Static Type Inference for Parametric Classes[*]

Atsushi Ohori Peter Buneman

Department of Computer and Information Science
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104-6389

June 9, 2002

Abstract

Central features of object-oriented programming are *method inheritance* and *data abstraction* attained through hierarchical organization of classes. Recent studies show that method inheritance can be nicely supported by ML style type inference when extended to labeled records. This is based on the fact 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 that allows the programmer to build a hierarchy of classes connected by multiple inheritance declarations. Moreover, classes can be parameterized by types allowing "generic" definitions. The type correctness of class declarations is statically checked by the type system. The type system also infers a principal scheme for any type correct program containing methods and objects defined in classes.

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. In contrast, languages such as Ada [IBH$^+$79], CLU [LAB$^+$81], ML [HMM86] and Miranda [Tur85] — to name a few — provide a generic or polymorphic type system that allows code to be refined by the instantiation of type variables. Moreover, in the polymorphic type system of ML and related languages, this refinement is automatically done by the type *inference* mechanism. The type system infers both a most general polymorphic type of a function and an appropriate type instantiation needed for each function application. By this mechanism ML achieves much of the flexibility of dynamically typed languages in 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 propose 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. To achieve this goal we exploit 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

[*] This research was supported in part by grants NSF IRI86-10617, ARO DAA6-29-84-k-0061 and ONR NOOO-14-88-K-0634. The first author was also supported in part by OKI Electric Industry Co., Japan.

This is an authors' version of the paper appeared in **ACM OOPSLA Conference, pages 445–456, 1989.**

that one naturally use to implement 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 the type inference system for these structures with explicit type declarations that represent classes. The main technical contribution of this paper is to establish 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 — the analogs of the results Damas and Milner proved for the language ML [DM82]. Based on these results a prototype programming language embodying the type system described in this paper (with the exception of parametric classes) has been implemented at University of Pennsylvania. The "core" of the language, i.e. the language without class construct, was described in [OBBT89].

To give examples of the different ways code sharing is achieved in object-oriented and polymorphic languages, we can define a class *person* with a method *increment_age* that increment a person's age by 1. We may define a subclass, *employee*, of *person* and expect that the same method, *increment_age*, can be applied to instances of the class *employee*. In ML one may define a polymorphic function *reverse* that reverses a list. This is a function of type $list(t) \to list(t)$ where t is a type variable. One may subsequently apply this function to, say, a list of integers, i.e. a value of type $list(int)$. Moreover ML is able to infer that $list(t) \to list(t)$ is the most general (polymorphic) type of *reverse* from a definition of *reverse* that contains no mention of types.

Wand observed [Wan87] that the method inheritance can be supported in 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 the polymorphic types 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 $[Name : string, Age : int] \to [Name : string, Age : int]$ but also the type $[Name : string, Age : int, Salary : int] \to [Name : string, Age : int, Salary : int]$.

Wand's system, however, does not share the ML's feature of existence of *principal* typing scheme [Mil78], on which ML's type inference is based (see [OB88] for the analysis of this problem). Based on Wand's observation, [OB88] extended the notion of a *principal typing* [Mil78] to allow conditions on type variables. This extension allows ML polymorphism to be extended to wide range of operations including labeled record and labeled variants. (See also [FM88, Sta88, Car88, JM88, Rem89] for related studies.) 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 *conditional type-scheme* is inferred:

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

The notation $[(t)\ l_1 : \tau_1, \ldots, l_n : \tau_n]$ represent a *conditional type variable* whose substitutions are restricted to those θ such that $\theta(t)$ is a record type that contains 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, $[Name : string, Age : int] \to [Name : string, Age : int]$ and $[Name : string, Age : int, Salary : int] \to [Name : string, Age : int, Salary : int]$ are both instances of the type of the above conditional type-scheme. This mechanism guarantees that 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 can be a proper integration of method inheritance and static type system with ML polymorphism. This approach also eliminates the problem of "loss of type information" associated with type systems based on the subtype relation [Car84] – the problem observed by Cardelli and Wegner [CW85] but not completely eliminated (see [OB88] for an analysis of 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 classes achieving *data abstraction*.

Galileo [ACO85] integrates inheritance and class hierarchy in a static type system by combining the subtype relation and abstract type declarations. Galileo, however, does not integrate polymorphism nor 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 abstract data types and multiple inheritance by extending the type inference method presented in [OB88]. In particular, our proposal achieves a proper integration of multiple inheritance in object-oriented programming and type parameterization in ML style abstract data types. The class declarations we describe can be regarded as a generalization of ML's abstract data types, but there is 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 $make\_person(n, a) = [Name = n, Age = a]$
        $: (string * int) \to person$;
    fun $name(p) = p.Name :$ sub $\to string$;
    fun $age(p) = p.Age :$ sub $\to int$;
    fun $increment\_age(p) =$ modify$(p, Age, p.Age + 1)$
        : sub → sub;
end
```

Outside of the definition, the actual structure of objects of the type *person* is hidden and can only be manipulated through the explicitly defined set of interface functions (methods). This is enforced by treating classes and the set of interface functions as if they were base types and 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 $person$
    fun $make\_employee(n, a) =$
            $[Name = n, Age = a, Sal = 0]$
        $: (string * int) \to employee$;
    fun $add\_salary(e, s) =$ modify$(e, Sal, e.Sal + s)$
        : sub $\to int \to$ sub;
    fun $salary(e) = e.Sal :$ sub $\to int$
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 could define classes

```
class $student =$
    $[Name : string, Age : int, Grade : string]$
isa $person$ with
    ⋮
end

class $research\_student =$
    $[Name : string, Age : int, Grade : string, Sal : int]$
isa $\{employee, student\}$ with
    ⋮
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

fun $raise\_salary(p) = add\_salary(p, salary(p)/10)$

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

$$(t < employee) \to (t < employee)$$

where $(t < employee)$ is a *conditional type variable* representing arbitrary subclasses of *employee*. By this type inference mechanism, the type system achieves a proper integration of ML style polymorphism and inheritance. The above function can be applied to objects of any subclass of *employee*. 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 $pre\_list = ($rec $t.\langle Empty : unit, List : [Tail : t]\rangle)$
with
    $nil = \langle Empty = ()\rangle :$ sub;
    fun $tl(x) =$ case $x$ of
                    $\langle Empty = y\rangle \Rightarrow \ldots error \ldots$;
                    $\langle List = z\rangle \Rightarrow z.Tail$;
                end : sub → sub
    fun $null(x) =$ case $x$ of
                    $\langle Empty = y\rangle \Rightarrow true$;
                    $\langle List = z\rangle \Rightarrow false$;
                end : sub $\to bool$;
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** expression. 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. ⟨Empty : unit, List : [Head : a, Tail : t]⟩)
isa pre_list
with
    fun cons(h,t) = ⟨List = [Head = h, Tail = t]⟩
        : (a * sub) → sub;
    fun hd(x) = case x of
                        ⟨Empty = y⟩ ⇒ ... error ...;
                        ⟨List = z⟩ ⇒ z.Head;
                end : sub → a;
end
```

which is a class for polymorphic lists much as they appear in ML. 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 the type inference also extends to methods of generic 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. We omit proofs of some of the results. Their detailed proofs can be found in [Oho89b]. In a final section 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 that the correctness and the existence of the type inference hold. 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:

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

where b stands for base types and $(\mathbf{rec}\ v.\,\tau(v))$ represents recursive types. $\tau(v)$ in $(\mathbf{rec}\ v.\,\tau(v))$ is a type expression possibly containing the symbol v. In $(\mathbf{rec}\ v.\,\tau(v))$, $\tau(v)$ must be either record type, variant type or 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 $(\mathbf{rec}\ v.\,\tau(v))$ represents a regular tree that is a solution to the equation $v = \tau(v)$. By the restriction of $\tau(v)$, $(\mathbf{rec}\ v.\,\tau(v))$ always denotes a regular tree. For convenience, we assume special set of 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:

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

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. Recursion is represented by a fixed point combinator, which is definable in the core language. For example, the following definition of Y can be used to define recursive function under the usual "call-by-value" evaluation:

$$\begin{aligned} \mathbf{fun}\ Y(f) = & (\mathbf{fn}\ x \Rightarrow (\mathbf{fn}\ y \Rightarrow (f(x(x)))(y))) \\ & (\mathbf{fn}\ x \Rightarrow (\mathbf{fn}\ y \Rightarrow (f(x(x)))(y))) \end{aligned}$$

A recursive function definition of the form **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**-expression is compatible to the type system we will develop in this paper and can be easily added to our language. Interested readers are referred to [Oho89a] for a formal treatment of **let**-expression and ML polymorphism.

A raw term is associated with types. We call such associations *typings*. A *type assignment* $\mathcal{A}$ is a function from a subset of variables to types. For a given type assignment $\mathcal{A}$, we write $\mathcal{A}\{x_1 : v_1,\ldots,x : v_n\}$ for the type assignment $\mathcal{A}'$ such that $dom(\mathcal{A}') = dom(\mathcal{A}) \cup \{x_1,\ldots,x_n\}$, $\mathcal{A}'(x) = \mathcal{A}(x)$ for all $x \notin \{x_1,\ldots,x_n\}$, and $\mathcal{A}(x_i) = v_i, 1 \le i \le n$. A typing is then defined as a formula of the form $\mathcal{A} \rhd e : \tau$ that is derivable in the inference system shown in Figure 1. We write $\vdash \mathcal{A} \rhd e : \tau$ if $\mathcal{A} \rhd e : \tau$ is derivable.

A raw term in general has infinitely many typings. One important feature of 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-scheme* (ranged over by ρ) is the set of regular trees represented by the following syntax:

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

$$
\begin{array}{ll}
(\textsc{const}) & \mathcal{A} \rhd c^b \,:\, b \\
(\textsc{var}) & \mathcal{A} \rhd x \,:\, \tau \qquad \text{if } \mathcal{A}(x) = \tau \\
(\textsc{record}) & \dfrac{\mathcal{A} \rhd e_1 \,:\, \tau_1,\ \cdots,\ \mathcal{A} \rhd e_2 \,:\, \tau_n}{\mathcal{A} \rhd [l_1 = e_1, \ldots, l_n : e_n] \,:\, [l_1 : \tau_1, \ldots, l_n : \tau_n]} \\
(\textsc{select}) & \dfrac{\mathcal{A} \rhd e \,:\, \tau}{\mathcal{A} \rhd e.l \,:\, \tau'} \qquad \text{if } \tau \text{ is a record type containing } l : \tau' \\
(\textsc{modify}) & \dfrac{\mathcal{A} \rhd e_1 \,:\, \tau \qquad \mathcal{A} \rhd e_2 \,:\, \tau'}{\mathcal{A} \rhd \mathbf{modify}(e_1, l, e_2) \,:\, \tau} \qquad \text{if } \tau \text{ is a record type containing } l : \tau' \\
(\textsc{variant}) & \dfrac{\mathcal{A} \rhd e \,:\, \tau}{\mathcal{A} \rhd \langle l = e \rangle \,:\, \tau'} \qquad \text{if } \tau' \text{ is a variant type containing } l : \tau \\
(\textsc{case}) & \dfrac{\mathcal{A} \rhd e \,:\, \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle,\ \mathcal{A}\{x_1 : \tau_1\} \rhd e_1 \,:\, \tau,\ \cdots,\ \mathcal{A}\{x_n : \tau_n\} \rhd e_n \,:\, \tau}{\mathcal{A} \rhd \mathbf{case}\ e\ \mathbf{of}\ \langle l_1 = x_1 \rangle \Rightarrow e_1 \mid \cdots \mid \langle l_n = x_n \rangle \Rightarrow e_n\ \mathbf{end} \,:\, \tau} \\
(\textsc{app}) & \dfrac{\mathcal{A} \rhd e_1 \,:\, \tau_1 \to \tau_2 \qquad \mathcal{A} \rhd e_2 \,:\, \tau_1}{\mathcal{A} \rhd e_1(e_2) \,:\, \tau_2} \\
(\textsc{abs}) & \dfrac{\mathcal{A}\{x : \tau_1\} \rhd e \,:\, \tau_2}{\mathcal{A} \rhd \mathbf{fn}\ x \Rightarrow e \,:\, \tau_1 \to \tau_2}
\end{array}
$$

Figure 1: Type Inference System for the Core Language

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 $[\rho_1/t_1, \ldots, \rho_n/t_n]$ for the substitution θ such that $\{t | \theta(t) \neq t\} = \{t_1, \ldots, t_n\}$ and $\theta(t_i) = \rho_i, 1 \leq i \leq n$. 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 substitution θ is *ground for* ρ if $\theta(\rho)$ is a type. A *type assignment scheme* Γ is a function form 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 $\mathcal{A} \rhd e \,:\, \tau$, $(\mathcal{A}\lceil^{dom(\Gamma)}, \tau)$ is a ground instance of (Σ, ρ), where $f\lceil^X$ 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 shown that [Mil78, DM82] for any typable raw term, there is a principal typing scheme. Moreover, there is an algorithm to compute a principal typing scheme for any typable raw terms. For example, ML type inference algorithm computes the following principal typing scheme for the function $id = \mathbf{fn}(x) \Rightarrow x$:

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

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 the 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 has a principal typing scheme because of the conditions associated with the rules (SELECT), (MODIFY) and (VARIANT). In [OB88] we have solved this problem 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:

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

$[(t)l \,:\, T, \ldots, l \,:\, T]$ 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 $\langle (t')\ldots \rangle$ respectively by t and t'. The substitution is extended to conditional type-scheme 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 for $\langle (t)l_1 : T_1, \ldots, l_n : T_n \rangle$. A ground substitution is admissible for a conditional type-scheme T if it is admissible for all conditional types 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* Σ is a function form a finite subset of variables to conditional type-schemes. A *conditional typing scheme* is a formula of the form $\Sigma \rhd e : T$ such that all its admissible instances are typings. We write $\vdash \Sigma \rhd e : T$ if it is a conditional typing scheme. A conditional typing scheme $\vdash \Sigma \rhd e : T$ is *principal* if for any typing $\vdash \mathcal{A} \rhd e : \tau$, $\mathcal{A}\lceil^{dom(\Sigma)} \rhd e : \tau$ is an admissible instance of $\Sigma \rhd e : \tau$. 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 otherwise reports failure.* ▌

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

$$\begin{array}{l}\emptyset \rhd \textbf{fn}\ x \Rightarrow \textbf{modify}(x, Age, x.Age + 1)\\ \quad : [(t)Age : int] \to [(t)Age : int]\end{array}$$

This property guarantees that we can statically check the type correctness of any given raw term. For example, the above function can be applied to any record type containing $Age : int$ field yielding a record of the same type.

3 Formulation of Classes

In this section, we first present a proof system for class declarations as an extension of the core language. We then shows the soundness of the proof system with respect to 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* names (ranged over by c) and a set of *method names* (ranged over by m). A class constructor of arity 0 correspond to a constant class. The set of types is extended by class constructors:

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

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:

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

In particular, we call type-schemes of the form $c(\rho_1, \ldots, \rho_n)$ *class-schemes.* We write $c(\bar{t}\,)$ 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:

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

$c(\bar{t}\,)$ is the class-scheme being defined by this definition. $\bar{t}$ in $c(\bar{t}\,)$ are type parameters, which must contain the set of all type variables that appear in the body of the definition. ρ is the implementation type-scheme of the class $c(\bar{t}\,)$, 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 $c(\bar{t})$ directly inherits methods. We will show below that the subclass relationship is obtained from this immediate **isa** relation by taking the closure of transitivity and instantiation. Note that class definitions allow both *multiple inheritance* and type parameterization. If the set of super classes is empty then **isa** declaration is omitted. If the set is a singleton set then we omit the braces { and }. Each m_i is the name of method implemented by the code e_i. 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:

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

Class declarations are forms of bindings for which we need some mechanism to resolve naming conflict, 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 a 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* proposed by Cardelli and Wegner [CW85]. The method type M containing **sub** corresponds to $\forall \textbf{sub} < c(\bar{t}\,).\, M$ where $c(\bar{t}\,)$ 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 relation on class schemes containing:

1. $\mathcal{D} \vdash c(\bar{t}\,) < c(\bar{t}\,)$ if $\mathcal{D}$ contains a class definition of the form **class** $c(\bar{t}) = \rho\ \cdots$ **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 $(\overline{\rho_1'}, \overline{\rho_2'})$,
4. $\mathcal{D} \vdash c_1(\overline{\rho_1}) < c_2(\overline{\rho_2})$ if $\mathcal{D} \vdash c_1(\overline{\rho_1}) < c_3(\overline{\rho_3})$ and $\mathcal{D} \vdash c_3(\overline{\rho_3}) < c_2(\overline{\rho_2})$ for some $c_3(\overline{\rho_3})$.

The combination of multiple inheritance and type parameterization requires certain condition on **isa** declarations. A context $\mathcal{D}$ is *coherent* if $\mathcal{D} \vdash c_1(\overline{\rho_1}) < c_2(\overline{\rho_2})$ 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 nor.* ▌

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 relation between c_1 and c_2.

Under the coherent 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 $(\bar{t}, \bar{\rho})$ or failure such that if it retuns $(\bar{t}, \bar{\rho})$ then $\mathcal{D} \vdash c_1(\bar{t}) < c_2(\bar{\rho})$ 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 judgements:

$\vdash \mathcal{D}$	$\mathcal{D}$ is a well typed class definition,
$\vdash \mathcal{D}, \mathcal{A} \rhd e : \tau$	the typing $\mathcal{D}, \mathcal{A} \rhd e : \tau$ is derivable.

The proof systems for those two forms of judgements are defined simultaneously.

Let D be a class definition of the form **class** $c(\bar{t}) = \rho_c \cdots$ **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:

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

where $[\phi_D(\bar{\rho})/\bar{t}]$ denotes $[\rho_1/t_1, \ldots, \rho_k/t_k]$ (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
  $m_1 = e_1 : M_1$;
  ⋮
  $m_n = e_n : M_n$
end.
```

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

(a) it is coherent,

(b) if a class name c' appear in some of $\rho, \rho_{c_1}, \ldots, \rho_{c_n}$ 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 $\phi_D(M_i[\rho/\textbf{sub}])$,

(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 $M_m[\overline{\rho'}/\overline{t'}, \rho/\textbf{sub}]$.

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-types that are already 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 is given by extending the proof rules for typings of the core language by the following rule:

(METHOD) $\vdash \mathcal{D}, \mathcal{A} \triangleright m : \tau$
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 $M[\bar{\rho}/\bar{t}, c'(\overline{t'})/\mathbf{sub}]$ for some $\mathcal{D} \vdash c'(\overline{t'}) < c(\bar{\rho}\,)$.

The well definedness of these two mutually dependent definition is shown by the induction on the length of $\mathcal{D}$. Since we have shown (lemma 1, 2 and the computability of $\phi_D(M_i[\rho/\mathbf{sub}])$) the decidability of all conditions for $\vdash \mathcal{D}$ except for the typing judgements $\vdash \mathcal{D}, \mathcal{A} \triangleright e : \tau$, the decidability of the both form of judgements follows form the decidability of the typing judgements, which will be established by the type inference algorithm we will develop in section 5.

4 Soundness of the Type System

We show the correctness of the type system for the extended language with respect to the type system of the core language.

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 $expose_{\mathcal{D}}(\tau) = \tau$,
2. if $\mathcal{D} = \mathcal{D}'; D$ then $expose_{\mathcal{D}}(\tau) = expose_{\mathcal{D}'}(\phi_D(\tau))$.

Intuitively, $expose_{\mathcal{D}}(\tau)$ is the type obtained from τ by recursively replacing all its classes by their implementation types. We extend $expose_{\mathcal{D}}$ to syntactic structures containing type-schemes.

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

1. if $\mathcal{D} = \emptyset$ then $unfld_{\mathcal{D}}(e) = e$,
2. if

$$\begin{array}{l} \mathcal{D} = \mathcal{D}';\mathbf{class} \ldots \mathbf{with} \\ \qquad m_1 = e_1 : M_1; \\ \qquad \vdots \\ \qquad m_n = e_n : M_n \\ \quad \mathbf{end}, \end{array}$$

then $unfld_{\mathcal{D}}(e) = unfld_{\mathcal{D}'}(e[e_1/m_1, \ldots, e_n/m_n])$.

$unfld_{\mathcal{D}}(e)$ is the raw term obtained from e by recursively replacing all method names defined in $\mathcal{D}$ with their implementations.

Theorem 2 *If $\vdash \mathcal{D}, \mathcal{A} \triangleright e : \tau$ then $\vdash expose_{\mathcal{D}}(\mathcal{A}) \triangleright unfld_{\mathcal{D}}(e) : expose_{\mathcal{D}}(\tau)$.*

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 *unfld*. Case for $e = m$ is proved by the typing rule (METHOD) and the definitions of *expose*, *unfld*. ▮

This theorem establishes the correctness of the type system with respect to the type system of the core language. 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 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, $\vdash \emptyset \triangleright expose(minor(c)) : bool$ for any *car* object c. But $\vdash \mathcal{D}, \mathcal{A} \triangleright minor(c) : bool$ is not provable unless we declare (by a sequences of **isa** declarations) that *car* is a subtype of *person*. This prevents illegal use of a method via 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:

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

where $(t < \{T,\ldots,T\})$ stands for new form of conditional type variables, called *bounded type variables.* Intuitively, $(t < \{T_1,\ldots,T_n\})$ represens the set of all instances $\theta(t)$ that are subtypes 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 include bounded type variables, i.e. $erase(T)$ also erase the subclass conditions from $(t < \{T_1,\ldots,T_n\})$. The definition of instances is the same as before. The admissibility of substitution 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 $1 \leq i \leq n$. The rules for other forms of conditional type variables are the same as before. A ground substitution is admissible for a conditional type T under a context $\mathcal{D}$ if it is admissible for all conditional types in T under $\mathcal{D}$. A type τ is an *admissible instance of* T *under* $\mathcal{D}$ if there is a 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. We write $\vdash \mathcal{D}, \Sigma \rhd e : T$ for $\Sigma \rhd e : T$ is a conditional typing scheme under the context $\mathcal{D}$.

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 $\mathcal{D}$. $\vdash \mathcal{D}, \Sigma \rhd e : T$ if for any admissible instance $\mathcal{A} \rhd e : \tau$ of $\Sigma \rhd e : T$ under $\mathcal{D}$, $\vdash \mathcal{D}, \mathcal{A} \rhd e : \tau$. The definition for principal conditional typing schemes 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 otherwise reports failure.*

Proof (sketch) This is proved by using the similar technique we 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 condition), $[(l : T) \in T']$ (representing field inclusion relation on record types) and $\langle (l : T) \in T' \rangle$ (representing field inclusion relation on variant types). The algorithm then reduce 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 1. 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 definition 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 method 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

```
    fun sum(l) = if null(l) then 0
                 else hd(l) + sum(tl(l))
```

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

```
class genintlist(b) =
    (rec t. ⟨Empty : unit,
             List : [Head : [Ival : int, Cont : b],
                     Tail : t]⟩)
isa list([Ival : int, Cont : b])
with
    ⋮
end
```

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:

```
class genintlist(b) =
    (rec t. ⟨Empty : unit,
             List : [Head : int, Cont : b, Tail : t]⟩)
isa list(int)
with
    ⋮
end
```

In this class, all the methods of *pre_list* could be inherited but the method *cons* of *list*(a) cannot be inherited

```
-> 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

-> fun wealthy e = salary(e) > 100000;
>> val wealth = fn :  ('a < employee) -> bool
```

Figure 2: A Simple Interactive Session with Classes

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.

7 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 implement *overriding* of methods. It is possible that some form of overloading could be added to the language to support this.

Another limitation is the restriction we imposed on inheritance declarations in connection to type parameters. We required that if a class $c(t_1, \ldots, t_n)$ is a subtype of both $c'(\tau_1, \ldots, \tau_n)$ and $c'(\tau'_1, \ldots, \tau'_n)$ then $\tau_j = \tau_j$ for all $1 \leq i \leq n$. 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 $m(x) = m(x) : \textbf{sub} \to t$
end

class $C_2 = \tau'$ isa $\{C_1(int), C_1(bool)\}$ with
      $c = e : C_2$
      ⋮
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 natural 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 terminates on any input. The authors could not construct any natural example that is type consistent but does not satisfy this coherence condition.

From a practical perspective, checking the type-correctness of a class definition with **isa** declaration requires the consistency checking of all methods of all super-types already defined. A naive way to do this would involve recursively unfolding definitions of types and method and then type-checking for the resulting raw term in the type system of the core language, which will be prohibitively expensive when the class hierarchy become large. This problem is avoided using the existence of a principal conditional typing scheme for any typable raw term in the extended language. At the time of a definition of each method, we can save its principal conditional typing scheme. The type correctness of the method against a newly defined subtype can then be checked by checking whether the required method type is an instance of its principal conditional type or not. This eliminates repeated type-checking of method bodies but still requires checking whether the required method type is an instance of its principal conditional type or not against the set of all inherited methods. This can be also avoided. The set of all possible implementation types of subclass of a class can be represented by a single principal conditional type. As an example, consider the example of *person* we defied in the introduction. The most general conditional type of the type variable **sub** in the definition of *person* can be computed as $[(t)\ Name : string, Age : int]$. Using this property, the type correctness of a subclass declaration can be checked by checking that the implementation type is an instance of $[(t)\ Name : string, Age : int]$ without checking the consistencies of each method. While the number of inherited methods might become very large, we expect that the number of super-types is relatively small even in development of a large system and therefore that this strategy yields an efficient implementation of a static type-checking of large class hierarchy.

8 Conclusion

We have presented a type inference system for classes that supports inheritance and parametricity in a statically typed language similar to ML. 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 proposed by David MacQueen [Mac86].

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 $\exists \mathbf{sub} : K.\ (M_1 \times \ldots \times M_n)$ 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.

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.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, Lecture Notes in Computer Science 173*. Springer-Verlag, 1984.

[Car88] L. Cardelli. Structural subtyping and the notion of power types. In *Proc. ACM Symposium on Principles of Programming Languages*, San Diedo, California, January 1988.

[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 Surverys*, 17(4):471–522, December 1985.

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

[FM88] Y-C. Fuh and P. Mishra. Type inference with subtypes. In *Proceedings of ESOP '88*, pages 94–114, 1988. Springer LNCS 300.

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

[HMM86] R. Harper, D. B. MacQueen, and R. Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, March 1986.

[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. *ACM SIGPLAN notices*, 14(6), 1979.

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

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU*

Reference Manual, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

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

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

[MP85] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 37–51, New Oreans, January 1985.

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

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

[Oho89a] A. Ohori. A simple semantics for ML polymorphism. In *Proceedings of ACM/IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 281–292, London, England, September 1989.

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

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

[Sta88] R. Stansifer. Type inference with subtypes. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 88–97, 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. Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, June 1987. a corrigendum in *Proc. Symposium on Logic in Computer Science*, 1988.