Static Type Inference for Parametric Classes

Static Type Inference for Parametric Classes*

Atsushi Ohori Peter Buneman

Abstract

Method inheritance and data abstraction are central features of object-oriented programming that are attained through a hierarchical organization of classes. Recent studies have shown that method inheritance can be supported by an ML style type inference when extended to labeled records. This is based on the observation that a function that selects a field f of a record can be given a polymorphic type that enables it to be applied to any record which contains a field f. Several type systems also provide data abstraction through abstract type declarations. However, these two features have not yet been properly integrated in a statically checked polymorphic type system.

This paper proposes a static type system that achieves this integration in an ML-like polymorphic language by adding a class construct which allows the programmer to build a hierarchy of classes connected by multiple inheritance declarations. Classes can be parameterized by types allowing "generic" definitions. The type correctness of class declarations is statically checked, and a principal scheme is inferred for any type correct program containing methods and objects defined in classes. Moreover, the type system can be extended to include the structures and operations needed for database programming and therefore can serve as a basis of an object-oriented database programming language.

1 Introduction

Code sharing is a term that implies the ability to write one piece of code that can be applied to different kinds of data. What this means in practice depends on what we mean by "kinds of data". In object-oriented languages [GR83] each data element (object) belongs to a unique member of a class hierarchy. The code that is applicable to that object is not only the code that is defined in its own class but also the one defined in its super-classes. As an example, we can define a class *person* with a method *increment_age* that increments a person's age by 1. We may define a subclass,

* Published in a book **"Theoretical Aspects of Object-Oriented Programming Types, Semantics and Language Design", C. Gunter and J. Mitchell, editors, pages 121 – 148, MIT Press, 1994.**

This is an extended version of the paper appeared in Proc. ACM OOPSLA Conference. Pages 445–456. New Orleans, LA., 1989.

This research was supported in part by grants NSF IRI86-10617, ARO DAA6-29-84-k-0061 and ONR NOOO-14-88-K-0634.

Authors' current address: Atsushi Ohori: Oki Electric Industry, Kansai Laboratory, Crystal Tower, 1-2-27 Shiromi, Chuo-ku, Osaka 540, JAPAN. Peter Buneman: Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104-6389.

employee, of *person* and expect that the same method, *increment_age*, can be applied to instances of the class *employee*.

In contrast, languages such as Ada [IBH+79], CLU [LAB+81], Standard ML [HMT88] and Miranda [Tur85] — to name a few — provide a generic or polymorphic type system that allows code to be refined by instantiating type variables. Moreover, in the type system of ML, a most general polymorphic type-scheme may be inferred for an (untyped) function and the correctness of the application of that function is checked by finding a suitable instantiation of the type variables. For example, in ML, one may define a polymorphic function *reverse* that reverses a list. From a definition of *reverse* that contains no mention of types, the ML type system is able to infer that $list(t) \to list(t)$ is a most general polymorphic type-scheme of *reverse* where t is a type variable. One may subsequently apply this function to a list of arbitrary type, i.e. a value of any type of the form $list(\tau)$. Through this polymorphic type inference mechanism, ML achieves much of the flexibility of programming without the need to express type information, as in dynamically typed languages, while maintaining most of the advantages — for program correctness and efficiency — of a static type system. A drawback to ML is that it does not combine data abstraction with inheritance in the same sense that object-oriented languages do this. While ML provides data abstraction through abstract data type declarations, it does not allow these to be organized into a class hierarchy.

The purpose of this paper is to develop a static type system that supports both forms of code sharing by combining ML polymorphism and explicit class definitions. In our type system a programmer can define a hierarchy of classes. A class can be parametric and can contain multiple inheritance declarations. The type correctness of such a class definition (including the type consistency of all inherited methods) is statically checked by the type system. Moreover, apart from the type assertions needed in the definition of a class, the type system has a static type inference similar to that available in ML.

We develop such a type system by exploiting a form of type inference for labeled records and labeled disjoint unions originally suggested by Wand [Wan87]. Labeled records and labeled disjoint unions are the structures that one naturally uses to implement a class hierarchy. For example, to implement a subclass in object-oriented languages one usually adds instance variables to those of the parent class; but one can equally well think of this as adding fields to a record type that implements the parent class. We combine a type inference system for these structures with explicit type declarations that represent classes. The main technical contribution is to demonstrate that such a combination is possible. We show that the resulting type system is sound with respect to the underlying type system and that it has a type inference algorithm. These results can be used to develop a programming language that integrates central features of object-oriented programming and those of statically typed polymorphic languages. Moreover, it is also possible to extend the type system to incorporate a number of structures and operations for database programming. The extended type system can serve as a basis for database programming languages, where databases are represented as typed data structures and queries and other database operations are cleanly represented as statically typed polymorphic functions. Parametric classes and polymorphism capture various aspects of object-oriented databases. Based on these results a prototype programming language embodying the type system described in

this paper (with the exception of class parameterization) has been implemented at University of Pennsylvania. The "core" of the language, i.e. the language without class construct, was described in [OBBT89]. A more detailed description of the language can be found in [BO90].

Wand [Wan87] observed that method inheritance can be supported in an ML-like strongly typed language by extending ML's type inference mechanism to labeled records and labeled disjoint unions. In this paradigm, classes correspond to record types and inheritance is realized by polymorphic typing of functions on records. For example, if we represent the classes *person* and *employee* by the record types $[Name : string, Age : int]$ and $[Name : string, Age : int, Salary : int]$ then the requirement that the method *increment_age* defined on the class *person* should be inherited by *employee* simply means that the type of the function *increment_age* should be a polymorphic type whose instances include not only the type $[Name : string, Age : int] \rightarrow [Name : string, Age : int]$ but also the type $[Name : string, Age : int, Salary : int] \rightarrow [Name : string, Age : int, Salary : int]$.

Wand's system, however, does not share ML's feature of existence of *principal* typing schemes (see [OB88, Wan88] for an analysis of this issue.) Based on Wand's general observation, [OB88] extended the notion of principal type-schemes to include conditions on type variables. (See also [Sta88, JM88, Rem89, Wan89] for related studies.) This extension allows ML polymorphism to be extended to standard operations on records and variants and also to various database operations such as *join* and *projection*. (See also [Wan89, CM89, HP91, Rem91] for proposals for other operations on records.) For example, the function *increment_age* can be implemented by the following code:

$$\textbf{fun } increment\_age(p) = \textbf{modify}(p, Age, p.Age + 1)$$

where $e.l$ selects the l field from the record e, and $\textbf{modify}(p, l, e)$ returns the new record that is same as p except that its l field is changed to e. For this function, the following principal *conditional type-scheme* is inferred:

$$[(t)Age : int] \rightarrow [(t)Age : int]$$

The notation $[(t)\, l_1 : \tau_1, \ldots, l_n : \tau_n]$ represents a *conditional type variable* t, for which substitutions are restricted to those θ such that $\theta(t)$ is a record type containing the fields $l_i : \theta(\tau_i), 1 \leq i \leq n$. Since both $[Name : string, Age : int]$ and $[Name : string, Age : int, Salary : int]$ satisfy the condition, $[Name : string, Age : int] \rightarrow [Name : string, Age : int]$ and $[Name : string, Age : int, Salary : int] \rightarrow [Name : string, Age : int, Salary : int]$ are both instances of the above conditional type-scheme. By this mechanism the function *increment_age* can be safely applied not only to *person* objects but also to *employee* objects.

The type inference method suggested by this example shows an integration of method inheritance and a static type system with ML polymorphism. This approach is not subject to the phenomenon of "loss of type information" associated with type systems based on the subsumption rule [Car88] – the problem observed by Cardelli and Wegner [CW85] (see [BTBO89] for a discussion on this problem). However this approach relies on the structure of record types of objects: inheritance is derived from the polymorphic nature of field selection. We would like to borrow from object-oriented languages the idea that the programmer can control the sharing of methods

through an explicitly defined hierarchy of classes and that objects are manipulated only through methods defined in these classes, achieving *data abstraction*.

Galileo [ACO85] integrates inheritance and class hierarchies in a static type system by combining the subtype relation in [Car88] and abstract type declarations. Galileo, however, does not integrate polymorphism or type inference. [JM88] suggests the possibility of using their type inference method to extend ML's abstract data types to support inheritance. Here we provide a formal proposal that achieves the integration of ML style parametric abstract data types and multiple inheritance by extending the type inference method presented in [OB88]. The class declarations we describe can be regarded as a generalization of ML's abstract data types, but there seems to be no immediate connection with the notion of abstract types as existential types in [MP85].

As an example, the class *person* can be implemented by the following class definition:

class $person = [Name : string, Age : int]$ **with**
    fun $make\_person(n, a) = [Name = n, Age = a]$
        $: (string * int) \rightarrow person$;
    fun $name(p) = p.Name : \mathbf{sub} \rightarrow string$;
    fun $age(p) = p.Age : \mathbf{sub} \rightarrow int$;
    fun $increment\_age(p) = \mathbf{modify}(p, Age, p.Age + 1)$
        $: \mathbf{sub} \rightarrow \mathbf{sub}$;
end

Outside of this definition, the actual structure of objects of the type *person* is hidden and *person* objects can only be manipulated through the explicitly defined set of interface functions (methods). This is enforced by treating classes as if they were base types and the methods as the primitive operations associated with them.

As in Miranda's abstract data types, we require the programmer to specify the type (type-scheme) of each method. The keyword **sub** in the type specifications of methods is a special type variable representing all possible subclasses of the class being defined. It is to be regarded as an assertion by the programmer (which may later prove to be inconsistent with a subclass definition) that a method can be applied to values of any subclass. For example, we may define a subclass

class $employee = [Name : string, Age : int, Sal : int]$
isa *person* **with**
    fun $make\_employee(n, a) =$
            $[Name = n, Age = a, Sal = 0]$
        $: (string * int) \rightarrow employee$;
    fun $add\_salary(e, s) = \mathbf{modify}(e, Sal, e.Sal + s)$
        $: (\mathbf{sub} * int) \rightarrow \mathbf{sub}$;
    fun $salary(e) = e.Sal : \mathbf{sub} \rightarrow int$
end

which inherits the methods *name*, *age* and *increment_age*, but not *make_person* from the class *person* because there is no **sub** in the type specification of *make_person*. For reasons that will emerge later we have given the complete record type required

to implement *employee*, not just the additional fields we need to add to the implementation of *person*. It is possible that for simple record extensions such as these we could invent a syntactic shorthand that is more in line with object-oriented languages. Continuing in the same fashion, we may define classes

class $student =$
$\quad [Name : string, Age : int, Grade : real]$
isa $person$ **with**
$\quad \vdots$
end

class $researchFellow =$
$\quad [Name : string, Age : int, Grade : real, Sal : int]$
isa $\{employee, student\}$ **with**
$\quad \vdots$
end

The second of these illustrates the use of multiple inheritance.

The type system we are proposing can statically check the type correctness of these class definitions containing multiple inheritance declarations. Moreover, the type system always infers a principal conditional type-scheme for expressions containing methods defined in classes. For example, for the following function

$$\textbf{fun } raise\_salary(p) = add\_salary(p, \textbf{div}(salary(p), 10))$$

which raises the salary of an object by approximately 10%, the type system infers the following principal conditional type-scheme:

$$(t < employee) \rightarrow (t < employee)$$

where $(t < employee)$ is a conditional type variable whose instances are restricted to subclasses of *employee*. This function can be applied to objects of any subclass of *employee* and the type correctness of such applications is statically checked.

To demonstrate the use of type parameters, consider how a class for lists might be constructed. We start from a class which defines a "skeletal" structure for lists.

class $pre\_list = (\textbf{rec } t.\langle Empty : unit, List : [Tail : t]\rangle)$
with
$\quad nil = \langle Empty = ()\rangle : \textbf{sub};$
$\quad \textbf{fun } tl(x) = \textbf{case } x \textbf{ of}$
$\qquad\qquad \langle Empty = y\rangle \Rightarrow \ldots error \ldots;$
$\qquad\qquad \langle List = z\rangle \Rightarrow z.Tail;$
$\qquad\quad \textbf{end} : \textbf{sub} \rightarrow \textbf{sub}$
$\quad \textbf{fun } null(x) = \textbf{case } x \textbf{ of}$
$\qquad\qquad\quad \langle Empty = y\rangle \Rightarrow true;$
$\qquad\qquad\quad \langle List = z\rangle \Rightarrow false;$
$\qquad\qquad \textbf{end} : \textbf{sub} \rightarrow bool;$
end

This example shows the use of recursive types (**rec** $t.\,\tau$) and labeled variant types ($\langle l_1 : \tau_1, ...l_n : \tau_n\rangle$) with the associated **case** expressions. By itself, the class *pre_list* is useless for it provides no method for constructing non-empty lists. We may nevertheless derive a useful subclass from it.

class $list(a) =$
  (**rec** $t.\,\langle Empty : unit, List : [Head : a, Tail : t]\rangle$)
isa *pre_list*
with
  fun $cons(h,t) = \langle List = [Head = h, Tail = t]\rangle$
    $: (a * \mathbf{sub}) \to \mathbf{sub}$;
  fun $hd(x) =$**case** x **of**
      $\langle Empty = y\rangle \Rightarrow \ldots error \ldots$;
      $\langle List = z\rangle \Rightarrow z.Head$;
    end : $\mathbf{sub} \to a$;
end

which provides the usual polymorphic operations on lists. Separating the definition into two parts may seem pointless here but we may be able to define other useful subclasses of *pre_list*. Moreover, since a may itself be a record type, we may be able to define further useful subclasses of *list*. This is something we shall demonstrate in Section 6. The type correctness of these parametric class declarations is also statically checked by the type system and type inference also extends to methods of parametric classes.

In the following sections we define a simple core language and describe type inference for this language. We then extend the core language with class declarations and show that the extended type system is correct with respect to the underlying type system and provide the necessary results to show that there is a type inference algorithm. To simplify the presentation, we omit sets and database operations treated in [OB88]. However, the theory of parametric classes is completely compatible with these structures. We briefly describe the method to extend the type system presented here to sets and database operations in Section 7. We also omit proofs of some of the results. Their detailed proofs as well as the full treatment of sets and database operations can be found in [Oho89b]. In Section 8 we consider the limitations and implementation aspects of our type system. The combination of multiple inheritance with type parameters requires certain restrictions, and some care is needed to make sure of the existence and correctness of the type inference method. Even if some other formulation of classes in a statically typed polymorphic language is preferable to the system proposed here, we believe that similar issues will arise.

2 The Core Language

The set of types (ranged over by τ) of the core language, i.e. the language without class definitions, is given by the following abstract syntax:

$$\begin{aligned}\tau \quad ::= \quad & b \mid [l : \tau, \ldots, l : \tau] \mid \langle l : \tau, \ldots, l : \tau\rangle \mid \\ & \tau \to \tau \mid (\mathbf{rec}\ v.\,\tau(v))\end{aligned}$$

where b stands for base types and (**rec** $v.\,\tau(v)$) represents recursive types. $\tau(v)$ in (**rec** $v.\,\tau(v)$) is a type expression possibly containing the symbol v. In (**rec** $v.\,\tau(v)$), $\tau(v)$ must be either a record type, a variant type or a function type. The same restriction will apply to similar notations defined below. Formally, the set of types is defined as the set of *regular trees* [Cou83] constructed from base types and type constructors. The above syntax should be regarded as representations of regular trees. In particular (**rec** $v.\,\tau(v)$) represents a regular tree that is a unique solution to the equation $v = \tau(v)$. By the restriction of $\tau(v)$, (**rec** $v.\,\tau(v)$) always denotes a regular tree. For convenience, we assume a set of special labels $\#1, \ldots, \#n, \ldots$ and treat a product type $(\tau_1 * \tau_2 * \cdots * \tau_n)$ as a shorthand for the record type $[\#1 : \tau_1, \ldots, \#n : \tau_n]$.

The set of raw terms (un-checked untyped terms, ranged over by e) is given by the following syntax:

$$\begin{array}{rcl} e & ::= & c^{\tau} \mid x \mid \mathbf{fn}\ x \Rightarrow e \mid e(e) \mid \\ & & [l = e, \ldots, l = e] \mid e.l \mid \mathbf{modify}(e, l, e) \mid \langle l = e \rangle \mid \\ & & \mathbf{case}\ e\ \mathbf{of}\ \langle l = x \rangle \Rightarrow e; \cdots; \langle l = x \rangle \Rightarrow e\ \mathbf{end} \end{array}$$

where c^{τ} stands for constants of type τ, x stands for a given set of variables and $\langle l = e \rangle$ stands for injections to variants. We write $\mathbf{fn}\ (x_1, \ldots, x_n) \Rightarrow e$ for the shorthand for $\mathbf{fn}\ x \Rightarrow e'$ where e' is the term obtained form e by substituting x_i with $x.\#i$ $(1 \leq i \leq n)$. Recursion is represented by a fixed point combinator, which is definable in the core language. The following definition of Y given in [Plo75] can be used to define recursive functions under the usual "call-by-value" evaluation:

$$\begin{array}{l} \mathbf{fun}\ Y(f) = (\mathbf{fn}\ x \Rightarrow (\mathbf{fn}\ y \Rightarrow (f(x(x)))(y))) \\ \qquad\qquad\quad (\mathbf{fn}\ x \Rightarrow (\mathbf{fn}\ y \Rightarrow (f(x(x)))(y))) \end{array}$$

A recursive function definition of the form $\mathbf{fun}\ f(x) = e$ where f appears in the body e is regarded as a shorthand for $f = Y(\mathbf{fn}\ f \Rightarrow \mathbf{fn}\ x \Rightarrow e)$. ML's *let* polymorphism [Mil78, DM82] is compatible with the type system we will develop in this paper and polymorphic *let* binding can be easily added to the language. Interested readers are referred to [Oho89a] for a formal treatment of adding *let*-expressions in a type system like the one defined in this paper.

An association between a raw term and a type is called a *typing*. A *type assignment* $\mathcal{A}$ is a function from a subset of variables to types. For a given type assignment $\mathcal{A}$, we write $\mathcal{A}\{x : v\}$ for the type assignment $\mathcal{A}'$ such that $dom(\mathcal{A}') = dom(\mathcal{A}) \cup \{x\}$, $\mathcal{A}(x) = v$, and $\mathcal{A}'(y) = \mathcal{A}(y)$ for all $y \in dom(\mathcal{A}), y \neq x$. A typing is then defined as a formula of the form $\mathcal{A} \rhd e : \tau$ that is derivable in the proof system shown in Figure 1. We write $\vdash \mathcal{A} \rhd e : \tau$ if $\mathcal{A} \rhd e : \tau$ is derivable.

In general a raw term has infinitely many typings. One important feature of the ML family of languages is the existence of a type inference algorithm, which is based on the existence of a *principal typing scheme* for any typable raw term. The set of *type-schemes* (ranged over by ρ) is the set of regular trees represented by the following syntax:

$$\begin{array}{rcl} \rho & ::= & t \mid b \mid [l : \rho, \ldots, l : \rho] \mid \langle l : \rho, \ldots, l : \rho \rangle \mid \\ & & \rho \rightarrow \rho \mid (\mathbf{rec}\ v.\,\rho(v)) \end{array}$$

where t stands for type variables. A substitution θ is a function from type variables to type-schemes such that $\theta(t) \neq t$ for only finitely many t. We write $[\rho_1/t_1, \ldots, \rho_n/t_n]$

(CONST) $\mathcal{A} \rhd c^b : b$

(VAR) $\mathcal{A} \rhd x : \tau \qquad \text{if } \mathcal{A}(x) = \tau$

(RECORD) $$\frac{\mathcal{A} \rhd e_1 : \tau_1,\ \cdots,\ \mathcal{A} \rhd e_2 : \tau_n}{\mathcal{A} \rhd [l_1 = e_1, \ldots, l_n : e_n] : [l_1 : \tau_1, \ldots, l_n : \tau_n]}$$

(SELECT) $$\frac{\mathcal{A} \rhd e : \tau}{\mathcal{A} \rhd e.l : \tau'} \quad \text{if } \tau \text{ is a record type containing } l : \tau'$$

(MODIFY) $$\frac{\mathcal{A} \rhd e_1 : \tau \qquad \mathcal{A} \rhd e_2 : \tau'}{\mathcal{A} \rhd \mathbf{modify}(e_1, l, e_2) : \tau} \quad \text{if } \tau \text{ is a record type containing } l : \tau'$$

(VARIANT) $$\frac{\mathcal{A} \rhd e : \tau}{\mathcal{A} \rhd \langle l = e \rangle : \tau'} \quad \text{if } \tau' \text{ is a variant type containing } l : \tau$$

(CASE) $$\frac{\mathcal{A} \rhd e : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle,\ \mathcal{A}\{x_i : \tau_i\} \rhd e_i : \tau\ (1 \le i \le n)}{\mathcal{A} \rhd \mathbf{case}\ e\ \mathbf{of}\ \langle l_1 = x_1 \rangle \Rightarrow e_1\ ; \cdots ;\ \langle l_n = x_n \rangle \Rightarrow e_n\ \mathbf{end} : \tau}$$

(APP) $$\frac{\mathcal{A} \rhd e_1 : \tau_1 \to \tau_2 \qquad \mathcal{A} \rhd e_2 : \tau_1}{\mathcal{A} \rhd e_1(e_2) : \tau_2}$$

(ABS) $$\frac{\mathcal{A}\{x : \tau_1\} \rhd e : \tau_2}{\mathcal{A} \rhd \mathbf{fn}\ x \Rightarrow e : \tau_1 \to \tau_2}$$

Figure 1: Typing Rules for the Core Language

for the substitution θ such that $\{t | \theta(t) \neq t\} = \{t_1, \ldots, t_n\}$ and $\theta(t_i) = \rho_i, 1 \le i \le n$. A substitution uniquely extends to type-schemes (and other syntactic structures containing type-schemes). For finite types, this is the unique homomorphic extension of θ. For general regular trees, see [Cou83] for a technical definition. A type-scheme ρ is an *instance* of a type-scheme ρ' if there is a substitution θ such that $\rho = \theta(\rho')$. An instance ρ is *ground* if it is a type. A substitution θ is *ground for* ρ if $\theta(\rho)$ is a type. A *type assignment scheme* Γ is a function from a finite subset of variables to type-schemes. A typing scheme is then defined as a formula of the form $\Gamma \rhd e : \rho$ such that all its ground instances are typings. A typing scheme $\Gamma \rhd e : \rho$ is *principal* if for any typing $\mathcal{A} \rhd e : \tau$, $(\mathcal{A}\uparrow^{dom(\Gamma)}, \tau)$ is a ground instance of (Γ, ρ), where $f\uparrow^X$ is the function restriction of f to X. A principal typing scheme can be also characterized syntactically as a *most general* typing scheme with respect to an ordering induced by substitutions.

For ML it is well-known ([Mil78, DM82]) that for any typable raw term, there is a principal typing scheme; moreover, there is an algorithm to compute this typing scheme. For example, the ML type inference algorithm computes the following principal typing scheme for the function $id \equiv \mathbf{fn}(x) \Rightarrow x$:

$$\emptyset \rhd id : t \to t$$

The set of all typings of id is correctly represented by the set of all ground instances of the above typing scheme (with possible weakening of type assignments). By this mechanism, ML achieves static type-checking and polymorphism (when combined with the binding mechanism of *let*). In the above example, the function id can be safely used as a function of any type of the form $\tau \to \tau$.

In our core language, however, a typable raw term does not necessarily have a principal typing scheme because of the conditions associated with the rules (SELECT), (MODIFY) and (VARIANT). In [OB88] this problem is resolved by extending type-schemes to include conditions on substitutions of type variables. The set of *conditional type-schemes* (ranged over by T) is the set of regular trees represented by the following syntax:

$$\begin{aligned} T \quad ::= \quad & t \mid [(t)l : T, \ldots, l : T] \mid \langle (t)l : T, \ldots, l : T \rangle \mid b \mid \\ & [l : T, \ldots, l : T] \mid \langle l : T, \ldots, l : T \rangle \mid T \to T \mid \\ & (\mathbf{rec}\ v.\, T(v)) \end{aligned}$$

$[(t)l : T, \ldots, l : T]$ and $\langle (t)l : T, \ldots, l : T \rangle$ are *conditional type variables*. Intuitively, $[(t)l_1 : T_1, \ldots, l_n : T_n]$ and $\langle (t)l_1 : T_1, \ldots, l_n : T_n \rangle$ respectively represent record types and variant types that contain the set of fields $l_1 : T_1, \ldots, l_n : T_n$. This intuition is made precise by the notion of *admissible instances*. For a conditional type-scheme T, the *condition erasure* of T, denoted by $erase(T)$, is the type scheme obtained form T by "erasing" all conditions from conditional type variables, i.e. by replacing all conditional type variables of the form $[(t)\ldots]$ and $\langle (t')\ldots \rangle$ by t and t' respectively. Substitutions are extended to conditional type-schemes as $\theta(T) = \theta(erase(T))$. A ground substitution θ for $[(t)l_1 : T_1, \ldots, l_n : T_n]$ is *admissible* for $[(t)l_1 : T_1, \ldots, l_n : T_n]$ if $\theta(t)$ is a record type containing $l_1 : \theta(T_1), \ldots, l_n : \theta(T_n)$. Similarly, θ is admissible for $\langle (t)l_1 : T_1, \ldots, l_n : T_n \rangle$ if $\theta(t)$ is a variant type containing $l_1 : \theta(T_1), \ldots, l_n : \theta(T_n)$. A ground substitution is admissible for a conditional type-scheme T if it is admissible for all conditional type variables in T. A type τ is an *admissible instance* of T if there is an admissible ground substitution θ for T such that $\tau = \theta(T)$. A conditional type-scheme denotes the set of all its admissible instances. For example, the conditional type-scheme

$$[(t)Age : int] \to [(t)Age : int]$$

denotes the set of all types of functions on records containing $Age : int$ field that return a record of the same type.

By using conditional type-schemes, Damas and Milner's result for ML can be extended to our language. A *conditional type assignment scheme* Γ is a function from a finite subset of variables to conditional type-schemes. A *conditional typing scheme* is a formula of the form $\Gamma \rhd e : T$ such that all its admissible instances are typings. We write $\vdash \Gamma \rhd e : T$ if it is a conditional typing scheme. A conditional typing scheme $\vdash \Gamma \rhd e : T$ is *principal* if for any typing $\vdash \mathcal{A} \rhd e : \tau$, $\mathcal{A}\uparrow^{dom(\Gamma)} \rhd e : \tau$ is an admissible instance of $\Gamma \rhd e : \tau$. As in ML a principal conditional typing scheme of e represents the set of all typings for e.

In [OB88] the following property is show for a language containing labeled records and a number of structures and operations for databases:

Theorem 1 *For any raw term e, if e has a typing then it has a principal conditional typing scheme. Moreover, there is an algorithm which, given any raw term, computes its principal conditional typing scheme if one exists and reports failure if not.* ▌

This result can be easily adapted to our language. The following is an examples of a principal conditional typing scheme:

$\emptyset \triangleright \mathbf{fn}\ x \Rightarrow \mathbf{modify}(x, Age, x.Age + 1)$
$\quad : [(t)Age : int] \to [(t)Age : int]$

This property guarantees that we can statically check the type correctness of any given raw term.

3 Formulation of Classes

In this section, we first present a proof system for class declarations as an extension to the core language. We then show the soundness of the proof system relative to the soundness of the type system of the core language and develop a type inference algorithm for the extended language.

3.1 Proof System for Classes

We assume that there is a given ranked alphabet of *class constructors* (ranged over by c) and a set of *method names* (ranged over by m). A class constructor of arity 0 is a constant (non-parametric) class. The set of types is extended by class constructors:

$$\begin{aligned} \tau \quad ::= \quad & b \mid [l : \tau, \ldots, l : \tau] \mid \langle l : \tau, \ldots, l : \tau \rangle \mid \\ & \tau \to \tau \mid c(\tau, \ldots, \tau) \mid (\mathbf{rec}\ v.\, \tau(v)) \end{aligned}$$

The set of raw terms is extended by method names:

$$e ::= m \mid c^{\tau} \mid \ldots$$

In order to allow parametric class declarations, we extend the set of type-schemes with class constructors:

$$\begin{aligned} \rho \quad ::= \quad & t \mid b \mid [l : \rho, \ldots, l : \rho] \mid \langle l : \rho, \ldots, l : \rho \rangle \mid \\ & \rho \to \rho \mid c(\rho, \ldots, \rho) \mid (\mathbf{rec}\ v.\, \rho(v)) \end{aligned}$$

In particular, we call type-schemes of the form $c(\rho_1, \ldots, \rho_k)$ *class-schemes*. We write $c(\overline{t}\,)$ and $c(\overline{\rho})$ for $c(t_1, \ldots, t_k)$ and $c(\rho_1, \ldots, \rho_k)$ where k is the arity of c.

A *class definition* D has the following syntax:

class $c(\overline{t}\,) = \rho$ **isa** $\{c_1(\overline{\rho_{c_1}}), \ldots, c_n(\overline{\rho_{c_n}})\}$ **with**
$\quad m_1 = e_1 \colon M_1;$
$\qquad \vdots$
$\quad m_n = e_n \colon M_n;$
end

$c(\overline{t}\,)$ is the class-scheme being defined by this definition. $\overline{t}$ in $c(\overline{t}\,)$ are type parameters, which must contain all the type variables that appear in the class definition. ρ is the implementation type-scheme of the class $c(\overline{t}\,)$, which must not be a type variable. $\{c_1(\overline{\rho_{c_1}}), \ldots, c_n(\overline{\rho_{c_n}})\}$ is the set of immediate super-class schemes from which $c(\overline{t})$ directly inherits methods. We will show below that the subclass relationship is obtained from this immediate **isa** relation by taking the closure under transitivity and instantiation. Note that class definitions allow both *multiple inheritance* and type parameterization. If the set of super-classes is empty then the **isa** declaration is

omitted. If the set is a singleton set then we omit the braces { and }. Each m_i is the name of a method implemented by the code e_i. Method names m_i should not appear in any method bodies e_j in the same class definition. This restriction is enforced by the type system defined below. It should be noted however that this restriction does not imply that we disallow (mutually) recursive method definitions, which can be provided by the following syntactic sugar:

$$\begin{array}{ll}\textbf{rec} & m_1 = e_1 : M_1; \\ \textbf{and} & \cdots \\ & \vdots \\ \textbf{and} & m_n = e_n : M_n;\end{array}$$

defined as

$$\begin{array}{l} m_1 = (Y\ (\textbf{fn}\ (x_1, \ldots, x_n) \Rightarrow (e'_1, \ldots, e'_n))).\#1 : M_1 \\ \quad \vdots \\ m_n = (Y\ (\textbf{fn}\ (x_1, \ldots, x_n) \Rightarrow (e'_1, \ldots, e'_n))).\#n : M_n \end{array}$$

where e'_i is the term obtained form e_i by substituting m_i by x_i $(1 \leq i \leq n)$. M_i is a *method type* specifying the type of m_i, whose syntax is given below:

$$\begin{array}{rcl} M & ::= & \textbf{sub} \mid t \mid b \mid [l : M, \ldots, l : M] \mid \\ & & \langle l : M, \ldots, l : M \rangle \mid M \to M \mid c(M, \ldots, M) \end{array}$$

sub is a distinguished type variable ranging over all subclasses of the class being defined. Note that we restrict method types to be finite types. This is necessary to ensure the decidability of type-checking of class definitions.

A *class context* (or simply *context*) $\mathcal{D}$ is a finite sequence of class definitions:

$$\mathcal{D} ::= \emptyset \mid \mathcal{D}; D$$

where $\emptyset$ is the empty sequence.

Class declarations are forms of bindings for which we need some mechanism to resolve naming conflicts, such as visibility rules and explicit name qualifications. Here we ignore this complication and assume that method names and class constructor names are unique in a given context. Like a typing scheme, a class definition containing type variables intuitively represents the set of all its instances. The scope of type variables is the class definition in which they appear.

The special type variable **sub** that appears in method type specifications denotes the set of all possible subclasses that the programmer will declare later. This can be regarded as a form of *bounded quantification* [CW85]. The method type M containing **sub** corresponds to $\forall \textbf{sub} < c(\overline{t}).M$ where $c(\overline{t})$ is the class being defined. The relation $<$ is the *subclass relation under a context* $\mathcal{D}$, denoted by $\mathcal{D} \vdash c_1(\overline{\rho_1}) < c_2(\overline{\rho_2})$, which is defined as the smallest transitive relation on class schemes containing:

1. $\mathcal{D} \vdash c(\overline{t}) < c(\overline{t})$ if $\mathcal{D}$ contains a class definition of the form **class** $c(\overline{t}) = \rho\ \cdots$ **end**,
2. $\mathcal{D} \vdash c_1(\overline{t_1}) < c_2(\overline{\rho_2})$ if $\mathcal{D}$ contains a class definition of the form **class** $c_1(\overline{t_1}) = \rho$ **isa** $\{\ldots, c_2(\overline{\rho_2}), \ldots\}$ **with** $\cdots$ **end**,
3. $\mathcal{D} \vdash c_1(\overline{\rho_1}) < c_2(\overline{\rho_2})$ if $\mathcal{D} \vdash c_1(\overline{\rho'_1}) < c_2(\overline{\rho'_2})$ and $(\overline{\rho_1}, \overline{\rho_2})$ is an instance of $(\overline{\rho'_1}, \overline{\rho'_2})$,

The combination of multiple inheritance and type parameterization requires certain conditions on **isa** declarations. A context $\mathcal{D}$ is *coherent* if whenever $\mathcal{D} \vdash c_1(\overline{\rho_1}) < c_2(\overline{\rho_2})$ and $\mathcal{D} \vdash c_1(\overline{\rho_1}) < c_2(\overline{\rho_2'})$, then $\overline{\rho_2} = \overline{\rho_2'}$. We require a context to be coherent. This condition is necessary to develop a type inference algorithm. The following property is easily shown.

Lemma 1 *For a given context $\mathcal{D}$, it is decidable whether $\mathcal{D}$ is coherent or not.* ∎

We say that a subclass relation $\mathcal{D} \vdash c_1(\overline{\rho_1}) < c_2(\overline{\rho_2})$ is more general than $\mathcal{D} \vdash c_1(\overline{\rho_1'}) < c_2(\overline{\rho_2'})$ if $(\overline{\rho_1'}, \overline{\rho_2'})$ is an instance of $(\overline{\rho_1}, \overline{\rho_2})$. A subclass relation $\mathcal{D} \vdash c_1(\overline{\rho_1}) < c_2(\overline{\rho_2})$ is *principal* if it is more general than all provable subclass relations between c_1 and c_2.

Under the coherence condition, the subclass relation has the following property:

Lemma 2 *For any coherent context $\mathcal{D}$ and any method names c_1, c_2, if $\mathcal{D} \vdash c_1(\overline{\rho_1}) < c_2(\overline{\rho_2})$ then there is a principal subclass relation $\mathcal{D} \vdash c_1(\overline{t_1}) < c_2(\overline{\rho_2'})$. Moreover, there is an algorithm which, given a coherent context $\mathcal{D}$ and a pair c_1, c_2, returns either $(\overline{t}, \overline{\rho})$ or failure such that if it returns $(\overline{t}, \overline{\rho})$ then $\mathcal{D} \vdash c_1(\overline{t}) < c_2(\overline{\rho})$ is a principal subclass relation between c_1, c_2 otherwise there is no subclass relation between c_1 and c_2.* ∎

Note that since the substitution relation is decidable, this result implies that the subclass relation is decidable.

The extended type system has the following forms of judgments:

$$\begin{array}{rl} \vdash \mathcal{D} & \mathcal{D} \text{ is a well typed class context,} \\ \vdash \mathcal{D}, \mathcal{A} \rhd e : \tau & \text{the typing } \mathcal{D}, \mathcal{A} \rhd e : \tau \text{ is derivable.} \end{array}$$

The proof systems for these two forms of judgments are defined simultaneously.

Let D be a class definition of the form **class** $c(\overline{t}) = \rho_c \cdots$ **end**. D induces the *tree substitution* ϕ_D on type-schemes. For finite type-schemes, $\phi_D(\rho)$ is defined by induction on the structure of ρ as follows:

$$\begin{array}{rcl} \phi_D(b) & = & b \\ \phi_D(t) & = & t \\ \phi_D(f(\rho_1, \ldots, \rho_n)) & = & f(\phi_D(\rho_1), \ldots, \phi_D(\rho_n)) \text{ for any} \\ & & \text{type constructor } f \text{ s.t. } f \neq c \\ \phi_D(c(\overline{\rho})) & = & \rho_c[\phi_D(\overline{\rho})/\overline{t}] \end{array}$$

where $[\phi_D(\overline{\rho})/\overline{t}]$ denotes $[\phi_D(\rho_1)/t_1, \ldots, \phi_D(\rho_k)/t_k]$ (with k the arity of c). Since ρ_c is not a type variable, ϕ_D is a *non-erasing second-order substitution* on trees [Cou83], which extends uniquely to regular trees. See [Cou83] for the technical details. Since regular trees are closed under second-order substitution [Cou83], $\phi_D(\rho)$ is a well defined type-scheme.

The rule for $\vdash \mathcal{D}$ is defined by induction on the length of $\mathcal{D}$:

1. The empty context is a well typed context, i.e. $\vdash \emptyset$.

2. Suppose $\vdash \mathcal{D}$. Let D be the following class definition:

class $c(\bar{t}\,) = \rho$ **isa** $\{c_1(\overline{\rho_{c_1}}), \ldots, c_n(\overline{\rho_{c_n}})\}$
with
$m_1 = e_1 : M_1;$
$\vdots$
$m_n = e_n : M_n$
end.

Then $\vdash \mathcal{D}; D$ if the following conditions hold:

(a) it is coherent,

(b) if a class name c' appears in some of $\rho, c_1(\overline{\rho_{c_1}}), \ldots, c_n(\overline{\rho_{c_n}})$ then $\mathcal{D}$ contains a definition of the form **class** $c'(\overline{t'}) \cdots$ **end**,

(c) $\vdash \mathcal{D}, \emptyset \rhd e_i : \tau$ for any ground instance τ of $\phi_D(M_i[\rho/\mathbf{sub}])$,

(d) for any method $m = e_m : M_m$ defined in some declaration of class $c'(\overline{t'})$ in $\mathcal{D}$ such that $\mathcal{D}; D \vdash c(\bar{t}\,) < c'(\overline{\rho'})$, $\vdash \mathcal{D}, \emptyset \rhd e_m : \tau$ for any ground instance τ of $M_m[\overline{\rho'}/\overline{t'}, \rho/\mathbf{sub}]$.

We have already discussed the necessity of the condition (a). The necessity of the condition (b) is obvious. The condition (c) states that each method defined in the definition of the class $c(\bar{t}\,)$ is type consistent with its own implementation. Note that since M_i is finite, $\phi_D(M_i[\rho/\mathbf{sub}])$ is effectively computable by the inductive definition of ϕ_D. The condition (d) ensures that all methods of all super-classes that are defined in $\mathcal{D}$ are also applicable to the class $c(\bar{t})$. This is done by checking the type consistency of each method e_m defined in a super-class against the type-scheme obtained from M_m by instantiating its type variables with type-schemes specified in **isa** declaration in the definition of the class $c(\bar{t})$ and replacing the variable **sub** with the implementation type ρ of the class $c(\bar{t}\,)$.

The proof rules for typings are given by extending the proof rules for typings of the core language by the following rule:

(METHOD) $\vdash \mathcal{D}, \mathcal{A} \rhd m : \tau$
if $\vdash \mathcal{D}$ and there is a method $m = e : M$ of a class $c(\bar{t}\,)$ in $\mathcal{D}$ such that τ is an instance of $M[\overline{\rho}/\bar{t}, c'(\overline{t'})/\mathbf{sub}]$ for some $\mathcal{D} \vdash c'(\overline{t'}) < c(\overline{\rho}\,)$.

The well definedness of these two mutually dependent definitions can be checked by induction on the length of $\mathcal{D}$. Since the decidability of judgments of the form $\vdash \mathcal{D}$ will follow from Lemma 1, 2, and the decidability of typing judgments, the decidability of static typechecking of the entire type system is established by the existence of a complete type inference algorithm for typing judgments, which we will develop in Section 5. But first, we establish the soundness of the type system. The following property is useful:

Lemma 3 *If $\vdash \mathcal{D}; D$ and $m = e : M$ is defined in D then m does not appear any body of method definition in $\mathcal{D}; D$.* ∎

4 Soundness of the Type System

Let $\mathcal{D}$ be a given context and τ be a type. The *exposure of τ under $\mathcal{D}$*, denoted by $expose_{\mathcal{D}}(\tau)$, is the type given by the following inductive definition on the length of $\mathcal{D}$:

1. if $\mathcal{D} = \emptyset$ then $\mathit{expose}_{\mathcal{D}}(\tau) = \tau$,

2. if $\mathcal{D} = \mathcal{D}'; D$ then $\mathit{expose}_{\mathcal{D}}(\tau) = \mathit{expose}_{\mathcal{D}'}(\phi_D(\tau))$.

By the condition (b) of the definition for $\vdash \mathcal{D}$, if $\vdash \mathcal{D}, \mathcal{A} \rhd e : \tau$ then $\mathit{expose}_{\mathcal{D}}(\tau)$ does not contain any class name and therefore a type in the core language. Intuitively, $\mathit{expose}_{\mathcal{D}}(\tau)$ is the type obtained from τ by recursively replacing all its classes by their implementation types. We extend $\mathit{expose}_{\mathcal{D}}$ to syntactic structures containing type-schemes.

The *unfold* of a raw term e under a context $\mathcal{D}$, denoted by $\mathit{unfold}_{\mathcal{D}}(e)$, is the raw term given by the following inductive definition on the length of $\mathcal{D}$:

1. if $\mathcal{D} = \emptyset$ then $\mathit{unfold}_{\mathcal{D}}(e) = e$,

2. if $\mathcal{D} = \mathcal{D}'$;**class** $\ldots$ **with**
$$\begin{array}{l} m_1 = e_1 : M_1; \\ \vdots \\ m_n = e_n : M_n \end{array}$$
end,
then $\mathit{unfold}_{\mathcal{D}}(e) = \mathit{unfold}_{\mathcal{D}'}(e[e_1/m_1, \ldots, e_n/m_n])$.

By Lemma 3, if $\vdash \mathcal{D}, \mathcal{A} \rhd e : \tau$ then $\mathit{unfold}_{\mathcal{D}}(e)$ does not contain any method name and is therefore is a raw term in the core language. $\mathit{unfold}_{\mathcal{D}}(e)$ corresponds to the raw term obtained from e by recursively replacing all method names defined in $\mathcal{D}$ with their implementations. We then have the following theorem.

Theorem 2 *If* $\vdash \mathcal{D}, \mathcal{A} \rhd e : \tau$ *then* $\vdash \mathit{expose}_{\mathcal{D}}(\mathcal{A}) \rhd \mathit{unfold}_{\mathcal{D}}(e) : \mathit{expose}_{\mathcal{D}}(\tau)$ *in the core language.*

Proof (Sketch) The proof is by induction on the length of $\mathcal{D}$. The basis is trivial. The induction step is by induction on the structure of e. Cases other than $e = m$ follow directly from the properties of *expose* and *unfold*. The case for $e = m$ is proved by the typing rule (METHOD) and the definitions of *expose*, *unfold*. ∎

Since the soundness of the core language can be shown by using, for example, the techniques developed in [Tof88], the above theorem establishes the soundness of the type system with parametric classes. In particular, since the type system of the core language prevents all run-time type errors, a type correct program in the extended language cannot produce a run-time type error.

The converse of this theorem, of course, does not hold, but we would not expect it to hold, for one of the advantages of data abstraction is that it allows us to distinguish two methods that may have the same implementation. As an example, suppose $\mathcal{D}$ contains definitions for the classes *car* and *person* whose implementation types coincide and *person* has a method *minor* which determines whether a person is older than 21 or not. By the coincidence of the implementations, $\vdash \emptyset \rhd \mathit{expose}_{\mathcal{D}}(\mathit{minor}(c)) : \mathit{bool}$ for any *car* object c. But $\vdash \mathcal{D}, \mathcal{A} \rhd \mathit{minor}(c) : \mathit{bool}$ is not provable unless we declare (by a sequences of **isa** declarations) that *car* is a subclass of *person*. This prevents illegal use of a method through a coincidence of the implementation schemes.

5 Type Inference for the Extended Language

We next show that there is a static type inference algorithm for the extended language. The set of conditional type-schemes is extended with classes and new conditional type variables:

$$T ::= c(T, \ldots, T) \mid (t < \{T, \ldots, T\}) \mid \cdots$$

where $(t < \{T, \ldots, T\})$ stands for new form of conditional type variables, called *bounded type variables*. Intuitively, $(t < \{T_1, \ldots, T_n\})$ represents the set of all instances $\theta(t)$ that are subclasses of all of $\theta(T_1), \ldots, \theta(T_n)$ under a given context $\mathcal{D}$. This intuition is made precise by extending the notion of condition erasure $erase(T)$, substitution instances $\theta(T)$ and the admissibility of substitutions. The condition erasure $erase(T)$ of T is extended to bounded type variables, i.e. $erase$ also replaces conditional type variables of the form $(t < \{T_1, \ldots, T_n\})$ by t. The definition of instances is the same as before. The admissibility of substitutions is now defined relative to a context $\mathcal{D}$. A ground substitution θ is *admissible for* $(t < \{T_1, \ldots, T_n\})$ *under a context* $\mathcal{D}$ if $\mathcal{D} \vdash \theta(t) < \theta(T_i)$ for all $1 \leq i \leq n$. Note that for a bounded type variable $(t < \{T_1, \ldots, T_n\})$ to have an admissible substitution, each T_i must be a type-scheme of the form $c(T, \cdots, T)$. The rules for other forms of conditional type variables are the same as before. A ground substitution is admissible for a conditional type-scheme T under a context $\mathcal{D}$ if it is admissible for all conditional types variables in T under $\mathcal{D}$. A type τ is an *admissible instance of* T *under* $\mathcal{D}$ if there is an admissible ground substitution θ for T under $\mathcal{D}$ such that $\tau = \theta(T)$. A conditional type-scheme denotes the set of all its admissible instances under a given context.

The relationship between the provability of conditional typing schemes and typings is similar to the one in the core language except it is now defined relative to a given context $\mathcal{D}$. $\Gamma \rhd e : T$ is a *conditional typing scheme under* $\mathcal{D}$, denoted by $\vdash \mathcal{D}, \Gamma \rhd e : T$, if $\vdash \mathcal{D}, \mathcal{A} \rhd e : \tau$ holds for any admissible instance $\mathcal{A} \rhd e : \tau$ of $\Gamma \rhd e : T$ under $\mathcal{D}$. The definition for *principality* is also the same. We then have the following theorem which is an extension of Theorem 1:

Theorem 3 *For any raw term e, and any well typed context $\mathcal{D}$ if e has a typing under $\mathcal{D}$ then it has a principal conditional typing scheme under $\mathcal{D}$. Moreover, there is an algorithm which, given any raw term and any well typed context $\mathcal{D}$, computes its principal conditional typing scheme under $\mathcal{D}$ if one exists, and reports failure otherwise.*

Proof (sketch) The strategy is based on that used in [OB88]. The algorithm to compute a principal conditional typing scheme is defined in two steps. It first constructs a typing scheme and a set of conditions of the forms $T < T'$ (representing bound conditions), $[(l : T) \in T']$ (representing field inclusion relation on record types) and $\langle (l : T) \in T' \rangle$ (representing field inclusion relation on variant types). The algorithm then reduces the set of conditions to conditional type-schemes. For a condition of the from $T < T'$, the reduction is done by producing a most general substitution θ such that $\mathcal{D} \vdash \theta(T) < \theta(T')$. This is possible because of the property shown in Lemma 2. The reduction of conditions of the forms $[(l : T) \in T']$ and $\langle (l : T) \in T \rangle$ is done by producing a substitution θ and a set of conditional types of the form $[(t)l : T, \ldots]$ and $\langle (t)l : T, \ldots \rangle$. ∎

6 Further Examples

In Section 1, we defined the classes *person* and *employee*. The sequence of the two definitions is indeed a type correct class context in our type system. Figure 2 shows an interactive session involving these class definitions in our prototype implementation, whose syntax mostly follows that of ML. `->` is input prompt followed by user input. `>>` is output prefix followed by the system output. `('a < person)` and `('a < employee)` are bounded type variables. As seen in the example, the system displays the set of all inherited methods for each type correct class definition.

Let us look briefly at some further examples of how type parameterization can interact with inheritance. At the end of Section 1 we defined a polymorphic list class $list(a)$. We could immediately use this by implicit instantiation of a. For example, the function

$$\begin{aligned}&\textbf{fun } sum(l) = \textbf{if } null(l) \textbf{ then } 0\\ &\qquad\qquad\qquad\textbf{else } hd(l) + sum(tl(l))\end{aligned}$$

will be given the type $list(int) \rightarrow int$, as would happen in ML. However we can instantiate the type variable a in other ways. For example, we could construct a class

$$\begin{aligned}&\textbf{class } genintlist(b) =\\ &\quad(\textbf{rec } t.\ \langle Empty : unit,\\ &\qquad\quad List : [Head : [Ival : int, Cont : b],\\ &\qquad\qquad\quad Tail : t]\rangle)\\ &\textbf{isa } list([Ival : int, Cont : b])\\ &\textbf{with}\\ &\quad\vdots\\ &\textbf{end}\end{aligned}$$

which could be used, say, as the implementation type for a "bag" of values of type b. In this case all the methods of *pre_list* and *list* are inherited. However, we might also attempt to create a subclass of *list* with the following declaration in which we directly extend the record type of the *List* variant of the implementation:

$$\begin{aligned}&\textbf{class } genintlist(b) =\\ &\quad(\textbf{rec } t.\ \langle Empty : unit,\\ &\qquad\quad List : [Head : int, Cont : b, Tail : t]\rangle)\\ &\textbf{isa } list(int)\\ &\textbf{with}\\ &\quad\vdots\\ &\textbf{end}\end{aligned}$$

In this class, all the methods of *pre_list* could be inherited but the method *cons* of $list(a)$ cannot be inherited because the implementation type of $genintlist(b)$ is incompatible with any of the possible types of *cons*. In this case, the type checker reports an error.

```
-> class person = [Name:string,Age:int]
   with  ···  end;

>> class person with
      make_person :  (string*int) -> person
      name :  ('a < person) -> string
      age :  ('a < person) -> int
      increment_age : ('a < person) -> ('a < person)


-> class employee = [Name:string,Age:int,Sal:int]
   with   ···  end;

>> class employee isa person with
      make_employee :  (string*int) -> employee
      add_salary :  (employee*int) -> employee
      salary :  ('a < employee) -> int
   inherited methods:
      name :  ('a < person) -> string
      age :  ('a < person) -> int
      increment_age : ('a < person) -> ('a < person)

-> val joe = make_person("Joe",21);
>> val joe = _ :  person

-> val helen = make_employee("Helen",31)
>> val helen = _ :  employee

-> age(joe);
>> 21 :  int

-> val helen = increment_age(helen);
>> val helen = _ :  employee

-> age(helen);
>> 32 :  int
```

Figure 2: A Simple Interactive Session with Classes

7 Extension for Database Programming

The type system described so far can be further extended to incorporate the structures and operations necessary for databases. Indeed, the core type system of [OB88] on which this paper is based includes set data types and a number of database operations. This extension together with the mechanism of parametric classes presented here makes the language appropriate as the basis of an object-oriented database programming language. Here we briefly describe the extension. For the detailed type inference system and its relevance to database programming, see respectively [OB88, BO90].

Since sets and most database operations require decidable equality on terms, they cannot be introduced on arbitrary terms. For this reason, we identify subsets of terms and types as what we call *description terms* and *description types*. Description types are those that do not contain function types (outside the scope of a reference type). Description types are a generalization of ML's *eqtypes* and also have available a number of useful database operations such as *join* and *projection*.

It is not difficult to introduce a set type constructor on description types. The crucial step toward a satisfactory integration of databases and a polymorphic type system is to introduce database operations that are powerful enough to manipulate complex database objects. One important operation common in databases is to *join* two records of consistent information. For example $[Name = "Joe", Age = 21]$ and $[Name = "Joe", Sal = 30,000]$ join to form $[Name = "Joe", Age = 21, Sal = 30,000]$. This can be regarded as a form of record concatenation [Wan89, HP91, Rem91], but can be generalized to arbitrary complex description terms to form natural join of complex database objects. In [BJO91, Oho90], we have achieved this by exploiting *information orderings* $\sqsubseteq$ and $\ll$ respectively on description terms and description types. $d_1 \sqsubseteq d_2$ represents our intuition that d_2 is a better description than d_2 and $\delta_1 \ll \delta_2$ denotes the fact that the structure represented by δ_2 is "more informative" than that represented by δ_1. Here is a simple example of $\ll$.

$$\begin{aligned}&\{[Name : [Fn : string, Ln : string], Age : int]\}\\ &\ll \{[Name : [Fn : string, Mi : char, Ln : string],\\ &\qquad Age : int, Salary : int]\}\end{aligned}$$

The natural join is then regarded as the operator which "combines" two consistent descriptions and is generalized to arbitrary description terms (even those involving cyclic definitions) as:

$$\mathbf{join}(d_1, d_2) = d_1 \sqcup_{\sqsubseteq} d_2$$

with the following polymorphic type:

$$\mathbf{join} : (\delta_1 * \delta_2) \to \delta_1 \sqcup_{\ll} \delta_2$$

Figure 3 shows an example of the generalized natural join of complex values. Other database operations can also be defined using the orderings.

It should be noted that the ordering on types, although somewhat similar to that used in [Car88], is in no sense a part of record polymorphism. In particular, it has no connection to the notion of structural subtyping. We introduced the orderings only to represent *join* and other database operations. This should be apparent from the fact that we have already incorporated field selection and other operations as polymorphic operations without having to make use of subtyping.

```
join({ [Name=[Last="Ludford"], Children={"Jeremy", "Christopher"}],
       [Name=[Last="Gurman"], Children={"Adam", "Benjamin"}]},
     { [Name=[First="Bridget", Last="Ludford"],
        Address=[Street="33 Cleveden Dr", City="Glasgow"]],
       [Name=[First="Wilfred", Last="Anderson"],
        Address=[Street="13 Princes St", City="Edinburgh"]]})
  = { [Name=[First="Bridget", Last="Ludford"],
       Children={"Jeremy","Christopher"},
       Address=[Street="33 Cleveden Dr", City="Glasgow"]]}
```

Figure 3: An Example of Higher-Order Join

To integrate sets, *join* and other database operations in the type system, the necessary extensions are a new class of type variables that ranges only over description types, and new forms of conditions on type variables that capture polymorphic nature of database operations. In the case of *join*, the necessary condition is of the form $\sigma = jointype(\sigma_1, \sigma_2)$. It was shown in [OB88] that these extensions preserve the existence of principal conditional typing schemes and the extended system still has a complete type inference algorithm. Since our mechanism of parametric classes relies only on the existence of a type inference algorithm, the entire language can be extended to database structures. Using operations on sets and *join*, database queries including SQL like expressions of the form

$$\textbf{select} \cdots \textbf{from} \cdots \textbf{where} \cdots$$

can be defined and freely combined with class structures. This achieves a proper integration of object-oriented programming and database programming in a static type system.

For example, the classes *student* and *employee* we have defined earlier can be used to construct a database containing sets of types $\{student\}$ and $\{employee\}$ where $\{\ \}$ is the set type constructor. For such a database, queries can be easily defined as polymorphic functions as shown in the following example:

$$\begin{aligned}
&\textbf{fun } wealthy\ S\ = \textbf{select } name(x)\\
&\qquad\qquad\qquad\quad \textbf{from } x \in S\\
&\qquad\qquad\qquad\quad \textbf{where } salary(x) > 100000\\
&\qquad : \{\,(t < employee)\,\} \rightarrow \{string\}
\end{aligned}$$

$$\begin{aligned}
&\textbf{fun } good_students\ S\ = \textbf{select } name(x)\\
&\qquad\qquad\qquad\quad \textbf{from } x \in S\\
&\qquad\qquad\qquad\quad \textbf{where } grade(x) > 3.7\\
&\qquad : \{\,(t < student)\,\} -> \{string\}
\end{aligned}$$

$$\begin{aligned}
&\textbf{fun } good_fellows\ S =\\
&\qquad \textbf{intersection}(wealthy(S), good_students(S))\\
&\qquad : \{\,(t < \{student, employee\})\,\} -> \{string\}
\end{aligned}$$

Moreover, by representing *object identity* by reference types as implemented in Stan-

dard ML, the type system can capture various aspects of object-oriented databases.

8 Limitations and Implementation

First, we should point out that the language we have proposed differs in some fundamental ways from object-oriented languages in the Smalltalk tradition. A static type system does not fit well with *late binding* – a feature of many object-oriented languages. One reason to have late binding seems to be to implement *overriding* of methods. It is possible that some form of overloading could be added to the language to support this.

One limitation in our type system is the restriction we imposed on inheritance declarations in connection with type parameters. We required that if a class $c(t_1, \ldots, t_k)$ is a subclass of both $c'(\tau_1, \ldots, \tau_j)$ and $c'(\tau'_1, \ldots, \tau'_j)$ then $\tau_i = \tau_i$ for all $1 \leq i \leq j$. This is necessary to preserve the existence of principal conditional typing schemes for all typable raw terms. This disallows certain type consistent declarations such as:

class $C_1(t) = \tau$ **with**
    fun $m(x) = m(x) : \mathbf{sub} \to t$
end

class $C_2 = \tau'$ **isa** $\{C_1(int), C_1(bool)\}$ **with**
    $c = e : C_2$
    $\vdots$
end

which is type consistent in any implementation types τ, τ' but creates a problem that terms like $m(c)$ do not have a principal conditional typing scheme. However, we believe that the condition is satisfied by virtually all ordinary class declarations. Note that in the above example the result type of the method m is the free type variable t without any dependency of its domain type **sub**, which reflects the property that the method m does not terminate on any input. The authors could not construct any natural example that is type consistent but that does not satisfy this coherence condition.

We have only allowed a single **sub** variable. This restricts the expressiveness of method types. For example, we may want to define a method *older* which compares any two objects of any subclasses of *person*. One possible definition of *older* in the definition of a class *person* is

$$older = \mathbf{fn}\ (x, y) \Rightarrow x.Age > y.Age : (\mathbf{sub} * \mathbf{sub}) \to bool$$

The type system infers the following conditional type-scheme for *older*

$$((t < person) * (t < person)) \to bool$$

But this type-scheme requires *older* to be applied to a pair of objects of the *same* type. This problem can be solved by introducing multiple **sub** variables. Since possible subclasses of a given class is always finite, it is not hard to extend our formal type system to allow multiple **sub** variables and to show that the extension preserves the soundness of the type system and the existence of a complete type inference

algorithm. As we will note below, however, such extension makes it more difficult to implement the type inference algorithm.

From a practical perspective, our type system does not immediately yield an efficient implementation. To typecheck a new class definition, the type system requires the typechecking of the raw terms that correspond to new methods defined in the class, and also the consistency checking of all methods of all super-classes already defined against the implementation type of the new class. A naive way to do this would involve recursively unfolding definitions of methods and repeated type-checking of the resulting raw terms in the type system of the core language, which will be prohibitively expensive when the class hierarchy becomes large. Fortunately, this problem can be avoided using the existence of a principal conditional typing scheme for any typable raw term in the extended language. One strategy is to *save* the principal conditional typing scheme of a method when it is first defined. Typechecking of new methods involving this method name can be done by using the saved principal typing scheme of the method. The type consistency of this method against implementation types of newly defined subclasses can be determined by checking whether the required method types are instances of the saved principal conditional type-scheme or not. These techniques eliminate both recursive unfolding and repeated type-checking of method bodies. Since checking whether a method type is an instance of a given principal conditional type-scheme or not can be done efficiently, this strategy yields an efficient implementation of static type-checking of class hierarchies.This strategy, however, relaies on the fact that the type system only allows a single **sub** variable. An efficient implementation strategy for a type system with multiple **sub** variables remains to be investigated.

9 Conclusion

We have presented a type inference system for classes that supports inheritance and parametricity in a statically typed language similar to ML. This achieves a proper integration of object-oriented programming and ML style polymorphic typing. Moreover, the type system can be further extended to include the structures and operations needed for database systems, and can therefore serve as a basis of object-oriented databases.

Some further syntactic sugaring may be appropriate, and we need to investigate scoping rules and overloading to bring our system into line with conventional object-oriented languages. It is also possible that there may be some integration between what we have proposed and the system of modules for Standard ML [Mac86] and its refinement [MMM91].

Another interesting question is a semantics of class definitions. A definition of a class determines a subset of types that are compatible with the set of methods (i.e. the set of raw lambda terms that implement the methods). This suggests that a class definition could be regarded as a form of existential type $\exists \mathbf{sub} : K.\ (M_1 * \ldots * M_n)$ where K denotes the subset of types that are compatible to the set of methods and $M_1, \ldots, M_n$ are the types of the methods defined in the class definition. This is a form of *bounded existential types* introduced in [CW85] but differs from theirs in that the kind K reflects directly the implementations of methods. Semantics of such types

should explain not only the functionality of the set of methods (as is done in [MP85]) but also the structure of a kind K determined by a set of raw lambda terms.

Acknowledgment

We would like to thank Anthony Kosky for his helpful comments on a draft of this paper. In particular, he pointed out the limitation of single **sub** variable mentioned above.

References

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[BJO91] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, 91(1):23–56, 1991.

[BO90] P. Buneman and A. Ohori. Polymoprhism and type inference in database programming. Technical report, Universities of Glasgow and Pennsylvania, 1990. To appear in *ACM Transaction on Database Systems*.

[BTBO89] V. Breazu-Tannen, P. Buneman, and A. Ohori. Can object-oriented databases be statically typed? In *Proc. 2nd International Workshop on Database Programming Languages*, pages 226 – 237, 1989. Morgan Kaufmann Publishers.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. (Special issue devoted to Symp. on Semantics of Data Types, 1984).

[CM89] L. Cardelli and J. Mitchell. Operations on records. In *Proc. Mathematical Foundation of Programming Semantics, Lecture Notes in Computer Science 442*, pages 22–52, 1989.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 207–212, 1982.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983.

[HMT88] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press. 1990.

[HP91] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 131–142, 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.

[IBH+79] J.H. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Bruckner, O. Roubine, and B.A. Wichmann. Rationale of the design of the programming language Ada. *SIGPLAN Notices*, 14(6), 1979.

[JM88] L. A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. ACM Conf. on LISP and Functional Programming*, pages 198–211, 1988.

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, *Lecture Notes in Computer Science 114*. Springer-Verlag, 1981.

[Mac86] D.B. MacQueen. Using dependent types to express modular structure. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 277–286, 1986.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MMM91] J. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 270–278, 1991.

[MP85] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Transaction on Programming Languages and Systems*, 10(3):470–520, 1988.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conf. on LISP and Functional Programming*, pages 174–183, 1988.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proc. ACM SIGMOD Conference*, pages 46–57, 1989.

[Oho89a] A. Ohori. A simple semantics for ML polymorphism. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 281–292, 1989.

[Oho89b] A. Ohori. *A Study of Types, Semantics and Languages for Databases and Object-oriented Programming.* PhD thesis, University of Pennsylvania, 1989.

[Oho90] A. Ohori. Semantics of types for database objects. *Theoretical Computer Science*, 76:53–91, 1990.

[Plo75] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Rem89] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 242–249, 1989.

[Rem91] D. Rémy. Typing record concatenation for free. Technical report, INRIA–Rocquencourt, Le Chesnay Cedex, France., 1991.

[Sta88] R. Stansifer. Type inference with subtypes. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 88–97, 1988.

[Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Department of Computer Science, University of Edinburgh, 1988.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201*, pages 1–16. Springer-Verlag, 1985.

[Wan87] M. Wand. Complete type inference for simple objects. In *Proc. Symp. on Logic in Computer Science*, pages 37–44, 1987.

[Wan88] M. Wand. Corrigendum : Complete type inference for simple object. In *Proc. Symp. on Logic in Computer Science*, 1988.

[Wan89] M. Wand. Type inference for records concatenation and simple objects. In *Proc. Symp. on Logic in Computer Science*, pages 92–97, 1989.