indicates the existence of *Name* field of type t. This typing is equivalent to the typing in our calculus in the sense that the two denote the same set of ground instances. However, his system is more powerful in that it can also represent an operation that extends a record with a new field and one that removes an existing field from a record, which are not representable in our type system. A restricted form of record extension operation is supported in Jategaonkar and Mitchell [1993]. Explicitly typed second-order calculi for extensible records have also been proposed [Cardelli and Mitchell 1989; Harper and Pierce 1991]. Unavailability of these extension operations is a limitation of our type system. However, those record calculi based on row variables appear to be difficult to compile. To the author's knowledge, there is no known systematic compilation method for any of these record calculi. A significant advantage of our type system is that it allows us to develop an efficient compilation method that always compile out labeled-field access into a direct index operation, as we shall explain in the next subsection.

In addition to record extension operations, various forms of record concatenation operations have also been proposed [Harper and Pierce 1991; Rémy 1992; Wand 1989]. Inclusion of any of these operations significantly complicates both the type theoretical analysis and compilation. Also, it is not obvious which of these powerful operations is really needed. For example, in database programming — a typical application area where labeled records play an important role — the form of record-merging operations commonly considered is not record concatenations but *natural join* for which row variables may not be a suitable mechanism. It is possible to generalize the relational natural join operation to general record structures [Buneman et al. 1991; Ohori 1990] and to extend a polymorphic type system with the

generalized natural join [Buneman and Ohori 1995; Ohori and Buneman 1988]. However, we are not sure that such an operation should be in the polymorphic core of programming languages.

The operations considered in this article support a wide range of programming with records while maintaining the existence of an efficient compilation method. We therefore claim that the calculus proposed in the present article serves as a basis for developing a practical polymorphic language with record polymorphism. We further believe that when our type system is extended with recursive types, it will support various features of object-oriented programming as discussed in Cardelli [1988] and Cardelli and Wegner [1985]. This issue is outside of the scope of the present article, and the author would like to investigate it elsewhere.

1.2 Compilation Method for Record Polymorphism

The second technical challenge in developing a practical programming language with record polymorphism is compilation. An important property of labeled records is the ability to access an element in a record not by position but by label, i.e., symbolic name. In a statically typed monomorphic language, this does not cause any difficulty in compilation. Since the actual position of each labeled field is statically determined from the type of a record, labeled-field access is easily compiled into an index operation, which is usually implemented by a single machine instruction. In a language with record polymorphism, however, compilation is a difficult problem. Consider the function $\lambda x.x.\mathsf{Name}$ again. Since actual arguments differ in the position of the Name field, it appears to be impossible to compile this function into a function that performs an index operation.

One straightforward approach might be to predetermine the offsets of all possible labels and to represent a record as a potentially very large structure with many empty slots. Cardelli [1994] took this strategy to represent records in a pure calculus of subtyping. Although this approach would be useful for studying formal properties of record polymorphism, it is unrealistic in practice. Another naive approach is to directly implement the intended semantics of the labeled-field access by dynamically searching for the specified label in a record represented as an association list of labels and values. An obvious drawback to such an approach is inefficiency in runtime execution. Since field access is the basic operation that is frequently invoked, such a method is unacceptable for serious application development.

A more-realistic approach for dynamic field lookup is to use a form of hashing. Rémy [1994a] presented an efficient, dynamic, field lookup method using a form of hashing similar to extendible hashing [Fagin et al. 1979] and showed that field selection can be implemented with relatively small runtime overhead both in execution time and in extra memory usage. This can be a reasonable implementation technique for various record calculi where static determination of the position of labeled fields is impossible. A drawback of this method is that labeled-field access always incurs extra runtime overhead, even when the program is completely monomorphic, and therefore the positions of labels can be statically determined. It would be unfortunate if we were forced to pay extra penalty for monomorphic labeled-field access when we move to a supposedly more advanced language with a polymorphic type system. Another drawback of hashing is that there is no guarantees to work for arbitrary records. Remy's technique, for example, does not work

well for large records such as those with 50 or 100 fields.

For a polymorphic record calculus to become a basis of practical programming languages, we must develop a compilation method that always achieves both compactness in the representation of records and efficiency in the execution of labeled-field access. Connor et al. [1989] considered this problem in the context of an explicitly typed language with subtyping and suggested an implementation strategy. However, they did not provide a systematic method to deal with arbitrary expressions, nor did they consider a type inference system. The second goal of this article is to develop such a compilation method and to establish that compilation achieves the intended operational behavior of a polymorphic record calculus.

Our strategy is to translate a polymorphic record calculus into an *implementation calculus*. In the implementation calculus, a labeled record is represented as a vector of values ordered by a canonical ordering of the set of labels, and fields are accessed by direct indexing. A variant value is represented as a value tagged with a natural number indicating the position in the vector of functions in a switch statement. To deal with polymorphic field selection and polymorphic variants, the implementation calculus contains *index variables* and *index abstraction*. For example, from an untyped term

```
let name = λx. x.Name in (name {Name = "Joe", Office=403},
                          name {Name="Hanako", Age=21, Phone=7222})
```

the translation algorithm produces the following implementation code:

```
let name = λIλx. x[I] in ((name 1) {"Joe",403},  (name 2) {21,"Hanako",7222})
```

where I is an index variable; λI. M is index abstraction; x[I] is an index expression; {"Joe",403} and {21,"Hanako",7222} are vector representations of the records whose fields are ordered by the lexicographical ordering on labels; and (name 1) and (name 2) are *index application* supplying appropriate index values to the index variable I. Similarly, an untyped term

```
let payment = ⟨Pound=100 0⟩
in (case payment of ⟨Pound=λx.x, Dollar=λx.x * 0.68⟩,
    case payment of ⟨Pound=λx.real_to_int(x * 150 0), Yen = λx.x⟩)
```

is translated into the following code in the implementation calculus:

```
let payment = λI ⟨I=100 0⟩
in (switch (payment 2) of ⟨λx.x * 0.68, λx.x⟩),
    switch (payment 1) of ⟨λx.real_to_int(x * 150.0), λx.x⟩)
```

where a polymorphic variant payment is represented as a term containing index abstraction whose index value is supplied through index applications — (payment 1), (payment 2) — and is used to select the corresponding function in the function vector in a switch statement whose elements are sorted by the lexicographical ordering on the variant labels.

Our compilation method works for arbitrary records and variants and does not introduce any runtime overhead for monomorphic programs. For polymorphic record functions and variants, it requires extra function applications to pass index values before applying them. However, as we shall show in the following development, extra index applications are done only when polymorphic terms are instantiated. So we believe that their cost is negligible.

The general idea of passing index values was suggested in Connor et al. [1989]. One of our original contributions is (1) to establish a systematic compilation algorithm that always constructs a correct implementation term for any type correct raw term of a polymorphic record calculus and (2) to establish its correctness formally.

1.3 Outline of the Development

To establish a rigorous typing discipline for record polymorphism, in Section 2 we define a second-order lambda calculus with record polymorphism, which we call $\Lambda^{\forall,\bullet}$, and show the subject reduction property. We then define in Section 3 an ML-style polymorphic record calculus, $\lambda^{let,\bullet}$, and define its call-by-value operational semantics. Typing derivations of $\lambda^{let,\bullet}$ correspond to terms of a predicative subcalculus $\Lambda^{let,\bullet}$ of $\Lambda^{\forall,\bullet}$, which can be regarded as an extension of Core XML [Harper and Mitchell 1993] with record polymorphism. The operational semantics of $\lambda^{let,\bullet}$ serves as a canonical model for evaluating a record calculus. A Damas-Milner-style polymorphic type discipline is shown to be sound with respect to this semantics. We next refine Robinson's unification algorithm by incorporating kind constraints on type variables, and using this unification we (1) give an algorithm to infer, for any typable raw term of $\lambda^{let,\bullet}$, both its principal typing and the corresponding explicitly typed term of $\Lambda^{let,\bullet}$and (2) prove its soundness and completeness. The constructed $\Lambda^{let,\bullet}$ term contains the necessary type information for compilation.

Next, in Section 4, we develop a compilation algorithm for $\lambda^{let,\bullet}$ and prove its correctness. We first define an implementation calculus $\lambda^{let,[]}$ and its call-by-value operational semantics. This calculus serves as an efficient abstract machine for record calculi. In particular, record field access and case branch selection are performed by direct indexing. In order to establish the correctness of the compilation algorithm, we present the calculus as a polymorphically typed functional calculus. We then develop an algorithm to compile $\lambda^{let,\bullet}$ into $\lambda^{let,[]}$ using the explicitly typed calculus $\Lambda^{let,\bullet}$ as an intermediate language.

There is one subtlety in using $\Lambda^{let,\bullet}$ as an intermediate language for compiling $\lambda^{let,\bullet}$. As shown in Ohori [1989], a translation of ML typings to Core XML terms cannot be coherent, and the same phenomenon occurs in construction of a $\Lambda^{let,\bullet}$ term from a $\lambda^{let,\bullet}$ term. One of the sources of failure of coherence is the free type variables that do not appear in the typing. Eliminating these "vacuous" type variables by selecting an appropriate canonical term of $\Lambda^{let,\bullet}$ is crucial for obtaining a correct compilation algorithm.

The compilation algorithm is shown to preserve typing. Furthermore, we also establish that the compilation preserves the operational behavior of a program by applying the idea of logical relations to set up an appropriate relationship between the operational semantics of $\lambda^{let,\bullet}$ and the operational semantics of $\lambda^{let,[]}$. (See Mitchell [1990] for a survey on logical relations and their applications.) Figure 2 shows the relationship among the four calculi defined in this article.

Based on the type system and the compilation method presented in this article, Standard ML has been extended with polymorphic record operations, and its compiler, SML$^\sharp$, has been implemented by modifying the Standard ML of New Jersey system [Appel and MacQueen 1991]. The following shows two ways of writing the function *name* in SML$^\sharp$:

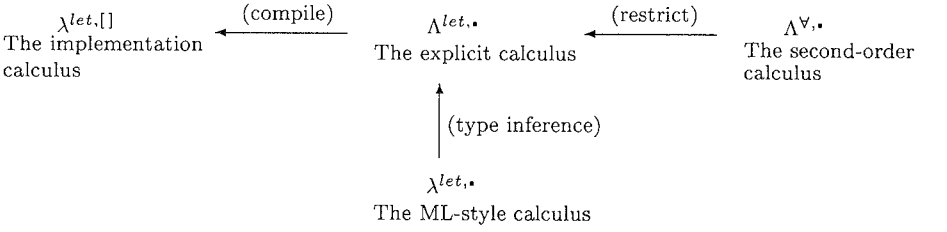

Fig. 2. Relationship among the calculi.

```
- val name = #Name;
val name = fn :  'b#{Name:'a,...} -> 'a
- fun name {Name=x,...} = x;
val name = fn :  'b#{Name:'a,...} -> 'a
```

where the second definition uses Standard ML's pattern machining mechanism. This is part of an interactive session in the implemented system. `{Name=x,...}` is a pattern that matches any record containing Name field, and `'b#{Name:'a,...} -> 'a` represents kinded quantification $\forall a::U.\forall b::\{\!\{Name : a\}\!\}.b \to a$. The notation "..." used in SML$^\sharp$ signifies that there may be more fields and should not be confused with the metanotation "$\cdots$" we shall use in describing various formal systems below. Section 5 describes the outline of the implementation and demonstrates its usefulness by examples.

2. POLYMORPHIC TYPE DISCIPLINE FOR RECORDS AND VARIANTS

This section defines a second-order polymorphic record calculus, $\Lambda^{\forall,\bullet}$, and proves its basic syntactic properties.

2.1 Types, Kinds, and Kinded Substitutions

The sets of types (ranged over by σ) and kinds (ranged over by k) are given by the following syntax:

$$\sigma ::= b \mid t \mid \sigma \to \sigma \mid \{l : \sigma, \cdots, l : \sigma\} \mid \langle l : \sigma, \cdots, l : \sigma\rangle \mid \forall t::k.\sigma$$
$$k ::= U \mid \{\!\{l : \sigma, \cdots, l : \sigma\}\!\} \mid \langle\!\langle l : \sigma, \cdots, l : \sigma\rangle\!\rangle$$

where b stands for a given set of *base types*, t for a given countably infinite set of *type variables*, l for a given set of *labels*, $\{l_1 : \sigma_1, \cdots, l_n : \sigma_n\}$ for record types, $\langle l_1 : \sigma_1, \cdots, l_n : \sigma_n\rangle$ for variant types, and $\forall t::k.\sigma$ for second-order types where type variable t is quantified over the set of types denoted by kind k. U is the universal kind denoting the set of all types. $\{\!\{l_1 : \sigma_1, \cdots, l_n : \sigma_n\}\!\}$ and $\langle\!\langle l_1 : \sigma_1, \cdots, l_n : \sigma_n\rangle\!\rangle$ are a record kind and a variant kind, respectively. The labels $l_1, \cdots, l_n$ appearing in a type or a kind must be pairwise distinct, and the order of their occurrence is insignificant.

The construct $\forall t::k.\sigma$ binds type variable t in σ but not in k. The set of *free type variables* of a type σ or a kind k are denoted by $FTV(\sigma)$ and $FTV(k)$, respectively. For second-order types, it is defined as $FTV(\forall t::k.\sigma) = FTV(k) \cup (FTV(\sigma)) \setminus \{t\})$. FTV for other types and kinds are defined as usual. We say that a type σ is *closed* if $FTV(\sigma) = \emptyset$. We identify types that differ only in the names of bound type variables and further adopt the usual "bound variable convention" on type

$$\mathcal{K} \vdash \sigma :: U \quad \text{for any } \sigma \text{ that is well formed under } \mathcal{K}$$
$$\mathcal{K} \vdash t :: \{\!\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}\!\} \quad \text{if } \mathcal{K}(t) = \{\!\{l_1 : \sigma_1, \ldots, l_n : \sigma_n, \cdots\}\!\}$$
$$\mathcal{K} \vdash \{l_1 : \sigma_1, \ldots, l_n : \sigma_n, \cdots\} :: \{\!\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}\!\}$$
$$\text{if } \{l_1 : \sigma_1, \ldots, l_n : \sigma_n, \cdots\} \text{ is well formed under } \mathcal{K}$$
$$\mathcal{K} \vdash t :: \langle\!\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n\rangle\!\rangle \quad \text{if } \mathcal{K}(t) = \langle l_1 : \sigma_1, \ldots, l_n : \sigma_n, \cdots\rangle$$
$$\mathcal{K} \vdash \langle l_1 : \sigma_1, \ldots, l_n : \sigma_n, \cdots\rangle :: \langle\!\langle l_1 : \sigma_1, \ldots, l_n : \sigma_n\rangle\!\rangle$$
$$\text{if } \langle l_1 : \sigma_1, \ldots, l_n : \sigma_n, \cdots\rangle \text{ is well formed under } \mathcal{K}$$

Fig. 3. Kinding rules for the second-order calculus $\Lambda^{\forall,\bullet}$.

variables, i.e., we assume that the set of all bound type variables are distinct and are different from any free type variables and that this property is preserved by substitution.

In our calculus, any free type variables must be *kinded* by a *kind assignment* $\mathcal{K}$, which is a mapping from a finite set of type variables to kinds. We sometimes regard a kind assignment as a set of pairs of a type variable and a kind and write $\emptyset$ for the empty kind assignment. Any type variables appearing in $\mathcal{K}$ must also be properly kinded by $\mathcal{K}$ itself. This is expressed by the following condition. A kind assignment $\mathcal{K}$ is *well formed* if for all $t \in dom(\mathcal{K})$, $FTV(\mathcal{K}(t)) \subseteq dom(\mathcal{K})$, where $dom(f)$ denotes the domain of a function f. Unless we explicitly say otherwise, we implicitly assume that any kind assignment appearing in the rest of the development is well formed. We write $\mathcal{K}\{t::k\}$ for $\mathcal{K} \cup \{(t, k)\}$ if $\mathcal{K}$ is well formed, $t \notin dom(\mathcal{K})$, and $FTV(k) \subseteq dom(\mathcal{K})$. We also write $\mathcal{K}\{t_1::k_1, \cdots, t_n::k_n\}$ for $((\mathcal{K}\{t_1::k_1\})\{t_2::k_2\}\cdots)\{t_n::k_n\}$. Note that $\mathcal{K}\{t_1::k_1, \cdots, t_n::k_n\}$ implies that $t_i \notin FTV(k_j)$ for any $1 \leq j \leq i \leq n$.

A type σ is *well formed under a kind assignment* $\mathcal{K}$ if $FTV(\sigma) \subseteq dom(\mathcal{K})$. This notion is naturally extended to other syntactic objects containing types, except for substitutions whose well-formedness condition is defined separately.

A type σ *has a kind* k *under* $\mathcal{K}$, denoted by $\mathcal{K} \vdash \sigma :: k$, if it is derivable by the set of *kinding rules* given in Figure 3. Note that if $\mathcal{K} \vdash \sigma :: k$ then both k and σ are well formed under $\mathcal{K}$.

A *type substitution*, or simply *substitution*, is a function from a finite set of type variables to types. We write $[\sigma_1/t_1, \cdots, \sigma_n/t_n]$ for the substitution that maps each t_i to σ_i. A substitution S is extended to the set of all type variables by letting $S(t) = t$ for all $t \notin dom(S)$, and it in turn is extended uniquely to record types, variant types, and function types. The result of applying a substitution S to a second-order type $\forall t::k.\sigma$ is the type obtained by applying S to its all *free type variables*. Under the bound type variable convention, we can simply take $S(\forall t::k.\sigma) = \forall t::S(k).S(\sigma)$. In what follows, we identify a substitution with its extension to types and write $S = S'$ if they are equal as functions on types. However, we maintain that the domain $dom(S)$ of a substitution S always means the domain of the original finite function S. If S_1 and S_2 are substitutions, we write $S_1 \circ S_2$ for the substitution S such that $dom(S) = dom(S_1) \cup dom(S_2)$; $S(t) = S_1(S_2(t))$ if $t \in dom(S_2)$; or $S(t) = S_1(t)$ if $t \in dom(S_1) \setminus dom(S_2)$. We further assume that this operation associates to the right so that $S_1 \circ S_2 \circ S_3$ means $S_1 \circ (S_2 \circ S_3)$.

Since in our type discipline type variables are *kinded* by a kind assignment,

the conventional notion of substitutions must be refined by incorporating kind constraints. A substitution S is *well formed under a kind assignment* $\mathcal{K}$ if for any $t \in dom(S)$, $S(t)$ is well formed under $\mathcal{K}$. A *kinded substitution* is a pair $(\mathcal{K}, S)$ of a kind assignment $\mathcal{K}$ and a substitution S that is well formed under $\mathcal{K}$. The kind assignment $\mathcal{K}$ in $(\mathcal{K}, S)$ specifies kind constraints of the *result* of the substitution. A kinded substitution $(\mathcal{K}, S)$ is *ground* if $\mathcal{K} = \emptyset$. We usually write S for a ground kinded substitution $(\emptyset, S)$.

A kinded substitution $(\mathcal{K}_1, S)$ *respects* a kind assignment $\mathcal{K}_2$ if for any $t \in dom(\mathcal{K}_2)$, $\mathcal{K}_1 \vdash S(t) :: S(\mathcal{K}_2(t))$. This notion specifies the condition under which a substitution can be applied, i.e., if $(\mathcal{K}_1, S)$ respects $\mathcal{K}$ then it can be applied to a type σ kinded by $\mathcal{K}$, yielding a type $S(\sigma)$ kinded by $\mathcal{K}_1$. The following lemma is easily proved.

LEMMA 2.1.1. *If* $\mathcal{K} \vdash \sigma :: k$, *and a kinded substitution* $(\mathcal{K}_1, S)$ *respects* $\mathcal{K}$, *then* $\mathcal{K}_1 \vdash S(\sigma) :: S(k)$.

As a simple corollary of this, if $(\mathcal{K}_1, S_1)$ respects $\mathcal{K}$ and $(\mathcal{K}_2, S_2)$ respects $\mathcal{K}_1$ then $(\mathcal{K}_2, S_2 \circ S_1)$ respects $\mathcal{K}$.

As seen from the above definitions, a kind assignment is regarded as a constraint on possible substitutions of type variables, i.e., those that respect it. In this view, our well-formedness condition of kind assignments is weak in the sense that it allows cyclic kind assignments like $\{t_1 :: \{\!\{l_1 : t_2\}\!\}, t_2 :: \{\!\{l_2 : t_1\}\!\}\}$, which is in some sense useless since there is no ground substitution that respects it. In fact, we could have adopted a stronger well-formedness condition on kind assignments requiring that a kind assignment must be of the form $\{t_1 :: k_1, \cdots, t_n :: k_n\}$ such that for any $1 \leq i \leq n$, k_i is well formed under $\{t_1 :: k_1, \cdots, t_{i-1} :: k_{i-1}\}$. The reason why we have not taken this approach is because under this stronger condition the complexity of the type inference algorithm would increase due to the well-formedness checking of a kind assignment every time types are unified. The current definition of well-formedness of kind assignments allows us to delay the check of circularity until a type variable is abstracted. The notation $\mathcal{K}\{t :: k\}$ is introduced for this purpose where t should not appear in k. This will be used for specifying the rule for type abstraction. Since this approach does not change the set of derivable *closed* typings, it still yields a sound type system that detects all the type errors of a program. We will comment on this issue again when we define a unification algorithm in Section 3.4.

2.2 Terms, Reduction, and Typing Rules

The set of *terms of* $\Lambda^{\forall,\cdot}$ is given by the grammar:

$$\begin{aligned} M \ ::= \ & x \mid c^b \mid \lambda x : \sigma.M \mid M\ M \mid \lambda t :: k.M \mid M\ \sigma \\ & \mid \{l{=}M, \cdots, l{=}M\} \mid M.l \mid \mathsf{modify}(M, l, M) \\ & \mid (\langle l{=}M\rangle{:}\sigma) \mid \mathsf{case}\ M\ \mathsf{of}\ \langle l{=}M, \cdots, l{=}M\rangle \end{aligned}$$

c^b is a constant of base type b. $\lambda t :: k\ M$ is kinded type abstraction. In a variant term $(\langle l{=}M\rangle{:}\sigma)$, type specification is necessary to preserve the explicit typing of the calculus. In a record expression or a case expression, the order of the fields is insignificant. We identify the terms that differ only in the names of bound variables and assume the bound variable convention for term variables. We write $[M/x]N$ for the term obtained from N by substituting M for all the free occurrences of

$$(\beta) \qquad (\lambda x : \sigma.M)\ N \Longrightarrow [N/x]M$$

$$(\text{type-}\beta) \qquad (\lambda t::k.M)\ \sigma \Longrightarrow [\sigma/t]M$$

$$(\text{dot}) \qquad \{l_1{=}M_1,\cdots,l_n{=}M_n\}.l_i \Longrightarrow M_i\ (1 \leq i \leq n)$$

$$(\text{modify}) \qquad \mathsf{modify}(\{l_1{=}M_1,\cdots,l_n{=}M_n\},l_i,N) \Longrightarrow \{l_1{=}M_1,\cdots,l_i{=}N,\cdots,l_n{=}M_n\}$$

$$(\text{case}) \qquad \mathsf{case}\ (\langle l_i{=}M\rangle{:}\sigma)\ \mathsf{of}\ \langle l_1{=}M_1,\cdots,l_n{=}M_n\rangle \Longrightarrow M_i\ M$$

Fig. 4. The reduction rules for the second-order system $\Lambda^{\forall,\bullet}$.

x in N. We write $FTV(M)$ for the set of free type variables of a term M. For type abstraction, it is defined as $FTV(\lambda t::k.M) = FTV(k) \cup (FTV(M) \setminus \{t\})$. The definitions for other terms are as usual. We use the equality symbol "=" for syntactic equality on terms and types. Since this article does not deal with an equational proof system, this does not cause any confusion.

In the above definition of the terms, we have only included constants of base types. Inclusion of constants of general types will not complicate the type system. A constant having a polymorphic type, however, requires a special treatment in both operational semantics and compilation we shall develop later. It is not hard to extend them for general constants by assuming that, for each constant, its operational behavior and its compilation scheme are given. But the extension would significantly complicate the presentation of our framework without giving much additional insight. For this reason, we only consider constants of base types.

The reduction axioms for this calculus are given in Figure 4. We say that M *reduces to* N *in one step*, written $M \rightarrow N$, if N is obtained from M by applying one of the reduction axioms to some subterm of M. We omit its routine formal definition. The reduction relation $M \twoheadrightarrow N$ is defined as the reflexive transitive closure of $\rightarrow$. This reduction system is Church-Rosser. This is seen from the following observation. It is easily verified that the reduction relation induced by (dot), (modify), and (case) is Church-Rosser. It is also easily verified that this relation commutes with the reduction relation generated by β and type-β. Since the latter reduction is well known to be Church-Rosser, by the Hindley-Roger theorem [Barendregt 1984, ch.3], the entire relation is Church-Rosser.

For the reduction system, we can also include η rule for lambda abstraction. However, there seems to be no easy way to include other extensionality rules for type abstraction, records, and variants. In the presence of record polymorphism, a straightforward inclusion of any of these rules causes both the confluence and subject reduction property to fail. For example, if we would have included type-η rule, then terms like $\lambda t::\{\!\{a : int, b : int\}\!\}.(\lambda s::\{\!\{a : int\}\!\}.\lambda x{:}s.x)\ t$ would have two different normal forms having different types.

Since in our system types may depend on type variables other than their own free type variables, we need to extend the notion of free type variables of a type. For a type σ well formed under $\mathcal{K}$, the set of *essentially free type variables of* σ *under* $\mathcal{K}$, denoted by $EFTV(\mathcal{K},\sigma)$, is the smallest set satisfying:

—$FTV(\sigma) \subseteq EFTV(\mathcal{K},\sigma)$.
—if $t \in EFTV(\mathcal{K},\sigma)$ then $FTV(\mathcal{K}(t)) \subseteq EFTV(\mathcal{K},\sigma)$.

Intuitively, $t \in EFTV(\mathcal{K},\sigma)$ if σ contains t either directly or through kind con-

$$\text{VAR}\quad \mathcal{K},\mathcal{T} \rhd x : \sigma \quad \text{if } \mathcal{T} \text{ is well formed under } \mathcal{K} \text{ and } \mathcal{T}(x) = \sigma$$

$$\text{CONST}\quad \mathcal{K},\mathcal{T} \rhd c^b : b \quad \text{if } \mathcal{T} \text{ is well formed under } \mathcal{K}$$

$$\text{ABS}\quad \frac{\mathcal{K},\mathcal{T}\{x:\sigma_1\} \rhd M_1 : \sigma_2}{\mathcal{K},\mathcal{T} \rhd \lambda x:\sigma_1.M_1 : \sigma_1 \to \sigma_2}$$

$$\text{APP}\quad \frac{\mathcal{K},\mathcal{T} \rhd M_1 : \sigma_1 \to \sigma_2 \quad \mathcal{K},\mathcal{T} \rhd M_2 : \sigma_1}{\mathcal{K},\mathcal{T} \rhd M_1\ M_2 : \sigma_2}$$

$$\text{TABS}\quad \frac{\mathcal{K}\{t::k\},\mathcal{T} \rhd M : \sigma}{\mathcal{K},\mathcal{T} \rhd \lambda t::k.M : \forall t::k.\sigma} \quad \text{if } t \notin FTV(\mathcal{T})$$

$$\text{TAPP}\quad \frac{\mathcal{K},\mathcal{T} \rhd M : \forall t::k.\sigma_1 \qquad \mathcal{K} \vdash \sigma_2 :: k}{\mathcal{K},\mathcal{T} \rhd M\ \sigma_2 : [\sigma_2/t](\sigma_1)}$$

$$\text{RECORD}\quad \frac{\mathcal{K},\mathcal{T} \rhd M_i : \sigma_i \quad (1 \le i \le n)}{\mathcal{K},\mathcal{T} \rhd \{l_1{=}M_1,\cdots,l_n{=}M_n\} : \{l_1 : \sigma_1,\cdots,l_n : \sigma_n\}}$$

$$\text{DOT}\quad \frac{\mathcal{K},\mathcal{T} \rhd M : \sigma_1 \quad \mathcal{K} \vdash \sigma_1 :: \{\!\{l : \sigma_2\}\!\}}{\mathcal{K},\mathcal{T} \rhd M.l : \sigma_2}$$

$$\text{MODIFY}\quad \frac{\mathcal{K},\mathcal{T} \rhd M_1 : \sigma_1 \quad \mathcal{K},\mathcal{T} \rhd M_2 : \sigma_2 \quad \mathcal{K} \vdash \sigma_1 :: \{\!\{l : \sigma_2\}\!\}}{\mathcal{K},\mathcal{T} \rhd \mathsf{modify}(M_1,l,M_2) : \sigma_1}$$

$$\text{VARIANT}\quad \frac{\mathcal{K},\mathcal{T} \rhd M : \sigma_1 \qquad \mathcal{K} \vdash \sigma_2 :: \langle\!\langle l : \sigma_1 \rangle\!\rangle}{\mathcal{K},\mathcal{T} \rhd (\langle l{=}M\rangle\ \sigma_2) : \sigma_2}$$

$$\text{CASE}\quad \frac{\mathcal{K},\mathcal{T} \rhd M : \langle l_1 : \sigma_1,\cdots,l_n : \sigma_n\rangle \qquad \mathcal{K},\mathcal{T} \rhd M_i : \sigma_i \to \sigma\ (1 \le i \le n)}{\mathcal{K},\mathcal{T} \rhd \mathsf{case}\ M\ \mathsf{of}\ \langle l_1{=}M_1,\cdots,l_n{=}M_n\rangle : \sigma}$$

Fig. 5. Type system of the second-order calculus $\Lambda^{\forall,\bullet}$.

strains specified by $\mathcal{K}$. For example, t_1 is essentially free in t_2 under $\{t_1::U, t_2::\{\!\{l : t_1\}\!\}\}$. This notion naturally extends to other syntactic structures containing types.

A *type assignment* $\mathcal{T}$ is a mapping from a finite set of variables to types. We write $\{x_1 : \sigma_1,\cdots,x_n : \sigma_n\}$ for the type assignment that binds x_i to σ_i $(1 \le i \le n)$. We also write $\mathcal{T}\{x : \sigma\}$ for $\mathcal{T} \cup \{x : \sigma\}$ provided that $x \notin dom(\mathcal{T})$. The type system is defined as a proof system to derive a *typing* of the form $\mathcal{K},\mathcal{T} \rhd M : \sigma$. The set of typing rules is given in Figure 5. In the rule TABS, the condition $t \notin FTV(\mathcal{T})$ is equivalent to $t \notin EFTV(\mathcal{K}\{t::k\},\mathcal{T})$ under our assumption on $\mathcal{K}\{t::k\}$. We write $\Lambda^{\forall,\bullet} \vdash \mathcal{K},\mathcal{T} \rhd M : \sigma$ if $\mathcal{K},\mathcal{T} \rhd M : \sigma$ is derivable in this proof system.

Unlike the polymorphic type discipline for records based on subtyping, this type system has the following property.

PROPOSITION 2.2.1. *For any $\mathcal{K},\mathcal{T},M$ there is at most one σ such that $\Lambda^{\forall,\bullet} \vdash \mathcal{K},\mathcal{T} \rhd M : \sigma$. Moreover, a derivation of $\Lambda^{\forall,\bullet} \vdash \mathcal{K},\mathcal{T} \rhd M : \sigma$ is unique.*

PROOF. We proceed by induction on the structure of M. The case of $M.l$ follows from the induction hypothesis and the fact that for given $\mathcal{K},\sigma_1$, and l there is at most one σ_2 such that $\mathcal{K} \vdash \sigma_1 :: \{\!\{l : \sigma_2\}\!\}$. Other cases are straightforward. □

The following basic properties hold, which are proved by routine induction.

LEMMA 2.2.2.

(1) *If $\Lambda^{\forall,\bullet} \vdash \mathcal{K},\mathcal{T} \rhd M : \sigma$ then $\mathcal{T}$, M, and σ are well formed under $\mathcal{K}$.*

(2) *If* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}\{t::k\}, \mathcal{T} \rhd M : \sigma$ *and* $t \notin (FTV(\mathcal{T}) \cup FTV(M) \cup FTV(\sigma))$ *then* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M : \sigma$.

(3) *If* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M : \sigma$ *then* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}\{t::k\}, \mathcal{T} \rhd M : \sigma$.

(4) *If* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M : \sigma$ *then* $FV(M) \subseteq dom(\mathcal{T})$.

(5) *If* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T}\{x : \sigma_0\} \rhd M : \sigma$ *and* $x \notin FV(M)$ *then* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M : \sigma$.

(6) *If* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M : \sigma$ *and* σ_0 *is well formed under* $\mathcal{K}$ *then* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T}\{x : \sigma_0\} \rhd M : \sigma$.

The corresponding lemma holds for all the calculi we shall define in this article.

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

LEMMA 2.2.3. *If* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_1, \mathcal{T} \rhd M : \sigma$ *and* $(\mathcal{K}_2, S)$ *respects* $\mathcal{K}_1$ *then* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_2, S(\mathcal{T}) \rhd S(M) : S(\sigma)$.

PROOF. Suppose $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_1, \mathcal{T} \rhd M : \sigma$ and $(\mathcal{K}_2, S)$ respects $\mathcal{K}_1$. Proof is by induction on the structure of M.

Case x. Since $\mathcal{T}$ is well formed under $\mathcal{K}_1$ and $(\mathcal{K}_2, S)$ respects $\mathcal{K}_1$, $S(\mathcal{T})$ is well formed under $\mathcal{K}_2$. Also $S(\mathcal{T})(x) = S(\sigma)$. Therefore $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_2, S(\mathcal{T}) \rhd x : S(\sigma)$.

Case $\lambda t::k.M_1$. We must have $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_1\{t::k\}, \mathcal{T} \rhd M_1 : \sigma_1$ for some σ_1 such that $\sigma = \forall t::k.\sigma_1$ and $t \notin FTV(\mathcal{T})$. By the bound type variable convention, we can assume that t does not appear in S or $\mathcal{K}_2$. Since k is well formed under $\mathcal{K}_1$ and $(\mathcal{K}_2, S)$ respects $\mathcal{K}_1$, $S(k)$ is well formed under $\mathcal{K}_2$. Then $\mathcal{K}_2\{t::S(k)\}$ is well formed, and $(\mathcal{K}_2\{t::S(k)\}, S)$ respects $\mathcal{K}_1\{t::k\}$. By the induction hypothesis $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_2\{t::S(k)\}, S(\mathcal{T}) \rhd S(M_1) : S(\sigma_1)$. Since $t \notin FTV(S(\mathcal{T}))$, by the rule TABS $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_2, S(\mathcal{T}) \rhd \lambda t::S(k).S(M_1) : \forall t::S(k).S(\sigma_1)$.

Case $M_1\ \sigma_1$. We must have $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_1, \mathcal{T} \rhd M_1 : \forall t::k.\sigma_2$, $\mathcal{K}_1 \vdash \sigma_1 :: k$ and $\sigma = [\sigma_1/t](\sigma_2)$ for some σ_2, t, k. By the induction hypothesis and the bound type variable convention, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_2, S(\mathcal{T}) \rhd S(M_1) : \forall t::S(k).S(\sigma_2)$. By Lemma 2.1.1, $\mathcal{K}_2 \vdash S(\sigma_1) :: S(k)$. By the typing rule TAPP, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_2, S(\mathcal{T}) \rhd S(M_1)\ S(\sigma_1) : [S(\sigma_1)/t](S(\sigma_2))$. But since $t \notin dom(S)$, $[S(\sigma_1)/t](S(\sigma_2)) = S([\sigma_1/t](\sigma_2)) = S(\sigma)$, as desired.

Case $M.l$. We must have $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_1, \mathcal{T} \rhd M : \sigma_1$, for some σ_1 such that $\mathcal{K}_1 \vdash \sigma_1 :: \{\!\{l : \sigma\}\!\}$. By the induction hypothesis, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_2, S(\mathcal{T}) \rhd S(M) : S(\sigma_1)$. By Lemma 2.1.1, $\mathcal{K}_2 \vdash S(\sigma_1) :: \{\!\{l : S(\sigma)\}\!\}$. By the rule DOT, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}_2, S(\mathcal{T}) \rhd S(M).l : S(\sigma)$.

The cases for modify(M_1,l,M_2) and ($\langle l{=}M\rangle{:}\sigma$) are similar to that of $M_1.l$. Other cases easily follow from the induction hypotheses. □

This lemma holds for all the calculi we shall define in this article.

We also have the following substitution lemma.

LEMMA 2.2.4. *If* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T}\{x : \sigma_1\} \rhd M : \sigma_2$ *and* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd N : \sigma_1$ *then* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd [N/x]M : \sigma_2$.

PROOF. Suppose $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T}\{x : \sigma_1\} \rhd M : \sigma_2$ and $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd N : \sigma_1$. Proof is by induction on the structure of M.

Case $\lambda t::k.M_1$. We must have $\Lambda^{\forall,\cdot} \vdash \mathcal{K}\{t::k\}, \mathcal{T}\{x : \sigma_1\} \rhd M_1 : \sigma_3$ for some σ_3 such that $\sigma_2 = \forall t::k.\sigma_3$ and $t \notin FTV(\mathcal{T}\{x : \sigma_1\})$. By Lemma 2.2.2, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd N : \sigma_1$ implies $\Lambda^{\forall,\cdot} \vdash \mathcal{K}\{t::k\}, \mathcal{T} \rhd N : \sigma_1$. By the induction hypothesis, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}\{t::k\}, \mathcal{T} \rhd [N/x]M : \sigma_3$. Since $t \notin FTV(\mathcal{T}) \subseteq FTV(\mathcal{T}\{x : \sigma_1\})$, by the typing rule TABS we have $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd \lambda t::k.[N/x]M : \forall t::k.\sigma_3$.

Case $M_1.l$. We must have $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T}\{x : \sigma_1\} \rhd M_1 : \sigma_3$ and $\mathcal{K} \vdash \sigma_3 :: \{\!\{l : \sigma_2\}\!\}$ for some σ_3. By the induction hypothesis, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd [N/x]M_1 : \sigma_3$. By the typing rule DOT, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd [N/x]M_1.l : \sigma_2$.

Cases for modify(M_1,l,M_2) and $(\langle l{=}M\rangle{:}\sigma)$ are shown similarly to $M.l$. Other cases are proved using Lemma 2.2.2, similarly to the second-order lambda calculus. □

Using these properties, we can prove the following subject reduction theorem.

THEOREM 2.2.5. *If* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M : \sigma$ *and* $M \longrightarrow\!\!\!\!\!\rightarrow N$ *then* $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd N : \sigma$.

PROOF. This is proved by showing that each reduction axiom preserves typing.

(β): Suppose $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd (\lambda x : \sigma_1.M_1)\ M_2 : \sigma$. Then $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T}\{x : \sigma_1\} \rhd M_1 : \sigma$ and $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M_2 : \sigma_1$. By Lemma 2.2.4, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd [M_2/x]M_1 : \sigma$.

(type-β): Suppose $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd (\lambda t::k.M_1)\ \sigma_1 : \sigma$. Then $\sigma = [\sigma_1/t](\sigma_2)$ for some σ_2 such that $\Lambda^{\forall,\cdot} \vdash \mathcal{K}\{t::k\}, \mathcal{T} \rhd M_1 : \sigma_2$ and $\mathcal{K} \vdash \sigma_1 :: k$. By the bound type variable convention, t does not appear free elsewhere other than σ_2 and M_1. Then the kinded substitution $(\mathcal{K}, [\sigma_1/t])$ respects $\mathcal{K}\{t::k\}$, and by Lemma 2.2.3, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, [\sigma_1/t](\mathcal{T}) \rhd [\sigma_1/t](M_1) : [\sigma_1/t](\sigma_2)$, i.e., $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd [\sigma_1/t](M_1) : \sigma$.

(dot): Suppose $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd \{l_1{=}M_1,\cdots,l_n{=}M_n\}.l_i : \sigma$. Then there are $\sigma_1,\cdots,\sigma_n$ such that $\mathcal{K}, \mathcal{T} \rhd M_j : \sigma_j$ $(1 \leq j \leq n)$, and $\mathcal{K} \vdash \{l_1 : \sigma_1,\cdots,l_n : \sigma_n\} :: \{\!\{l_i : \sigma\}\!\}$. By the definition of kinding, $\sigma = \sigma_i$ and $\mathcal{K}, \mathcal{T} \rhd M_i : \sigma$.

(modify): Suppose $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd \mathsf{modify}(\{l_1{=}M_1,\cdots,l_n{=}M_n\},l_i,N) : \sigma$. Then there are some $\sigma_1,\cdots,\sigma_n,\sigma'$ such that $\sigma = \{l_1 : \sigma_1,\cdots,l_n : \sigma_n\}$, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M_j : \sigma_j$ $(1 \leq j \leq n)$, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd N : \sigma'$, and $\mathcal{K} \vdash \{l_1 : \sigma_1,\cdots,l_n : \sigma_n\} :: \{\!\{l_i : \sigma'\}\!\}$. By the definition of kinding, $\sigma_i = \sigma'$. Then by the typing rule RECORD, $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd \{l_1 = M_1,\cdots,l_i = N,\cdots,l_n = M_n\} : \sigma$.

(case): Suppose $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd \mathsf{case}\ (\langle l_i{=}M\rangle{:}\sigma_0)\ \mathsf{of}\ \langle l_1{=}M_1,\cdots,l_n{=}M_n\rangle : \sigma$. Then there are $\sigma_1,\cdots,\sigma_n$ such that $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M_j : \sigma_j \to \sigma$ $(1 \leq j \leq n)$, and $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd (\langle l_i = M\rangle : \sigma_0) : \langle l_1 : \sigma_1,\cdots,l_n : \sigma_n\rangle$. The last typing implies that $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M : \sigma'$ and $\mathcal{K} \vdash \langle l_1 : \sigma_1,\cdots,l_n : \sigma_n\rangle :: \langle\!\langle l_i : \sigma'\rangle\!\rangle$ for some σ'. By the definition of kindings, $\sigma' = \sigma_i$. Then by the typing rule APP, we have $\Lambda^{\forall,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M_i\ M : \sigma$. □

Let us show simple examples of terms in this calculus. The field selection function *name* and variant term *payment* given in Section 1 are represented as the terms

$$NAME = \lambda t_1::U.\lambda t_2::\{\!\{Name : t_1\}\!\}.\lambda x{:}t_2.x.\mathsf{Name}$$
$$PAYMENT = \lambda t::\langle\!\langle Pound : real\rangle\!\rangle\ (\langle \mathsf{Pound}{=}100.0\rangle{:}\ t)$$

and are given the following typings:

$$\emptyset,\emptyset \rhd NAME : \forall t_1::U.\forall t_2::\{\!\{Name : t_1\}\!\}.t_2 \to t_1$$
$$\emptyset,\emptyset \rhd PAYMENT : \forall t::\langle\!\langle Pound : real\rangle\!\rangle.t$$

With appropriate type applications, these terms can be used polymorphically. The expressions containing *name* and *payment* given in Section 1.2 are represented as the following terms.

$(\lambda$name:$\forall t_1::U.\forall t_2::\{\!\{Name : t_1\}\!\}.t_2 \to t_1$.
  (name *string* $\{Name : string, Office : int\}$ {Name="Joe", Office=403},
   name *string* $\{Name : string, Age : int, Phone : int\}$
        {Name="Hanako", Age=21, Phone=7222})) $NAME$

$(\lambda$payment:$\forall t::\langle\!\langle Pound : real\rangle\!\rangle.t$
  (payment $\langle Pound : real, Dollar : real\rangle$ ⟨Pound=λx:*real*.x, Dollar=λx:*real*.x * 0.68⟩,
   payment $\langle Pound : real, Yen : int\rangle$
        ⟨Pound=λx:*real*.real_to_int(x * 150.0), Yen=λx:*int* x ⟩))
$PAYMENT$

For this calculus, the equational proof system can be defined. We also believe that a semantic framework of the second-order lambda calculus such as Bruce et al. [1990] and Breazu-Tannen and Coquand [1988] can be extended to this calculus and that the soundness and completeness of the equational proof system can be proved. A detailed studies of semantic properties of the calculus is beyond the scope of the present article. Our main focus is an ML-style type inference system and compilation. We now turn to the first of the two.

3. ML-STYLE TYPE INFERENCE SYSTEM

In this section, we define an ML-style, implicitly typed, polymorphic record calculus $\lambda^{let,\bullet}$, show that typing derivations of $\lambda^{let,\bullet}$ correspond to terms of a predicative subcalculus $\Lambda^{let,\bullet}$ of $\Lambda^{\forall,\bullet}$, give a type inference algorithm for $\lambda^{let,\bullet}$, and prove its soundness and completeness. We also give a call-by-value operational semantics and prove the soundness of the type system of $\lambda^{let,\bullet}$.

3.1 An ML-Style Polymorphic Record Calculus : $\lambda^{let,\bullet}$

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

$$\begin{aligned} e ::= {} & x \mid c^b \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ x{=}e\ \mathsf{in}\ e \\ & \mid \{l{=}e,\cdots,l{=}e\} \mid e.l \mid \mathsf{modify}(e,l,e) \\ & \mid \langle l{=}e\rangle \mid \mathsf{case}\ e\ \mathsf{of}\ \langle l{=}e,\cdots,l{=}e\rangle \end{aligned}$$

Following Damas and Milner's presentation of ML, we divide the set of types into *monotypes* (ranged over by τ) and *polytypes* (ranged over by σ) as follows:

$$\tau ::= t \mid b \mid \tau \to \tau \mid \{l : \tau,\cdots,l : \tau\} \mid \langle l : \tau,\cdots,l : \tau\rangle$$
$$\sigma ::= \tau \mid \forall t::k.\,\sigma$$

In what follows, we indicate the fact that a type is restricted to be a monotype by our usage of a metavariable τ. The set of kinds is given by the following grammar.

$$k ::= U \mid \{\!\{l : \tau,\cdots,l : \tau\}\!\} \mid \langle\!\langle l : \tau,\cdots,l : \tau\rangle\!\rangle$$

$$\mathcal{K} \vdash \tau :: U \quad \text{for any } \tau \text{ well formed under } \mathcal{K}$$
$$\mathcal{K} \vdash t :: \{\!\{l_1 : \tau_1, \quad , l_n \cdot \tau_n\}\!\} \text{ if } \mathcal{K}(t) = \{\!\{l_1 : \tau_1, \cdots, l_n : \tau_n, \cdot \cdot\}\!\}$$
$$\mathcal{K} \vdash \{l_1 \cdot \tau_1, \cdots, l_n : \tau_n, \cdot \cdot \} :: \{\!\{l_1 : \tau_1, \cdot \cdot, l_n \cdot \tau_n\}\!\}$$
$$\text{if } \{l_1 : \tau_1, \cdots, l_n \cdot \tau_n, \cdot \cdot\} \text{ is well formed under } \mathcal{K}$$
$$\mathcal{K} \vdash t :: \langle\!\langle l_1 \cdot \tau_1, \cdots, l_n : \tau_n\rangle\!\rangle \text{ if } \mathcal{K}(t) = \langle l_1 : \tau_1, \cdots, l_n : \tau_n, \cdots\rangle$$
$$\mathcal{K} \vdash \langle l_1 : \tau_1, \cdots, l_n : \tau_n, \cdots\rangle :: \langle\!\langle l_1 \cdot \tau_1, \cdots, l_n : \tau_n\rangle\!\rangle$$
$$\text{if } \langle l_1 : \tau_1, \cdots, l_n : \tau_n, \cdots\rangle \text{ is well formed under } \mathcal{K}$$

Fig. 6. Kinding rules for the ML-style type inference system $\lambda^{let,\bullet}$.

As in $\Lambda^{\forall,\bullet}$, a kind assignment is a mapping from a finite set of type variables to kinds. The set of kinding rules is given in Figure 6. Note that unlike the second-order calculus $\Lambda^{\forall,\bullet}$, a kind denotes a subset of monotypes, and a type variable ranges only over monotypes.

Let σ_1 be a polytype well formed under $\mathcal{K}$. We say that σ_2 is a *generic instance* of σ_1 under $\mathcal{K}$, written $\mathcal{K} \vdash \sigma_1 \geq \sigma_2$, if $\sigma_1 = \forall t_1^1 :: k_1^1 \cdots \forall t_n^1 :: k_n^1 . \tau_1$, $\sigma_2 = \forall t_1^2 :: k_1^2 \cdots \forall t_m^2 :: k_m^2 . \tau_2$, and there is a substitution S such that $dom(S) = \{t_1^1, \cdots, t_n^1\}$, $(\mathcal{K}\{t_1^2 :: k_1^2, \cdots, t_m^2 :: k_m^2\}, S)$ respects $\mathcal{K}\{t_1^1 :: k_1^1, \cdots, t_n^1 :: k_n^1\}$ and $\tau_2 = S(\tau_1)$. It is easily checked that if $\mathcal{K} \vdash \sigma_1 \geq \sigma_2$ then σ_2 is well formed under $\mathcal{K}$. This is a refinement of the usual definition of generic instance. We have the following expected property, which can be proved using Lemma 2.1.1.

LEMMA 3.1.1. *If* $\mathcal{K} \vdash \sigma_1 \geq \sigma_2$ *and* $\mathcal{K} \vdash \sigma_2 \geq \sigma_3$ *then* $\mathcal{K} \vdash \sigma_1 \geq \sigma_3$.

Let $\mathcal{T}$ and τ be well formed under $\mathcal{K}$. The *closure of* τ *under* $\mathcal{T}, \mathcal{K}$, denoted by $Cls(\mathcal{K}, \mathcal{T}, \tau)$, is a pair $(\mathcal{K}', \forall t_1 :: k_1 \cdots \forall t_n :: k_n . \tau)$ such that $\mathcal{K}'\{t_1 :: k_1, \cdots, t_n :: k_n\} = \mathcal{K}$ and $\{t_1, \cdots, t_n\} = EFTV(\mathcal{K}, \tau) \setminus EFTV(\mathcal{K}, \mathcal{T})$. Note that if $\lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd e : \tau$ and $Cls(\mathcal{K}, \mathcal{T}, \tau) = (\mathcal{K}', \sigma)$ then $\mathcal{T}$ and σ are well formed under $\mathcal{K}'$. A type assignment is a mapping from a finite set of variables to polytypes. The set of typing rules for $\lambda^{let,\bullet}$ is given in Figure 7.

In this type system, polymorphic generalization and let abstraction are separated into two rules: GEN and LET. It is possible to combine these two into a single rule. The presentation adopted here has the advantage of making it easier to prove various properties by induction on typing derivations.

The following lemma allows us to strengthen the type assignment, which is proved by routine induction on typing derivation of e.

LEMMA 3.1.2. *If* $\mathcal{K}, \mathcal{T}\{x : \sigma_1\} \rhd e : \tau$ *and* $\mathcal{K} \vdash \sigma_2 \geq \sigma_1$ *then* $\mathcal{K}, \mathcal{T}\{x : \sigma_2\} \rhd e : \tau$.

The example terms *name* and *payment* given in Section 1 have the following typings in $\lambda^{let,\bullet}$.

$$\lambda^{let,\bullet} \vdash \emptyset, \emptyset \rhd \lambda x.x.\mathsf{Name} : \forall t_1 :: U . \forall t_2 :: \{\!\{Name : t_1\}\!\} . t_2 \to t_1$$
$$\lambda^{let,\bullet} \vdash \emptyset, \emptyset \rhd \langle \mathsf{Pound}{=}100.0\rangle : \forall t :: \langle\!\langle Pound : real\rangle\!\rangle . t$$

Moreover, they are principal typings and are automatically inferred by a type inference algorithm, as we shall show later.

$$\text{VAR}\quad \mathcal{K},\mathcal{T} \rhd x : \tau \quad \text{if } \mathcal{T} \text{ is well formed under } \mathcal{K} \text{ and } \mathcal{K} \vdash \mathcal{T}(x) \geq \tau$$

$$\text{CONST}\quad \mathcal{K},\mathcal{T} \rhd c^b : b \quad \text{if } \mathcal{T} \text{ is well formed under } \mathcal{K}$$

$$\text{APP}\quad \frac{\mathcal{K},\mathcal{T} \rhd e_1 : \tau_1 \to \tau_2 \quad \mathcal{K},\mathcal{T} \rhd e_2 : \tau_1}{\mathcal{K},\mathcal{T} \rhd e_1\ e_2 : \tau_2}$$

$$\text{ABS}\quad \frac{\mathcal{K},\mathcal{T}\{x : \tau_1\} \rhd e_1 : \tau_2}{\mathcal{K},\mathcal{T} \rhd \lambda x.e_1 : \tau_1 \to \tau_2}$$

$$\text{RECORD}\quad \frac{\mathcal{K},\mathcal{T} \rhd e_i : \tau_i \quad (1 \leq i \leq n)}{\mathcal{K},\mathcal{T} \rhd \{l_1{=}e_1,\cdots,l_n{=}e_n\} : \{l_1 : \tau_1,\cdots,l_n : \tau_n\}}$$

$$\text{DOT}\quad \frac{\mathcal{K},\mathcal{T} \rhd e : \tau_1 \quad \mathcal{K} \vdash \tau_1 :: \{\!\{l : \tau_2\}\!\}}{\mathcal{K},\mathcal{T} \rhd e.l : \tau_2}$$

$$\text{MODIFY}\quad \frac{\mathcal{K},\mathcal{T} \rhd e_1 : \tau_1 \quad \mathcal{K},\mathcal{T} \rhd e_2 : \tau_2 \quad \mathcal{K} \vdash \tau_1 :: \{\!\{l : \tau_2\}\!\}}{\mathcal{K},\mathcal{T} \rhd \mathsf{modify}(e_1,l,e_2) : \tau_1}$$

$$\text{VARIANT}\quad \frac{\mathcal{K},\mathcal{T} \rhd e : \tau_1 \quad \mathcal{K} \vdash \tau_2 :: \langle\!\langle l : \tau_1 \rangle\!\rangle}{\mathcal{K},\mathcal{T} \rhd \langle l{=}e\rangle : \tau_2}$$

$$\text{CASE}\quad \frac{\mathcal{K},\mathcal{T} \rhd e : \langle l_1 : \tau_1,\cdots,l_n : \tau_n\rangle \quad \mathcal{K},\mathcal{T} \rhd e_i : \tau_i \to \tau \ (1 \leq i \leq n)}{\mathcal{K},\mathcal{T} \rhd \mathsf{case}\ e\ \mathsf{of}\ \langle l_1{=}e_1,\cdots,l_n{=}e_n\rangle : \tau}$$

$$\text{GEN}\quad \frac{\mathcal{K},\mathcal{T} \rhd e : \tau}{\mathcal{K}',\mathcal{T} \rhd e : \sigma} \quad \text{if } Cls(\mathcal{K},\mathcal{T},\tau) = (\mathcal{K}',\sigma)$$

$$\text{LET}\quad \frac{\mathcal{K},\mathcal{T} \rhd e_1 : \sigma \quad \mathcal{K},\mathcal{T}\{x : \sigma\} \rhd e_2 : \tau}{\mathcal{K},\mathcal{T} \rhd \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 : \tau}$$

Fig. 7. Typing rules for ML-style record calculus $\lambda^{let,\bullet}$.

3.2 Operational Semantics of $\lambda^{let,\bullet}$

As a model of an ML-style programming language, we require $\lambda^{let,\bullet}$ to have a stronger property of type soundness than the subject reduction property, i.e., the property being that evaluation of a closed term of some type always yields a value of that type. To establish this property, we define a call-by-value operational semantics using evaluation contexts of Felleisen et al. [1987] and prove the type soundness theorem with respect to this semantics. This semantics serves as an evaluation model of a polymorphic programming language with records and variants.

Figure 8 gives the definitions of the set of *values* (ranged over by v), the set of call-by-value *evaluation contexts* (ranged over by $ev[\,]$), and call-by-value *context-rewriting axioms*, where $[\cdot]$ denotes the empty context and where $ev[e]$ is the term obtained by placing e in the hole of the context $ev[\,]$. A *one-step evaluation relation* $e \xrightarrow{ev} e'$ is then defined as: there exist $ev[\,]_1, e_1, e_2$ such that $e = ev[e_1]_1$, $ev[e_1]_1 \longrightarrow ev[e_2]_1$, and $e' = ev[e_2]_1$. We write $\xrightarrow{ev}\!\!\!\!\!\rightarrow$ for the reflexive transitive closure of $\xrightarrow{ev}$, and we write $e \downarrow e'$ if $e \xrightarrow{ev}\!\!\!\!\!\rightarrow e'$ and if there is no e'' such that $e' \xrightarrow{ev} e''$.

To show the type soundness with respect to this operational semantics, we first define a type-indexed family of predicates on closed values. For a *closed* type σ, let $value^\sigma$ be the set $\{v | \lambda^{let,\bullet} \vdash \emptyset, \emptyset \rhd v : \sigma\}$ and define $p^\sigma \subseteq value^\sigma$ by induction on σ as follows:

—$v \in p^b$ iff $v = c^b$ for some constant c^b.

$$v ::= c^b \mid \lambda x\ e \mid \{l{=}v,\cdots,l{=}v\} \mid \langle l{=}v\rangle$$

$$\begin{aligned} ev[\,] ::= {} & [\cdot] \mid ev[\,]\ e \mid v\ ev[\,] \mid \mathsf{let}\ x{=}ev[\,]\ \mathsf{in}\ e \mid \{l_1{=}v_1,\cdots,l_{i-1}{=}v_{i-1},\ l_i{=}ev[\,],\cdots\} \\ & \mid ev[\,].l \mid \mathsf{modify}(ev[\,],l,e) \mid \mathsf{modify}(v,l,ev[\,]) \mid \langle l{=}ev[\,]\rangle \mid \mathsf{case}\ ev[\,]\ \mathsf{of}\ \langle l{=}e,\cdots,l{=}e\rangle \end{aligned}$$

$$\begin{aligned} ev[(\lambda x.e)\ v] &\longrightarrow ev[[v/x]e] \\ ev[\{l_1{=}v_1,\cdots,l_n{=}v_n\}.l_i] &\longrightarrow ev[v_i] \\ ev[\mathsf{modify}(\{l_1{=}v_1,\cdots,l_n{=}v_n\},l_i,v)] &\longrightarrow \{l_1{=}v_1,\cdots,l_i{=}v,\cdots,l_n{=}v_n\} \\ ev[\mathsf{case}\ \langle l_i{=}v\rangle\ \mathsf{of}\ \langle l_1{=}e_1,\cdots,l_n{=}e_n\rangle] &\longrightarrow ev[e_i\ v] \\ ev[\mathsf{let}\ x = v\ \mathsf{in}\ e] &\longrightarrow ev[[v/x]e] \end{aligned}$$

Fig. 8. Call-by-value operational semantics of $\lambda^{let,\bullet}$

—$v \in p^{\tau_1 \to \tau_2}$ iff for any $v_0 \in p^{\tau_1}$, if $(v\ v_0) \downarrow e$ then $e \in p^{\tau_2}$.
—$v \in p^{\{l_1:\tau_1,\cdots,l_n:\tau_n\}}$ iff $v = \{l_1 = v_1, \cdots, l_n = v_n\}$ such that $v_i \in p^{\tau_i}$ $(1 \le i \le n)$.
—$v \in p^{\langle l_1:\tau_1,\cdots,l_n:\tau_n\rangle}$ iff $v = \langle l_i = v'\rangle$ for some i $(1 \le i \le n)$ such that $v' \in p^{\tau_i}$.
—$v \in p^{\forall t_1::k_1\cdots t_n::k_n.\tau}$ iff for any ground substitution S such that $dom(S) = \{t_1, \cdots, t_n\}$ and it satisfies $\{t_1::k_1.\cdots t_n::k_n\}$, $v \in p^{S(\tau)}$.

Let $\mathcal{T}$ be a closed type assignment. A *$\mathcal{T}$-environment* is a function η such that $dom(\eta) = dom(\mathcal{T})$ and for any $x \in dom(\mathcal{T})$, $\eta(x) \in value^{\mathcal{T}(x)}$. If η is an environment, we write $\eta(e)$ for the term obtained from e by substituting $\eta(x)$ for each free occurrence of x in e. For a function f, if $x \notin dom(f)$ then we write $f\{x \mapsto v\}$ for the extension f' of f to x such that $f'(x) = v$.

THEOREM 3.2.1. *If $\lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd e : \sigma$ then for any ground substitution S that respects $\mathcal{K}$, and for any $S(\mathcal{T})$-environment η, if $\eta(e) \downarrow e'$ then $e' \in p^{S(\sigma)}$.*

PROOF. This is proved by induction on typing derivation. The proof proceeds by cases in terms of the last rule used in the derivation. Case for the rules CONST is by definition. Let S be any ground substitution respecting $\mathcal{K}$, and let η be any $S(\mathcal{T})$-environment.

Case VAR. Suppose $\mathcal{K}, \mathcal{T} \rhd x : \tau$. Then $\mathcal{K} \vdash \mathcal{T}(x) \ge \tau$. Let $\forall t_1::k_1 \cdots t_n::k_n.\tau_0 = \mathcal{T}(x)$. Then there is some S_0 such that $dom(S_0) = \{t_1, \cdots, t_n\}$, $\tau = S_0(\tau_0)$, and $\mathcal{K} \vdash S_0(t_i) :: S_0(k_i)$. By Lemma 2.1.1, $\emptyset \vdash S(S_0(t_i)) :: S(S_0(k_i))$. By the bound type variable convention, $S(S_0(\tau_0)) = (S \circ S_0)(S(\tau_0))$ and $S(S_0(k_i)) = (S \circ S_0)(S(k_i))$. So, $S \circ S_0$ is a ground substitution respecting $\{t_1::S(k_1), \cdots, t_n::S(k_n)\}$. Now suppose $\eta(x) \downarrow e'$. Then by the assumption, $e' \in P^{\forall t_1::S(k_1)\cdots t_n::S(k_n)\ S(\tau_0)}$. By the definition of the predicate P, $e' \in P^{(S\circ S_0)(S(\tau_0))}$, i.e., $e' \in P^{S(\tau)}$.

Case ABS. Suppose $\mathcal{K}, \mathcal{T} \rhd \lambda x.e_1 : \tau_1 \to \tau_2$ is derived from $\mathcal{K}, \mathcal{T}\{x : \tau_1\} \rhd e_1 : \tau_2$. $\eta(\lambda x.e_1) = \lambda x.\eta(e_1) \downarrow \lambda x.\eta(e_1)$. Let v be any element in $p^{S(\tau_1)}$ and suppose $((\lambda x.\eta(e_1))\ v) \downarrow e'$. By the definition of evaluation contexts, $[v/x](\eta(e_1)) \downarrow e'$, i.e., $\eta\{x \mapsto v\}(e_1) \downarrow e'$. Since $\eta\{x \mapsto v\}$ is a $S(\mathcal{T}\{x : \tau_1\})$-environment, by the induction hypothesis, $e' \in p^{S(\tau_2)}$. This proves $\lambda x.\eta(e_1) \in p^{S(\tau_1)\to S(\tau_2)}$.

Case APP. Suppose $\mathcal{K}, \mathcal{T} \rhd e_1\ e_2 : \tau_1$ is derived from $\mathcal{K}, \mathcal{T} \rhd e_1 : \tau_2 \to \tau_1$ and $\mathcal{K}, \mathcal{T} \rhd e_2 : \tau_2$. Also suppose $\eta(e_1\ e_2) \downarrow e'$. By the definition of evaluation

contexts, $\eta(e_1) \downarrow e_1'$ and $(e_1'\ \eta(e_2)) \downarrow e'$. By the induction hypothesis for e_1, $e_1' = v_1 \in p^{S(\tau_2)\to S(\tau_1)}$ for some value v_1. Then by the definition of evaluation contexts, $\eta(e_2) \downarrow e_2'$ and $(v_1\ e_2') \downarrow e'$. By the induction hypothesis for e_2, $e_2' = v_2 \in p^{S(\tau_2)}$ for some value v_2. Then by the definition of the predicate p, we have $e' \in p^{S(\tau_1)}$.

Case DOT. Suppose $\mathcal{K}, \mathcal{T} \rhd e_1.l : \tau$ is derived from $\mathcal{K}, \mathcal{T} \rhd e_1 : \tau_1$ and $\mathcal{K} \vdash \tau_1 :: \{\!\{l : \tau\}\!\}$. Suppose $\eta(e_1.l) \downarrow e'$. By the definition of evaluation contexts, $\eta(e_1) \downarrow e_1'$ and $e_1'.l \downarrow e'$. By the induction hypothesis, $e_1' = v \in p^{S(\tau_1)}$ for value v. Since S is a ground substitution respecting $\mathcal{K}$, by Lemma 2.1.1, $\emptyset \vdash S(\tau_1) :: \{\!\{l : S(\tau)\}\!\}$. This implies that $S(\tau_1)$ is a ground record type of the form $\{\cdots, l : S(\tau), \cdots\}$. By the definition of p, $v = \{\cdots, l = v', \cdots\}, v' \in p^{S(\tau)}$. But $\{\cdots, l = v', \cdots\}.l \downarrow v'$.

Case GEN. Suppose $\mathcal{K}, \mathcal{T} \rhd e : \sigma$ is derived from $\mathcal{K}', \mathcal{T} \rhd e : \tau$ such that $Cls(\mathcal{K}', \mathcal{T}, \tau) = (\mathcal{K}, \sigma)$. Then there are some $t_1, \cdots, t_n$ and $k_1, \cdots, k_n$ such that $\mathcal{K}' = \mathcal{K}\{t_1 :: k_1, \cdots, t_n :: k_n\}$ and $\sigma = \forall t_1 :: k_1 \cdots \forall t_n :: k_n.\tau$. By the bound type variable convention, we can assume that any of $\{t_1, \cdots, t_n\}$ do not appear in $\mathcal{S}$. Then $S(\sigma) = \forall t_1 :: S(k_1) \cdots t_n :: S(k_n).S(\tau)$. Let S' be any ground substitution such that $dom(S') = \{t_1, \cdots, t_n\}$ and S' respects $\{t_1 :: S(k_1), \cdots t_n :: S(k_n)\}$. Then $S' \circ S$ is a ground substitution that respects $\mathcal{K}\{t_1 :: k_1, \cdots, t_n :: k_n\}$, and η is a $(S' \circ S)(\mathcal{T})$-environment. By the induction hypothesis, if $\eta(e) \downarrow e'$ then $e' \in p^{S'(S(\tau))}$. This proves that $e' \in p^{S(\sigma)}$ by the definition of p.

Cases for MODIFY and VARIANT are similar to that of DOT. The case for CASE is similar to that of APP. Cases for RECORD and LET follow from the corresponding induction hypotheses. □

From this theorem, we have the following corollary.

COROLLARY 3.2.2. *If $\lambda^{let,\bullet} \vdash \emptyset, \emptyset \rhd e : \sigma$ and $e \downarrow e'$ then e' is a value of type σ.*

This says that a well-typed program (closed term) of type σ evaluates to a value of type σ, and in particular, a well-typed program will not produce a runtime type error.

3.3 Explicitly Typed Calculus $\Lambda^{let,\bullet}$ Corresponding to $\lambda^{let,\bullet}$

We define an explicitly typed calculus, $\Lambda^{let,\bullet}$, corresponding to $\lambda^{let,\bullet}$. This calculus will be used as an intermediate language for compilation of $\lambda^{let,\bullet}$.

The set of types, kinds, and kinding rules are the same as those of $\lambda^{let,\bullet}$. The set of terms of $\Lambda^{let,\bullet}$ (ranged over by M) is given by the syntax:

$$\begin{aligned} M ::= {} & (x\ \tau \cdots \tau) \mid c^b \mid \lambda x{:}\tau.M \mid M\ M \mid \mathsf{Poly}(M{:}\sigma) \mid \mathsf{let}\ x{:}\sigma = M\ \mathsf{in}\ M \\ & \mid \{l{=}M, \cdots, l{=}M\} \mid M{:}\tau.l \mid \mathsf{modify}(M.\tau, l, M) \\ & \mid (\langle l{=}M\rangle{:}\tau) \mid \mathsf{case}\ M\ \mathsf{of}\ \langle l{=}M, \cdots, l{=}M\rangle \end{aligned}$$

$(x\ \tau \cdots \tau)$ is polymorphic instantiation, and $\mathsf{Poly}(M{:}\sigma)$ is polymorphic generalization. To make the development of the compilation algorithm easier, we require type specification in a field selection term and in a field modification term. The set of free type variables of a term is written as $FTV(M)$. For a polymorphic generalization term, it is defined as $FTV(\mathsf{Poly}(M{:}\forall t_1 :: k_1 \cdots t_n :: k_n.\tau)) = FTV(M) \setminus \{t_1, \cdots, t_n\}$. The definitions for other terms are as usual. The set of typing rules is given in Figure 9.

$$\text{VAR}\quad \mathcal{K}, \mathcal{T}\{x : \forall t_1{::}k_1 \cdots \forall t_n{::}k_n.\tau_0\} \rhd (x\ \tau_1 \cdots \tau_n) \quad [\tau_1/t_1, \cdots, \tau_n/t_n](\tau_0)$$
if $\mathcal{T}\{x : \forall t_1{::}k_1 \cdots \forall t_n{::}k_n.\tau_0\}$ is well formed under $\mathcal{K}$
and $(\mathcal{K}, [\tau_1/t_1, \cdots, \tau_n/t_n])$ respects $\mathcal{K}\{t_1{::}k_1, \cdots, t_n{::}k_n\}$

$$\text{CONST}\quad \mathcal{K}, \mathcal{T} \rhd c^b \ .\ b \quad \text{if } \mathcal{T} \text{ is well formed under } \mathcal{K}$$

$$\text{APP}\quad \frac{\mathcal{K}, \mathcal{T} \rhd M_1 : \tau_1 \to \tau_2 \quad \mathcal{K}, \mathcal{T} \rhd M_2 : \tau_1}{\mathcal{K}, \mathcal{T} \rhd M_1\ M_2 : \tau_2}$$

$$\text{ABS}\quad \frac{\mathcal{K}, \mathcal{T}\{x : \tau_1\} \rhd M_1 : \tau_2}{\mathcal{K}, \mathcal{T} \rhd \lambda x.\tau_1.M_1 : \tau_1 \to \tau_2}$$

$$\text{RECORD}\quad \frac{\mathcal{K}, \mathcal{T} \rhd M_i : \tau_i \quad (1 \le i \le n)}{\mathcal{K}, \mathcal{T} \rhd \{l_1{=}M_1, \cdots, l_n{=}M_n\} \ .\ \{l_1 : \tau_1, \cdots, l_n : \tau_n\}}$$

$$\text{DOT}\quad \frac{\mathcal{K}, \mathcal{T} \rhd M : \tau_1 \quad \mathcal{K} \vdash \tau_1 :: \{\!\{l : \tau_2\}\!\}}{\mathcal{K}, \mathcal{T} \rhd M{:}\tau_1.l : \tau_2}$$

$$\text{MODIFY}\quad \frac{\mathcal{K}, \mathcal{T} \rhd M_1 : \tau_1 \quad \mathcal{K}, \mathcal{T} \rhd M_2 \cdot \tau_2 \quad \mathcal{K} \vdash \tau_1 .{::} \{\!\{l : \tau_2\}\!\}}{\mathcal{K}, \mathcal{T} \rhd \mathsf{modify}(M_1\ \tau_1, l, M_2) \cdot \tau_1}$$

$$\text{VARIANT}\quad \frac{\mathcal{K}, \mathcal{T} \rhd M : \tau_1 \quad \mathcal{K} \vdash \tau_2 :: \langle\!\langle l : \tau_1 \rangle\!\rangle}{\mathcal{K}, \mathcal{T} \rhd (\langle l{=}M\rangle\ \tau_2) : \tau_2}$$

$$\text{CASE}\quad \frac{\mathcal{K}, \mathcal{T} \rhd M : \langle l_1 \cdot \tau_1, \cdots, l_n : \tau_n\rangle \quad \mathcal{K}, \mathcal{T} \rhd M_i : \tau_i \to \tau \ (1 \le i \le n)}{\mathcal{K}, \mathcal{T} \rhd \mathsf{case}\ M\ \mathsf{of}\ \langle l_1{=}M_1, \cdots, l_n{=}M_n\rangle \ .\ \tau}$$

$$\text{GEN}\quad \frac{\mathcal{K}, \mathcal{T} \rhd M : \tau}{\mathcal{K}', \mathcal{T} \rhd Poly(M : \sigma) : \sigma} \quad \text{if } Cls(\mathcal{K}, \mathcal{T}, \tau) = (\mathcal{K}', \sigma)$$

$$\text{LET}\quad \frac{\mathcal{K}, \mathcal{T} \rhd M_1 \cdot \sigma \quad \mathcal{K}, \mathcal{T}\{x : \sigma\} \rhd M_2 : \tau}{\mathcal{K}, \mathcal{T} \rhd \mathsf{let}\ x\ \sigma = M_1\ \mathsf{in}\ M_2 \quad \tau}$$

Fig 9. Typing rules for the explicitly typed record calculus $\Lambda^{let,\bullet}$.

By regarding let $x{:}\sigma{=}M_1$ in M_2 as $(\lambda x{:}\sigma.M_2)\ M_1$, $(x\ \tau \cdots \tau)$ as the nested type application $(\cdots(x\ \tau)\cdots\tau)$, and Poly$(M{:}\sigma)$ as the nested type abstraction determined by σ, $\Lambda^{let,\bullet}$ can be regarded as a subcalculus of $\Lambda^{\forall,\bullet}$. Each of the typing rules of $\Lambda^{let,\bullet}$ is derivable in $\Lambda^{\forall,\bullet}$ under the above correspondence of the terms.

As ML corresponds to Core XML [Harper and Mitchell 1993], $\lambda^{let,\bullet}$ corresponds to $\Lambda^{let,\bullet}$ in the following sense, which can be proved similarly to the corresponding proof in Harper and Mitchell [1993].

PROPOSITION 3.3.1. *There is a one-to-one correspondence between the terms of* $\Lambda^{let,\bullet}$ *and the typing derivations of* $\lambda^{let,\bullet}$.

3.4 Kinded Unification

In order to develop a type inference algorithm, we need to refine Robinson's [1965] unification algorithm to incorporate kind constraints on type variables.

A *kinded set of equations* is a pair $(\mathcal{K}, E)$ consisting of a kind assignment $\mathcal{K}$ and a set E of pairs of types such that E is well formed under $\mathcal{K}$. We say that a substitution S *satisfies* E if $S(\tau_1) = S(\tau_2)$ for all $(\tau_1, \tau_2) \in E$. A kinded substitution $(\mathcal{K}_1, S)$ is a *unifier* of a kinded set of equations $(\mathcal{K}, E)$ if it respects $\mathcal{K}$ and if S satisfies E. $(\mathcal{K}_1, S)$ is a *most general unifier* of $(\mathcal{K}_2, E)$ if it is a unifier of $(\mathcal{K}_2, E)$ and if for any unifier $(\mathcal{K}_3, S_2)$ of $(\mathcal{K}_2, E)$ there is some substitution S_3 such that

$(\mathcal{K}_3, S_3)$ respects $\mathcal{K}_1$ and $S_2 = S_3 \circ S$.

We define a kinded unification algorithm $\mathcal{U}$ in the style of Gallier and Snyder [1989] by transformation. In our system each rule transforms a 4-tuple of the form $(E, \mathcal{K}, S, \mathcal{SK})$ consisting of a set E of type equations, a kind assignment $\mathcal{K}$, a substitution S, and a (not necessarily well-formed) kind assignment $\mathcal{SK}$. Intended roles of these components are: E keeps the set of equations to be unified; $\mathcal{K}$ specifies kind constrains to be verified; S records "solved" equations as a form of substitution; and $\mathcal{SK}$ records "solved" kind constraints that has already been verified for S.

In specifying rules, we treat functions $\mathcal{K}, \mathcal{SK}$, 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 $\{\!\{F\}\!\}$ to denote the record type identified by F and the record kind identified by F, respectively. Similar notations are used for variant types and variant kinds. For two functions F_1 and F_2 we write $F_1 \pm F_2$ for the function F such that $dom(F) = dom(F_1) \cup dom(F_2)$ and such that for $l \in dom(F)$, $F(l) = F_1(l)$ if $l \in dom(F_1)$; otherwise $F(l) = F_2(l)$. Figure 10 gives the set of transformation rules.

Let $(\mathcal{K}, E)$ be a given kinded set of equations. The algorithm $\mathcal{U}$ first transforms the system $(E, \mathcal{K}, \emptyset, \emptyset)$ to $(E', \mathcal{K}', S, \mathcal{SK})$ until no more rules can apply. It then returns the pair $(\mathcal{K}', S)$ if $E' = \emptyset$; otherwise it reports failure. We then have the following theorem, whose proof is deferred to the Appendix.

THEOREM 3.4.1. *The algorithm $\mathcal{U}$ takes any kinded set of equations, computes a most general unifier if one exists, and reports failure otherwise.*

The careful reader may have noticed that we could have required a stronger "occur check" condition when eliminating a type variable. For example, in the rule II, we could have required $t \notin EFTV(\mathcal{K} \cup \{(t, U)\}, \tau)$ instead of $t \notin FTV(\tau)$. Requiring this stronger condition corresponds to disallowing kind assignments having "cyclic dependencies" such as $\{t_1 :: \{\!\{l : t_2\}\!\}, t_2 :: \{\!\{l' : t_1\}\!\}\}$ we have mentioned in Section 2. The rationale behind not taking this approach is that the stronger condition would increase the complexity of the unification algorithm due to the extra check of acyclicity every time a substitution is generated. Since unification is repeatedly performed, this would slow down the type inference algorithm. Although our approach allows some useless open terms, such as $\{t_1 :: \{\!\{l : t_1 \to int\}\!\}\}\{x : t_1\} \rhd (x.l)\ x : int$, the typability on closed terms does not change and therefore does not create any problems. Also, if we extend the type system to recursive types using regular trees [Courcelle 1983], allowing those "cyclic" kind assignments would become essential. Buneman and Ohori [1995] discusse possible usefulness of recursive programming with record polymorphism, and Vasconcelos' recent work [Vasconcelos 1994] extends our kinded unification to infinite regular trees.

3.5 The Type Inference Algorithm

Using the kinded unification, Milner's [1978] type inference algorithm is extended with record polymorphism. Figure 11 gives the algorithm, $\mathcal{WK}$, which simultaneously infers, for a given typable raw term in $\lambda^{let,\bullet}$, its principal typing (in the sense of the theorem below) and the corresponding explicitly typed term of $\Lambda^{let,\bullet}$. In this definition, it is implicitly assumed that the algorithm fails if unification or any of the recursive calls on subterms fail.

(I) $(E \cup \{(\tau,\tau)\},\ \mathcal{K},\ S,\ \mathcal{SK}) \Longrightarrow (E,\ \mathcal{K},\ S,\ \mathcal{SK})$

(II) $(E \cup \{(t,\tau)\},\ \mathcal{K} \cup \{(t,U)\},\ S,\ \mathcal{SK}) \Longrightarrow$
$([\tau/t](E),\ [\tau/t](\mathcal{K}),\ [\tau/t](S) \cup \{(t,\tau)\},\ [\tau/t](\mathcal{SK}) \cup \{(t,U)\})$ if $t \notin FTV(\tau)$

(III) $(E \cup \{(t_1,t_2)\},\ \mathcal{K} \cup \{(t_1,\{\!|F_1|\!\}),(t_2,\{\!|F_2|\!\})\},\ S,\ \mathcal{SK}) \Longrightarrow$
$([t_2/t_1](E \cup \{(F_1(l),F_2(l))|l \in dom(F_1) \cap dom(F_2)\}),$
$[t_2/t_1](\mathcal{K}) \cup \{(t_2,[t_2/t_1](\{\!|F_1 \pm F_2|\!\}))\},$
$[t_2/t_1](S) \cup \{(t_1,t_2)\},\ [t_2/t_1](\mathcal{SK}) \cup \{(t_1,\{\!|F_1|\!\})\})$

(IV) $(E \cup \{(t_1,\{F_2\})\},\ \mathcal{K} \cup \{(t_1,\{\!|F_1|\!\})\},\ S,\ \mathcal{SK}) \Longrightarrow$
$([\{F_2\}/t_1](E \cup \{(F_1(l),F_2(l))|l \in dom(F_1)\}),\ [\{F_2\}/t_1](\mathcal{K}),$
$[\{F_2\}/t_1](S) \cup \{(t_1,\{F_2\})\},\ [\{F_2\}/t_1](\mathcal{SK}) \cup \{(t_1,\{\!|F_1|\!\})\})$
if $dom(F_1) \subseteq dom(F_2)$ and $t \notin FTV(\{F_2\})$

(V) $(E \cup \{(\{F_1\},\{F_2\})\},\mathcal{K},S,\mathcal{SK}) \Longrightarrow (E \cup \{(F_1(l),F_2(l))|l \in dom(F_1)\},\mathcal{K},S,\mathcal{SK})$
if $dom(F_1) = dom(F_2)$

(VI) $(E \cup \{(t_1,t_2)\},\ \mathcal{K} \cup \{(t_1,\langle\!\langle F_1 \rangle\!\rangle),(t_2,\langle\!\langle F_2 \rangle\!\rangle)\},\ S,\ \mathcal{SK}) \Longrightarrow$
$([t_2/t_1](E \cup \{(F_1(l),F_2(l))|l \in dom(F_1) \cap dom(F_2)\}),$
$[t_2/t_1](\mathcal{K}) \cup \{(t_2,[t_2/t_1](\langle\!\langle F_1 \pm F_2 \rangle\!\rangle))\},$
$[t_2/t_1](S) \cup \{(t_1,t_2)\},\ [t_2/t_1](\mathcal{SK}) \cup \{(t_1,\langle\!\langle F_1 \rangle\!\rangle)\})$

(VII) $(E \cup \{(t_1,\langle F_2 \rangle)\},\ \mathcal{K} \cup \{(t_1,\langle\!\langle F_1 \rangle\!\rangle)\},\ S,\ \mathcal{SK}) \Longrightarrow$
$([\langle F_2 \rangle/t_1](E \cup \{(F_1(l),F_2(l))|l \in dom(F_1)\}),\ [\langle F_2 \rangle/t_1](\mathcal{K}),$
$[\langle F_2 \rangle/t_1](S) \cup \{(t_1,\langle F_2 \rangle)\},\ [\langle F_2 \rangle/t_1](\mathcal{SK}) \cup \{(t_1,\langle\!\langle F_1 \rangle\!\rangle)\})$
if $dom(F_1) \subseteq dom(F_2)$ and $t_1 \notin FTV(\langle F_2 \rangle)$

(VIII) $(E \cup \{(\langle F_1 \rangle,\langle F_2 \rangle)\},\ \mathcal{K},\ S,\ \mathcal{SK}) \Longrightarrow (E \cup \{(F_1(l),F_2(l))|l \in dom(F_1)\},\ \mathcal{K},\ S,\ \mathcal{SK})$
if $dom(F_1) = dom(F_2)$

(IX) $(E \cup \{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2)\},\ \mathcal{K},\ S,\ \mathcal{SK}) \Longrightarrow (E \cup \{(\tau_1^1,\tau_2^1),(\tau_1^2,\tau_2^2)\},\ \mathcal{K},\ S,\ \mathcal{SK})$

For a notation of the form $X \cup Y$ appeared in the left hand side of each rule, we assume that X and Y are disjoint.

Fig. 10. Transformation rules for kinded unification.

For a term M of $\Lambda^{let,\bullet}$, the *type erasure* of M, denoted by $erase(M)$, is the term of $\lambda^{let,\bullet}$ obtained from M by erasing all type information. The definition of $erase(M)$ is obtained by extending the following clauses inductively according to the other term constructors.

$$
\begin{aligned}
erase((x\ \tau_1 \cdots \tau_n)) &= x \\
erase(\lambda x:\tau.M_1) &= \lambda x.erase(M_1) \\
erase(M_1 : \tau.l) &= erase(M_1).l \\
erase(\mathsf{modify}(M_1{:}\tau,l,M_2)) &= \mathsf{modify}(erase(M_1),l,erase(M_2)) \\
erase((\langle l{=}M_1\rangle{:}\tau)) &= \langle l{=}erase(M_1)\rangle \\
erase(\mathsf{Poly}(M_1{:}\sigma)) &= erase(M_1) \\
erase(\mathsf{let}\ x{:}\sigma = M_1\ \mathsf{in}\ M_2) &= \mathsf{let}\ x{=}erase(M_1)\ \mathsf{in}\ erase(M_2)
\end{aligned}
$$

$$\mathcal{WK}(\mathcal{K},\mathcal{T},x) = \text{if } x \notin dom(\mathcal{T}) \text{ then } failure$$
$$\text{else let } \forall t_1{::}k_1 \ldots \forall t_n{::}k_n.\tau = \mathcal{T}(x),$$
$$S = [s_1/t_1,\cdots,s_n/t_n] \ (s_1,\cdots,s_n \text{ are fresh})$$
$$\text{in } (\mathcal{K}\{s_1{::}S(k_1),\cdots,s_n{::}S(k_n)\},\ \emptyset,\ (x\ s_1\cdots s_n),\ S(\tau))$$

$$\mathcal{WK}(\mathcal{K},\mathcal{T},\lambda x.e_1) = \text{let } (\mathcal{K}_1,S_1,M_1,\tau_1) = \mathcal{WK}(\mathcal{K}\{t{::}U\},\mathcal{T}\{x:t\},e_1)\ (t \text{ fresh})$$
$$\text{in } (\mathcal{K}_1,S_1,\lambda x:S_1(t).M_1,S_1(t)\rightarrow\tau_1).$$

$$\mathcal{WK}(\mathcal{K},\mathcal{T},e_1\ e_2) = \text{let } (\mathcal{K}_1,S_1,M_1,\tau_1) = \mathcal{WK}(\mathcal{K},\mathcal{T},e_1)$$
$$(\mathcal{K}_2,S_2,M_2,\tau_2) = \mathcal{WK}(\mathcal{K}_1,S_1(\mathcal{T}),e_2)$$
$$(\mathcal{K}_3,S_3) = \mathcal{U}(\mathcal{K}_2,\{(S_2(\tau_1),\tau_2\rightarrow t)\})\ (t \text{ fresh})$$
$$\text{in } (\mathcal{K}_3,\ S_3\circ S_2\circ S_1,\ (S_3\circ S_2(M_1)\ S_3(M_2)),\ S_3(t)).$$

$$\mathcal{WK}(\mathcal{K},\mathcal{T},\{l_1{=}e_1,\cdots,l_n{=}e_n\}) =$$
$$\text{let } (\mathcal{K}_1,S_1,M_1,\tau_1) = \mathcal{WK}(\mathcal{K},\mathcal{T},e_1)$$
$$(\mathcal{K}_i,S_i,M_i,\tau_i) = \mathcal{WK}(\mathcal{K}_{i-1},S_{i-1}\circ\cdots\circ S_1(\mathcal{T}),e_i)\ (2\le i\le n)$$
$$\text{in } (\mathcal{K}_n,\ S_n\circ\cdots\circ S_2\circ S_1,$$
$$\{l_1{=}S_n\circ\cdots\circ S_2(M_1),\cdots,\ l_i{=}S_n\circ\cdots\circ S_{i+1}(M_i),\cdots,\ l_n{=}M_n\},$$
$$\{l_1:S_n\circ\cdots S_2(\tau_1),\cdots,l_i:S_n\circ\cdots\circ S_{i+1}(\tau_i),\cdots,l_n:\tau_n\})$$

$$\mathcal{WK}(\mathcal{K},\mathcal{T},e_1.l) = \text{let } (\mathcal{K}_1,S_1,M_1,\tau_1) = \mathcal{WK}(\mathcal{K},\mathcal{T},e_1)$$
$$(\mathcal{K}_2,S_2) = \mathcal{U}(\mathcal{K}_1\{t_1{::}U,t_2{::}\{\!\{l:t_1\}\!\}\},\{(t_2,\tau_1)\})\ (t_1,t_2 \text{ fresh})$$
$$\text{in } (\mathcal{K}_2,\ S_2\circ S_1,\ S_2(M_1):S_2(t_2).l,\ S_2(t_1)).$$

$$\mathcal{WK}(\mathcal{K},\mathcal{T},\mathsf{modify}(e_1,l,e_2)) =$$
$$\text{let } (\mathcal{K}_1,S_1,M_1,\tau_1) = \mathcal{WK}(\mathcal{K},\mathcal{T},e_1)$$
$$(\mathcal{K}_2,S_2,M_2,\tau_2) = \mathcal{WK}(\mathcal{K}_1,S_1(\mathcal{T}),e_2)$$
$$(\mathcal{K}_3,S_3) = \mathcal{U}(\mathcal{K}_2\{t_1{::}U,t_2{::}\{\!\{l:t_1\}\!\}\},\{(t_1,\tau_2),(t_2,S_2(\tau_1))\})\ (t_1,t_2 \text{ fresh})$$
$$\text{in } (\mathcal{K}_3,\ S_3\circ S_2\circ S_1,\ \mathsf{modify}(S_3\circ S_2(M_1):S_3(t_2),l,S_3(M_2)),\ S_3(t_2)).$$

$$\mathcal{WK}(\mathcal{K},\mathcal{T},\mathsf{case}\ e_0\ \mathsf{of}\ \langle l_1{=}e_1,\cdots,l_n{=}e_n\rangle) =$$
$$\text{let } (\mathcal{K}_0,S_0,M_0,\tau_0) = \mathcal{WK}(\mathcal{K},\mathcal{T},e_0)$$
$$(\mathcal{K}_i,S_i,M_i,\tau_i) = \mathcal{WK}(\mathcal{K}_{i-1},S_{i-1}\circ\cdots\circ S_0(\mathcal{T}),e_i)\ (1\le i\le n)$$
$$(\mathcal{K}_{n+1},S_{n+1}) = \mathcal{U}(\mathcal{K}_n\{t_0{::}U,\ \cdots,t_n{::}U\},\ \{(S_n\circ\cdots S_1(\tau_0),\langle l_1:t_1,\cdots,l_n\cdot t_n\rangle)\}$$
$$\cup\{(S_n\circ\cdots S_{i+1}(\tau_i),t_i\rightarrow t_0)|1\le i\le n\})\quad (t_0,\cdots,t_n \text{ fresh})$$
$$\text{in } (\mathcal{K}_{n+1},\ S_{n+1}\circ\cdots\circ S_0,$$
$$(\mathsf{case}\ S_{n+1}\circ\cdots\circ S_1(M_0)\ \mathsf{of}\ \langle\cdots,l_i{=}S_{n+1}\circ\cdots\circ S_{i+1}(M_i),\cdots\rangle),\ S_{n+1}(t_0))$$

$$\mathcal{WK}(\mathcal{K},\mathcal{T},\langle l{=}e_1\rangle) = \text{let } (\mathcal{K}_1,S_1,M_1,\tau_1) = \mathcal{WK}(\mathcal{K},\mathcal{T},e_1)$$
$$\text{in } (\mathcal{K}_1\{t{::}\langle\!\langle l:\tau_1\rangle\!\rangle\},\ S_1,\ (\langle l=M_1\rangle:t),\ t)\ (t \text{ fresh})$$

$$\mathcal{WK}(\mathcal{K},\mathcal{T},\mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2) = \text{let } (\mathcal{K}_1,S_1,M_1,\tau_1) = \mathcal{WK}(\mathcal{K},\mathcal{T},e_1)$$
$$(\mathcal{K}_1',\sigma_1) = Cls(\mathcal{K}_1,S_1(\mathcal{T}),\tau_1)$$
$$(\mathcal{K}_2,S_2,M_2,\tau_2) = \mathcal{WK}(\mathcal{K}_1',(S_1(\mathcal{T}))\{x:\sigma_1\},e_2)$$
$$\text{in } (\mathcal{K}_2,\ S_2\circ S_1,\ \mathsf{let}\ \mathsf{x}.S_2(\sigma_1) = \mathsf{Poly}(S_2(M_1:\sigma_1))\ \mathsf{in}\ M_2,\ \tau_2)$$

Fig. 11. Type inference algorithm.

This algorithm is sound and complete in the following sense.

THEOREM 3.5.1. *If* $\mathcal{WK}(\mathcal{K},\mathcal{T},e) = (\mathcal{K}',S,M,\tau)$ *then the following properties hold:*

(1) $(\mathcal{K}',S)$ *respects* $\mathcal{K}$ *and* $\lambda^{let,\bullet} \vdash \mathcal{K}', S(\mathcal{T}) \rhd e : \tau$,

(2) $erase(M) = e$ *and* $\Lambda^{let,\bullet} \vdash \mathcal{K}', S(\mathcal{T}) \rhd M : \tau$,

(3) *if* $\lambda^{let,\bullet} \vdash \mathcal{K}_0, S_0(\mathcal{T}) \rhd e : \tau_0$ *for some* $(\mathcal{K}_0, S_0)$ *and* τ_0 *such that* $(\mathcal{K}_0, S_0)$ *respects* $\mathcal{K}$ *then there is some* S' *such that* $(\mathcal{K}_0, S')$ *respects* $\mathcal{K}'$, $\tau_0 = S'(\tau)$, *and* $S_0(\mathcal{T}) = S'(S(\mathcal{T}))$.

If $\mathcal{WK}(\mathcal{K},\mathcal{T},e)$ *fails then there is no* $(\mathcal{K}_0, S_0)$ *and* τ_0 *such that* $(\mathcal{K}_0, S_0)$ *respects* $\mathcal{K}$ *and* $\lambda^{let,\bullet} \vdash \mathcal{K}_0, S_0(\mathcal{T}) \rhd e : \tau_0$.

The proof is deferred to the Appendix.

We end this section by showing some examples. For the example field selection function *name* and variant term *payment* given before, the algorithm computes the following data:

$$\mathcal{WK}(\emptyset,\emptyset,\lambda\text{x.x.Name}) = (\{t_2{::}U, t_3{::}\{\!\{Name : t_2\}\!\}\}, [t_3/t_1], \lambda\text{x:}t_3\text{.x.Name}, t_3 \to t_2)$$

$$\mathcal{WK}(\emptyset,\emptyset,\langle\text{Pound=100.0}\rangle) = (\{t{::}\langle\!\langle Pound : real\rangle\!\rangle\}, \emptyset, (\langle\text{Pound=100.0}\rangle{:}t), t)$$

where t_1 in the first example is a type variable introduced during the type inference process and is irrelevant to the final result. We indeed have the following typings.

$$\begin{aligned}
\lambda^{let,\bullet} \vdash \{t_2{::}U, t_3{::}\{\!\{Name : t_2\}\!\}\}, \emptyset &\rhd \lambda\text{x.x.Name} : t_3 \to t_2\\
\Lambda^{let,\bullet}\{t_2{::}U, t_3{::}\{\!\{Name : t_2\}\!\}\}, \emptyset &\rhd \lambda\text{x:}t_3\text{.x.Name} : t_3 \to t_2\\
\lambda^{let,\bullet}\{t{::}\langle\!\langle Pound : real\rangle\!\rangle\}, \emptyset &\rhd \langle\text{Pound=100 0}\rangle : t\\
\Lambda^{let,\bullet}\{t{::}\langle\!\langle Pound : real\rangle\!\rangle\}, \emptyset &\rhd (\langle\text{Pound=100.0}\rangle{:}t) : t
\end{aligned}$$

When these terms are let bound, the type variables are abstracted. The following are results of type inference for the expression containing *name* and *payment* given in Section 1.2.

```
WK(∅,∅,let name=λx x.Name in
          (name {Name="Joe",Office=403},
           name {Name="Hanako",Age=21,Phone=7222})) =
 (∅, S, let name.∀t1::U.∀t2::{{Name : t1}}.t2 → t1 =
          Poly(λx·t2 x.Name . ∀t1::U.∀t2::{{Name : t1}}.t2 → t1)
       in ((name string {Name : string, Office : string}) {Name="Joe",Office=403},
           (name string {Name : string, Age : int, Phone : int})
                        {Name="Hanako",Age=21,Phone=7222}),
  string × string))

WK(∅,∅,let payment=⟨Pound=100.0⟩ in
          (case payment of ⟨Pound=λx x, Dollar=λx.x * 0 68⟩,
           case payment of ⟨Pound=λx real_to_int(x * 150 0), Yen = λx x⟩)) =
 (∅, S, let payment·∀t::⟪Pound : real⟫.t =
          Poly((⟨Pound=100 0⟩ . t)  ∀t::⟪Pound : real⟫.t)
       in (case (payment ⟨Pound : real, Dollar : real⟩) of
             ⟨Pound=λx:real x, Dollar=λx real x * 0.68⟩,
```

case (payment ⟨*Pound* : *real*, *Yen* : *int*⟩) of
⟨Pound=λx:*real*.real_to_int(x * 150.0), Yen=λx:*int*.x)))), *real* × *int*))

4. COMPILATION

This section develops an algorithm to compile the ML-style polymorphic record calculus $\lambda^{let,\bullet}$ into an implementation calculus $\lambda^{let,[]}$, defined below.

4.1 Implementation Calculus : $\lambda^{let,[]}$

We define an implementation calculus with directly indexable vectors and switch statements on integer tags as an efficient abstract machine for polymorphic record calculi.

As we shall see later, index values are always computed statically by our compilation algorithm defined later, and there is no need to treat them as first-class values. So we introduce the following new syntactic category of *indexes* (ranged over by $\mathcal{I}$) and treat them specially:

$$\mathcal{I} ::= I \mid i$$

where I stands for a given set of *index variables* and i for natural numbers. The set of raw terms of $\lambda^{let,[]}$ (ranged over by C) is given by the syntax:

$$\begin{aligned} C \ ::= \ & x \mid c^b \mid \lambda x.C \mid C\ C \mid \mathsf{let}\ x{=}C\ \mathsf{in}\ C \mid \{C,\cdots,C\} \mid C\,[\mathcal{I}] \\ & \mid \mathsf{modify}(C,\mathcal{I},C) \mid \langle \mathcal{I}{=}C \rangle \mid \mathsf{switch}\ C\ \mathsf{of}\ C,\cdots,C \mid \lambda I.C \mid C\ \mathcal{I} \end{aligned}$$

where $\{C_1,\cdots,C_n\}$ is a vector representation of a record; $C\,[\mathcal{I}]$ is index expression retrieving the element of index value $\mathcal{I}$ from vector C; switch C of $C_1,\cdots,C_n$ analyzes the integer tag of a variant C and applies the corresponding function C_i to the value of C; $\lambda I.C$ is index abstraction; and $C\ \mathcal{I}$ is index application.

As in $\lambda^{let,\bullet}$, call-by-value operational semantics is defined using *evaluation contexts* (ranged over by $EV[]$), the set of *values* (ranged over by V), and call-by-value *context-rewriting axioms* of the form $EV[C_1] \longrightarrow EV[C_2]$. We say that C *evaluates to* C' *in one step*, written $C \xrightarrow{EV} C'$, if there are $EV[]_1, C_1, C_2$ such that $C = EV[C_1]_1, EV[C_1]_1 \longrightarrow EV[C_2]_1$, and $C' = EV[C_2]_1$. We write $\xrightarrow{EV}\!\!\!\!\!\rightarrow$ for the reflexive transitive closure of $\xrightarrow{EV}$; we write $C \downarrow C'$ if $C \xrightarrow{EV}\!\!\!\!\!\rightarrow C'$ and if there is no C'' such that $C' \xrightarrow{EV} C''$; and we write $C \downarrow$ if $C \downarrow C'$ for some C'. Figure 12 gives a mutual recursive definitions of the set of values, call-by-value evaluation contexts, and the set of context-rewriting axioms of $\lambda^{let,[]}$.

4.2 The Type System of $\lambda^{let,[]}$

To establish the correctness of the compilation algorithm defined in the following subsection, we define a type system for the implementation calculus.

To represent labeled records and labeled variants in the implementation calculus, we assume a total order $\ll$ on the set of labels and restrict that a record type $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ or a variant type $\langle l_1 : \tau_1, \cdots, l_n : \tau_n \rangle$ must satisfy the condition $l_1 \ll \cdots \ll l_n$. The usual choice for $\ll$ is the lexicographical ordering on the string representations of labels. If τ is one of the above forms, we define the *index of a label* l_i *in* τ to be i. A record term of the above record type is a vector whose ith element is l_i field. An l_i variant of the above variant type is a value tagged

$$V ::= c^b \mid \lambda x\, C \mid \{V,\cdots,V\} \mid \langle \mathcal{I}{=}V\rangle \mid \lambda I\, C' \text{ (for some } C' \text{ such that } C' \downarrow C').$$

$$\begin{aligned} EV[] ::= {} & [\cdot] \mid EV[]\ C \mid V\ EV[] \mid \mathsf{let}\ x{=}EV[]\ \mathsf{in}\ C \mid \{V,\cdots,V,EV[],\cdots\} \mid EV[][\mathcal{I}] \\ & \mid \mathsf{modify}(EV[],\mathcal{I},C) \mid \mathsf{modify}(V,\mathcal{I},EV[]) \mid \langle \mathcal{I}{=}EV[]\rangle \mid \mathsf{switch}\ EV[]\ \mathsf{of}\ C,\cdots,C \\ & \mid EV[]\ \mathcal{I} \mid \lambda I.EV[] \end{aligned}$$

$$\begin{aligned} EV[(\lambda x.C)\ V] &\longrightarrow EV[[V/x]C] \\ EV[\{V_1,\cdots,V_n\}[i]] &\longrightarrow EV[V_i] \\ EV[\mathsf{modify}(\{V_1,\cdots,V_{i-1},V_i,V_{i+1},\cdots,V_n\},i,V)] &\longrightarrow EV[\{V_1,\cdots,V_{i-1},V,V_{i+1},\cdots,V_n\}] \\ EV[\mathsf{switch}\ \langle i{=}V\rangle\ \mathsf{of}\ C_1,\cdots,C_n] &\longrightarrow EV[C_i\ V] \\ EV[(\lambda I.C')\ \mathcal{I}] &\longrightarrow EV[[\mathcal{I}/I]C] \text{ if } C' \downarrow C' \\ EV[\mathsf{let}\ x = V\ \mathsf{in}\ C] &\longrightarrow EV[[V/x]C] \end{aligned}$$

Fig. 12. Call-by-value evaluation operational semantics of $\lambda^{let,[]}$.

with integer i manipulated by a switch statement containing a vector of functions whose jth element corresponds to the function for l_j variant. For example, if a record type consists of an *Age* field of type *int* and a *Name* field of type *string*, then it must be of the form $\{Age : int, Name : string\}$, and thus the index of *Name* in this record type is 2. A possible term of this type includes {21,"Joe"}, which corresponds to {Name="Joe", Age=21} in $\lambda^{let,\bullet}$. Similarly, a variant type with *Pound* variant of type *real* and *Dollar* variant of type *real* must be of the form $\langle Dollar : real, Pound : real\rangle$, and thus the index of *Pound* in the type is 2. A switch statement for this type consists of a vector of functions for *Dollar* and *Pound* in this order, and a *Pound* variant of 100.0 of this type is represented as $\langle 2{=}100.0\rangle$, which corresponds to a (monomorphic) term of $\langle$Pound=100.0$\rangle$ of the above type in $\lambda^{let,\bullet}$.

To account for polymorphic operations, we introduce a new form of types $idx(l,\tau)$ for index values. When τ is a record type or a variant type, this type denotes the index of l in τ. We write $|idx(l,\tau)|$ for the index value denoted by $idx(l,\tau)$. For example, $|idx(Name,\{Age : int, Name : string\})| = 2$. When τ is a type variable t, then $idx(l,t)$ denotes possible index values depending on instantiations of t, and $|idx(l,t)|$ is undefined.

The set of types of the implementation calculus is given by the following syntax:

$$\begin{aligned} \tau &::= t \mid c^b \mid \tau \rightarrow \tau \mid \{l : \tau,\cdots,l : \tau\} \mid \langle l : \tau,\cdots,l : \tau\rangle \mid idx(l,\tau) \Rightarrow \tau \\ \sigma &::= \tau \mid \forall t :: k.\sigma \end{aligned}$$

where $idx(l,\tau_1) \Rightarrow \tau_2$ denotes functions that take an index value denoted by $idx(l,\tau_1)$ and yield a value of type τ_2. Since index values are not first-class objects, it is not necessary to included index types $idx(l,\tau)$ as separate types. The set of kinds and the kinding rules are the same as those of $\lambda^{let,\bullet}$.

The type system of this calculus is defined as a proof system for the following forms of judgments:

$$\begin{aligned} \mathcal{K},\mathcal{L},\mathcal{T} \rhd C : \tau &\quad \text{typing judgment} \\ \mathcal{L} \vdash \mathcal{I} : idx(l,\tau) &\quad \text{index judgment} \end{aligned}$$

where $\mathcal{K}$ is a kind assignment and $\mathcal{T}$ a type assignment as in the previous calculi. $\mathcal{L}$ is an *index assignment*, which is a mapping from a set of index variables to index types of the form $idx(l, \tau)$. A type τ is *well formed under* $\mathcal{K}$ if $FTV(\tau) \subseteq dom(\mathcal{K})$ and if for any type of the form $idx(l, \tau')$ appearing in τ, τ' has a record kind or variant kind containing l field under $\mathcal{K}$. A type $\forall t_1::k_1 \cdots t_n::k_n.\tau$ is *well formed under* $\mathcal{K}$ if τ is well formed under $\mathcal{K}\{t_1::k_1 \cdots t_n::k_n\}$. $\mathcal{L}$ is *well formed under* $\mathcal{K}$ if each type in $\mathcal{L}$ is well formed under $\mathcal{K}$. A type assignment $\mathcal{T}$ is *well formed under* $\mathcal{K}$ if each type in $\mathcal{T}$ is well formed under $\mathcal{K}$.

The set of typing rules is given in Figure 13. Since we are not concerned with type inference of $\lambda^{let,[]}$, we adopt a more-general and more-natural rule for GEN than the one we used for $\lambda^{let,\bullet}$. This makes the proof of the subject reduction property below slightly easier. We write $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C : \sigma$ if $\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C : \sigma$ is derived in this poof system. It is easily verified that if $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C : \sigma$ then σ is well formed under $\mathcal{K}$.

For this type system, we show the subject reduction property, which will be useful later in establishing that our compilation algorithm preserves the operational behavior of $\lambda^{let,\bullet}$. Since our usage of $\lambda^{let,[]}$ is as an abstract machine to implement $\lambda^{let,\bullet}$, we do not need a stronger property of type soundness of $\lambda^{let,[]}$ itself. The type soundness of $\lambda^{let,\bullet}$ with respect to the operational semantics of the compiled term of $\lambda^{let,[]}$ will follow from the correctness of compilation we shall establish later.

The reduction axioms for $\lambda^{let,[]}$ are given in Figure 14. We say that C_1 *reduces to* C_2 *in one step*, written $C_1 \rightarrow C_2$, if C_2 is obtained from C_1 by applying one of the reduction axioms to some subterm of C_1. The reduction relation $C_1 \rightarrow\!\!\!\!\rightarrow C_2$ is defined as the reflexive transitive closure of $\rightarrow$. The following substitution lemmas are useful in proving the subject reduction theorem.

LEMMA 4.2.1. *If* $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}, \mathcal{T}\{x : \sigma_1\} \rhd C_1 : \sigma_2$ *and* $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_2 : \sigma_1$ *then* $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd [C_2/x]C_1 : \sigma_2$.

PROOF. The proof is by induction on the typing derivation of C_1. The only interesting case is variable axiom. Other cases are similar to Lemma 2.2.4.

Suppose $\mathcal{K}, \mathcal{L}, \mathcal{T}\{x : \sigma_1\} \rhd y : \tau_2$ is a VAR axiom. The case for $x \neq y$ is trivial. Suppose $x = y$. Let $\sigma_1 = \forall t_1::k_1 \cdots t_n::k_n.\tau_1$. Then $\mathcal{K} \vdash \forall t_1::k_1 \cdots t_n::k_n.\tau_1 \geq \tau_2$ and therefore there must be some S such that $dom(S) = \{t_1, \cdots, t_n\}$, $(\mathcal{K}, S)$ respects $\mathcal{K}\{t_1::k_1 \cdots t_n::k_n\}$ and $S(\tau_1) = \tau_2$. If $n = 0$, i.e., σ_1 is a monotype, then $\sigma_1 = \tau_2$, and therefore $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_2 : \tau_2$. Otherwise $\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_2 : \sigma_1$ must be derived by GEN, and therefore $\lambda^{let,[]} \vdash \mathcal{K}\{t_1::k_1 \cdots t_n::k_n\}, \mathcal{L}, \mathcal{T} \rhd C_2 : \tau_1$ such that $\{t_1, \cdots, t_n\}$ does not appear in $\mathcal{T}$ or $\mathcal{L}$. By Lemma 2.2.3 for $\lambda^{let,\bullet}$, we have $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_2 : \tau_2$. □

LEMMA 4.2.2. *If* $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}\{I : idx(l, \tau_1)\}, \mathcal{T} \rhd C : \tau$ *and* $\mathcal{L} \vdash \mathcal{I} : idx(l, \tau_1)$ *then* $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd [\mathcal{I}/I]C : \tau$.

PROOF. This is provde by induction on the typing derivation of C. We only show the case for rule INDEX. The cases for IAPP, MODIFY, and VARIANT can be shown similarly. All the other cases follow directly from the induction hypothesis.

Suppose $\mathcal{K}, \mathcal{L}\{I : idx(l, \tau_1)\}, \mathcal{T} \rhd C_1[\mathcal{I}_1] : \tau$ is derived by rule INDEX from $\mathcal{K}, \mathcal{L}\{I : idx(l, \tau_1)\}, \mathcal{T} \rhd C_1 : \tau_2$, from $\mathcal{K} \vdash \tau_2 :: \{\!\{l' : \tau\}\!\}$, and from $\mathcal{L}\{I : idx(l, \tau_1)\} \vdash \mathcal{I}_1 : idx(l', \tau_2)$. By the induction hypothesis, $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd [\mathcal{I}/I]C_1 : \tau_2$. There

$$\text{IVAR}\quad \mathcal{L}\{I : idx(l,\tau)\} \vdash I : idx(l,\tau)$$

$$\text{ICONST1}\quad \mathcal{L} \vdash i : idx(l_i, \{l_1 : \tau_1, \cdots, l_n : \tau_n\}) \quad 1 \le i \le n$$

$$\text{ICONST2}\quad \mathcal{L} \vdash i : idx(l_i, \langle l_1 : \tau_1, \cdots, l_n : \tau_n\rangle) \quad 1 \le i \le n$$

$$\text{VAR}\quad \mathcal{K}, \mathcal{L}, \mathcal{T}\{x : \sigma\} \rhd x : \tau \quad \text{if } \mathcal{T}\{x : \sigma\} \text{ and } \mathcal{L} \text{ are well formed under } \mathcal{K} \text{ and } \mathcal{K} \vdash \sigma \ge \tau$$

$$\text{CONST}\quad \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd c^b : b \quad \text{if } \mathcal{T} \text{ and } \mathcal{L} \text{ are well formed under } \mathcal{K}$$

$$\text{APP}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_1 : \tau_1 \to \tau_2 \quad \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_2 : \tau_1}{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_1\ C_2 : \tau_2}$$

$$\text{ABS}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T}\{x : \tau_1\} \rhd C_1 : \tau_2}{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd \lambda x.C_1 : \tau_1 \to \tau_2}$$

$$\text{IABS}\quad \frac{\mathcal{K}, \mathcal{L}\{I : idx(l,\tau_1)\}, \mathcal{T} \rhd C_1 : \tau_2}{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd \lambda I.C_1 : idx(l,\tau_1) \Rightarrow \tau_2}$$

$$\text{IAPP}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C : idx(l,\tau_1) \Rightarrow \tau_2 \quad \mathcal{L} \vdash \mathcal{I} : idx(l,\tau_1)}{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C\ \mathcal{I} : \tau_2}$$

$$\text{RECORD}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_i : \tau_i \quad (1 \le i \le n)}{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd \{C_1, \cdots, C_n\} : \{l_1 : \tau_1, \cdots, l_n : \tau_n\}}$$

$$\text{INDEX}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_1 : \tau_1 \quad \mathcal{K} \vdash \tau_1 :: \{\!\{l : \tau_2\}\!\} \quad \mathcal{L} \vdash \mathcal{I} : idx(l,\tau_1)}{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_1[\mathcal{I}] : \tau_2}$$

$$\text{MODIFY}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_1 : \tau_1 \quad \mathcal{K} \vdash \tau_1 :: \{\!\{l : \tau_2\}\!\} \quad \mathcal{L} \vdash \mathcal{I} : idx(l,\tau_1) \quad \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_2 : \tau_2}{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd \mathsf{modify}(C_1, \mathcal{I}, C_2) : \tau_1}$$

$$\text{VARIANT}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C : \tau_1 \quad \mathcal{K} \vdash \tau_2 :: \langle\!\langle l : \tau_1 \rangle\!\rangle \quad \mathcal{L} \vdash \mathcal{I} : idx(l,\tau_2)}{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd \langle \mathcal{I} = C \rangle : \tau_2}$$

$$\text{SWITCH}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C : \langle l_1 : \tau_1, \cdots, l_n : \tau_n\rangle \quad \mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_i : \tau_i \to \tau \ (1 \le i \le n)}{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd \mathsf{switch}\ C\ \mathsf{of}\ C_1, \cdots, C_n : \tau}$$

$$\text{GEN}\quad \frac{\mathcal{K}\{t_1 :: k_1 \cdots t_n :: k_n\}, \mathcal{L}, \mathcal{T} \rhd C : \tau}{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C : \forall t_1 :: k_n \cdots t_n :: k_n . \tau} \quad \text{if } t_i \notin FTV(\mathcal{L} \cup \mathcal{T})\ (1 \le i \le n)$$

$$\text{LET}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd C_1 : \sigma \quad \mathcal{K}, \mathcal{L}, \mathcal{T}\{x : \sigma\} \rhd C_2 : \tau}{\mathcal{K}, \mathcal{L}, \mathcal{T} \rhd \mathsf{let}\ x = C_1\ \mathsf{in}\ C_2 : \tau}$$

Fig. 13. Typing rules for the implementation calculus $\lambda^{let,[]}$.

$$(\beta)\quad (\lambda x.C_1)\ C_2 \Longrightarrow [C_2/x]C_1$$

$$\text{(index)}\quad \{C_1, \cdots, C_n\}[i] \Longrightarrow C_i\ (1 \le i \le n)$$

$$\text{(modify)}\quad \mathsf{modify}(\{C_1, \cdots, C_n\}, i, C) \Longrightarrow \{C_1, \cdots, C_{i-1}, C, C_{i+1}, \cdots, C_n\} (1 \le i \le n)$$

$$\text{(switch)}\quad \mathsf{switch}\ \langle i = C\rangle\ \mathsf{of}\ C_1, \cdots, C_n \Longrightarrow C_i\ C (1 \le i \le n)$$

$$\text{(iapp)}\quad (\lambda I.C)\ \mathcal{I} \Longrightarrow [\mathcal{I}/I]C$$

$$\text{(let)}\quad \mathsf{let}\ x = C_1\ \mathsf{in}\ C_2 \Longrightarrow [C_1/x]C_2$$

Fig. 14. The reduction rules for the implementation calculus $\lambda^{let,[]}$.

are two cases to be considered. First, suppose $\mathcal{I}_1 = I$. Then we have $\tau_1 = \tau_2$ and $l = l'$, and therefore $\mathcal{K} \vdash \tau_2 :: \{\!\{l : \tau\}\!\}$ and $\mathcal{L} \vdash \mathcal{I} : idx(l, \tau_2)$. By the typing rule, $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{T} \rhd ([\mathcal{I}/I]C_1)[\mathcal{I}] : \tau$, i.e., $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{T} \rhd [\mathcal{I}/I](C_1[I]) : \tau$. Second, suppose $\mathcal{I}_1 \neq I$. Then either $|idx(l', \tau_2)|$ is defined and $\mathcal{I}_1 = |idx(l', \tau_2)|$ or $\mathcal{I}_1 \in dom(\mathcal{L})$. In either case, $\mathcal{L} \vdash \mathcal{I}_1 : idx(l', \tau_2)$ and by the typing rule, $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{T} \rhd ([\mathcal{I}/I]C_1)[\mathcal{I}_1] : \tau$. But $[\mathcal{I}/I](C_1[\mathcal{I}_1]) = ([\mathcal{I}/I]C_1)[\mathcal{I}_1]$. □

THEOREM 4.2.3. *If* $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{T}, \mathcal{L} \rhd C_1 : \sigma$ *and* $C_1 \twoheadrightarrow C_2$ *then* $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{T}, \mathcal{L} \rhd C_2 : \sigma$.

PROOF. It is sufficient to show the theorem for monotypes. The proof is similar to Theorem 2.2.5 using the above two substitution lemmas. □

4.3 Compilation Algorithm

We develop a compilation algorithm for $\lambda^{let,\bullet}$ using type information obtained by type inference. The type inference algorithm has already converted a given $\lambda^{let,\bullet}$ term into an explicitly typed term of $\Lambda^{let,\bullet}$, which contains all the type information necessary for compilation. So we present the compilation algorithm as an algorithm to compile $\Lambda^{let,\bullet}$ terms into $\lambda^{let,[]}$ terms.

As explained in the introduction, our strategy for compiling polymorphic functions containing polymorphic record operations is to insert appropriate index abstractions. Under this strategy, a polymorphic function of type σ in $\lambda^{let,\bullet}$ is compiled into a term having the type that is obtained from σ by inserting necessary index abstractions indicated by the kinded type quantifiers of σ. To establish the relationship formally between the type of a source code and the type of the compiled code, we first define the following auxiliary notions.

The *set of index types contained in* t *of kind* k, denoted by $IdxSet(t::k)$, is defined as the following set.

$$\begin{aligned} IdxSet(t::U) &= \emptyset \\ IdxSet(t::\{\!\{F\}\!\}) &= \{idx(l,t) | l \in dom(F)\} \\ IdxSet(t::\langle\!\langle F \rangle\!\rangle) &= \{idx(l,t) | l \in dom(F)\} \end{aligned}$$

This definition is extended to polytypes and kind assignments as follows:

$$\begin{aligned} IdxSet(\forall t_1::k_1 \cdots t_n::k_n.\tau) &= IdxSet(t_1::k_1) \cup \cdots \cup IdxSet(t_n::k_n) \\ IdxSet(\mathcal{K}) &= \bigcup \{IdxSet(t::k) | (t::k) \in \mathcal{K}\} \end{aligned}$$

For a given type σ of $\lambda^{let,\bullet}$, the corresponding type $(\sigma)^*$ of $\lambda^{let,[]}$ is defined as

$$(\forall t_1::k_1 \cdots t_n::k_n.\tau)^* = \forall t_1::k_1 \cdots t_n::k_n.idx(l_1, t'_1) \Rightarrow \cdots idx(l_m, t'_m) \Rightarrow \tau$$

such that $idx(l_1, t'_1), \cdots, idx(l_m, t'_m)$ is the set of index types in $IdxSet(t_1::k_1) \cup \cdots \cup IdxSet(t_n::k_n)$ ordered as: $idx(l, t_i)$ precedes $idx(l', t_j)$ iff $i < j$ or $i = j$ and $l \ll l'$. In particular, $(\tau)^* = \tau$ for any monotype τ. The following is an example.

$(\forall t_2::\{\!\{a : bool, b : int\}\!\}.\forall t_3::\{\!\{a : t_2\}\!\}.t_2 \rightarrow t_3)^* =$

$\forall t_2::\{\!\{a : bool, b : int\}\!\}.\forall t_3::\{\!\{a : t_2\}\!\}.idx(a, t_2) \Rightarrow idx(b, t_2) \Rightarrow idx(a, t_3) \Rightarrow t_2 \rightarrow t_3$

This definition is extended to type assignments as follows:

$$(\mathcal{T})^* = \{x : (\mathcal{T}(x))^* | x \in dom(\mathcal{T})\}$$

For a kind assignment $\mathcal{K}$, define the index assignment $\mathcal{L}_\mathcal{K}$ determined by $\mathcal{K}$ as $\mathcal{L}_\mathcal{K} = \{I : idx(l,t) | idx(l,t) \in IdxSet(\mathcal{K}),$ each I fresh$\}$.

The compilation algorithm is given in Figure 15 as an algorithm $\mathcal{C}$ that takes $\mathcal{L}_\mathcal{K}, (\mathcal{T})^*$ and M and computes a term of the implementation calculus. Since $\mathcal{L}_\mathcal{K}$ has the property that there is at most one $(I, idx(l,t)) \in \mathcal{L}_\mathcal{K}$ for any pair (l,t), each $\mathcal{I}$ mentioned in the algorithm is unique, and therefore $\mathcal{C}$ is a deterministic algorithm.

The compilation preserves types as shown in the following theorem.

THEOREM 4.3.1. *If* $\Lambda^{let,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M : \sigma$, *then* $\mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, M)$ *succeeds with* C *such that* $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}_\mathcal{K}, (\mathcal{T})^* \rhd C : (\sigma)^*$.

PROOF. This is provded by induction on the structure of M. Here we show the cases for variables, field selection, and generalization. Cases for variants and modify expressions can be shown similar to that of field selection. All the other cases follow easilily from the corresponding induction hypotheses.

$(x\ \tau_1 \cdots \tau_n)$: Suppose $\Lambda^{let,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd (x\ \tau_1 \cdots \tau_n) : \tau$. Let $S = [\tau_1/t_1, \cdots, \tau_n/t_n]$. Then $\mathcal{T}(x) = \forall t_1{::}k_1 \cdots \forall t_n{::}k_n.\tau_0$, $\mathcal{K} \vdash S(t_i) :: S(k_i)$ and $\tau = S(\tau_0)$. By the definition of $(\mathcal{T})^*$, $(\mathcal{T})^*(x) = \forall t_1{::}k_1 \cdots \forall t_n{::}k_n.idx(l_1, t'_1) \Rightarrow \cdots idx(l_m, t'_m) \Rightarrow \tau_0$ such that $\{idx(l_1,t'_1), \cdots, idx(l_m,t'_m)\} = IdxSet(\forall t_1{::}k_1 \cdots \forall t_n{::}k_n.\tau_0)$. By the rule VAR, $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}_\mathcal{K}, (\mathcal{T})^* \rhd x : idx(l_1, S(t'_1)) \Rightarrow \cdots idx(l_m, S(t'_m)) \Rightarrow S(\tau_0)$. Let $\mathcal{I}_i$ be those mentioned in the algorithm. For each $idx(l, S(t'_i))$, if $S(t'_i)$ is a type variable t then $t \in dom(\mathcal{K})$, and therefore by the property of $\mathcal{L}_\mathcal{K}$, there is I_i such that $(I_i : idx(l, S(t'_i))) \in \mathcal{L}_\mathcal{K}$, and $\mathcal{I}_i = I_i$. If $S(t'_i)$ is not a type variable then $\mathcal{I}_i = |idx(l_i, S(t'_i))|$. Therefore in either case $\mathcal{L}_\mathcal{K} \vdash \mathcal{I}_i : idx(l, S(t'_i))$. Therefore the algorithm succeeds with $(x\ \mathcal{I}_1 \cdots \mathcal{I}_m)$ and $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}_\mathcal{K}, (\mathcal{T})^* \rhd (x\ \mathcal{I}_1 \cdots \mathcal{I}_m) : \tau$.

$M_1 : \tau_1.l$: Suppose $\Lambda^{let,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M_1 : \tau_1.l : \tau_2$. Then $\Lambda^{let,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd M_1 : \tau_1$ and $\mathcal{K} \vdash \tau_1 :: \{\!\{l : \tau_2\}\!\}$. By the induction hypothesis, $\mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, M_1) = C_1$ such that $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}_\mathcal{K}, (\mathcal{T})^* \rhd C_1 : \tau_1$. Let $\mathcal{I}$ be the one mentioned in the algorithm. If τ_1 is a type variable t, then $t \in dom(\mathcal{K})$. Since $\mathcal{K} \vdash t :: \{\!\{l : \tau_2\}\!\}$, $\mathcal{K}(t) = \{\!\{F\}\!\}$ such that F contains $l : \tau_2$. Then by the property of $\mathcal{L}_\mathcal{K}$, there is some I such that $(I : idx(l,t)) \in \mathcal{L}_\mathcal{K}$ and $\mathcal{I} = I$. If τ_1 is not a type variable then $|idx(l, \tau_1)| = i$ for some integer i and $\mathcal{I} = i$. Therefore in either case $\mathcal{L}_\mathcal{K} \vdash \mathcal{I} : idx(l, \tau_1)$. Then $\mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, M_1 : \tau_1.l)$ succeeds with $C_1\,[\mathcal{I}]$ and $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}_\mathcal{K}, (\mathcal{T})^* \rhd C_1\,[\mathcal{I}] : \tau_2$.

$\mathsf{Poly}(M{:}\sigma)$: Suppose $\Lambda^{let,\cdot} \vdash \mathcal{K}, \mathcal{T} \rhd \mathsf{Poly}(M{:}\sigma) : \sigma$. Then $\sigma = \forall t_1{::}k_1 \cdots \forall t_n{::}k_n.\tau$ such that $\Lambda^{let,\cdot} \vdash \mathcal{K}', \mathcal{T} \rhd M : \tau$, $Cls(\mathcal{K}', \mathcal{T}, \tau) = (\mathcal{K}, \forall t_1{::}k_1 \cdots \forall t_n{::}k_n.\tau)$. Let $\forall t_1{::}k_1 \cdots \forall t_n{::}k_n.idx(l_1, t'_1) \Rightarrow \cdots \Rightarrow idx(l_m, t'_m) \Rightarrow \tau_1 = (\sigma)^*$. Then $\mathcal{L}_{\mathcal{K}'} = \mathcal{L}_\mathcal{K}\{I_1 : idx(l_1, t'_1), \cdots, I_m : idx(l_m, t'_m)\}$ ($I_1, \cdots, I_m$ fresh). By the induction hypothesis, $\mathcal{C}(\mathcal{L}_{\mathcal{K}'}, (\mathcal{T})^*, M) = C$ such that $\lambda^{let,[]} \vdash \mathcal{K}', \mathcal{L}_{\mathcal{K}'}, (\mathcal{T})^* \rhd C : \tau$. Then $\mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, \mathsf{Poly}(M{:}\sigma))$ succeeds with $\lambda I_1 \cdots \lambda I_m.C$. By applying the rule IABS to $\lambda^{let,[]} \vdash \mathcal{K}', \mathcal{L}_{\mathcal{K}'}, (\mathcal{T})^* \rhd C : \tau$ repeatedly, we have $\lambda^{let,[]} \vdash \mathcal{K}', \mathcal{L}_\mathcal{K}, \mathcal{T} \rhd \lambda I_1 \cdots \lambda I_m.C : idx(l_1, t'_1) \Rightarrow \cdots idx(l_m, t'_m) \Rightarrow \tau$. Since $\mathcal{L}_\mathcal{K}$ is well formed under $\mathcal{K}$, $t_i \notin FTV(\mathcal{L}_\mathcal{K} \cup (\mathcal{T})^*)$ $(1 \le i \le n)$. Therefore we have $\lambda^{let,[]} \vdash \mathcal{K}, \mathcal{L}_\mathcal{K}, (\mathcal{T})^* \rhd C : (\forall t_1{::}k_1 \cdots \forall t_n{::}k_n.\tau)^*$, as desired. □

Combining this result with Theorem 3.5.1, we have the following.

$$\mathcal{C}(\mathcal{L},\mathcal{T},(x\ \tau_1\cdots\tau_n)) = \text{let } (\forall t_1{::}k_1\cdots t_n{::}k_n.\imath dx(l_1,t'_1)\Rightarrow\cdots\imath dx(l_m,t'_m)\Rightarrow\tau) = \mathcal{T}(x)$$
$$S=[\tau_1/t_1,\cdots,\tau_n/t_n]$$
$$\mathcal{I}_i=\begin{cases} i & \text{if } |\imath dx(l,S(t'_i))|=i \\ I & \text{if } |\imath dx(l,S(t'_i))| \text{ is undefined and } (I:\imath dx(l,S(t'_i)))\in\mathcal{L}\end{cases}$$
$$\text{in } (x\ \mathcal{I}_1\ \cdots\ \mathcal{I}_m)$$

$$\mathcal{C}(\mathcal{L},\mathcal{T},c^b)=c^b$$

$$\mathcal{C}(\mathcal{L},\mathcal{T},\lambda x:\tau.M)=\lambda x.\mathcal{C}(\mathcal{L},\mathcal{T}\{x:\tau\},M)$$

$$\mathcal{C}(\mathcal{L},\mathcal{T},M_1\ M_2)=\mathcal{C}(\mathcal{L},\mathcal{T},M_1)\ \mathcal{C}(\mathcal{L},\mathcal{T},M_2)$$

$$\mathcal{C}(\mathcal{L},\mathcal{T},\{l_1=M_1,\cdots,l_n=M_n\})=\{\mathcal{C}(\mathcal{L},\mathcal{T},M_1),\cdots,\mathcal{C}(\mathcal{L},\mathcal{T},M_n)\}$$

$$\mathcal{C}(\mathcal{L},\mathcal{T},M:\tau.l) = \text{let } C=\mathcal{C}(\mathcal{L},\mathcal{T},M) \text{ and}$$
$$\mathcal{I}=\begin{cases} i & \text{if } |idx(l,\tau)|=i \\ I & \text{if } |idx(l,\tau)| \text{ is undefined and } (I:idx(l,\tau))\in\mathcal{L}\end{cases}$$
$$\text{in } C[\mathcal{I}]$$

$$\mathcal{C}(\mathcal{L},\mathcal{T},\mathsf{modify}(M_1{:}\tau,l,M_2)) = \text{let } C_1=\mathcal{C}(\mathcal{L},\mathcal{T},M_1),$$
$$C_2=\mathcal{C}(\mathcal{L},\mathcal{T},M_2), \text{ and}$$
$$\mathcal{I}=\begin{cases} i & \text{if } |idx(l,\tau)|=i \\ I & \text{if } |idx(l,\tau)| \text{ is undefined and } (I:idx(l,\tau))\in\mathcal{L}\end{cases}$$
$$\text{in } \mathsf{modify}(C_1,\mathcal{I},C_2)$$

$$\mathcal{C}(\mathcal{L},\mathcal{T},(\langle l{=}M\rangle{:}\tau)) = \text{let } C=\mathcal{C}(\mathcal{L},\mathcal{T},M) \text{ and}$$
$$\mathcal{I}=\begin{cases} i & \text{if } |idx(l,\tau)|=i \\ I & \text{if } |idx(l,\tau)| \text{ is undefined and } (I:idx(l,\tau))\in\mathcal{L}\end{cases}$$
$$\text{in } \langle\mathcal{I}{=}C\rangle$$

$$\mathcal{C}(\mathcal{L},\mathcal{T},\mathsf{case}\ M\ \mathsf{of}\ \langle l_1{=}M_1,\cdots,l_n{=}M_n\rangle) =$$
$$\mathsf{switch}\ \mathcal{C}(\mathcal{L},\mathcal{T},M)\ \mathsf{of}\ \mathcal{C}(\mathcal{L},\mathcal{T},M_1),\cdots,\mathcal{C}(\mathcal{L},\mathcal{T},M_n)$$

$$\mathcal{C}(\mathcal{L},\mathcal{T},\mathsf{Poly}(M_1:\forall t_1{::}k_1\cdots\forall t_n{::}k_n.\tau_1)) =$$
$$\text{let } \forall t_1{::}k_1\cdots\forall t_n{::}k_n.idx(l_1,t'_i)\Rightarrow idx(l_m,t'_m)\Rightarrow\tau_1$$
$$=(\forall t_1{::}k_1\cdots\forall t_n{::}k_n.\tau_1)^*$$
$$C_1=\mathcal{C}(\mathcal{L}\{I_1:idx(l_1,t'_1),\cdots,I_n:idx(l_m,t'_m)\},\mathcal{T},M_1)\ (I_1,\cdots,I_m \text{ fresh})$$
$$\text{in } \lambda I_1\cdots\lambda I_m.C_1$$

$$\mathcal{C}(\mathcal{L},\mathcal{T},\mathsf{let}\ x{:}\sigma{=}M_1\ \mathsf{in}\ M_2) = \text{let } C_1=\mathcal{C}(\mathcal{L},\mathcal{T},M_1)$$
$$C_2=\mathcal{C}(\mathcal{L},\mathcal{T}\{x:(\sigma)^*\},M_2)$$
$$\text{in let } x{=}C_1 \text{ in } C_2$$

Fig. 15. Compilation algorithm.

COROLLARY 4.3.2. *If* $\mathcal{WK}(\mathcal{K},\mathcal{T},e)=(\mathcal{K}',S,M,\sigma)$ *then* $\lambda^{let,\bullet}\vdash\mathcal{K}',S(\mathcal{T})\rhd e:\sigma$ *and* $\mathcal{C}(\mathcal{L}_{\mathcal{K}'},(S(\mathcal{T}))^*,M)$ *succeeds with* C *such that* $\lambda^{let,[]}\vdash\mathcal{K}',\mathcal{L}_{\mathcal{K}'},(S(\mathcal{T}))^*\rhd C:(\sigma)^*$.

The above result shows that the compilation algorithm maps a term of type σ to a term of type $(\sigma)^*$. Since $(\tau)^*=\tau$, the compilation preserves all the monotypes.

4.4 Eliminating Vacuous Type Variables from $\lambda^{let,\bullet}$ Typing

The above algorithm translates a kinded typing of $\Lambda^{let,\bullet}$ into a kinded typing of $\lambda^{let,[]}$. For this to serve as a compilation algorithm for $\lambda^{let,\bullet}$, there is one subtle point to be taken care of. This is related to the problem of coherence [Breazu-Tannen et al. 1991]. As shown in Ohori [1989], Damas-Milner system of ML is not coherent with respect to Core XML, and the same is true for the relationship between $\lambda^{let,\bullet}$ and $\Lambda^{let,\bullet}$. (See also Harper and Mitchell [1993] for a related discussion.)

The source of the failure of coherence is free type variables used in a typing derivation that do not appear in the type or the type assignment in the result typing. Those type variables also cause a problem in applying the compilation algorithm developed in the previous subsection. To see this, consider the raw term $(\lambda x.c^b)\ (\lambda x.(x.l)+1)$. The type inference algorithm produces the following typing in $\lambda^{let,\bullet}$,

$$\{t::\{\!\{l:int\}\!\}\},\emptyset\rhd(\lambda x.c^b)\ (\lambda x.(x.l)+1):b$$

corresponding to the following typing in $\Lambda^{let,\bullet}$:

$$\{t::\{\!\{l:int\}\!\}\},\emptyset\rhd(\lambda x{:}t\to int.c^b)\ (\lambda x{:}t.(x.l)+1):b$$

The kinded type variable t is introduced to typecheck polymorphic field selection $x.l$, but it does not appear in the type assignment or the result type and therefore will never be further instantiated. As a consequence, the given closed term is translated to an open term in $\Lambda^{let,\bullet}$ containing a free index variable denoting the position of l which will not be determined.

Our solution to this problem is to refine the Milner-style type inference algorithm given in Section 3 to eliminate these "redundant" or *vacuous* type variables. We say that a type variable t in $\mathcal{K},\mathcal{T}\rhd e:\tau$ is *vacuous* if $t\in dom(\mathcal{K})$ and $t\notin EFTV(\mathcal{K},\tau)\cup EFTV(\mathcal{K},\mathcal{T})$.

We assume that there is a predefined base type b_0. The choice of b_0 is unimportant. Let $\mathcal{K},\mathcal{T}\rhd e:\tau$ be a typing, and let t be a vacuous type variable of the typing such that $t\notin FTV(\mathcal{K}(t))$. Then $\mathcal{K}$ is written as $\mathcal{K}'\{t::k\}$. Define the canonical instance τ_t of t in $\mathcal{K}$ as follows:

$$\tau_t=\begin{cases}b_0 & \text{if } k=U\\ \{F\} & \text{if } k=\{\!\{F\}\!\}\\ \langle F\rangle & \text{if } k=\langle\!\langle F\rangle\!\rangle\end{cases}$$

We can eliminate t from the typing by applying kinded substitution $(\mathcal{K}',[\tau_t/t])$. If the set of vacuous type variables has no mutual cyclic dependency in $\mathcal{K}$, then it has a sequence $t_1,\cdots,t_n$ such that $t_i\notin FTV(\mathcal{K}(t_j))$ if $1\le i\le j\le n$. Then by repeating the above process for $t_1,\cdots,t_n$, we obtain a sequence of kinded substitution $(\mathcal{K}_i,[\tau_{t_i}/t_i])$. We define a *canonical instantiation* for $\mathcal{K},\mathcal{T}\rhd e:\tau$ as a kinded

substitution $(\mathcal{K}_n, [\tau_n/t_n] \circ \cdots \circ [\tau_i/t_1])$. From this definition, the following results can be easily proved.

LEMMA 4.4.1. *If $(\mathcal{K}_0, S_0)$ is a canonical instantiation of a typing $\mathcal{K}, \mathcal{T} \rhd e : \tau$, then $(\mathcal{K}_0, S_0)$ respects $\mathcal{K}$.*

By Lemma 2.2.3, we have the following.

COROLLARY 4.4.2. *If $\lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd e : \tau$ and $(\mathcal{K}_0, S_0)$ is a canonical instantiation of $\mathcal{K}, \mathcal{T} \rhd e : \tau$ then $\lambda^{let,\bullet} \vdash \mathcal{K}_0, \mathcal{T} \rhd e : \tau$.*

This shows that if the set of vacuous type variables has no cyclic dependency then we can eliminate them without affecting the typing property of the term. We call $\mathcal{K}_0, \mathcal{T} \rhd e : \tau$ the *canonical instance* of $\lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd e : \tau$.

We identify a *program* in $\lambda^{let,\bullet}$ as a *closed* typing of the form:

$$\emptyset, \emptyset \rhd e : \sigma$$

We refine the type inference algorithm defined in the previous section so that just before the type abstraction at the top level it takes a canonical instance of the inferred typing if one exists; otherwise it reports type error. Since a program must have a closed typing, and therefore its derivation does not contain a kind assignment with cyclic dependency, this process does not change the typability of programs. From a program of the above form of $\lambda^{let,\bullet}$ the refined type inference algorithm produces the following closed typing of $\Lambda^{let,\bullet}$

$$\emptyset, \emptyset, \emptyset \rhd \mathsf{Poly}(M{:}\sigma) : \sigma$$

We regard these closed typings as units of separate compilation.

With this refinement, the compilation algorithm given in the previous subsection serves as a compilation algorithm for $\lambda^{let,\bullet}$. Corollary 4.3.2 becomes the following.

COROLLARY 4.4.3. *If e is a well typed $\lambda^{let,\bullet}$ program, then $\mathcal{WK}(\emptyset, \emptyset, e)$ succeeds with $(\emptyset, S, M, \sigma)$ for some S, M, σ such that $\lambda^{let,\bullet} \vdash \emptyset, \emptyset \rhd e : \sigma$, $\Lambda^{let,\bullet} \vdash \emptyset, \emptyset \rhd M : \sigma$, and $\mathcal{C}(\emptyset, \emptyset, M)$ succeeds with C such that $\lambda^{let,[]} \vdash \emptyset, \emptyset, \emptyset \rhd C : (\sigma)^*$.*

Let us show examples of compilation. From a $\lambda^{let,\bullet}$ term λx.x.Name, the type inference process produces the following program

$$\begin{aligned}\Lambda^{let,\bullet} \vdash \emptyset, \emptyset \ \rhd\ & \mathsf{Poly}(\lambda \mathsf{x}{:}t_2.\mathsf{x}.\mathsf{Name} : \forall t_1{::}U.\forall t_2{::}\{\!\{Name : t_1\}\!\}.t_2 \to t_1) \\ & : \forall t_1{::}U.\forall t_2{::}\{\!\{Name : t_1\}\!\}.t_2 \to t_1\end{aligned}$$

For this program, the compilation algorithm produces the following result

$$\mathcal{C}(\emptyset, \emptyset, \mathsf{Poly}(\lambda \mathsf{x}{:}t_2.\mathsf{x}.\mathsf{Name} : \forall t_1{::}U.\forall t_2{::}\{\!\{Name : t_1\}\!\}.t_2 \to t_1)) = \lambda \mathsf{l}.\lambda \mathsf{x}.\mathsf{x}[\mathsf{l}]$$

which has the following typing:

$$\lambda^{let,[]} \vdash \emptyset, \emptyset, \emptyset \rhd \lambda \mathsf{l}.\lambda \mathsf{x}.\mathsf{x}\,[\mathsf{l}] \ : \ \forall t_1{::}U.\forall t_2{::}\{\!\{Name : t_1\}\!\}.idx(Name, t_2) \Rightarrow t_2 \to t_1$$

A program

```
let name=λx.x.Name in (name {Name="Joe",Office=403},
                       name {Name="Hanako",Age=21,Phone=7222})
```

is converted to the following program in $\Lambda^{let,\bullet}$ as seen in the previous section:

$$E \equiv \text{let name } \forall t_1{::}U.\forall t_2{::}\{\!\{Name : t_1\}\!\}.t_2 \to t_1$$
$$= \text{Poly}(\lambda x{\cdot}t_2\ \text{x.Name}\ .\ \forall t_1{::}U.\forall t_2{::}\{\!\{Name : t_1\}\!\}.t_2 \to t_1)$$
$$\text{in } ((\text{name } string\ \{Name : string, Office : string\})$$
$$\{\text{Name="Joe",Office=403}\},$$
$$(\text{name } string\ \{Name : string, Age\ .\ int, Phone : int\})$$
$$\{\text{Name="Hanako",Age=21,Phone=7222}\})$$

and the compilation algorithm produces the following result

$$\mathcal{C}(\emptyset,\emptyset,E) = \text{let name=}\lambda\text{l }\lambda\text{x x[l] in (name 1 \{"Joe",403\},}$$
$$\text{name 2 \{21,"Hanako",7222\})}$$

which has the expected typing and evaluates to ("Joe","Hanako").

The next is an example of a program involving polymorphic variant and vacuous type variable elimination. The following program of $\lambda^{let,\cdot}$

```
let point = ⟨Cartesian={X=2 0,Y=3 0}⟩ in
  case point of ⟨Cartesian=λc.sqroot(square(c.X) + square(c.Y)), Polar=λp.R⟩
```

is converted into the following program in $\Lambda^{let,\cdot}$.

$$F \equiv \text{let point.}\forall t{::}\langle\!\langle Cartesian : \{X : real, Y : real\}\rangle\!\rangle.t$$
$$= \text{Poly}((\langle\text{Cartesian=\{X=2.0,Y=3 0}\}\rangle\ t)$$
$$\forall t{::}\langle\!\langle Cartesian : \{X : real, Y : real\}\rangle\!\rangle.t)$$
$$\text{in case (point } \langle Cartesian : \{X : real, Y : real\}, Polar : \{R : b_0\}\rangle) \text{ of}$$
$$\langle\text{Cartesian= }\lambda\text{c }\{X : real, Y : real\}\ \text{sqroot(square(c.X)+square(c.Y)),}$$
$$\text{Polar=}\lambda\text{p }\{R : b_0\}\text{.x.R}\rangle$$

From this, the compilation algorithm produces the following code:

$$\mathcal{C}(\emptyset,\emptyset,F) = \text{let point=}\lambda\text{l.}\langle\text{l=\{2 0,3 0}\}\rangle$$
$$\text{in switch (point 1) of } \langle\lambda\text{c.sqroot(square(c[1]) + square(c[2])), }\lambda\text{x x[1]}\rangle$$

Note that vacuous type variable elimination is properly performed for Polar branch of the case statement, and the unused field extension x.R is compiled into index expression with the default index value 1.

4.5 Correctness of Compilation

In Section 4, we have shown that the compilation algorithm preserves typing. This section shows that the compilation algorithm also preserves the operational behavior of a program. Since we have shown that the type system of $\lambda^{let,\cdot}$ is sound with respect to its operational semantics, the preservation of operational behavior will also establish that the type system of $\lambda^{let,\cdot}$ is sound with respect to the operational semantics of the compiled code in $\lambda^{let,[]}$.

For terms of base types, the desired property is simply being that the original term and the compiled term evaluate to the same constant value. We need to generalize this to arbitrary types including polytypes. Our strategy is to apply the idea of logical relations to lift the above relationship to arbitrary types.

Let σ be a closed type of $\lambda^{let,\cdot}$. Let $term^\sigma$ be the set $\{e|\lambda^{let,\cdot} \vdash \emptyset,\emptyset \rhd e : \sigma\}$, and let $Term^\sigma$ be the set $\{M|\lambda^{let,[]} \vdash \emptyset,\emptyset,\emptyset \rhd M : (\sigma)^*\}$. We define a type-indexed family of relations $\{R^\sigma \subseteq term^\sigma \times Term^\sigma\}$ by induction on σ as follows.

$$(e,C) \in R^\sigma \iff (1)\ e\downarrow \text{ iff } C\downarrow \text{ and}$$
$$(2) \text{ one of the following conditions holds}$$

—if $\sigma = b$ then if $e \downarrow e'$ and $C \downarrow C'$ then $e' = C'$.

—if $\sigma = \tau_1 \to \tau_2$ then for any e_0, C_0 such that $(e_0, C_0) \in R^{\tau_1}$, $(e\ e_0, C\ C_0) \in R^{\tau_2}$.

—if $\sigma = \{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ then if $e \downarrow e'$ and $C \downarrow C'$ then $e' = \{l_1 = e_1, \cdots, l_n = e_n\}, C' = \{C_1, \cdots, C_n\}$ such that $(e_i, C_i) \in R^{\tau_i}$ for all $1 \leq i \leq n$.

—if $\sigma = \langle l_1 : \tau_1, \cdots, l_n : \tau_n \rangle$ then if $e \downarrow e'$ and $C \downarrow C'$ then there is some i such that $e' = \langle l_i = e'' \rangle, C' = \langle i = C'' \rangle$ and $(e'', C'') \in R^{\tau_i}$.

—if $\sigma = \forall t_1 :: k_1. \cdots t_n :: k_n.\tau$ such that

$$(\sigma)^* = \forall t_1 :: k_1. \cdots t_n :: k_n. idx(l_1, t_1') \Rightarrow \cdots idx(l_m, t_m') \Rightarrow \tau$$

then for any ground substitution S satisfying $\{t_1 :: k_1. \cdots t_n :: k_n\}$,

$$(e, (\cdots (C\ i_1) \cdots i_m)) \in R^{S(\tau)}$$

where $i_j = |S(idx(l_j, t_j'))|$ $(1 \leq j \leq m)$.

Note that by the type soundness theorem (Theorem 3.2.1) of $\lambda^{let,\bullet}$ and the subject reduction theorem (Theorem 4.2.3) of $\lambda^{let,[]}$, if $e \in term^\sigma$, $C \in Term^\sigma$, $e \downarrow e'$, and $C \downarrow C'$ then $e' \in term^\sigma$, $C' \in Term^\sigma$. Furthermore, by the definition of R^σ, if $(e, C) \in R^\sigma$ then $(e', C') \in R^\sigma$.

Let $\mathcal{T}$ be a closed type assignment of $\lambda^{let,\bullet}$. A $\mathcal{T}$-environment in $\lambda^{let,\bullet}$ is a function η^1 such that $dom(\eta^1) = dom(\mathcal{T})$ and for any $x \in dom(\eta^1)$, $\eta^1(x) \in term^{\mathcal{T}(x)}$. A $(\mathcal{T})^*$-environment in $\lambda^{let,[]}$ is a function η^2 such that $dom(\eta^2) = dom(\mathcal{T})$ and for any $x \in dom(\eta^2)$, $\eta^2(x) \in Term^{(\mathcal{T})^*(x)}$. Let $\mathcal{L}$ be a well-formed, closed, index assignment. A $(\mathcal{T})^*$-environment η^2 in $\lambda^{let,[]}$ is uniquely extended to the function defined on $dom(\mathcal{T}) \cup dom(\mathcal{L})$ by setting its value of I to be $|\mathcal{L}(I)|$ for all $I \in dom(\mathcal{L})$. We write $\eta^2_{\mathcal{L}}$ for the extension of η^2 to $dom(\mathcal{L})$.

The relation R is extended to environments. $R^{\mathcal{T}}$ is the relation between $\mathcal{T}$-environments in $\lambda^{let,\bullet}$ and $(\mathcal{T})^*$-environments in $\lambda^{let,[]}$ such that $(\eta^1, \eta^2) \in R^{\mathcal{T}}$ iff for any $x \in dom(\mathcal{T})$, $(\eta^1(x), \eta^2(x)) \in R^{\mathcal{T}(x)}$.

We now have the following theorem, whose proof is deferred to the Appendix.

THEOREM 4.5.1. *Let $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd M : \sigma$ be any typing. If $\mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, M) = C$ then for any ground substitution S that respects $\mathcal{K}$, and for any pair of environments $(\eta^1, \eta^2) \in R^{S(\mathcal{T})}$, $(\eta^1(erase(M)), \eta^2_{S(\mathcal{L}_{\mathcal{K}})}(C)) \in R^{S(\sigma)}$.*

For a program, we have the following.

COROLLARY 4.5.2. *If e is a well-typed $\lambda^{let,\bullet}$ program, then $\mathcal{WK}(\emptyset, \emptyset, e)$ succeeds with $(\emptyset, S, M, \sigma)$ for some S, M, σ and $\mathcal{C}(\emptyset, \emptyset, M)$ succeeds with C for some C such that $(e, C) \in R^\sigma$.*

If we define the set of *observable types* by the following syntax

$$\omega ::= b \mid \{l : \omega, \cdots, l : \omega\} \mid \langle l : \omega, \cdots, l : \omega \rangle$$

then the relation R^ω is essentially the identity (modulo representation of records and variants), and therefore a program of an observable type in $\lambda^{let,\bullet}$ and its compiled term evaluates to the essentially same value.

5. IMPLEMENTATION

Using the polymorphic typing and the compilation method presented in this article we have extended Standard ML with polymorphic record operations, and we have implemented its compiler, called SML#. SML# is an extension of the Standard ML of New Jersey compiler [Appel and MacQueen 1991] on which it is based. The extended language deals with the features of Standard ML including pattern matching, weak type variables, and explicit type declarations, which can be freely mixed with polymorphic record operations. Moreover, it preserves the efficiency of the original compiler and extends it to polymorphic manipulation of records. SML# produces the same code for monomorphic record operations. For polymorphic manipulation of records, it is necessary to perform extra function application to pass necessary index values. But they occur only when polymorphic record functions are instantiated. So we believe that the overhead due to index applications is in most cases negligible. Indeed, a simple iteration of the example function *name* does not show meaningful difference in execution speed compared to the corresponding monomorphic function written in Standard ML of New Jersey. (1,000,000 calls of `#Name` in SML# take 20.74 seconds while the same number of calls of `(#Name:{Name:string, Age:int} -> string)` in Standard ML of New Jersey takes 20.50 seconds.) This implementation substantiates this article's claim that the method presented here provides a theoretically sound and practical basis for extending ML with record polymorphism.

SML# does not include polymorphic variants. The current definition of Standard ML couples *monomorphic* variants (defined through `datatype` declarations) with other language features including user-defined recursive types, pattern matching, and constructor binding. As a consequence, the introduction of polymorphic variants requires either substantial language changes or the introduction of a new class of syntactic objects whose role largely overlaps with ML's datatypes. To include them, it is therefore essential to redesign Standard ML. Another limitation of the current version of SML# is that it does not evaluate the inside of index abstraction, and therefore it does not necessary preserve the order of evaluation. In the current version of SML#, the author adopted the strategy to implement index abstraction using the ordinary closure creation mechanism of the New Jersey system. To implement completely the operational semantics defined in this article, it is necessary to develop a new evaluation scheme for index abstraction and index application. We shall discuss possible strategies to overcome these limitations in the conclusions.

The rest of this section outlines the extension of Standard ML and the implementation of the SML# compiler.

5.1 Extension to Standard ML

Standard ML's type expressions are extended with kind constraints. This is done by introducing the following syntax for type variables having a record kind:

$$\texttt{'a\#}\{l_1 : ty_1, \cdots, l_n : ty_n, \ldots\}$$
$$\texttt{''b\#}\{l_1 : ty'_1, \cdots, l_n : ty'_n, \ldots\}$$

which represent a type variable a and an equality type variable b under the kind assignment of the form $\{a::\{\!\{l_1 : ty_1, \cdots l_n : ty_n\}\!\}, \cdots\}$ and $\{b::\{\!\{l_1 : ty'_1, \cdots l_n : ty'_n\}\!\}, \cdots\}$, respectively, where $ty'_1, \cdots, ty'_n$ are restricted to equality types. In SML#

both syntax can appear wherever `'a` or `''b` are allowed.

Standard ML already contains records and field selection. The syntax for record formation is identical to the one used in this article. In addition, ML provides the following pattern for record operations:

$$pat ::= \cdots \mid \{l = pat, \cdots, l = pat\} \mid \{l = pat, \cdots, l = pat, ...\}$$

where the last "..." is a part of ML syntax for "flexible records" in ML parlance and should not be confused with the metanotation we have used. There is also the following syntax

#*l* for field selection function λx.x.l

as explained in the introduction. In SML$^\sharp$, these patterns and expressions are freely used without type specifications, as shown in the following example:

```
- fun FirstName {Name=x,...} = #First x;
val FirstName = fn :  'c#{Name:'b#{First:'a,...},...} -> 'a
```

which shows an interactive session of SML$^\sharp$ where the user's input is prompted by "-" and where the system's output is printed in the format "`val` *x* `=` *value* `:` *type*."

In addition, SML$^\sharp$ introduces the following term constructor:

`#>` *e* `=>` $\{l_1{=}e_1, \cdots, l_n{=}e_n\}$

for nested field modification $\mathsf{modify}(\cdots(\mathsf{modify}(e,l_1,e_1)\cdots),l_n,e_n)$ whose syntax is chosen to make it compatible with the rest of the language definition.

Figure 16 shows programming examples. The second half of the example demonstrates the usefulness of record polymorphism for data-intensive applications such as database programming. In particular, the last function demonstrates that an SQL-style database query language can be cleanly integrated into a polymorphic record calculus. (The interested reader is referred to Buneman and Ohori [1995] for more discussion on polymorphism and type inference in database programming.)

5.2 Implementation Strategies

The implementation has been done by modifying the Standard ML of New Jersey compiler (version 0.75). The main modification consists of (1) the replacement of the type inference module with a new one which incorporates the kinded unification and the compilation algorithm and (2) a refinement of pattern match compilation and value binding.

The new type inference module closely follows the algorithm presented in this article with additional refinements for Standard ML's equality types and weak polymorphism. To integrate these features, the actual kind of a type variable in SML$^\sharp$ consists of the product of record/variant kind, equality flag, and weakness measure. With this refinement, our record polymorphism can be freely mixed with these features.

Some work was needed to refine the compilation for pattern matching and value binding. Consider, for example, the following simple binding:

```
val (x,y) = (#A,#B)
```

Since `x` and `y` will be used independently, the compiler should produce the binding of the following types:

```
(* simple examples *)
- fun moveX point = #> point  => {x = #x point + 1};
val moveX = fn : 'a#{x:int,...} -> 'a#{x:int,...}
- moveX {x=1,y=2};
val it = {x=2,y=2} : {x:int,y:int}
- moveX {x=1, y=2, z=3, color="Green"};
val it = {color="Green",x=2,y=2,z=3} : {color:string,x:int,y:int,z:int}

(* database like examples *)
- fun wealthy {Salary =s,...} = s > 100000;
val wealthy = fn : 'a#{Salary:int,...} -> bool
- fun young x = #Age x < 24;
val young = fn : 'a#{Age:int,...} -> bool
- fun youngAndWealthy x = wealthy x andalso young x;
val youngAndWealthy = fn : 'a#{Age:int,Salary:int,...} -> bool
- fun select display l pred  =
      fold (fn (x,y) => if pred x then (display x)::y else y) l nil;
val select = fn : ('a -> 'b) -> 'a list -> ('a -> bool) -> 'b list
- fun youngAndWealthyEmployees l = select #Name l youngAndWealthy;
val youngAndWealthyEmployees = fn
     : 'b#{Age:int,Name:'a,Salary:int,...} list -> 'a list
```

Fig. 16. Interactive programming session in SML$^\sharp$

```
x : 'a#{A:'b} -> 'b
y : 'a#{B:'b} -> 'b
```

Therefore, index abstraction insertion and polymorphic generalization must be done for x and y separately according to the corresponding portion of the code. To achieve this effect, the SML$^\sharp$ compiler transforms ML's value binding of the form

val *pat* = *expr*

where *pat* is a pattern containing variables $\{x_1, \cdots, x_n\}$ into the following form

$$\texttt{val } (x_1,\cdots,x_n) \texttt{ = let } f \texttt{ = } expr \texttt{ in } ((\texttt{fn } pat \texttt{ => } x_1)\ f,\ \vdots\ (\texttt{fn } pat \texttt{ => } x_n)\ f)$$

The inner binding is the ordinary let binding and is transformed by the method described in this article. The compiler then transforms the outer val binding by eliminating vacuous type variables separately from the typing of each variable x_i, performing index abstraction and polymorphic type generalization for each component corresponding to x_i separately, and finally translating to a binding of the implementation calculus. In the actual implementation, all of these steps are done in one step using the type information of the original term. For example, the above example is transformed into the following binding

```
val (x,y) = let F = λI1λI2.(λx.x[I1], λy.y[I2])
            in (λI3.(F I3 1),λI4.(F 1 I4))
```

where 1 is the default index value introduced in the process of vacuous type variable elimination explained earlier. A similar treatment is necessary for mutually

recursive function definitions of the form:

$$\begin{array}{l}\texttt{fun } f_1\ pat_1\ \texttt{=}\ expr_1\\ \qquad\qquad\vdots\\ \texttt{and } f_n\ pat_n\ \texttt{=}\ expr_n\end{array}$$

This is transformed into the following code

$$\begin{array}{l}\texttt{val } (f_1,\cdots,f_n)\ \texttt{=}\ \texttt{let } F\ \texttt{=}\ \texttt{rec } f_1\ pat_1\ \texttt{=}\ expr_1\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\vdots\\ \qquad\qquad\qquad\qquad\qquad\qquad\texttt{and } f_n\ pat_n\ \texttt{=}\ expr_n\\ \qquad\qquad\qquad\texttt{in } (F\texttt{.1},\cdots,F\texttt{.n})\end{array}$$

where `rec` f pat `=` $expr$ $\cdots$ `and` pat `=` $expr$ is the construct for mutually recursive functions without polymorphic generalization. The binding is then transformed into a term in the implementation calculus in a similar way as in the case of the complex value binding explained above. By this treatment, SML$^\sharp$ allows record polymorphism to be freely mixed with arbitrary complex value binding and mutually recursive function definitions.

A prototype SML$^\sharp$ system is available from Kyoto University. The interested reader should copy the README file from `ftp.kurims.kyoto-u.ac.jp` at the directory `pub/paper/member/ohori` by anonymous FTP or consult the web page `http://www.kurims.kyoto-u.ac.jp/~ohori/smlsharp.html`.

6. CONCLUSIONS

We have given a polymorphic type discipline for records and variants as an extension of the Girard-Reynolds second-order lambda calculus and have defined an ML-style polymorphic record calculus that corresponds to a predicative subcalculus of the second-order system. The type system of the ML-style record calculus is shown to be sound with respect to its operational semantics. For this calculus, we have given a type inference algorithm and proved its soundness and completeness with respect to its polymorphic type system. We have then developed an efficient compilation method for the ML-style polymorphic record calculus. The compilation method translates any type-correct term in the polymorphic record calculus into a term in a calculus where (1) records are represented as directly indexable vectors and (2) variants are represented as values tagged with a natural number that is used as the index to the vector of functions in a switch statement. The correctness of the compilation algorithm is shown by applying the idea of logical relation to set up a desired relation between the operational behavior of the polymorphic record calculus and that of the implementation calculus. Based on these results, Standard ML has been extended with polymorphic record operations, and a full-scale prototype compiler has been implemented.

There are a number of further issues to be considered. Here we only briefly mention some of them.

A More Complete Implementation. As mentioned in the previous section, there are two major limitations of the current SML$^\sharp$ implementation: it lacks polymorphic variants and is not faithful to the operational semantics defined in this article because it does not evaluate inside of index abstraction.

Since the basic techniques of implementing polymorphic variants are the same as those required for polymorphic records, there is no technical difficulty in introducing them. In fact, pure variants can be encoded using records by regarding a variant type $\langle l_1 : \tau_1, \cdots, l_n : \tau_n \rangle$ as $\forall t::U.\{l_1 : \tau_1 \to t, \cdots, l_n : \tau_n \to t\} \to t$ and by encoding variant constructors and case statements analogous to the standard encoding of booleans and conditional statements. Therefore they can be implemented by the implementation mechanism for polymorphic records. However, a labeled variant type in Standard ML is coupled with user-defined recursive types. This coupling works well when labeled variants are restricted to be monomorphic, but is incompatible with polymorphic manipulation of labeled variants since the possibility of polymorphic manipulation is eliminated. To incorporate polymorphic variants into Standard ML, we need to redesign the language to separate the mechanism of labeled variants from the user-defined recursive types. We are currently considering a proper syntax and the corresponding formal definition of the Standard ML with polymorphic labeled variants.

It is more challenging to refine SML$^\sharp$ to make it faithful to the operational semantics defined in this article. Although we have not yet developed a complete implementation technique, we believe that it is possible to develop a reasonably efficient system that evaluates inside of index abstractions by treating index abstraction and index application specially. Let us briefly explain a possible strategy.

From the above observation on implementation of polymorphic variants, we can restrict our attention to polymorphic record operations. The compilation algorithm has the property that if it produces an implementation term of the form $\lambda I.C$ then I in C occurs inside of an ordinary lambda abstraction, and therefore evaluation of C does not involve evaluation of I. The evaluation of $\lambda I.C$ can then be done as follows. When evaluating $\lambda I.C$, the system first allocates a dummy entry for I in the environment and evaluates C. This should yield a closure whose environment contains I. This closure is saved as a template. The application of the function $\lambda I.C$ to an actual index value i can be implemented by making a copy of the template closure and updating the I entry in the environment to i. It is still needed to develop a method to propagate this technique for nested index abstraction of the form $\lambda I.\cdots(\lambda I'.C)\ I\cdots$. The author intends to develop a systematic method to achieve the desired semantics, which will also be useful for various type-inference-based program specialization discussed below.

Type-Inference-Based Program Specialization. The compilation method presented in this article can be characterized as specialization of polymorphic functions using type information. In this respect, our work shares the same motivation as the paradigm of "intentional type analysis" [Harper and Morrisett 1995]. Their framework is based on runtime type analysis and is therefore more general. For example, certain features of overloading can comfortably be represented in their framework. It is an interesting issue whether or not our method can be combined with their framework. There are also several recent papers on the efficient implementation of polymorphic languages using explicit type information. Examples include a polymorphic unboxed calculus [Ohori and Takamizawa 1995] which specialize polymorphic functions according to the size information obtained by type inference, mixed representation optimization [Leroy 1992] by inserting appropriate box-unbox co-

ercions guided by type information, tag-free garbage collection [Tolmach 1994] by translating a raw term to an explicitly typed second-order term, specialization of Haskell type classes [Wadler and Blott 1989] using type information [Hall et al. 1994; Peterson and Jones 1993], and list representation optimization [Hall 1994; Shao et al. 1994]. Since program translations in all of these approaches and ours appear to share some general structure, a detailed comparison among them may shed some light on the general property of a type-inference-based approach to program specialization.

Our compilation method may also be applied to optimizing the "soft type system" of Cartwright and Fagan [1991] and Wright and Cartwright [1994]. In the soft type system, types are implicitly treated as elements of a variant type. By checking type tag at runtime, this approach allows more-flexible typing than the conventional static typing. Our compilation method for polymorphic variants might be used to reduce the cost of a runtime check.

Compilation of a Calculus with Subtyping. We would like to extend our compilation technique to a polymorphic type discipline with subtyping, which is another important paradigm for flexible treatment of labeled records and labeled variants. Its flexibility is based on the following subsumption rule:

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

There seems to be an inherent difficulty in compiling a calculus containing this rule. To see the difficulty, consider the expression:

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

where e_1 is some boolean expression. With the existence of the subsumption rule, this expression has the type $\{B : bool\}$. However, the actual set of labels of the value denoted by this expression depends on the value denoted by e_1, and therefore the offset of the label B cannot be statically determined. It is therefore impossible to determine statically the necessary index value in a program such as $(\lambda x.\, x.B)\ e$. With the subsumption rule, a typing judgment of the form $\emptyset \rhd e : \{l_1 : \tau_1, \cdots, l_n : \tau_n\}$ no longer implies that e denotes a record value having the exact type $\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$. A similar problem arises in compiling a calculus that allows heterogeneous collections [Buneman and Ohori 1995]. This observation suggests that compilation should incorporate some degree of dynamic type testing at runtime. One approach might be to combine our method and the intensional type analysis of Harper and Morrisett [1995].

Application to Other Labeled Data Structures. Our flexible typing and compilation method for record polymorphism may also be applicable to several other systems where labels play an important role.

In Common Lisp [Steele 1984], parameters to a function may be labeled. It is straightforward to model simple labeled parameters using labeled records. However, Common Lisp also allows *optional arguments* with *default values.* This feature cannot easily be modeled in a simple type discipline. One approach to represent these features in a record calculus is to extend it with an "optional-field selection" operation $(e.l\ ?\ d)$ which behaves like $e.l$ if e is a record containing an l field;

otherwise it evaluates to the default value d. Our polymorphic type discipline and the compilation method can be extended to support this construct. Recently, Garrigue and Aït-Kaci [1994] developed a lambda calculus that supports a more-flexible labeled parameter passing mechanism where labeled currying and labeled partial application are allowed. Furuse and Garrigue [1995] extended it with optional arguments and developed a compilation method. It is an interesting further research to investigate the possibility of combining their approach and ours.

Our approach may also be applicable to object-oriented programming. A calculus with record subtyping is often considered as a formal model for object-oriented programming, where a "class" is represented as a record type and where an "object" is represented as a record containing methods; method invocation is done by field selection. In this model, the feature of "method inheritance" is achieved through flexible typing of field selection. Since kinded typing allows similar flexible typing of field selection, we believe that our second-order record calculus $\Lambda^{\forall,\cdot}$, when extended with recursive types and other features, can serve as an appropriate basis for type systems for object-oriented programming. Since kinded typing does not have complicated interaction with function type and since there is an efficient compilation algorithm for the calculus, we further believe that our record calculus, when properly extended, can be a simpler and practical alternative to a calculus with subtyping.

APPENDIX

PROOFS OF MAJOR THEOREMS

THEOREM 3.4.1. *The algorithm $\mathcal{U}$ takes any kinded set of equations and computes its most general unifier if one exists and reports failure otherwise.*

PROOF. We first show that if the algorithm returns a kinded substitution then it is a most general unifier of a given kinded set of equations.

It is easily verified that each transformation rule preserves the following property on 4-tuple $(E, \mathcal{K}, S, \mathcal{SK})$.

(1) $\mathcal{K}$ and $\mathcal{K} \cup \mathcal{SK}$ are well-formed kind assignments; E is well-formed under $\mathcal{K}$; S is a well-formed substitution under $\mathcal{K}$; $dom(\mathcal{K}) \cap dom(\mathcal{SK}) = \emptyset$; and $dom(\mathcal{SK}) = dom(S)$.

We establish that if the above property holds for the 4-tuple then each transformation rule also preserves the following properties on 4-tuples.

(2) For any kinded substitution $(\mathcal{K}_0, S_0)$, if $(\mathcal{K}_0, S_0)$ respect $\mathcal{K}$ and S_0 satisfies $E \cup S$ then $(\mathcal{K}_0, S_0)$ respect $\mathcal{SK}$.

(3) The set of unifiers of $(\mathcal{K} \cup \mathcal{SK}, E \cup S)$.

The case for Rule (I) is trivial, and those for (V) and (IX) follow from the assumptions. Since the rules (VI) – (VIII) have the same shape as the rules (II) – (IV) respectively, their proofs are the same as the corresponding proofs. Below, we show Properties 2 and 3 for the rules (II) – (IV).

Rule (II).

$$(E \cup \{(t,\tau)\},\ \mathcal{K} \cup \{(t,U)\},\ S,\ \mathcal{SK}) \Longrightarrow$$
$$([\tau/t](E),\ [\tau/t](\mathcal{K}),\ [\tau/t](S) \cup \{(t,\tau)\},\ [\tau/t](\mathcal{SK}) \cup \{(t,U)\}) \quad \text{if } t \notin FTV(\tau)$$

Property 2. Let S_0 satisfy $[\tau/t](E) \cup [\tau/t](S) \cup \{(t,\tau)\}$ and $(\mathcal{K}_0, S_0)$ respect $[\tau/t](\mathcal{K})$. Then $S_0 = S_0 \circ [\tau/t]$, and S_0 satisfies $E \cup \{(\tau,t)\} \cup S$. Since τ is well formed under $\mathcal{K}$, $S_0(\tau)$ is well formed under $\mathcal{K}_0$, and $\mathcal{K}_0 \vdash S_0(\tau) :: U$. Therefore $(\mathcal{K}_0, S_0)$ respects $\mathcal{K} \cup \{(t,U)\}$. Then by Property 2 of the premise of the rule, $(\mathcal{K}_0, S_0)$ respects $\mathcal{SK}$, and therefore $(\mathcal{K}_0, S_0)$ respects $[\tau/t](\mathcal{SK}) \cup \{(t,U)\}$.

Property 3. Let S_0 be any substitution. S_0 satisfies $E \cup \{(t,\tau)\} \cup S$ iff S_0 satisfies $[\tau/t](E) \cup [\tau/t](S) \cup \{(t,\tau)\}$. Let $\mathcal{K}_0$ be any kind assignment such that $(\mathcal{K}_0, S_0)$ is a kinded substitution. Suppose S_0 satisfies $E \cup \{(t,\tau)\} \cup S$. Then since $S_0 = S_0 \circ [\tau/t]$ and $t \notin dom(\mathcal{K})$, $(\mathcal{K}_0, S_0)$ respects $\mathcal{K}$ iff $(\mathcal{K}_0, S_0)$ respects $[\tau/t](\mathcal{K})$. Similarly, $(\mathcal{K}_0, S_0)$ respects $\mathcal{SK}$ iff $(\mathcal{K}_0, S_0)$ respects $[\tau/t](\mathcal{SK})$. Therefore $(\mathcal{K}_0, S_0)$ is a unifier of $(\mathcal{K} \cup \{(t,U)\} \cup \mathcal{SK},\ E \cup \{(t,\tau)\} \cup S)$ iff it is a unifier of $([\tau/t](\mathcal{K}) \cup [\tau/t](\mathcal{SK}) \cup \{(t,U)\},\ [\tau/t](E) \cup [\tau/t](S) \cup \{(t,\tau)\})$.

Rule (III).

$$
\begin{array}{l}
(E \cup \{(t_1,t_2)\},\ \mathcal{K} \cup \{(t_1,\{\!\{F_1\}\!\}),(t_2,\{\!\{F_2\}\!\})\},\ S,\ \mathcal{SK}) \Longrightarrow \\
([t_2/t_1](E \cup \{(F_1(l),F_2(l)) | l \in dom(F_1) \cap dom(F_2)\}), \\
\ [t_2/t_1](\mathcal{K}) \cup \{(t_2,[t_2/t_1](\{\!\{F_1 \pm F_2\}\!\}))\}, \\
\ [t_2/t_1](S) \cup \{(t_1,t_2)\},\ [t_2/t_1](\mathcal{SK}) \cup \{(t_1,\{\!\{F_1\}\!\})\})
\end{array}
$$

Property 2. Let S_0 satisfy $[t_2/t_1](E \cup \{(F_1(l),F_2(l)) | l \in dom(F_1) \cap dom(F_2)\})) \cup [t_2/t_1](S) \cup \{(t_1,t_2)\}$ and $(\mathcal{K}_0, S_0)$ respect $[t_2/t_1](\mathcal{K}) \cup \{(t_2,[t_2/t_1](\{\!\{F_1 \pm F_2\}\!\}))\}$. Then $S_0 = S_0 \circ [t_2/t_1]$; S_0 satisfies $E \cup \{(t_1,t_2)\} \cup S$; and $(\mathcal{K}_0, S_0)$ respects $\mathcal{K} \cup \{(t_1,\{\!\{F_1\}\!\}),(t_2,\{\!\{F_2\}\!\})\}$. Then by Property 2 of the premise of the rule, $(\mathcal{K}_0, S_0)$ respects $\mathcal{SK}$. Therefore $(\mathcal{K}_0, S_0)$ respects $[t_2/t_1](\mathcal{SK}) \cup \{(t_1,\{\!\{F_1\}\!\})\}$.

Property 3. Let S_0 be any substitution satisfying $\{(t_1,t_2)\}$. Then S_0 satisfies $E \cup S$ iff S_0 satisfies $[t_2/t_1](E) \cup [t_2/t_1](S)$. Let $\mathcal{K}_0$ be any kind assignment such that $(\mathcal{K}_0, S_0)$ is a kinded substitution. Since $S_0 = S_0 \circ [t_2/t_1]$ and $t_1 \notin dom(\mathcal{K})$, $(\mathcal{K}_0, S_0)$ respects $\mathcal{K}$ iff $(\mathcal{K}_0, S_0)$ respects $[t_2/t_1](\mathcal{K})$. Similarly, $(\mathcal{K}_0, S_0)$ respects $\mathcal{SK}$ iff $(\mathcal{K}_0, S_0)$ respects $[t_2/t_1](\mathcal{SK})$. We also have the following:

$$
\begin{array}{rl}
 & (\ S_0 \text{ satisfies } \{(F_1(l),F_2(l)) | l \in dom(F_1) \cap dom(F_2)\} \text{ and} \\
 & \quad \mathcal{K}_0 \vdash S_0(t_1) :: S_0(\{\!\{F_1\}\!\}) \text{ and } \mathcal{K}_0 \vdash S_0(t_2) :: S_0([t_2/t_1](\{\!\{F_1 \pm F_2\}\!\}))\ ) \\
\text{iff} & (\ \mathcal{K}_0 \vdash S_0(t_1) :: S_0(\{\!\{F_1\}\!\}) \text{ and } \mathcal{K}_0 \vdash S_0(t_2) :: S_0(\{\!\{F_2\}\!\})\ ).
\end{array}
$$

This proves Property 3.

Rule (IV).

$$
\begin{array}{l}
(E \cup \{(t_1,\{F_2\})\},\ \mathcal{K} \cup \{(t_1,\{\!\{F_1\}\!\})\},\ S,\ \mathcal{SK}) \Longrightarrow \\
([\{F_2\}/t_1](E \cup \{(F_1(l),F_2(l)) | l \in dom(F_1)\}),\ [\{F_2\}/t_1](\mathcal{K}), \\
\ [\{F_2\}/t_1](S) \cup \{(t_1,\{F_2\})\},\ [\{F_2\}/t_1](\mathcal{SK}) \cup \{(t_1,\{\!\{F_1\}\!\})\}) \\
\text{if } dom(F_1) \subseteq dom(F_2) \text{ and } t \notin FTV(\{F_2\})
\end{array}
$$

Property 2. Let $(\mathcal{K}_0, S_0)$ respect $[\{F_2\}/t_1](\mathcal{K})$ and S_0 satisfy $[\{F_2\}/t_1](E \cup \{(F_1(l),F_2(l)) | l \in dom(F_1)\}) \cup [\{F_2\}/t_1](S) \cup \{(t_1,\{F_2\})\}$. Then $S_0 = S_0 \circ [\{F_2\}/t_1]$, S_0 satisfies $E \cup \{(t_1,\{F_2\})\} \cup S$, and $(\mathcal{K}_0, S_0)$ respects $\mathcal{K}$. Since $dom(F_1) \subseteq dom(F_2)$ and S_0 satisfies $\{(F_1(l),F_2(l)) | l \in dom(F_1)\}$, $\mathcal{K}_0 \vdash S_0(t_1) :: S_0(\{\!\{F_1\}\!\})$. Then $(\mathcal{K}_0, S_0)$ respects $\mathcal{K} \cup \{(t_1,\{\!\{F_1\}\!\})\}$. By Property 2 of the premise of the rule,

$(\mathcal{K}_0, S_0)$ respects $\mathcal{SK}$, and therefore $(\mathcal{K}_0, S_0)$ respects $[\{F_2\}/t_1](\mathcal{SK})$. This proves Property 2.

Property 3. Let S_0 be any substitution satisfying $\{(t_1, \{F_2\})\}$. Then S_0 satisfies $E \cup S$ iff S_0 satisfies $[\{F_2\}/t_1](E) \cup [\{F_2\}/t_1](S)$. Let $\mathcal{K}_0$ be any kind assignment such that $(\mathcal{K}_0, S_0)$ is a kinded substitution. Since $S_0 = S_0 \circ [\{F_2\}/t]$ and $t \notin dom(\mathcal{K})$, $(\mathcal{K}_0, S_0)$ respects $\mathcal{K}$ iff $(\mathcal{K}_0, S_0)$ respects $[\{F_2\}/t](\mathcal{K})$. Similarly, $(\mathcal{K}_0, S_0)$ respects $\mathcal{SK}$ iff $(\mathcal{K}_0, S_0)$ respects $[\{F_2\}/t](\mathcal{SK})$. This proves Property 3.

We now conclude the proof of the correctness of the algorithm. Let $(\mathcal{K}, E)$ be a given kinded set of equations.

Suppose the algorithm terminates with $(\mathcal{K}', S)$. Then there is some $\mathcal{SK}$ such that $(E, \mathcal{K}, \emptyset, \emptyset)$ is transformed to $(\emptyset, \mathcal{K}', S, \mathcal{SK})$ by repeated applications of transformation rules. Property 1 trivially holds for $(E, \mathcal{K}, \emptyset, \emptyset)$. Then $(\mathcal{K}', S)$ is a kinded substitution; $dom(S) \cap dom(\mathcal{K}') = \emptyset$, and therefore $(\mathcal{K}', S)$ respects $\mathcal{K}'$. S also trivially satisfies $S \cup \emptyset$. Then by Property 2, $(\mathcal{K}', S)$ also respects $\mathcal{SK}$. Therefore $(\mathcal{K}', S)$ is a unifier of $(\mathcal{K}' \cup \mathcal{SK}, \emptyset \cup S)$. By Property 3, it is also a unifier of $(\mathcal{K}, E)$. Let $(\mathcal{K}_0, S_0)$ be any unifier of $(\mathcal{K}, E)$. By Property 3, it is also a unifier of $(\mathcal{K}' \cup \mathcal{SK}, \emptyset \cup S)$. But $(\mathcal{K}', S)$ is also a unifier of $(\mathcal{K}' \cup \mathcal{SK}, \emptyset \cup S)$, and $S_0 = S_0 \circ S$. Thus $(\mathcal{K}', S)$ is more general than $(\mathcal{K}_0, S_0)$.

Conversely, suppose the algorithm fails. Then $(E, \mathcal{K}, \emptyset, \emptyset)$ is transformed to $(E', \mathcal{K}', S', \mathcal{SK}')$ for some $E', \mathcal{K}', S, \mathcal{SK}$ such that $E' \neq \emptyset$, and no rule applies to $(E', \mathcal{K}', S', \mathcal{SK}')$. It is clear from the definition of each rule that $(\mathcal{K}' \cup \mathcal{SK}', E' \cup S')$ has no unifier, and therefore by Property 3, $(\mathcal{K}, E)$ has no unifier.

The termination can be proved by showing that each transformation rule decreases the complexity measure of the lexicographical pair consisting of the size of the set $dom(\mathcal{K})$ and the total number of occurrences of type constructors (including base types) in E. □

THEOREM 3.5.1. *If* $\mathcal{WK}(\mathcal{K}, \mathcal{T}, e) = (\mathcal{K}', S, M, \tau)$ *then the following properties hold:*

(1) $(\mathcal{K}', S)$ *respects* $\mathcal{K}$ *and* $\lambda^{let,\bullet} \vdash \mathcal{K}', S(\mathcal{T}) \rhd e : \tau$,

(2) $erase(M) = e$ *and* $\Lambda^{let,\bullet} \vdash \mathcal{K}', S(\mathcal{T}) \rhd M : \tau$,

(3) *if* $\lambda^{let,\bullet} \vdash \mathcal{K}_0, S_0(\mathcal{T}) \rhd e : \tau_0$ *for some* $(\mathcal{K}_0, S_0)$ *and* τ_0 *such that* $(\mathcal{K}_0, S_0)$ *respects* $\mathcal{K}$ *then there is some* S' *such that* $(\mathcal{K}_0, S')$ *respects* $\mathcal{K}'$, $\tau_0 = S'(\tau)$, *and* $S_0(\mathcal{T}) = S' \circ S(\mathcal{T})$.

If $\mathcal{WK}(\mathcal{K}, \mathcal{T}, e) = failure$ *then there is no* $(\mathcal{K}_0, S_0)$ *and* τ_0 *such that* $(\mathcal{K}_0, S_0)$ *respects* $\mathcal{K}$ *and* $\lambda^{let,\bullet} \vdash \mathcal{K}_0, S_0(\mathcal{T}) \rhd e : \tau_0$.

PROOF. Property 2 of the first statement follows directly from the proof of Property 1 and the relationship between the two type systems. It is also a routine matter to show that if the algorithm fails for some term then there is no typing for that term. In what follows, we show Properties 1 and 3 of the first statement. Proof is by induction on the structure of e. Here we only show the cases for x, $e_1.l$, and let $x{=}e_1$ in e_2. The cases for modify(e_1,l,e_2) and $\langle l{=}e_1 \rangle$ are similar to the case for $e_1.l$. Other cases are essentially the same as the corresponding proof for ML [Damas and Milner 1982].

Case x. Suppose $\mathcal{WK}(\mathcal{K},\mathcal{T},x)=(\mathcal{K}',S,M,\tau)$. Then $\mathcal{T}(x)=\forall t_1::k_1\cdots t_n::k_n.\tau'$ and $\mathcal{K}'=\mathcal{K}\{s_1::k_1',\cdots,s_n::k_n'\}, S=\emptyset, M=(x\ s_1\cdots s_n), \tau=[s_1/t_1,\cdots,s_n/t_n](\tau')$ where $s_1,\cdots,s_n$ are fresh and $k_i'=[s_1/t_1,\cdots,s_n/t_n](k_i)$ $(1\le i\le n)$.

Property 1. $(\mathcal{K}',\emptyset)$ trivially respects $\mathcal{K}$. Since $(\mathcal{K}',[s_1/t_1,\cdots,s_n/t_n])$ respects $\mathcal{K}'\{t_1::k_1,\cdots,t_n::k_n\}$, $\mathcal{K}'\vdash\forall t_1::k_1\cdots t_n::k_n.\tau'\ge\tau$. Then $\mathcal{K}',\mathcal{T}\rhd x:\tau$.

Property 3. Suppose $(\mathcal{K}_0,S_0)$ respects $\mathcal{K}$ and $\mathcal{K}_0,S_0(\mathcal{T})\rhd x:\tau_0$. By the bound type variable convention and the assumption being that $s_1,\cdots,s_n$ are fresh, we can assume that none of $t_1,\cdots,t_n,s_1,\cdots,s_n$ appears in $dom(S_0)$ or $\{S_0(t)|t\in dom(S_0)\}$. Then $(S_0(\mathcal{T}))(x)=\forall t_1::S_0(k_1)\cdots\forall t_n::S_0(k_n).S_0(\tau')$, and there is some S_1 such that $dom(S_1)=\{t_1,\cdots,t_n\}$, $S_1(S_0(\tau'))=\tau_0$, $\mathcal{K}_0\vdash S_1(t_i)::S_1(S_0(k_i))$ $(1\le i\le n)$. Let $S_2=S_1\circ S_0\circ[t_1/s_1,\cdots,t_n/s_n]$. Then $(\mathcal{K}_0,S_2)$ respects $\mathcal{K}$, and $S_2(s_i)=S_1(t_i)$ and $S_2(k_i')=S_1(S_0(k_i))$. Therefore $\mathcal{K}_0\vdash S_2(s_i)::S_2(k_i')$. Thus $(\mathcal{K}_0,S_2)$ respects $\mathcal{K}\{s_1::k_1',\cdots,s_n::k_n'\}$. Also, we have $\tau_0=S_1(S_0(\tau'))=S_2(S(\tau))$, and $S_0(\mathcal{T})=S_2(S(\mathcal{T}))$. This proves Property 3.

Case $e_1.l$. Suppose the algorithm succeeds for $e_1.l$. Then we must have the following. $\mathcal{WK}(\mathcal{K},\mathcal{T},e_1)=(\mathcal{K}_1,S_1,M_1,\tau_1)$, $\mathcal{U}(\mathcal{K}_1\{t_1::U,t_2::\{\!\{l:t_1\}\!\}\},\{(t_2,\tau_1)\})=(\mathcal{K}_2,S_2)$ $(t_1,t_2$ fresh), and $\mathcal{WK}(\mathcal{K},\mathcal{T},e_1.l)=(\mathcal{K}_2,S_2\circ S_1,S_2(M_1).l,S_2(t_1))$.

Property 1. By Theorem 3.4.1, $(\mathcal{K}_2,S_2)$ respects $\mathcal{K}_1$ and $\mathcal{K}_2\vdash S_2(\tau_1)::\{\!\{l:S_2(t_1)\}\!\}$. By the induction hypothesis, $(\mathcal{K}_1,S_1)$ respects $\mathcal{K}$ and $\mathcal{K}_1,S_1(\mathcal{T})\rhd e_1:\tau_1$. By Lemma 2.1.1, $(\mathcal{K}_2,S_2\circ S_1)$ respects $\mathcal{K}$. By Lemma 2.2.3, $\mathcal{K}_2,S_2\circ S_1(\mathcal{T})\rhd e_1:S_2(\tau_1)$. By the rule DOT, we have $\mathcal{K}_2,S_2\circ S_1(\mathcal{T})\rhd e_1.l:S_2(t_1)$.

Property 3. Suppose $(\mathcal{K}_0,S_0)$ respects $\mathcal{K}$ and $\mathcal{K}_0,S_0(\mathcal{T})\rhd e_1.l:\tau_0^1$. Then we must have $\mathcal{K}_0,S_0(\mathcal{T})\rhd e_1:\tau_0^2$ and $\mathcal{K}_0\vdash\tau_0^2::\{\!\{l:\tau_0^1\}\!\}$. By the induction hypothesis, there is some S_0^1 such that $(\mathcal{K}_0,S_0^1)$ respects $\mathcal{K}_1$; $\tau_0^2=S_0^1(\tau_1)$; and $S_0(\mathcal{T})=S_0^1\circ S_1(\mathcal{T})$. Then $\mathcal{K}_0\vdash S_0^1(\tau_1)::\{\!\{l:\tau_0^1\}\!\}$. Consider $S_0^2=[\tau_0^1/t_1,\tau_0^2/t_2]\circ S_0^1$. Since t_1,t_2 are fresh, we have $\mathcal{K}_0\vdash S_0^2(t_2)::\{\!\{l:S_0^2(t_1)\}\!\}$ and $\mathcal{K}_0\vdash S_0^2(t)::S_0^2(\mathcal{K}_1(t))$ for $t\in dom(\mathcal{K}_1)$. So $(\mathcal{K}_0,S_0^2)$ respects $\mathcal{K}_1\{t_1::U,t_2::\{\!\{l:t_1\}\!\}\}$. Also $S_0^2(t_2)=S_0^2(\tau_1)$. So $(\mathcal{K}_0,S_0^2)$ is a unifier of $(\mathcal{K}_1\{t_1::U,t_2::\{\!\{l:t_1\}\!\}\},\{(t_2,\tau_1)\})$. Therefore by Theorem 3.4.1, there is some S_0^3 such that $(\mathcal{K}_0,S_0^3)$ respects $\mathcal{K}_2$ and $S_0^2=S_0^3\circ S_2$. Then we have $\tau_0^1=S_0^3\circ S_2(t_1)$, $S_0(\mathcal{T})=S_0^1\circ S_1(\mathcal{T})=S_0^2\circ S_1(\mathcal{T})=S_0^3\circ S_2\circ S_1(\mathcal{T})$ as desired.

Case $\langle l=e_1\rangle$. Suppose the algorithm succeeds for $\langle l=e_1\rangle$. Then we must have: $\mathcal{WK}(\mathcal{K},\mathcal{T},e_1)=(\mathcal{K}_1,S_1,M_1,\tau_1)$ and $\mathcal{WK}(\mathcal{K},\mathcal{T},\langle l=e_1\rangle)=(\mathcal{K}_1\{t::\langle\!\langle l:\tau_1\rangle\!\rangle\},S_1,M_2,t)$ where t fresh.

Property 1. By induction hypothesis, $\mathcal{K}_1,S_1(\mathcal{T})\rhd e_1:\tau_1$. Since $\mathcal{K}_1\{t::\langle\!\langle l:\tau_1\rangle\!\rangle\}\vdash t::\langle\!\langle l:\tau_1\rangle\!\rangle$, by the typing rule $\mathcal{K}_1\{t::\langle\!\langle l:\tau_1\rangle\!\rangle\},S_1(\mathcal{T})\rhd\langle l=e_1\rangle:t$.

Property 3. Suppose $(\mathcal{K}_0,S_0)$ respects $\mathcal{K}$ and $\mathcal{K},S_0(\mathcal{T})\rhd\langle l=e_1\rangle:\tau_0$. Then by the typing rules, $\mathcal{K}_0,S_0(\mathcal{T})\rhd e_1:\tau_0^1$ and $\mathcal{K}_0\vdash\tau_0::\langle\!\langle l:\tau_0^1\rangle\!\rangle$. By induction hypothesis, there is some S_0^1 such that $(\mathcal{K}_0,S_0^1)$ respects $\mathcal{K}_1$; $S_0^1(\tau_1)=\tau_0^1$; and $S_0^1(S_1(\mathcal{T}))=S_0(\mathcal{T})$. Let $S_0^2=[\tau_0/t]\circ S_0^1$. Then since t is fresh, $\mathcal{K}_0\vdash S_0^2(t)::\langle\!\langle l:S_0^2(\tau_1)\rangle\!\rangle$. Therefore $(\mathcal{K}_0,S_0^2)$ respects $\mathcal{K}_1\{t::\langle\!\langle l:\tau_1\rangle\!\rangle\}$, $S_0^2(S_1(\mathcal{T}))=S_0^1(S_1(\mathcal{T}))=S_0(\mathcal{T})$. $S_0^2(t)=\tau_0$, as desired.

Case let x=e_1 in e_2. Suppose $\mathcal{WK}(\mathcal{K},\mathcal{T},\text{let } x=e_1 \text{ in } e_2)=(\mathcal{K}',S,M,\tau)$. Then we must have: $\mathcal{WK}(\mathcal{K},\mathcal{T},e_1)=(\mathcal{K}_1,S_1,M_1,\tau_1)$, $Cls(\mathcal{K}_1,S_1(\mathcal{T}),\tau_1)=(\mathcal{K}_1',\sigma_1)$,

$\mathcal{WK}(\mathcal{K}'_1, (S_1(\mathcal{T}))\{x : \sigma_1\}, e_2) = (\mathcal{K}_2, S_2, M_2, \tau_2)$, and $\mathcal{K}' = \mathcal{K}_2, S = S_2 \circ S_1, M =$ let $x{:}S_2(\sigma_1) = \mathsf{Poly}(S_2(M_1 : \sigma_1))$ in $M_2, \tau = \tau_2$.

Property 1. By the induction hypothesis, $(\mathcal{K}_1, S_1)$ respects $\mathcal{K}$ and $\mathcal{K}_1, S_1(\mathcal{T}) \rhd e_1 : \tau_1$. By the rule GEN, $\mathcal{K}'_1, S_1(\mathcal{T}) \rhd e_1 : \sigma_1$. By the induction hypothesis on e_2, $(\mathcal{K}_2, S_2)$ respects $\mathcal{K}'_1$, and $\mathcal{K}_2, S_2 \circ S_1(\mathcal{T})\{x : S_2(\sigma_1)\} \rhd e_2 : \tau_2$. We show that for any $t \in dom(\mathcal{K})$, $S(t)$ is well formed under $\mathcal{K}'_1$. By the definition of the type inference algorithm and the unification algorithm, if $t \notin EFTV(\mathcal{K}, \mathcal{T})$, then t does not appear in τ or S, and therefore $FTV(S(t)) \subseteq dom(\mathcal{K}'_1)$. Suppose $t \in EFTV(\mathcal{K}, \mathcal{T})$. By a simple induction on the derivation of $t \in EFTV(\mathcal{K}, \mathcal{T})$, it is shown that $FTV(S(t)) \subseteq EFTV(\mathcal{K}_1, S(\mathcal{T}))$, and therefore $FTV(S(t)) \subseteq dom(\mathcal{K}'_1)$. Thus in either case $S_1(t)$ is well formed under $\mathcal{K}'_1$. Then $(\mathcal{K}'_1, S_1)$ respects $\mathcal{K}$, and by Lemma 2.1.1, $(\mathcal{K}_2, S_2 \circ S_1)$ respects $\mathcal{K}$. By Lemma 2.2.3, $\mathcal{K}_2, S_2 \circ S_1(\mathcal{T}) \rhd e_1 : S_2(\sigma_1)$. Then by the typing rule LET, $\mathcal{K}_2, S_2 \circ S_1(\mathcal{T}) \rhd$ let $x{=}e_1$ in $e_2 : \tau_2$

Property 3. Suppose $(\mathcal{K}_0, S_0)$ respects $\mathcal{K}$, and $\mathcal{K}_0, S_0(\mathcal{T}) \rhd$ let $x{=}e_1$ in $e_2 : \tau_0$. Then we must have: $\mathcal{K}'_0, S_0(\mathcal{T}) \rhd e_1 : \tau_0^1$, $(\mathcal{K}_0, \sigma_0^1) = Cls(\mathcal{K}'_0, S_0(\mathcal{T}), \tau_0^1)$ and $\mathcal{K}_0, (S_0(\mathcal{T}))\{x : \sigma_0^1\} \rhd e_2 : \tau_0$. By the definition of Cls, we can write $\mathcal{K}'_0 = \mathcal{K}_0\{t_1^0{::}k_1^0, \cdots, t_m^0{::}k_m^0\}$ such that $\{t_1^0, \cdots, t_m^0\} = EFTV(\mathcal{K}'_0, \tau_0^1) \backslash EFTV(\mathcal{K}'_0, S_0(\mathcal{T}))$, and $\sigma_0^1 = \forall t_1^0{::}k_1^0 \cdots t_m^0{::}k_m^0.\tau_0^1$. By the induction hypothesis, there is some S_0^1 such that $(\mathcal{K}'_0, S_0^1)$ respects $\mathcal{K}_1$; $\tau_0^1 = S_0^1(\tau_1)$; and $S_0(\mathcal{T}) = S_0^1 \circ S_1(\mathcal{T})$. By the definition of Cls, $\mathcal{K}_1$ and σ_1 can also be written as: $\mathcal{K}_1 = \mathcal{K}'_1\{t_1{::}k_1, \cdots, t_n{::}k_n\}$ such that $\{t_1, \cdots, t_n\} = EFTV(\mathcal{K}_1, \tau_1) \setminus EFTV(\mathcal{K}_1, S_1(\mathcal{T}))$, and $\sigma_1 = \forall t_1{::}k_1 \cdots t_n{::}k_n.\tau_1$. By the bound type variable convention, $\{t_1, \cdots, t_n\} \cap \{t_1^0, \cdots, t_m^0\} = \emptyset$. Since $(\mathcal{K}'_0, S_0^1)$ respects $\mathcal{K}_1$, $\mathcal{K}_0\{t_1^0{::}k_1^0, \cdots, t_m^0{::}k_m^0\} \vdash S_0^1(t_i) :: S_0^1(k_i)$ $(1 \leq i \leq n)$. Let S_0^2 be the restriction of S_0^1 on $dom(S_0^1) \setminus \{t_1, \cdots, t_n\}$, and let S_0^3 be the substitution $[S_0^1(t_1)/t_1, \cdots, S_0^1(t_n)/t_n]$. We show that, for each $1 \leq i \leq n$, $S_0^2(k_i)$ is well formed under $\mathcal{K}_0\{t_1{::}S_0^2(k_1), \cdots, t_{i-1}{::}S_0^2(k_{i-1})\}$. Since $S_0^1(k_i)$ is well formed under $\mathcal{K}'_0$, it is enough to show that $FTV(S_0^2(k_i)) \cap \{t_1^0, \cdots, t_m^0\} = \emptyset$. Suppose $t \in FTV(S_0^2(k_i))$. Then there is some t' such that $t \in FTV(S_0^2(t'))$, $t' \in FTV(k_i)$. Since $t_i \in EFTV(\mathcal{K}_1, \tau_1)$, by the definition of $EFTV$, $t' \in EFTV(\mathcal{K}_1, \tau_1)$. By our assumption on $\mathcal{K}_1$, for any $j \geq i$, $t_j \notin FTV(k_i)$. Therefore either $t \in \{t_1, \cdots, t_{i-1}\}$, or $t \in S_0^2(EFTV(\mathcal{K}_1, S_1(\mathcal{T})))$. But it is shown by induction on the construction of $EFTV(\mathcal{K}_1, S_1(\mathcal{T}))$ that $S_0^2(EFTV(\mathcal{K}_1, S_1(\mathcal{T}))) \subseteq EFTV(\mathcal{K}'_0, S_0(\mathcal{T}))$. Thus $t \notin \{t_1^0, \cdots, t_m^0\}$, and $\mathcal{K}_0\{t_1{::}S_0^2(k_1), \cdots, t_n{::}S_0^2(k_n)\}$ is well formed. By similar argument, it is shown that $S_0^2(\tau_1)$ is well formed under $\mathcal{K}_0\{t_1{::}S_0^2(k_1), \cdots, t_n{::}S_0^2(k_n)\}$. Then $(\mathcal{K}_0\{t_1^0{::}k_1^0, \cdots, t_m^0{::}k_m^0\}, S_0^3)$ respects $\mathcal{K}_0\{t_1{::}S_0^2(k_1), \cdots, t_n{::}S_0^2(k_n)\}$, and $\tau_0^1 = S_0^3(S_0^2(\tau_1))$. Thus $\mathcal{K}_0 \vdash \forall t_1{::}S_0^2(k_1) \cdots \forall t_n{::}S_0^2(k_n).S_0^2(\tau_1) \geq \forall t_1^0{::}k_1^0 \cdots \forall t_m^0{::}k_m^0.\tau_0^1$. Then by Lemma 3.1.2, $\mathcal{K}_0, (S_0(\mathcal{T}))\{x : \forall t_1{::}S_0^2(k_1) \cdots \forall t_n{::}S_0^2(k_n).S_0^2(\tau_1)\}) \rhd e_2 : \tau_0$. Since $S_0^2(\sigma_1) = \forall t_1{::}S_0^2(k_1) \cdots \forall t_n{::}S_0^2(k_n).S_0^2(\tau_1)$ and $S_0(\mathcal{T}) = S_0^2(S_1(\mathcal{T}))$, $\mathcal{K}_0, S_0^2((S_1(\mathcal{T}))\{x : \sigma_1\}) \rhd e_2 : \tau_0$. Since $(\mathcal{K}_0, S_0^1)$ respects $\mathcal{K}_1$, $(\mathcal{K}_0, S_0^2)$ respects $\mathcal{K}'_1$. We can then apply the induction hypothesis to e_2 and conclude that there is some S_0^4 such that $(\mathcal{K}_0, S_0^4)$ respects $\mathcal{K}_2$ and $\tau_0 = S_0^4(\tau_2)$ and $S_0^4(S_2(S_1(\mathcal{T}))) = S_0^2(S_1(\mathcal{T})) = S_0^1(S_1(\mathcal{T})) = S_0(\mathcal{T})$. This proves Property 3 for the case of the let expression. □

THEOREM 4.5.1. *Let $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd M : \sigma$ be any typing. If $\mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, M) = C$ then for any ground substitution S that respects $\mathcal{K}$, and for any pair of environments $(\eta^1, \eta^2) \in R^{S(\mathcal{T})}$, $(\eta^1(erase(M)), \eta^2_{S(\mathcal{L}_\mathcal{K})}(C)) \in R^{S(\tau)}$.*

PROOF. By induction on the structure of M.

Case $(x\ \tau_1 \cdots \tau_n)$. Suppose $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd (x\ \tau_1 \cdots \tau_n) : \tau$. Then by the type system, there are some $k_1, \cdots, k_n, \tau_0$ such that $\mathcal{T}(x) = \forall t_1{::}k_1 \cdots t_n{::}k_n.\tau_0$, $\tau = [\tau_1/t_1, \cdots, \tau_n/t_n](\tau_0)$, and $(\mathcal{K}, [\tau_1/t_1, \cdots, \tau_n/t_n])$ respects $\mathcal{K}\{t_1{::}k_1, \cdots, t_n{::}k_n\}$. Let $\forall t_1{::}k_1 \cdots t_n{::}k_n.idx(l_1, t'_1) \Rightarrow \cdots idx(l_m, t'_m) \Rightarrow \tau_0 = (\mathcal{T})^*(x)$. There are $\mathcal{I}_j (1 \leq j \leq n)$ such that $\mathcal{L}_\mathcal{K} \vdash \mathcal{I}_j : idx(l_j, [\tau_1/t_1, \cdots, \tau_n/t_n](t'_j))$. Now suppose $\mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, (x\ \tau_1 \cdots \tau_n)) = M$. Then $M = x\ \mathcal{I}_1 \cdots \mathcal{I}_m$. By the assumption on η_1, η_2, $(\eta^1(x), \eta^2(x)) \in R^{\forall t_1::S(k_1)\cdots\forall t_n::S(k_n).S(\tau_0)}$. By the definition of $\mathcal{I}$, $\eta^2_{S(\mathcal{L}_\mathcal{K})}(\mathcal{I}_j) = |idx(l_j, [S(\tau_1)/t_1, \cdots, S(\tau_n)/t_n](t'_j)))|$. Since S respects $\mathcal{K}$ and $(\mathcal{K}, [\tau_1/t_1, \cdots, \tau_n/t_n])$ respects $\mathcal{K}\{t_1{::}k_1, \cdots, t_n{::}k_n\}$, $[S(\tau_1)/t_1, \cdots, S(\tau_n)/t_n]$ is a ground substitution respecting $\{t_1{::}S(k_1), \cdots, t_n{::}S(k_n)\}$. Then by the definition of R, $(\eta^1(x), (\eta^2_{S(\mathcal{L}_\mathcal{K})}(x\ \mathcal{I}_1 \cdots \mathcal{I}_m))) \in R^{S(\tau)}$, as desired.

Case $\lambda x : \tau_1.M$. Suppose $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd \lambda x : \tau_1.M_1 : \tau_1 \to \tau_2$. Then $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T}\{x : \tau_1\} \rhd M_1 : \tau_2$. Suppose $\mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, \lambda x : \tau_1.M_1) = C$. Then $C = \lambda x.C_1$ such that $C_1 = \mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T}\{x : \tau_1\})^*, M_1)$. Let $e_1 = erase(M_1)$. By the bound variable convention and the definition of evaluation contexts, $\eta^1(\lambda x.e_1) = \lambda x.\eta^1(e_1) \downarrow \lambda x.\eta_1(e_1)$, and $\eta^2_{S(\mathcal{L}_\mathcal{K})}(\lambda x.C_1) = \lambda x.\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1) \downarrow \lambda x.\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1)$. Let $(e_0, C_0) \in R^{S(\tau_1)}$ be any pair of related elements. Then $e_0 \downarrow$ iff $C_0 \downarrow$. Suppose $e_0 \downarrow e'_0$ and $C_0 \downarrow C'_0$. By the type soundness theorem (Theorem 3.2.1) of $\lambda^{let,\bullet}$ and the subject reduction theorem (Theorem 4.2.3) of $\lambda^{let,[]}$, e'_0, C'_0 are also terms of the same types of e_0, C_0 respectively, and therefore $(e'_0, C'_0) \in R^{S(\tau_1)}$. By the definition of evaluation contexts, $((\lambda x.\eta^1(e_1))\ e_0) \downarrow e'$ iff $[e'_0/x](\eta^1(e_1)) \downarrow e'$, and $((\lambda x.\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1))\ C_0) \downarrow C'$ iff $[C'_0/x](\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1)) \downarrow C'$. But $[e'_0/x]\eta^1(e_1) = \eta^1\{x \mapsto e'_0\}(e_1)$, and $[C'_0/x]\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1) = \eta^2\{x \mapsto C'_0\}_{S(\mathcal{L}_\mathcal{K})}(C_1)$. Since $(e'_0, C'_0) \in R^{S(\tau_1)}$, $(\eta^1\{x \mapsto e'_0\}, \eta^2\{x \mapsto C'_0\}) \in R^{S(\mathcal{T}\{x:\tau_1\})}$. By the induction hypothesis, $(((\lambda x.\eta^1(e_1))\ e_0), ((\lambda x.\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1))\ C_0)) \in R^{S(\tau_2)}$. Thus $(\lambda x.e_1), C) \in R^{S(\tau_1)\to S(\tau_2)}$.

Case $M_1\ M_2$. Suppose $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd M_1\ M_2 : \tau_1$. Then $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd M_1 : \tau_2 \to \tau_1$ and $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd M_2 : \tau_2$ for some τ_2. Suppose $\mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, M) = C$. Then $C = (C_1\ C_2)$ such that $C_1 = \mathcal{C}(\mathcal{L}_\mathcal{K}, \mathcal{T}, M_1)$ and $C_1 = \mathcal{C}(\mathcal{L}_\mathcal{K}, \mathcal{T}, M_2)$. By the induction hypotheses for M_1 and M_2, $(\eta^1(erase(M_1)), \eta^2_{S(\mathcal{L})}(C_1)) \in R^{S(\tau_2\to\tau_1)}$ and $(\eta^1(erase(M_2)), \eta^2_{S(\mathcal{L})}(C_2)) \in R^{S(\tau_2)}$. Then by the definition of the relations R, $(\eta^1(erase(M_1\ M_2)), \eta^2_{S(\tau_1)}(C_1\ C_2)) \in R^{S(\tau_1)}$.

Case $\{l_1{=}M_1, \cdots, l_n{=}M_n\}$. Suppose $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd \{l_1 = M_1, \cdots, l_n = M_n\} : \{l_1 : \tau_1, \cdots, l_n : \tau_n\}$. Then $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd M_i : \tau_i\ (1 \leq i \leq n)$. Suppose $\mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, \{l_1{=}M_1, \cdots, l_n{=}M_n\}) = C$. Then $C = \{C_1, \cdots, C_n\}$ such that $C_i = \mathcal{C}(\mathcal{K}, \mathcal{T} \rhd M_i : \tau_i)\ (1 \leq i \leq n)$. Let $e_i = erase(M_i)\ (1 \leq i \leq n)$. By the definitions of the reduction systems, $\eta^1(\{l_1 = e_1, \cdots, l_n = e_n\}) \downarrow$ iff $\eta^1(e_i) \downarrow$ for all i, and $\eta^2_{S(\mathcal{L})}(\{C_1, \cdots, C_n\}) \downarrow$ iff $\eta^2_{S(\mathcal{L})}(C_i) \downarrow$ for all i. Furthermore, if $\eta^1(\{l_1 = e_1, \cdots, l_n = e_n\}) \downarrow e'$ then $e' = \{l_1 = e'_1, \cdots, l_n = e'_n\}$ such that $\eta^1(e_i) \downarrow e'_i$ for $(1 \leq i \leq n)$. Similarly for $\eta^2_{S(\mathcal{L})}(\{C_1, \cdots, C_n\})$. By the induction hypotheses, $(\eta^1(e_i), \eta^2_{S(\mathcal{L})}(C_i)) \in R^{S(\tau_i)}$ for each i. Thus we have $(\eta^1(\{l_1 = e_1, \cdots, l_n = e_n\}), \eta^2_{S(\mathcal{L})}(\{C_1, \cdots, C_n\})) \in R^{S(\{l_1:\tau_1, \cdots, l_n:\tau_n\})}$.

Case $M_1{:}\tau_1.l$. Suppose $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd M_1 : \tau_1.l : \tau_2$. Then $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd M_1 : \tau_1$, $\mathcal{K} \vdash \tau_1 :: \{\!\{l : \tau_2\}\!\}$. Suppose $\mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, M_1{:}\tau_1.l) = C$. Then $C = C_1[\mathcal{I}]$ such that $C_1 = \mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, M_1)$ and $\mathcal{L}_\mathcal{K} \vdash \mathcal{I} : idx(l, \tau_1)$. Let $e_1 = erase(M_1)$. By the induction hypothesis, $(\eta^1(e_1), \eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1)) \in R^{S(\tau_1)}$. Since $\eta^1(e_1) \downarrow$ iff $\eta^1(e_1.l) \downarrow$, and $\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1) \downarrow$ iff $\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1[\mathcal{I}]) \downarrow$, we have $\eta^1(e) \downarrow$ iff $\eta^2_{S(\mathcal{L}_\mathcal{K})}(C) \downarrow$. Suppose $\eta^1(e_1) \downarrow e'_1$ and $\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1) \downarrow C'_1$. Since $\emptyset \vdash S(\tau_1) :: \{\!\{l : S(\tau_2)\}\!\}$, $S(\tau_1)$ is a ground record type of the form $\{\cdots, l : S(\tau_2), \cdots\}$. Then by the definition of R, $e'_1 = \{\cdots, l = e', \cdots\}$, $C'_1 = \{\cdots, C', \cdots\}$ such that C' is at the index $|idx(l, S(\tau_1))|$ and $(e', C') \in R^{S(\tau_2)}$. This proves $(\eta^1(e_1.l), \eta^2_{S(\mathcal{L}_\mathcal{K})}(C)) \in R^{S(\tau_2)}$.

Case $\mathsf{Poly}(M_1{:}\sigma)$. Let $\forall t_1{::}k_1 \cdots t_n{::}k_n.\tau_1 = \sigma$, and $\forall t_1{::}k_1 \cdots t_n{::}k_n.idx(l_1, t'_1) \Rightarrow \cdots idx(l_m, t'_m) \Rightarrow \tau_1 = (\sigma)^*$. Suppose $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd \mathsf{Poly}(M_1{:}\sigma) : \sigma$. Then $\Lambda^{let,\bullet} \vdash \mathcal{K}\{t_1{::}k_1 \cdots t_n{::}k_n\}, \mathcal{T} \rhd M_1 : \tau_1$. Suppose $\mathcal{C}(\mathcal{L}_\mathcal{K}, (\mathcal{T})^*, \mathsf{Poly}(M{\cdot}\sigma)) = C$. Then $C = \lambda I_1 \cdots \lambda I_m.C_1$ such that $C_1 = \mathcal{C}(\mathcal{L}_{\mathcal{K}\{I_1:idx(l_1,t'_1),\cdots,I_m:idx(l_m,t'_m)\}}, (\mathcal{T})^*, M_1)$. Let $e = erase(M_1)$. Let S' be any ground substitution such that $dom(S') = \{t_1, \cdots, t_n\}$ and such that it respects $\{t_1{::}S(k_1) \cdots t_n{::}S(k_n)\}$. Then $S' \circ S$ respects $\mathcal{K}\{t_1{::}k_1, \cdots, t_n{::}k_n\}$ and $(\eta_1, \eta_2) \in R^{S' \circ S(\mathcal{T})}$. By the induction hypothesis,

$$(\eta^1(e), \eta^2_{S' \circ S(\mathcal{L}_{\mathcal{K}\{t_1::k_1, \cdots, t_n::k_n\}})}(C_1)) \in R^{S' \circ S(\tau_1)}.$$

But $\mathcal{L}_{\mathcal{K}\{t_1::k_1,\cdots,t_n::k_n\}} = \mathcal{L}_\mathcal{K}\{I_1 : idx(l_1, t'_1), \cdots, I_m : idx(l_m, t'_m)\}$, and therefore $\eta^2_{S' \circ S(\mathcal{L}_{\mathcal{K}\{t_1::k_1, \cdots, t_n::k_n\}})} = \eta^2_{S(\mathcal{L}_\mathcal{K})\{I_1:idx(l_1,S'(t'_1)),\cdots,I_m:idx(l_m,S'(t'_m))\}}$. Let $i_j = |idx(l_j, S'(t'_j))|$ $(1 \leq j \leq m)$. Then we have

$$\eta^2_{S' \circ S(\mathcal{L}_{\mathcal{K}\{t_1::k_1, \cdots, t_n::k_n\}})}(C_1) = [i_1/I_1, \cdots, i_m/I_m](\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1)).$$

By the definition of evaluation, $\lambda I_1 \cdots I_m.\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1) \downarrow$ iff $\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1) \downarrow$. Since each t'_i is a type variable, by the type system, I_i in $\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1)$ does not appear in any subterm of the forms $\{C^1, \cdots, C^n\}[I_i]$, $\mathsf{modify}(\{C^1, \cdots, C^n\}, I_i, C^0)$, $\mathsf{switch}\ \langle I_i{=}C^0\rangle$ of $C^1, \cdots, C^n$. Now if $\lambda I_1 \cdots I_m.\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1) \downarrow \lambda I_1 \cdots I_m.C'_1$, then the same property holds for I_i and C'_i, otherwise the subject reduction property of $\lambda^{let,[]}$ (Theorem 4.2.3) would have been violated. Thus:

$$\begin{aligned} & \lambda I_1 \cdots I_m.\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1) \downarrow \lambda I_1 \cdots I_m.C'_1 \\ \text{iff}\ & ((\lambda I_1 \cdots \lambda I_m.\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1))\ \imath_1 \cdots \imath_m) \downarrow [\imath_1/I_1, \cdots, i_m/I_m](C'_1) \\ \text{iff}\ & [i_1/I_1, \cdots, i_m/I_m](\eta^2_{S(\mathcal{L}_\mathcal{K})}(C_1)) \downarrow [i_1/I_1, \cdots, i_m/I_m](C'_1) \end{aligned}$$

Then the induction hypothesis and the definition of the relation R implies that $(\eta^1(e), \eta^2_{S(\mathcal{L})}(\lambda I_1 \cdots \lambda I_m.C_1)) \in R^{S(\forall t_1::k_1 \cdots t_n::k_n.\tau_1)}$.

Case $\mathsf{let}\ x{:}\sigma = M_1\ \mathsf{in}\ M_2$. Suppose $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd \mathsf{let}\ x : \sigma = M_1\ \mathsf{in}\ M_2 : \tau$. Then $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T}\{x : \sigma\} \rhd M_2 : \tau$, $\Lambda^{let,\bullet} \vdash \mathcal{K}, \mathcal{T} \rhd M_1 : \tau$, $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ such that $e_1 = erase(M_1)$ and $e_2 = erase(M_2)$, $C = \mathsf{let}\ x{=}C_1\ \mathsf{in}\ C_2$ such that $C_1 = \mathcal{C}(\mathcal{L}_\mathcal{K}, \mathcal{T}, M_1)$, $C_2 = \mathcal{C}(\mathcal{L}_\mathcal{K}, \mathcal{T}\{x : \sigma\}, M_2)$. The desired result follows by the induction hypotheses.

The cases for $\mathsf{modify}(M_1, l, M_2)$ and $(\langle l{=}M\rangle{\cdot}\sigma)$ are similar to that of $M : \tau.l$. □

ACKNOWLEDGMENTS

The author would like to thank Yasuhiko Minamide for careful reading of a draft of this article and providing many useful comments, and Jacques Garrigue for helpful discussions. The author also thanks the anonymous referees for helpful comments for improving the presentation of the article.

REFERENCES

APPEL, A. W. AND MACQUEEN, D. B. 1991. Standard ML of New Jersey. In *Proceedings of the 3rd International Symposium on Programming Languages and Logic Programming*. Lecture Notes in Computer Science, vol. 528. Springer-Verlag, Berlin, 1–13.

BARENDREGT, H. 1984. *The Lambda Calculus: Its Syntax and Semantics*, rev. ed. Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland, Amsterdam.

BREAZU-TANNEN, V. AND COQUAND, C. 1988. Extensional models for polymorphism. *Inf. Comput. 59*, 85–114.

BREAZU-TANNEN, V., COQUAND, T., GUNTER, C., AND SCEDROV, A. 1991. Inheritance as explicit coercion. *Inf. Comput. 93*, 172–221.

BRUCE, K. B., MEYER, A. R., AND MITCHELL, J. C. 1990. The semantics of second-order lambda calculus. *Inf. Comput. 85*, 76–134.

BUNEMAN, P., JUNG, A., AND OHORI, A. 1991. Using powerdomains to generalize relational databases. *Theor. Comput. Sci. 91*, 1, 23–56.

BUNEMAN, P. AND OHORI, A. 1995. Polymorphism and type inference in database programming. *ACM Trans. Database Syst. to appear.*

CARDELLI, L 1988. A semantics of multiple inheritance. *Inf. Comput. 76*, 138–164.

CARDELLI, L. 1994. Extensible records in a pure calculus of subtyping. In *Theoretical Aspects of Object-Oriented Programming*, C. GUNTER AND J. MITCHELL, Eds. MIT Press, Cambridge, Mass., 373–426.

CARDELLI, L. AND MITCHELL, J. 1989. Operations on records. In *Proceedings of Mathematical Foundation of Programming Semantics*. Lecture Notes in Computer Science, vol. 442. Springer-Verlag, Berlin, 22–52.

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Comput Surv. 17*, 4 (Dec.), 471–522.

CARTWRIGHT, R. AND FAGAN, M. 1991. Soft typing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. ACM, New York, 278–292.

CONNOR, R., DEARLE, A., MORRISON, R., AND BROWN, F. 1989. An object addressing mechanism for statically typed languages with multiple inheritance. In *Proceedings of the ACM OOPSLA Conference* (New Orleans, La.). ACM, New York, 279–285.

COURCELLE, B. 1983. Fundamental properties of infinite trees. *Theor. Comput. Sci. 25*, 95–169.

DAMAS, L. AND MILNER, R. 1982. Principal type-schemes for functional programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, New York, 207–212.

FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H. 1979. Extendible hashing — a fast access method for dynamic files. *ACM Trans. Database Syst. 4*, 3, 315–344.

FELLEISEN, M., FRIEDMAN, D. P., KOHLBECKER, E., AND DUBA, B. 1987. A syntactic theory of sequential control. *Theor. Comput. Sci. 52*, 205–237.

FUH, Y.-C. AND MISHRA, P. 1988. Type inference with subtypes. In *Proceedings of ESOP '88*. Lecture Notes in Computer Science, vol. 300. Springer-Verlag, Berlin, 94–114.

FURUSE, J.P. AND GARRIGUE, J. 1995. A label selective lambda calculus with optional argument and its compilation method. RIMS Preprint 1041, Research Instit. for Mathematical Sciences, Kyoto Univ., Kyoto, Japan.

GALLIER, J. AND SNYDER, W. 1989. Complete sets of transformations for general E-unification. *Theor. Comput. Sci. 67*, 2, 203–260.

GARRIGUE, J. AND AÏT-KACI, H. 1994. The typed polymorphic label-selective λ calculus. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM, New York, 35–47

GIRARD, J.-Y 1971 Une extension de l'interpretation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et théorie des types. In the *2nd Scandinavian Logic Symposium.* North-Holland, Amsterdam

HALL, C. 1994. Using Hindley-Milner type inference to optimize list representation. In *Proceedings of the ACM Conference on Lisp and Functional Programming.* ACM, New York.

HALL, C , HAMMOND, K., PEYTON JONES, S., AND WADLER, P. 1994. Type class in Haskell Tech. rep., Univ. of Glasgow, Glasgow, Scotland.

HARPER, R AND MITCHELL, J. C 1993. On the type structure of Standard ML *ACM Trans. Program. Lang. Syst. 15,* 2, 211–252.

HARPER, R. AND MORRISETT, G 1995. Compiling polymorphism using intensional type analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM, New York, 130–141.

HARPER, R. AND PIERCE, B. 1991. A record calculus based on symmetric concatenation. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM, New York, 131–142.

JATEGAONKAR, L. A. AND MITCHELL, J. 1993. Type inference with extended pattern matching and subtypes *Fundam. Inform. 19,* 127–165.

LEROY, X. 1992. Unboxed objects and polymorphic typing. In *Procceedings of the ACM Symposium on Principles of Programming Languages.* ACM, New York, 177–188

MILNER, R. 1978 A theory of type polymorphism in programming. *J. Comput. Syst. Sci 17,* 348–375.

MILNER, R., TOFTE, M., AND HARPER, R 1990 *The Definition of Standard ML.* MIT Press, Cambridge, Mass.

MITCHELL, J. 1984. Type inference and type containment. In *Semantics of Data Types.* Lecture Notes in Computer Science, vol. 173. Springer-Verlag, Berlin, 257–277.

MITCHELL, J. 1990 Type systems for programming languages. In *Handbook of Theoretical Computer Science,* J. VAN LEEUWEN, Ed. MIT Press, Cambridge, Mass., 365–458

OHORI, A. 1989. A simple semantics for ML polymorphism. In *Proceedings of the ACM/IFIP Conference on Functional Programming Languages and Computer Architecture* (London, England). ACM, New York, 281–292

OHORI, A. 1990. Semantics of types for database objects. *Theor Comput. Sci. 76,* 53–91.

OHORI, A. 1992. A compilation method for ML-style polymorphic record calculi. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM, New York, 154–165.

OHORI, A. AND BUNEMAN, P. 1988. Type inference in a database programming language. In *Proceedings of the ACM Conference on LISP and Functional Programming* (Snowbird, Utah). ACM, New York, 174–183

OHORI, A. AND BUNEMAN, P. 1989. Static type inference for parametric classes. In *Proceedings of the ACM OOPSLA Conference* (New Orleans, La.). ACM, New York, 445–456 Extended version in *Theoretical Aspects of Object-Oriented Programming,* C GUNTER AND J. MITCHELL, Eds MIT Press, Cambridge, Mass , 121–148, 1994.

OHORI, A. AND TAKAMIZAWA, T 1995 A polymorphic unboxed calculus as an abstract machine for polymorphic languages. RIMS Preprint 1031, Research Instit. for Mathematical Sciences, Kyoto Univ., Kyoto, Japan.

PETERSON, J. AND JONES, M 1993. Implementing type classes. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* ACM, New York, 227–236

RÉMY, D 1989. Typechecking records and variants in a natural extension of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM, New York, 77–87

RÉMY, D. 1992. Typing record concatenation for free. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM, New York, 167–176.

RÉMY, D. 1994a. Efficient representation of extensible records. In *Proceedings of the ACM SIGPLAN Workshop on ML and Its Applications.* ACM, New York, 12–16.

RÉMY, D. 1994b. Type inference for records in a natural extension of ML. In *Theoretical Aspects of Object-Oriented Programming*, C. GUNTER AND J. MITCHELL, Eds. MIT Press, Cambridge, Mass., 67–96.

REYNOLDS, J. 1974. Towards a theory of type structure. In the *Paris Colloqium on Programming.* Springer-Verlag, Berlin, 408–425.

ROBINSON, J. A. 1965. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 23–41.

SHAO, Z., REPPY, J., AND APPLE, A. W. 1994. Unrolling lists. In *Proceedings of the ACM Conference on Lisp and Functional Programming.* ACM, New York.

STANSIFER, R. 1988. Type inference with subtypes. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM, New York, 88–97.

STEELE, G.L. 1984. *Common LISP: The Language.* Digital Press, Burlington, Ma.

TOLMACH, A. 1994. Tag-free garbage collection using explicit type parameters. In *Proceedings of the ACM Conference on Lisp and Functional Programming* ACM, New York, 1–11.

VASCONCELOS, V. 1994. A process-calculus approach to typed concurrent objects. Ph.D. thesis, Dept. of Computer Science, Keio Univ., Yokohama, Japan.

WADLER, P. AND BLOTT, S. 1989. How to make ad hoc polymorphism less ad hoc. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM, New York, 60–76.

WAND, M. 1987. Complete type inference for simple objects. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science* (Ithaca, New York). IEEE, New York, 37–44.

WAND, M. 1988. Corrigendum : Complete type inference for simple object. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science.* IEEE, New York, 132.

WAND, M. 1989. Type inference for records concatenation and simple objects. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science.* IEEE, New York, 92–97.

WRIGHT, A. AND CARTWRIGHT, R. 1994. A practical soft type system for scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming.* ACM, New York, 250–262.

Received April 1995; revised September 1995; accepted October 1995