An Interoperable Calculus for External Object Access

[Extended Abstract]

Atsushi Ohori*
School of Information Science
JAIST (Japan Advanced Institute of Science and Technology)
Ishikawa 923-1292, Japan
ohori@jaist.ac.jp

Kiyoshi Yamatodani†
School of Information Science
JAIST (Japan Advanced Institute of Science and Technology)
Ishikawa 923-1292, Japan

ABSTRACT

By extending an ML-style type system with record polymorphism, recursive type definition, and an ordering relation induced by field inclusion, it is possible to achieve seamless and type safe interoperability with an object-oriented language. Based on this observation, we define a polymorphic language that can directly access external objects and methods, and develop a type inference algorithm. This calculus enjoys the features of both higher-order programming with ML polymorphism and class-based object-oriented programming with dynamic method dispatch. To establish type safety, we define a sample object-oriented language with multiple inheritance as the target for interoperability, define an operational semantics of the calculus, and show that the type system is sound with respect to the operational semantics. These results have been implemented in our prototype interpretable language, which can access Java class files and other external resources.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Constructs and Features

General Terms: Languages

Keywords

interoperability, type inference, record polymorphism, ML, object-oriented language, Java

*Atsushi Ohori's work was partially supported by Grant-in-aid for scientific research on priority area "informatics" A01-08, grant no:14019403.

†Kiyoshi Yamatodani's current address: Ascade, Inc., 14-3 Takadanobaba 3, Shinjuku, Tokyo 169–0075, Japan. E-mail: yamatodani@ascade.co.jp.

ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.

1. INTRODUCTION

Recently emerging component architecture such as Java Virtual Machine classes [13], COM [4], and Microsoft .NET, brings a new software develop environment where the programmer can use rich collections of libraries implemented in various different languages without knowing the details of their implementation languages. Unfortunately, however, there does not seem to exist any easy and natural way to manipulate these components from an existing statically typed functional language. This is due to a form of "impedance mismatch" between interface specification of component architecture and static polymorphic typing of modern functional languages. One way to overcome this problem is to establish a type theoretical basis for integrating *object-oriented model*, which underlies most of component frameworks, and *ML-style polymorphism*, which underlies functional languages.

Although there are some similarities, these two paradigms differ in essential ways in their strength, expressiveness and programming styles. In object-oriented programming, the combination of an explicitly declared *inheritance* relation on classes and dynamic method dispatch achieves flexible object manipulation and finely controlled method sharing. For example, this paradigm can naturally represent heterogeneous collections of objects having some common properties, and generic methods performing the desired specialized action dependent on the target object class. These features are not easily representable in a typed higher-order functional calculus. In contrast, ML-style polymorphic type inference allows type-safe and flexible manipulation of higher-order functions without requiring type annotation. A program can be developed by composing polymorphic higher-order functions. This feature is not well supported in an object-oriented language.

These two paradigms are complementary ones, and we would like to have a language system where we can enjoy the benefits of both of them. Type theoretical foundations of object-oriented programming have been extensively studied and several methods for object-encoding have been proposed (see, for example, [9] for a collection of works on the subject). These results can be used to introduce object manipulation primitives in a polymorphic functional language. This approach is taken in OCaml [12, 20], Moby [7], and O'Haskell [17]. While this approach can introduce some features of object-oriented programming, it complicates the language

on which it is based, and even with sophisticated type theoretical machinery, it is rather difficult to provide the simplicity, flexibility and expressiveness of existing object-oriented languages.

In the present work, we take an alternative approach of developing an ML-style language with a minimum extension for interfacing with a class-based object-oriented language. This is based on our following observations.

- The typing mechanism for using classes should be much simpler than that for defining classes and methods. Although it appears to be rather difficult to extend an ML-style type inference system with the full features of object-oriented programming including inheritance and dynamic method dispatch, it would be feasible to extend it with a mechanism to use existing classes and methods implemented in an object-oriented language.
- This approach would also yields an ML-style *interoperable* language, where the programmer can freely use a rich collection of object-oriented classes within an advanced polymorphic type inference system.

The goal of this paper is to establish type theoretical basis for interoperability and to develop such an interoperable language.

We define (in Section 3) a functional calculus as an extension of the core of ML with the primitives for object creation and method invocation. The intended semantics of these primitives is to call an external object-oriented language and to perform object creation and dynamic method dispatch in the external language. The external object-oriented language can be any one as far as it is a class-based statically typed language such as Java. For the functional calculus, we develop a polymorphic type system and an ML-style type inference algorithm. To incorporate the two important features of objects – polymorphic method invocation and subsumption – we base our development on a polymorphic record calculus [18], whose type system is strong enough to represent objects and methods in a given class hierarchy. To show type safety of our interoperable primitives, we define (in Section 4) a sample object-oriented language and its operational semantics. We then define (in Section 5) an operational semantics of our calculus, which refers to the operational semantics of the object-oriented language, and establish that the type system is sound with respect to the operational semantics.

The interoperable calculus presented in this paper has been implemented with Java as the target for its interoperability. In this language, a program can import a collection of Java classes and invoke object constructors and methods, which are executed in a Java Virtual Machine runtime system through JNI interface. In Section 6, we discuss some practical issues in designing an interoperable language and describe our prototype implementation.

Before giving a technical development of our interoperable calculus, in the next section, we outline our approach and compare it with related works.

2. AN APPROACH TO INTEROPERABLE TYPE SYSTEM

The static structure of an object-oriented program is represented as a collection of classes connected with a subclass relation. A straightforward approach of extending ML with objects would be to add class hierarchy (a collection of classes and the associated subclass relation) with the subsumption rule of the form:

$$\frac{\mathcal{T} \rhd e : c \quad c <: c'}{\mathcal{T} \rhd e : c'}$$

This seemingly simple extension significantly complicates both the type inference algorithm and the representation of a type of a program. An inferred type of a polymorphic program involves a set of order constraints, and becomes difficult to understand for the programmer. For example, even a very simple function such as $\lambda f.\lambda x.f\ x$ can no longer be given a simple polymorphic type. Considering extensive usage of higher-order polymorphic functions in ML programming, we would like to avoid this complication.

Our strategy is to decompose the mechanism of class inheritance into parametric polymorphism and a limited form of object subsumption. The class inheritance in object-oriented programming appears to support the following two features.

1. Polymorphic method invocation.
   A method of a class can be applied to objects of its subclasses. For example, `move` method in a `point` class can be applied to an object of `colorPoint` class. Moreover, if a method is redefined in some of the subclasses, then the method code actually invoked is determined by the actual (run-time) class of the object.

2. Object subsumption.
   An object of a class can be used as an object of any of its superclasses. For example, one can form a `pointSet` containing objects of various subclasses of `point`, and iterate a method `move` over the collection.

In an object-oriented class system, these two are combined. However, these two aspects have different requirements in a static polymorphic type system.

The first feature is precisely represented by the combination of *record polymorphism* [18, 19] and external method invocation. We represent the type of an object as a form of a record type containing all the applicable method signatures. Suppose a class c contains a set of methods $m_1 : \rho_1, \ldots, m_n : \rho_n$. Then the type of an object of class c is represented as a record type of the form $\{m_1 : \rho_1, \ldots, m_n : \rho_n, \ldots\}$. Polymorphic typing of field selection expression $e.m$ in a record calculus is sufficient for typechecking polymorphic method invocation. We then bind the method invocation $e.m$ not to a particular method but to a code in the external language that performs dynamic method invocation. Conceptually, the meaning of $e.m$ is the η-expanded method invocation operation of the form

$$\lambda x.\texttt{invoke}(x, m)$$

in the target object-oriented language where `invoke` is an operation that performs dynamic method dispatch such as `invokevirtual` in JVM. In this way, we achieves the desired features of polymorphic method invocation with dynamic method dispatching semantics.

The above "structural" encoding may incorrectly equate classes that have the same set of method names and signatures. In order to rectify this problem and also to account for the second feature of object subsumption, we refine our

representation of a class c as follows

$$\{m_1 : \rho_1, \ldots, m_n : \rho_n, \ldots, c : unit, c_1 : unit, \ldots, c_n : unit\}$$

to include class name c and all its superclass names $c_1, \ldots, c_n$ as record labels. This encoding explicitly specifies the set of all classes to which an object belongs, and properly represents the uniqueness of each class. Furthermore, by combining a shallow subsumption relation on record types, this encoding allows us to type expressions such as "`[`p`,`cp`,...]`" or "`if` e `then` p `else` cp" where `[`e_1`,`e_2`,...]` is a list formation expression and p and cp are assumed to be of type `point` and `colorPoint` respectively. Since we do not extend this relation to other type constructors including function type, the necessary extension to polymorphic type inference is relatively small. We should note that this subsumption relation is not needed for polymorphic method invocation. In fact, eliminating this mechanism yields a slightly cleaner type system while retaining most of the features except for heterogeneous collections.

Since method signatures ρ in general contain classes, we represent a given set of classes as a set of mutually recursive type equations of the form $c = [\![c]\!]$ where $[\![c]\!]$ is the encoding of c described above.

To illustrate some flavor of the polymorphic interoperable calculus we shall develop in this paper, let us show some examples. Suppose we have a class `printable` containing a method `toString`, then we can easily write a polymorphic function

```
fun layout L = foldr (fn (obj,rest) =>
                      obj.toString ^ rest) "" L
:  ∀(t :: {{toString : t → string}}).list(t) → string
```

and map `layout` function over a list of printable objects, where `fun` is ML-style function definition and $list(t)$ is list type whose element type is t. The notation $\forall(t :: \{\{\texttt{toString} : t \rightarrow \texttt{string}\}\})$ indicates that type variable t ranges over object types containing the field $\texttt{toString} : t \rightarrow \texttt{string}$. When applied, this function invokes `toString` method on each object of the list in the target object-oriented language, performing the desired specialized action depending on the actual class of the object. Figure 1 shows some more examples. As seen from these examples, object-oriented features including method dispatch and manipulation of heterogeneous collections can be freely combined with the feature of higher-order functions with ML polymorphism.

Superficially, these examples may appear simple instances of well studied record calculi, but they are not. We note that the external method invocation such as `p.move` dispatches `move` method on the object denoted by `p`, achieving the effect of dynamically choosing the appropriate methods depending on the actual class of the object.

2.1 Comparison with other approaches

Most of existing approaches use some form of subtyping to represent subclass relation in object-oriented programming.

The simplest form of subtyping in type theory is *structural subtyping* [2], which is derived from the inclusion relation on sets of fields of record types. This has been the basis of polymorphic method definition and for object subsumption. However, a subclass relation in typical object-oriented languages is a *name* subtyping explicitly declared by the programmer, and is not directly representable by structural subtyping. One of the central issues in designing a functional language with object oriented features has been to properly capture name subtyping. Moby [7] combines structural subtyping with name subtyping to form a hybrid system. A type system of Objective ML [20], which underlies OCaml [12], introduces explicitly declared class hierarchy on top of a type system with structural subtyping. O'Haskell [17] adopts a name subtyping on record like constructs. Each of these approaches captures the features of subclass relation to some extent. Observing the complexity of required type theoretical machinery and also the difficulty of achieving simple and powerful inheritance of object-oriented language, Mondrian [14] avoids subtyping and resolves subclass relation dynamically with a mechanism for catching runtime type error.

Complications in these approaches come from their need to provide language constructs for defining class hierarchy. In contrast, our approach shows that record types and record polymorphism are sufficient to use an existing class hierarchy. The feature of name subtyping is represented by including class names in the representing record type.

Approaches most closely related to ours are those based on "phantom types". In [5], it is shown that interface hierarchy of COM is represented in Haskell using phantom types. However, this approach does not directly applicable to multiple interface inheritance found, for example in Java. This problem is solved in Lambada [15] by using type class mechanism [10] of Haskell based on the following techniques. Java class c is encoded as a type T using phantom types, and if c implements interfaces $I_1, \cdots, I_n$, a type class is declared for each I_i with T as its instance. While we see that this encoding works, type class itself is an elaborate system requiring certain amount of type theoretical machinery, and its relationship to subclass is not entirely clear. Also, in these proposals, the issue of type safety is not investigated.

In contrast, our system achieves seamless interoperability with existing classes using the basic properties of record structures, and its type soundness is formally established.

3. THE INTEROPERABLE CALCULUS

This section defines the interoperable calculus.

3.1 Syntax of the calculus

The set of terms of the calculus is given by the following abstract syntax.

$$\begin{array}{rcl} e & ::= & c^b \mid x \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x\ =\ e\ \texttt{in}\ e\ \texttt{end} \\ & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\ & \mid & \texttt{new}\ c(e, \cdots, e) \mid e.l \mid e.m(e, \cdots, e) \end{array}$$

The first two lines are those of core ML. The rest are those for invoking operations in an external object-oriented language. `new` $c(e_1, \cdots, e_n)$ creates a new object of class c by calling the constructor of c with n arguments. $e.l$ retrieves the value of field l of e. $e.m(e_1, \cdots, e_n)$ invokes method m on e with n arguments. Objects reside in a heap of an external object-oriented language, and all these operations are performed in the external language.

Although we only explicitly include external object types and base types, various (internal) data structures can also be added. As we mentioned, the introduction of the standard list type enables us to represent heterogeneous collections of external objects without any additional machinery. There are some issues in adding records depending on whether we

$type\ point = \{\texttt{X} : int,\ \texttt{Y} : int,\ \texttt{move} : int \times int \to unit,\ \texttt{point} : unit\}$
$and\ colorPoint = \{\texttt{X} : int,\ \texttt{Y} : int,\ \texttt{color} : string,\ \texttt{move} : int \times int \to unit,\ \texttt{point} : unit,\ \texttt{colorPoint} : unit\}$

```
fun getX p = p.X
```
$: \forall(t_2{::}U, t_1{::}\{\!\{\texttt{X} : t_2\}\!\}).t_1 \to t_2$
```
fun slide p = (p.move(getX p,0); p)   (* (e1; e2) is a sequence expression a la ML *)
```
$: \forall(t_2{::}U, t_3{::}U, t_1{::}\{\!\{\texttt{X} : t_2,\ \texttt{move} : t_2 \times int \to t_3\}\!\}).t_1 \to t_1$
```
val points = [new point (0,0), new cpoint (0,0,"red")]
```
$: list(\{\texttt{X} : int,\ \texttt{Y} : int,\ \texttt{move} : int \times int \to unit,\ \texttt{point} : unit\})$
```
map slide points
```
$: list(\{\texttt{X} : int,\ \texttt{Y} : int,\ \texttt{move} : int \times int \to unit,\ \texttt{point} : unit\})$

Figure 1: Example typings in the interoperable calculus

want to treat objects and records uniformly or not. We shall comment on this when we describe our prototype implementation in Section 6.

3.2 Assumptions on class structures

The calculus includes external objects. To develop a static type system for the calculus, we need to make some assumptions on static structure of classes to which those external objects belong.

We let c range over *class names* and b range over *base types*. We use o to denote either a class name or a base type, and ρ to denote a method type of the form $o_1 \times \cdots \times o_n \to o$ representing the type of a method which takes a tuple of objects of types $o_1, \ldots, o_n$ (other than the "receiver" object) and yields an object of type o.

In an actual programming language, the external classes to be used is declared by the programmer. In Section 6, we discuss a mechanism for importing external classes in the perspective of practical language design and implementation. However, since they do not affect the typing mechanism we develop in this paper, to simplify the presentation, we assume that we are given a fixed collection of classes, and that their static structure is described in a *class environment* $\mathcal{C}$ consisting of the following:

- the set of class names and the associated subclass relation,
- for each class name c, the set `fields` of field signatures $\{f_1 : o_1, \ldots, f_n : o_n\}$ of c (including all those inherited),
- for each class name c, the type of its constructor, and
- for each class name c, the set `methods` of method signatures $\{m_1 : \rho_1, \ldots, m_1 : \rho_1\}$ of c (including all those inherited).

We write $\mathcal{C} \vdash c <: c'$ if c is a subclass of c', and write $\mathcal{C}(c).\texttt{ancestors}$ for $\{c' | \mathcal{C} \vdash c <: c'\}$; we write $\mathcal{C}(c).\texttt{fields}$ and $\mathcal{C}(c).\texttt{methods}$ for the sets of field signatures and method signatures of c in $\mathcal{C}$, respectively. We also use the following notations.

- $\mathcal{C} \vdash f : o \in c$ if c has a field $f : o$,
- $\mathcal{C} \vdash m : \rho \in c$ if c has a method $m : \rho$, and
- $\mathcal{C} \vdash \texttt{new} : \rho \in c$ if ρ is the type of the constructor of c.

Given a set of class definitions in an object-oriented language such as Java class files, a class environment can easily be extracted.

3.3 Object types and classes

We write $\{l_1 : \rho_1, \cdots, l_n : \rho_n\}$ for a type of an external object whose static properties are determined by the set of fields $l_1 : \rho_1, \cdots, l_n : \rho_n$. To deal with mutual dependency between classes and method signatures, we represent a class as a recursive object type. Since the class structure $\mathcal{C}$ is already given, it is sufficient to introduce explicit recursive type declarations. Furthermore, since objects are created by primitive construct `new`, no special mechanism for recursive object creation is needed.

Let $c_1, \ldots, c_n$ be the set of class names defined in a given class environment $\mathcal{C}$. We regard $c_1, \ldots, c_n$ as type names in the calculus, and define for each c, the encoding $\mathcal{C}[\![c]\!]$ of the object structure of c under $\mathcal{C}$ as follows.

$$\begin{aligned}\mathcal{C}[\![c]\!] \quad = \quad & \{f_1 : o_1, \cdots, f_k : o_k, \\ & m_1 : \rho_1, \cdots, m_m : \rho_m, \\ & c_1 : unit, \ldots, c_n : unit\}\end{aligned}$$

where

$$\begin{aligned}&\mathcal{C}(c).\texttt{fields} = \{f_1 : o_1, \cdots, f_k : o_k\} \\ &\mathcal{C}(c).\texttt{methods} = \{\ m_1 : \rho_1, \cdots, m_m : \rho_m\} \\ &\mathcal{C}(c).\texttt{ancestors} = \{c_1, \ldots, c_n\}\end{aligned}$$

The encoding of the class hierarchy described in $\mathcal{C}$ is given by the following mutually recursive type equations.

$$\begin{aligned}&\texttt{type}\ c_1 = \mathcal{C}[\![c_1]\!] \\ &\qquad\vdots \\ &\texttt{and}\ \ c_n = \mathcal{C}[\![c_n]\!]\end{aligned}$$

The type system of the interoperable calculus is defined with respect to this set of recursive type equations. Different from `datatype` declarations in Standard ML, these are treated as *transparent equations*, i.e. c_i is equal to the corresponding object type.

3.4 The type system

The type system is given based on the kinded type system [18] for record polymorphism. The set of *monotypes* (ranger over by τ), *kinds* (ranged over by k), and *polytypes* (ranged

over by σ) are given by the following abstract syntax.

$$\begin{array}{lcl} \tau & ::= & t \mid o \mid \rho \mid \tau \to \tau \mid \{l:\tau, \cdots, l:\tau\} \\ k & ::= & U \mid \{\!\{l:\tau, \cdots, l:\tau\}\!\} \\ \sigma & ::= & \tau \mid \forall(t::k, \cdots, t::k).\sigma \end{array}$$

t stands for a given countably infinite set of *type variables*. U is the *universal kind*, denoting the set of all monotypes. $\{\!\{l:\tau, \cdots, l:\tau\}\!\}$ is an object kind (corresponding to a record kind in [18]) denoting the set of all object types containing the designated fields. $\forall(t::k, \cdots, t::k).\sigma$ is a kinded abstract types whose instances are restricted to those that satisfy kind constraints.

Types are considered modulo type equivalence defined in a given $\mathcal{C}$. In the following development, we generally assume type equations implicitly, and use type name c_i for the declared object type. When we need to mention type equivalent explicitly, we write $\mathcal{C} \vdash c = \tau$ if c is equal to τ under $\mathcal{C}$.

The calculus does not have any constructor for method type ρ or a tuple type $o_1 \times \cdots \times o_n$. These are only used in external object manipulation, and ρ appearing outside of object types are useless.

On the set of object types, we define the following shallow *subtype relation*.

$$\tau < \tau$$

$$\{l_1:\tau_1, \cdots, l_n:\tau_n, \cdots\} < \{l_1:\tau_1, \cdots, l_n:\tau_n\}$$

Different from the ordinary subtyping such as those [16, 2, 3] and subsequent studies on subtyping, it indicates inclusion of fields of object types, and it is identity on all the other types.

The following definitions on kinding are taken from [18].

The set of *free type variables* of a type σ or a kind k are denoted by $FTV(\sigma)$ and $FTV(k)$, respectively. A *kind assignment*, ranged over by $\mathcal{K}$, is a mapping from a finite set of type variables to kinds. A kind assignment $\mathcal{K}$ is *well formed* if for all $t \in dom(\mathcal{K})$, $FTV(\mathcal{K}(t)) \subseteq dom(\mathcal{K})$, where $dom(f)$ denotes the domain of function f. We implicitly assume that any kind assignment appearing in the rest of the development is well formed. A type σ has a kind k under $\mathcal{K}$, denoted by $\mathcal{K} \vdash \sigma :: k$, if it is derivable by the following set of kinding rules.

$$\mathcal{K} \vdash \tau :: U$$

$$\mathcal{K} \vdash t :: k \quad \text{if } \mathcal{K}(t) = k$$

$$\mathcal{K} \vdash \{l_1:\tau_1, \cdots, l_n:\tau_n\} :: \{\!\{l_1:\tau_1, \cdots, l_n:\tau_n\}\!\}$$

$$\frac{\mathcal{K} \vdash \tau :: \{\!\{l_1:\tau_1, \cdots, l_n:\tau_n, \ldots\}\!\}}{\mathcal{K} \vdash \tau :: \{\!\{l_1 :: \tau_1, \cdots, l_n:\tau_n\}\!\}}$$

A substitution S is *well formed under a kind assignment* $\mathcal{K}$ if for any $t \in dom(S)$, $S(t)$ is well formed under $\mathcal{K}$. A *kinded substitution* is a pair $(\mathcal{K}, S)$ of a kind assignment $\mathcal{K}$ and a substitution S that is well formed under $\mathcal{K}$. The kind assignment $\mathcal{K}$ in $(\mathcal{K}, S)$ specifies kind constraints of the *result* of the substitution. A kinded substitution $(\mathcal{K}, S)$ is *ground* if $\mathcal{K} = \emptyset$. We usually write S for a ground kinded substitution $(\emptyset, S)$. A kinded substitution $(\mathcal{K}_1, S)$ *respects* a kind assignment $\mathcal{K}_2$ if for any $t \in dom(\mathcal{K}_2)$, $\mathcal{K}_1 \vdash S(t) :: S(\mathcal{K}_2(t))$. Let σ_1 be a polytype well formed under $\mathcal{K}$. We say that σ_2 is a *generic instance* of σ_1 under $\mathcal{K}$, written $\mathcal{K} \vdash \sigma_1 \geq \sigma_2$, if $\sigma_1 = \forall(t_1^1::k_1^1, \cdots, t_n^1::k_n^1).\tau_1$, $\sigma_2 = \forall(t_1^2::k_1^2, \cdots, t_m^2::k_m^2).\tau_2$, and there is a substitution S such that $dom(S) = \{t_1^1, \cdots, t_n^1\}$, $(\mathcal{K}\{t_1^2::k_1^2, \cdots, t_m^2::k_m^2\}, S)$ respects $\mathcal{K}\{t_1^1::k_1^1, \cdots, t_n^1::k_n^1\}$ and $\tau_2 = S(\tau_1)$.

A *typing environment*, ranged over by $\mathcal{T}$, is a mapping from a finite set of variables to polytypes. We write $\mathcal{T}\{x:\sigma\}$ for the type environment $\mathcal{T}'$ such that $dom(\mathcal{T}') = dom(\mathcal{T}) \cup \{x\}$, $\mathcal{T}'(x) = \sigma$, and $\mathcal{T}'(y) = \mathcal{T}(y)$ for all $y \in dom(\mathcal{T}'), y \neq x$. The *closure of* τ *under* $\mathcal{T}, \mathcal{K}$, denoted by $Cls(\mathcal{K}, \mathcal{T}, \tau)$, is a pair $(\mathcal{K}', \forall(t_1::k_1, \cdots, t_n::k_n).\tau)$ s.t. $\mathcal{K}'\{t_1::k_1, \cdots, t_n::k_n\} = \mathcal{K}$ and $\{t_1, \cdots, t_n\}$ is the set of type variables free in τ under $\mathcal{K}$ but not free in $\mathcal{T}$ under $\mathcal{K}$.

Using these notations, the type system is defined as a proof system to derive a judgment of the form

$$\mathcal{K}, \mathcal{T} \rhd e : \tau$$

indicating the fact that e has type τ under $\mathcal{T}$ and $\mathcal{K}$. The set of typing rules is given in Figure 2.

The combination of our object representation, record polymorphism and object subsumption provides the sufficient power to use objects and methods within an ML-style polymorphic calculus. The typing examples shown in Figure 1 are indeed derivable typings in this type system.

$$\mathcal{K}, \mathcal{T} \rhd c^b : b$$

$$\frac{\mathcal{K} \vdash \sigma \geq \tau}{\mathcal{K}, \mathcal{T}\{x:\sigma\} \rhd x : \tau}$$

$$\frac{\mathcal{K}, \mathcal{T}\{x:\tau_1\} \rhd e : \tau_2}{\mathcal{K}, \mathcal{T} \rhd \lambda x.e : \tau_1 \to \tau_2}$$

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

$$\frac{\begin{array}{l}\mathcal{K}, \mathcal{T} \rhd e_1 : \tau_1 \\ Cls(\mathcal{K}, \mathcal{T}, \tau_1) = (\mathcal{K}', \sigma) \\ \mathcal{K}', \mathcal{T}\{x:\sigma\} \rhd e_2 : \tau_2\end{array}}{\mathcal{K}, \mathcal{T} \rhd \texttt{let val } x \texttt{ = } e_1 \texttt{ in } e_2 \texttt{ end} : \tau_2}$$

$$\frac{\mathcal{K}, \mathcal{T} \rhd e_0 : \texttt{bool} \quad \mathcal{K}, \mathcal{T} \rhd e_1 : \tau \quad \mathcal{K}, \mathcal{T} \rhd e_2 : \tau}{\mathcal{K}, \mathcal{T} \rhd \texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 : \tau}$$

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

$$\frac{\begin{array}{l}\mathcal{K}, \mathcal{T} \rhd e : \tau \\ \mathcal{K} \vdash \tau :: \{\!\{m:\tau_1 \times \cdots \times \tau_n \to \tau_0\}\!\} \\ \mathcal{K}, \mathcal{T} \rhd e_i : \tau_i \quad (1 \leq i \leq n)\end{array}}{\mathcal{K}, \mathcal{T} \rhd e.m(e_1, \cdots, e_n) : \tau_0}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash \texttt{new} : o_1 \times \cdots \times o_n \to o \in c \\ \mathcal{K}, \mathcal{T} \rhd e_i : o_i \quad (1 \leq i \leq n)\end{array}}{\mathcal{K}, \mathcal{T} \rhd \texttt{new } c(e_1, \cdots, e_n) : o}$$

$$\frac{\mathcal{K}, \mathcal{T} \rhd e : \tau_1 \quad \tau_1 < \tau_2}{\mathcal{K}, \mathcal{T} \rhd e : \tau_2}$$

Figure 2: The type system of the interoperable calculus

3.5 Type inference

One of the goal of designing this calculus is to preserve the benefit of ML-style polymorphic type inference. For

this purpose, we need to develop a type inference algorithm. We use the kinded unification algorithm [18] for record type inference.

A *kinded set of equations* is a pair $(\mathcal{K}, E)$ consisting of a kind assignment $\mathcal{K}$ and a set E of pairs of types such that E is well formed under $\mathcal{K}$. We say that a substitution S *satisfies* E if $S(\tau_1) = S(\tau_2)$ for all $(\tau_1, \tau_2) \in E$. A kinded substitution $(\mathcal{K}_1, S)$ is a *unifier* of a kinded set of equations $(\mathcal{K}, E)$ if it respects $\mathcal{K}$ and if S satisfies E. $(\mathcal{K}_1, S)$ is a *most general unifier* of $(\mathcal{K}_2, E)$ if it is a unifier of $(\mathcal{K}_2, E)$ and if for any unifier $(\mathcal{K}_3, S_2)$ of $(\mathcal{K}_2, E)$ there is some substitution S_3 such that $(\mathcal{K}_3, S_3)$ respects $\mathcal{K}_1$ and $S_2 = S_3 \circ S$. The following is shown in [18].

THEOREM 1. *The algorithm $\mathcal{U}$ takes any kinded set of equations, computes a most general unifier if one exists, and reports failure otherwise.*

In order to infer types under the implicit subtyping on objects, we also need the following two algorithms $\mathcal{S}$ and $\mathcal{F}$ for subtype checking and for the least upper bound computation:

- $\mathcal{S}(\mathcal{K}, \tau_1, \tau_2)$ returns $(\mathcal{K}', S)$ such that $\tau_0 = S(\tau_1) < S(\tau_2)$ if such substitution exists otherwise it returns *failure*.
- $\mathcal{F}(\mathcal{K}, \tau_1, \tau_2)$ returns $(\mathcal{K}', \tau_0, S)$ such that (1) τ_0 is the least one satisfying $\tau_0 < S(\tau_1)$ and $\tau_0 < S(\tau_2)$, and (2) τ_0 contains at least one class label, if such $\mathcal{K}, S, \tau_o$ exist, otherwise it returns *failure*.

These algorithms are given in Figure 3.

Using these auxiliary algorithms, we define a type inference algorithm $\mathcal{WK}$ which takes $(\mathcal{K}, \mathcal{T}, e)$ and returns $(\mathcal{K}', S, \tau)$ or *failure*. The algorithm is given in Figure 4. For this algorithm, we can show the soundness property similarly to the corresponding theorem in [18].

This algorithm is, however, not complete; there is a typable term for which the type inference algorithm reports failure. This occurs in combination of first-class functions and object subsumption. To show a typical counter example, we assume that $\mathcal{C} \vdash c <: c'$, $\mathcal{C} \vdash \texttt{new} : int \rightarrow \ c \in c$, and $\mathcal{C} \vdash \texttt{new} : int \rightarrow \ c' \in c'$. Then the following term can be typable but the algorithm reports failure.

$$(\lambda x.z\ (y\ x)\ (y\ (\texttt{new}\ c(0)))\ (y\ x))\ (\texttt{new}\ c'(0))$$

The problem is that type inference algorithm only consider possibility of applying subsumption rule for concrete object types but not for type variables. As a result, the type of x is (prematurely) unified to c.

This incompleteness would have been a severe limitation if subsumption rule were the source of polymorphism of method application. In our calculus, however, method invocation $(e.m(e_1, \ldots, e_n))$ is given polymorphic typing through kinded record polymorphism, and the subsumption rule is introduced for typing heterogeneous collections. We believe that typable terms that are rejected by our conservative type inference are negligible.

A trivial way to recover completeness of type inference is to eliminate the subsumption rule in the type system, and to modify the type inference algorithm so that $\mathcal{S}$ and $\mathcal{F}$ return failure if the two types are not unifiable. The resulting type system is still strong enough to interfacing with an object-oriented language. In fact, our prototype implementation does not contain subsumption rule for a different reason, which we shall discuss in Section 6, and it still demonstrates its expressiveness. The only missing feature is heterogeneous collections.

A more systematic way to recover completeness of type inference retaining the subsumption rule would be to perform type inference modulo subclass relation similarly to those in [16, 8, 21]. The rationale of not pursuing this direction is twofold: (1) it significantly complicates the type inference system and makes inferred types difficult to understand, and (2) the extra generality is negligible.

We believe that the current definition is a good compromise for adding the feature of heterogeneous collections without introducing much complication to the underlying ML-style type inference system.

$\mathcal{S}(\mathcal{K}, \{F_1\}, \{F_2\}) =$
  if $dom(F_1) \subseteq dom(F_2)$ then
    $\mathcal{U}(\mathcal{K}, \{(F_1(l), F_2(l)) | l \in dom(F_1)\})$
  else $failure$

$\mathcal{S}(\mathcal{K}, \tau_1, \tau_2) = \mathcal{U}(\mathcal{K}, \{(\tau_1, \tau_2)\})$

$\mathcal{F}(\mathcal{K}, \{F_1\}, \{F_2\}) =$
  let $L = dom(F_1) \cap dom(F_2)$
    $(\mathcal{K}_1, S_1) = \mathcal{U}(\mathcal{K}, \{(F_1(l), F_2(l)) | l \in L\})$
    $F_3 =$ the restriction of F_1 on L
  in if L does not contain class label then
      $failure$
    else $(\mathcal{K}_1, S_1, S_1(\{F_3\}))$

$\mathcal{F}(\mathcal{K}, \tau_1, \tau_2) =$
  let$(\mathcal{K}, S) = \mathcal{U}(\mathcal{K}, \{(\tau_1, \tau_2)\})$
  in$(\mathcal{K}, S, S(\tau_1))$

Figure 3: Subtyping checking and least upper bound computation

4. A SAMPLE OBJECT-ORIENTED LANGUAGE

We believe that the proposed calculus serves as a model of a language that achieves type safe interoperability with various object-oriented languages. To demonstrate this, we define a sample object-oriented language. It is not our purpose to study type theoretical issues in object-oriented languages but to define a language that is expressive enough to study the issues in interfacing with other systems. To this purpose, we believe a language with the following features is sufficient:

- a class system with multiple inheritance, and
- dynamic method dispatch.

Since the interoperable primitives only depends on class structures in its static semantics, and on method dispatch in its dynamic semantics, the results we shall establish with the sample language should be applicable to other object-oriented languages with various features such as interface definitions.

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

$\mathcal{WK}(\mathcal{K},\mathcal{T},e_1\ e_2) =$
  let $(\mathcal{K}_1,S_1,\tau_1) = \mathcal{WK}(\mathcal{K},\mathcal{T},e_1)$
    $(\mathcal{K}_2,S_2,\tau_2) = \mathcal{WK}(\mathcal{K}_1,S_1(\mathcal{T}),e_2)$
    $(\mathcal{K}_3,S_3) = \mathcal{U}(\mathcal{K}_2,\{(S_2(\tau_1),t \to s)\})$ $(t,s$ fresh$)$
    $(\mathcal{K}_4,S_4) = \mathcal{S}(\mathcal{K}_3,S_3(\tau_2),S_3(t))$
  in $(\mathcal{K}_4,S_4 \circ S_3 \circ S_2 \circ S_1,S_4 \circ S_3(s))$

$\mathcal{WK}(\mathcal{K},\mathcal{T},\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}) =$
  let $(\mathcal{K}_1,S_1,\tau_1) = \mathcal{WK}(\mathcal{K},\mathcal{T},e_1)$
    $(\mathcal{K}'_1,\sigma_1) = Cls(\mathcal{K}_1,S_1(\mathcal{T}),\tau_1)$
    $(\mathcal{K}_2,S_2,\tau_2) = \mathcal{WK}(\mathcal{K}'1,(S_1(\mathcal{T}))\{x : \sigma_1\},e_2)$
  in $(\mathcal{K}_2,S_2 \circ S_1,\tau_2)$

$\mathcal{WK}(\mathcal{K},\mathcal{T},\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2) =$
  let $(\mathcal{K}_0,S_0,\tau_0) = \mathcal{WK}(\mathcal{K},\mathcal{T},e_0)$
    $(\mathcal{K}_1,S_1) = \mathcal{U}(\mathcal{K}_0,\{(\tau_0,\texttt{bool})\})$
    $(\mathcal{K}_2,S_2,\tau_1) = \mathcal{WK}(\mathcal{K}_1,S_1 \circ S_0(\mathcal{T}),e_1)$
    $(\mathcal{K}_3,S_3,\tau_2) = \mathcal{WK}(\mathcal{K}_2,S_2 \circ S_1 \circ S_0(\mathcal{T}),e_2)$
    $(\mathcal{K}_4,S_4,\tau) = \mathcal{F}(\mathcal{K}_\exists,\tau_1,\tau_2)$
  in $(\mathcal{K}_4,S_4 \circ S_3 \circ S_2 \circ S_1 \circ S_0,\tau)$

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

$\mathcal{WK}(\mathcal{K},\mathcal{T},e.m(e_1,\cdots,e_n)) =$
  let $(\mathcal{K}_0,S_0,\tau_0) = \mathcal{WK}(\mathcal{K},\mathcal{T},e)$
    $(\mathcal{K}_1,S_1,\tau_1) = \mathcal{WK}(\mathcal{K}_0,S_0(\mathcal{T}),e_1)$
    $(\mathcal{K}_2,S_2,\tau_2) = \mathcal{WK}(\mathcal{K}_1,S_1 \circ S_0(\mathcal{T}),e_2)$
      $\vdots$
    $(\mathcal{K}_n,S_n,\tau_n) = \mathcal{WK}(\mathcal{K}_{n-1},S_{n-1} \circ \cdots \circ S_0(\mathcal{T}),e_n)$
    $(\mathcal{K}_{n+1},S_{n+1}) = \mathcal{U}(\mathcal{K}_n\{t :: U,s :: \{\!\{m : t\}\!\}\},$
        $\{(s,S_n \circ \cdots \circ S_1(\tau_0))\})(t,s$ fresh$)$
    $(\mathcal{K}_{n+2},S_{n+2}) =$
      $\mathcal{U}(\mathcal{K}_{n+1},\{(S_{n+1}(t),$
          $S_{n+1} \circ \cdots \circ S_2(\tau_1) \times$
          $\cdots \times S_{n+1}(\tau_n) \to r)\})$
        $(r$ fresh$)$
  in $(K_{n+2},S_{n+2} \circ \cdots \circ S_0,S_{n+2}(r))$

$\mathcal{WK}(\mathcal{K},\mathcal{T},\texttt{new}\ c(e_1,\cdots,e_n)) =$
  let $(\mathcal{K}_1,S_1,\tau_1) = \mathcal{WK}(\mathcal{K},\mathcal{T},e_1)$
    $(\mathcal{K}_2,S_2,\tau_2) = \mathcal{WK}(\mathcal{K}_1,S_1(\mathcal{T}),e_2)$
      $\vdots$
    $(\mathcal{K}_n,S_n,\tau_n) = \mathcal{WK}(\mathcal{K}_{n-1},S_{n-1} \circ \cdots \circ S_1(\mathcal{T}),e_n)$
    $(K_{n+1},S_{n+1}) =$
      $\mathcal{U}(\mathcal{K}_n,\{(\rho_0,$
          $S_n \circ \cdots \circ S_2(\tau_1) \times \cdots \times \tau_n \to t)\})$
        where $\mathcal{C} \vdash \texttt{new} : \rho_0 \in c$ and t fresh
  in $(K_{n+1},S_{n+1} \circ \cdots \circ S_1,S_{n+1}(t))$

Figure 4: Type inference algorithm (excerpts)

4.1 The syntax and the type system

We consider the language given by the following syntax.

$$
\begin{array}{rcl}
P & ::= & \texttt{classdef}\ C;\cdots;C\ \texttt{in}\ e \\
C & ::= & \texttt{class}\ c\ \texttt{extends}\ \{c,\ldots,c\}\ \texttt{=}\ F\ \texttt{with}\ M \\
F & ::= & \{f : o,\ldots,f : o\} \\
M & ::= & \{m : \rho = \lambda(x,\ldots,x).e,\ldots,m : \rho = \lambda(x,\ldots,x).e\} \\
e & ::= & c^b \mid x \mid \texttt{this} \mid e.f \mid e.m(e,\ldots,e) \mid \texttt{new}\ c(e,\ldots,e)
\end{array}
$$

A program is a sequence of class definitions $C_1;\cdots;C_n$ with a main program e. Each class definition consists of superclass declarations, field declarations F and method declarations M. A method body can use a special variable `this` which will be bound to the receiver object.

There are several potentially subtle issues in defining a type system for this calculus regarding the interaction between multiple inheritance, method overriding and the `this` variable. Although they are important in designing a sound type system for expressive object-oriented language, they are internal in typing a method and do not affect typing issues in interfacing with another language. For this reason, we do not consider these issues and make the following simplifying assumptions.

- The relation induced by `extends` is acyclic.
- The set of fields of a class contains all the fields defined in its superclasses.
- Overriding fields and methods have the same types.
- No name conflict occurs among superclasses.
- The constructor takes the set of field values in a fixed order determined by some predefined ordering on field names.

Under these assumptions, a set of class definition determines the following class environment $\mathcal{C}$.

1. The subclass relation $\mathcal{C} \vdash c <: c'$ is determined by the following rules.

$$\mathcal{C} \vdash c <: c$$

$$\frac{\text{“}\texttt{class}\ c_1\ \texttt{extends}\ \{\cdots c_2 \cdots\}\cdots\text{”} \in \mathcal{C}}{\mathcal{C} \vdash c_1 <: c_2}$$

$$\frac{\mathcal{C} \vdash c_1 <: c_2 \quad \mathcal{C} \vdash c_2 <: c_3}{\mathcal{C} \vdash c_1 <: c_3}$$

2. $\mathcal{C}(c).\texttt{fields}$ is the set of fields declared for c. By the assumption, if $\mathcal{C} \vdash c_1 <: c_2$ then

$$\mathcal{C}(c_1).\texttt{fields} \supseteq \mathcal{C}(c_2).\texttt{fields}$$

3. $\mathcal{C}(c).\texttt{methods}$ is the union of the sets of methods declared for c' such that $\mathcal{C} \vdash c <: c'$. By our assumption, if a method m is defined in both c and c' then there types must be the same.
4. $\mathcal{C} \vdash \texttt{new} : o_1 \times \cdots \times o_n \to c \in c$ where $o_1,\cdots,o_n$ is the types of fields in $\mathcal{C}(c).\texttt{fields}$ ordered by a predefined ordering on the labels.

Based on these assumptions, we define typing rules for expressions, method and each class definition *relative to a*

$$\mathcal{T} \rhd x : o \quad \text{if } x : o \in \mathcal{T}$$

$$\mathcal{T} \rhd c^b : b$$

$$\frac{\mathcal{T} \rhd e : c \quad \mathcal{C} \vdash f : o \in c}{\mathcal{T} \rhd e.f : o}$$

$$\frac{\mathcal{T} \rhd e : c \quad \mathcal{C} \vdash m : o_1 \times \cdots \times o_n \to o \in c \quad \mathcal{T} \rhd e_i : o_i}{\mathcal{T} \rhd e.m(e_1, \ldots, e_n) : o}$$

$$\frac{\mathcal{C}(c).\texttt{fields} = \{f_1 : o_1, \ldots, f_n : o_n\} \quad \mathcal{T} \rhd e_i : o_i \ \ (1 \le i \le n)}{\mathcal{T} \rhd \texttt{new}\ c(e_1, \ldots, e_n) : c}$$

$$\frac{\mathcal{T} \rhd e : c \quad \mathcal{C} \vdash c <: c'}{\mathcal{T} \rhd e : c'}$$

Figure 5: Expression typing in a sample object-oriented language

$$\delta \vdash_o (h, x) \Downarrow (h, \delta(x))$$

$$\delta \vdash_o (h, c^b) \Downarrow (h, c^b)$$

$$\frac{\delta \vdash_o (h, e) \Downarrow (h', p) \quad h'(p) = \langle \ldots, f = v, \ldots \rangle_c}{\delta \vdash_o (h, e.f) \Downarrow (h', v)}$$

$$\frac{\begin{array}{l}\delta \vdash_o (h, e) \Downarrow (h_1, p) \\ h_1(p) = \langle F \rangle_c \\ \mathcal{C} \vdash m : \rho = \lambda(x_1, \ldots, x_n).e_0 \in c \\ \delta \vdash_o (h_i, e_i) \Downarrow (h_{i+1}, v_i) \ \ (1 \le i \le n) \\ \delta\{\texttt{this} : p, x_1 : v_1, \ldots, x_n : v_n\} \vdash_o (h_{n+1}, e_0) \Downarrow (h_0, v)\end{array}}{\delta \vdash_o (h, e.m(e_1, \ldots, e_n)) \Downarrow (h_0, v)}$$

$$\frac{\begin{array}{l}\delta \vdash_o (h_{i-1}, e_i) \Downarrow (h_i, v_i) \ \ (1 \le i \le n) \\ \mathcal{C}(c).\texttt{fields} = \{f_1 : o_1, \ldots, f_n : o_n\} \\ h'_n = h_n\{p \mapsto \langle f_1 = v_1, \ldots, f_n = v_n \rangle_c\} \ (p \text{ fresh})\end{array}}{\delta \vdash_o (h_0, \texttt{new}\ c(e_1, \ldots, e_n)) \Downarrow (h'_n, p)}$$

Figure 6: An operational semantics of the object-oriented language

given class environment $\mathcal{C}$. We then define a type-checking rule for a program.

The type system of expressions is given in Figure 5. Under a given class environment $\mathcal{C}$, a method definition of the form

$$m : o_1 \times \cdots \times o_n \to o = \lambda(x_1, \ldots, x_n).e$$

is type correct under $\mathcal{C}$ if

$$\{\texttt{this} : c, x_1 : o_1, \ldots, x_n : o_n\} \rhd e : o$$

is derivable under $\mathcal{C}$.

A class definition of the form

$$\texttt{class}\ c\ \texttt{extends}\ \{c_1, \ldots, c_n\}\ \texttt{=}\ F\ \texttt{with}\ M$$

is type correct if all the methods in M are type correct.

We now define the type-checking rule for programs. A program

$$\texttt{classdef}\ C_1; \cdots; C_n\ \texttt{in}\ e$$

is type correct if the following conditions are met.

- The set of class definitions $C_1; \cdots; C_n$ is well formed (i.e. it satisfies the set of assumptions stated earlier), yielding a class environment $\mathcal{C}$.
- each class definition in C_i is type correct under $\mathcal{C}$, and
- e is type correct under $\mathcal{C}$, i.e. $\emptyset \rhd e : o$ is derivable for some o under $\mathcal{C}$.

4.2 Operational semantics and type soundness

In order to model realistic interoperability, we define an operational semantics of this language using heaps (ranged over by h) and heap addresses (ranged over by p). A heap is a mapping from a finite set of heap address to heap values. Heaps, heap values (ranged over by v_h) and runtime values (ranged over by v) are given as follows.

$$\begin{array}{rcl} h & ::= & \{p \mapsto v_h, \ldots, p \mapsto v_h\} \\ v_h & ::= & c^b \mid \langle f = v, \ldots, f = v \rangle_c \\ v & ::= & c^b \mid p \mid \textit{wrong} \end{array}$$

where $\langle f = v, \ldots, f = v \rangle_c$ is an object whose runtime type is c, and *wrong* denotes runtime error. We write $h\{p \mapsto v\}$ for the extension of h with $\{p \mapsto v\}$. A similar notation is used for H.

The operational semantics is defined as an evaluation relation of the form

$$\delta \vdash_o (h, e) \Downarrow (h', v)$$

indicating the fact that expression e yields a value v and a new heap h' when evaluated under environment δ and heap h. The set of evaluation rules is given in Figure 6.

This set of rules is taken with the implicit rules saying that the evaluation yields *wrong* if the conditions specified in a rule are not met or evaluation of some of its subexpressions yields *wrong*.

Runtime values may form cycles and sharing through object pointers. To define value typing without resorting to co-induction, we follow [11] and define types of values relative to a *heap type* (ranged over by H) specifying the structure of a heap, which is a function from a finite set of heap addresses to types. The following three relations determine value typing.

- $\models_o h : H$ (heap h satisfies heap type H.)
  if $dom(H) = dom(h)$, and for each $p \in dom(h)$, one of the following holds:
  1. $h(p) = c^b$ and $H(p) = b$,
  2. $H(p) = c$, $h(p) = \langle f_1 = v_1, \ldots, f_n = v_n, \ldots \rangle_{c'}$, $\mathcal{C} \vdash c' <: c$, $\mathcal{C}(c).\texttt{fields} = \{f_1 : o_1, \ldots, f_n : o_n\}$, and $H \models_o v_i : o_i$ $(1 \le i \le n)$.
- $H \models_o v : o$ (v has type o under H)
  if either $H \models_o c^b : b$ or $H \models_o p : c$ for some c such that $\mathcal{C} \vdash H(p) <: c$.
- $H \models_o \delta : \mathcal{T}$ (δ is a model of $\mathcal{T}$ under H)
  if $dom(\delta) = dom(\mathcal{T})$, and for all $x \in dom(\delta)$ $H \models_o \delta(x) : \mathcal{T}(x)$.

A heap type H' is an *extension* of H if $dom(H') \supseteq dom(H)$ and $H'(p) = H(p)$ for $p \in dom(H)$. The following theorem shows soundness of the type system with respect to this semantics.

THEOREM 2. *If $\mathcal{T} \rhd e : o$, $\models_o h : H$, $H \models_o \delta : \mathcal{T}$ then if $\delta \vdash_o (h, e) \Downarrow (h', v)$ then there is an extension H' of H such that $\models_o h' : H'$, $H' \models_o v : o$.*

Proof. The proof is by induction on the length of evaluation steps using the following simple lemma.

LEMMA 1. *1. If $H \models_o v : o$ and H' is an extension of H then $H' \models_o v : o$.*

2. If $H \models_o v : o$ and $\mathcal{C} \vdash o <: o'$ then $H \models_o v : o'$.

The proof of the theorem proceeds by cases in term of the rule used in the last evaluation step.

Case $e \equiv e_1.f$. By the evaluation rule, $\delta \vdash_o (h, e_1) \Downarrow (h', v_0)$ for some v_0. By the typing rule, there are some c, o' such that $\mathcal{T} \rhd e_1 : c$, $\mathcal{C} \vdash f : o' \in c$ and $\mathcal{C} \vdash f : o' <: o$. By the induction hypothesis, there is some extension H' of H such that $\models_o h' : H'$ and $H' \models_o v_0 : c$. By the definition of runtime typing, $v_0 = p$ such that $\mathcal{C} \vdash H'(p) <: c$. Since $\mathcal{C} \vdash f : o' \in c$ and the property of $\models_o h' : H'$, $h'(p)$ must be an object containing $f = v$ such that $H' \models_o v : o'$. Then by lemma we have $H' \models_o v : o$ as desired.

Case $e \equiv \texttt{new}\ c(e_1, \ldots, e_n)$. By the typing rule, $\mathcal{C}(c).\texttt{fields} = \{f_1 : o_1, \ldots, f_n : o_n\}$ for some types $o_1, \ldots, o_n$. By the evaluation rule, $\delta \vdash_o (h, e_i) \Downarrow (h_i, v_i)$ for some v_i $(1 \leq i \leq n)$. By induction hypothesis and Lemma 1, there are $H \subseteq H_1 \subseteq \cdots \subseteq H_n$ such that $\models_o h_i : H_i$, $H_i \models_o v_i : o_i$. Let $h'_n = h_n\{p \mapsto \langle f_1 = v_1, \ldots, f_n = v_n\rangle_c\}$ (p fresh). We can then take $H' = H_n\{p \mapsto c\}$, for which $\models_o h' : H'$ and $H' \models_o p : c$ by the definition of value and heap typing. Since $\mathcal{C} \vdash c <: o$, by lemma $H' \models_o p : o$ as desired.

The cases for variables and constants are trivial. The case for method invocation can be shown by combining the techniques of the above two cases. □

We define value typing and heap typing as follows.

- $h \models_o v : o$ (v has type o under h)
  if there is some H such that $\models_o h : H$ and $H \models_o v : o$.
- $h \models_o \delta : \mathcal{T}$ (δ satisfies $\mathcal{T}$ under h)
  if there is some H such that $\models_o h : H$ and $H \models_o \delta : \mathcal{T}$.

Then the above theorem yields the following.

COROLLARY 1. *If $\mathcal{T} \rhd e : o$ and $h \models_o \delta : \mathcal{T}$ then if $\delta \vdash_o (h, e) \Downarrow (h', v)$ then $h' \models_o v : o$.*

5. SEMANTICS AND TYPE SOUNDNESS OF THE INTEROPERABLE CALCULUS

We now give a formal operational semantics of our interoperable calculus with the sample object-oriented language as the target language for interoperability, and establish type soundness of the calculus.

5.1 Operational semantics

We write $\langle X\rangle$ to emphasize that X is an entity in the object-oriented language.

The set of runtime values (ranged over by V) is given by the following syntax.

$$V ::= c^b \mid \langle p\rangle \mid cls(\Delta, \lambda x.M)$$

We assume that constants (c^b) of base types are implicitly marshaled. $\langle p\rangle$ is a heap address in the external object-oriented language. $cls(\Delta, \lambda x.M)$ represents a function closure. Δ is a runtime environment, which is a mapping from a finite set of variables to runtime values.

The operational semantics is defined by specifying the set of evaluation rules of the form:

$$\Delta \vdash_\lambda (\langle h\rangle, M) \Downarrow (\langle h'\rangle, V)$$

indicating the fact that under environment Δ and external heap $\langle h\rangle$, expression M evaluates to value V yielding a modified heap $\langle h'\rangle$. The set of rules is given in Figure 7.

5.2 Value typing and type soundness

Typing of values and runtime environments are given as follows.

- $H \models_p c^b : b$
- $H \models_p \langle p\rangle : o$ if $H \models_o p : o$
- $H \models_p cls(\Delta, \lambda x.M) : \tau_1 \to \tau_2$
  if there is some $\mathcal{T}$ such that $H \models_p \Delta : \mathcal{T}$ and $\mathcal{T} \rhd \lambda x.M : \tau_1 \to \tau_2$.
- $H \models_p V : \sigma$
  if $H \models_p V : \tau$ for any kind-respecting instance τ of σ.
- $H \models_p \Delta : \mathcal{T}$
  if $dom(\Delta) = dom(\mathcal{T})$ and for all $x \in dom(\Delta)$, $H \models_p \Delta(x) : \mathcal{T}(x)$.

These are relative to a given class environment $\mathcal{C}$.

We can now show the following type soundness theorem.

THEOREM 3. *If $\mathcal{K}, \mathcal{T} \rhd M : \tau$, S is a $\mathcal{K}$-respecting ground substitution, $\models_o h : H$, $H \models_p \Delta : S(\mathcal{T})$ and $\Delta \vdash_\lambda (\langle h\rangle, M) \Downarrow (\langle h'\rangle, V)$ then there is some extension H' of H such that $\models_o h' : H'$ and $H' \models_p V : S(\tau)$.*

Proof. The proof is by induction on the length of evaluation steps using Theorem 1. Proof proceeds by cases in terms of the rule used in the last evaluation step. We only show the case for external method invocation. The other cases are simpler.

Case $M \equiv M_0.m(M_1, \ldots, M_n)$.
Suppose $\mathcal{K}, \mathcal{T} \rhd M_0.m(M_1, \ldots, M_n) : \tau$. Then $\emptyset, S(\mathcal{T}) \rhd M_0.m(M_1, \ldots, M_n) : S(\tau)$. By the typing rule, $\emptyset, S(\mathcal{T}) \rhd M_0 : \tau_0$ such that $\emptyset \vdash \tau_0 :: \{\!\{m : o_1 \times \cdots \times o_n \to o\}\!\}$, and $\emptyset, S(\mathcal{T}) \rhd M_i : o_i$. τ_0 is a ground type containing a field $m : o_1 \times \cdots o_n \to o$. By the definition of the type system, the only possible τ_0 satisfying this condition is some c such that $\mathcal{C} \vdash c = \tau$. By the evaluation relation, $\Delta \vdash_\lambda (\langle h\rangle, M_0) \Downarrow (\langle h_1\rangle, V_0)$. By induction hypothesis, there is some $H_0 \supseteq H$ such that $\models_o h' : H_0$, and $H_0 \models_p V_0 : c$. By the definition of value typings, $V_0 = p_0$ for some p_0 such that $H' \models_o p_o : c$. By evaluation relation and induction hypothesis, we can show that there are some $H_i (1 \leq i \leq n)$ such that $H_0 \subseteq H_1 \subseteq \cdots \subseteq H_n$, $\models_o h_i : H_i$, and $H_i \models_p V_i : o_i$. By the definition of value typing, either $V_i = c^b$ and $o = b$, or $V_i = \langle p\rangle$, $o_i = c_i$, and $H_i \models_o p_i : c_i$. Let $v_i = p_i$ if $V_i = p_i$ otherwise $v_i = V_i$. Then $H_{n+1} \models_o \{x : p, x_1 : v_1, \ldots, x_n : v_n\} : \{x : o, x_1 : o_1, \ldots, x_n : o_n\}$. By the encoding of c in the record calculus, we must have $\mathcal{C} \vdash m : o_1 \times \cdots \times o_n \to o \in c$, and therefore $\{x : o, x_1 : o_1, \ldots, x_n : o_n\} \rhd x.m(x_1, \ldots, x_n) : o$. By evaluation relation, $\{x : p, x_1 : p_1, \ldots, x_n : p_n\} \vdash_o (h_n, x.m(x_1, \cdots, x_n)) \Downarrow (h_{n+1}, p)$. By Theorem 1, there is some extension H_{n+1} of H_n such that $\models_o h_{n+1} : H_{n+1}$ and $H_{n+1} \models v : o$. Then either $v = c^b = V$ and $o = b$ or $v = p$, $V = \langle p\rangle$ and $o = c$. In either case, we can take H_{n+1} as the

$$\Delta \vdash_\lambda (\langle h\rangle, x) \Downarrow (\langle h\rangle, \Delta(x)) \qquad \Delta \vdash_\lambda (\langle h\rangle, c^b) \Downarrow (\langle h\rangle, c^b) \qquad \Delta \vdash_\lambda (\langle h\rangle, \lambda x.M) \Downarrow (\langle h\rangle, cls(\Delta, \lambda x.M))$$

$$\frac{\Delta \vdash_\lambda (\langle h\rangle, M_1) \Downarrow (\langle h_1\rangle, cls(\Delta', \lambda x.M_1')) \quad \Delta \vdash_\lambda (\langle h_1\rangle, M_2) \Downarrow (\langle h_2\rangle, V_2) \quad \Delta'\{x : v_2\} \vdash_\lambda (\langle h_2\rangle, M_1') \Downarrow (\langle h_3\rangle, V)}{\Delta \vdash_\lambda (\langle h\rangle, M_1\ M_2) \Downarrow (\langle h_3\rangle, V)}$$

$$\frac{\Delta \vdash_\lambda (\langle h\rangle, M) \Downarrow (\langle h_1\rangle, \langle p\rangle) \quad \Delta \vdash_\lambda (\langle h_{i-1}\rangle, M_i) \Downarrow (\langle h_i\rangle, \langle p_i\rangle)\ (1 \le i \le n) \quad \mathcal{C}(c).\texttt{fields} = \{f_1 : o_1, \ldots, f_n : o_n\} \quad \langle h_0\rangle = \langle h_n\{p : \langle f_1 : p_1, \ldots, f_n : p_n\rangle_c\}\ \ (p \text{ fresh})}{\Delta \vdash_\lambda (\langle h\rangle, \texttt{new}\ c(M_1, \cdots, M_n)) \Downarrow (\langle h_0\rangle, \langle p\rangle)}$$

$$\frac{\Delta \vdash_\lambda (\langle h\rangle, M_0) \Downarrow (\langle h_0\rangle, \langle p_0\rangle) \quad \Delta \vdash_\lambda (\langle h_{i-1}\rangle, M_i) \Downarrow (\langle h_i\rangle, V_i)\ (1 \le i \le n),\ V_i = \langle p_i\rangle \text{ or } V_i = c^b \quad \text{let } v_i = p_i \text{ if } V_i = p_i \text{ otherwise } v_i = V_i \quad \langle\{x : p_0, x_1 : v_1, \ldots, x_n : v_n\} \vdash_o (h_n, x.m(x_1, \cdots, x_n)) \Downarrow (h_{n+1}, p)\rangle}{\Delta \vdash_\lambda (\langle h\rangle, M_0.m(M_1, \cdots, M_n)) \Downarrow (\langle h_{n+1}\rangle, V)}$$

$$\frac{\Delta \vdash_\lambda (\langle h\rangle, M) \Downarrow (\langle h_1\rangle, \langle p\rangle) \quad \langle\{x : p\} \vdash_o (h_1, x.l) \Downarrow (h_2, p)\rangle}{\Delta \vdash_\lambda (\langle h\rangle, M.l) \Downarrow (\langle h_2\rangle, \langle p\rangle)}$$

Figure 7: Operational semantics

necessary H' for which we have $\models_p h' : H'$ and $H' \models_p V : o$, as desired. □

Similarly as before, we define the following value typing.

- $h \models_p V : \tau$ (v has type τ under h)
  if there is some H such that $\models_o h : H$ and $H \models_p V : \tau$.

- $h \models_p \Delta : \mathcal{T}$ (Δ satisfies $\mathcal{T}$ under h)
  if there is some H such that $\models_o h : H$ and $H \models_p \Delta : \mathcal{T}$.

We then have the following desired result.

COROLLARY 2. *If $\mathcal{K}, \mathcal{T} \rhd M : \tau$, S is a $\mathcal{K}$-respecting ground substitution, $h \models_p \Delta : S(\mathcal{T})$ and $\Delta \vdash_\lambda (\langle h\rangle, M) \Downarrow (\langle h'\rangle, V)$ then $h' \models_p V : S(\tau)$.*

6. EXTENSIONS AND IMPLEMENTATION

This work is part of our ongoing project of developing an *Interoperable ML*. The aim of the project is to design and develop a polymorphic language that is interoperable with other programming models, including object-oriented languages and database systems. As a step toward such an interoperable language, we have implemented an experimental prototype system[1] embodying the interoperable calculus we have presented in this paper.

In design and development of the prototype system, we have considered several issues other than those presented in this paper, and have implemented some of them. An important one of them is the development of a uniform mechanism for interfacing with external languages of various different programming models. In this paper, we have so far considered interoperability with a single object-oriented language. Foreign function interfaces of most existing functional languages also usually presuppose single external model, especially that of C language [1, 6]. In practice, however, it is desirable and sometimes essential to be able to manipulate objects of various different models, such as Java, CORBA, COM and .NET, simultaneously.

[1]The current experimental prototype system, which we tentatively called Amethyst, is available at `http://pllab.jaist.ac.jp:8080/amethyst/index.html`. The interested reader may copy and try the system. Note, however, that this is a prototype made available for evaluation, and not intended for general release. We are planning to develop Interoperable ML – a practical polymorphic interoperable language.

To handle objects in various models, it is necessary to have some language constructs for distinguishing name spaces and for specifying required access protocols. To achieve this in a modular and uniform way, we have introduced in our implementation a notion of a *domain* representing a particular object model, and let object types of some object model belong to the domain corresponding to the model. The usual built-in records can be treated as object types belonging to a special built-in domain. Our implementation extends the core of Standard ML (i.e. those except for the module system) with a mechanism for domains. Its runtime system consists of a bytecode executor and plug-in modules, each of which realizes a particular domain. Current prototype includes a module for accessing Java class libraries and one for accessing PostgreSQL database server. The module for Java domain is implemented using Java Native Language Interface (JNI). The bytecode compiler of the main language together with the Java module realizes the interoperable calculus presented in this paper.

The actual implementation differs from the formal presentation in that it does not implement subsumption on object types. The rationale of this deviation is due to our consideration of the relationship between external objects and internal records. Since they are similar in their properties and usage, the programmer may often want to treat them uniformly. Based on this observation, we make the elimination operations of records (field selection primitives and polymorphic record pattern matching) applicable also to external objects. The lack of object subsumption is the price of this uniform treatment. We could of course implement external objects as those independent of records by introducing two sets of elimination operations. We are currently investigating a typing mechanism that allows both object subsumption and uniform treatment of records and objects.

Let us show some examples in our prototype system below. First, types and functions necessary for manipulating Java objects are declared through `domain` statement as follows.

```
domain Java = imports "init" of "jnilib";

external type 't JObject =
  JObject of 't imports "JObject" of Java;

fun ObjOf (JObject obj) = obj;
```

In the first line, Java domain which is implemented in a mod-

ule specified by "jnilib" is declared, and "init" specifies the entry point of the module. external type statement declares types representing external data structures, and has the property similar to datatype declaration in Standard ML. Type JObject wraps external Java classes, and plays a role as a marker to distinguish them from object types of other domains. A type parameter of JObject is the representation of the class structure. The function defined in the bottom line extracts the wrapped object.

This extra layer of type definition is necessary to deal with multiple domains. If the only possible target language is Java, then we could have used the same type encoding as in our formal development.

Using these declarations, we show examples of using Java classes. Suppose we have the following Java class.

```
public class C{
  public String Name;
  private int Age;
  public C(String Name,int Age){
    this.Name = Name;
    this.Age = Age;
  }
  public int getAge(){
    return this.Age;
  }
}
```

Class C is imported as follows.

```
external type C =
  {Name:string "field:Name:Ljava/lang/String;",
   getAge:unit->int "method:C:getAge:()I",
   java'lang'Object:unit "class",
   C:unit "class"}
  imports "C" of Java;

external fun C : string -> int -> C JObject
  = imports "new:C:(Ljava/lang/String;I)V"
    of Java;
```

C is an object type representing Java class C containing four members. Name corresponds to the Name field of class C. The string literal "field:Name:Ljava/lang/String;" is used by Java domain to identify the correspondence between members of external type and those of Java class. The second member getAge corresponds to getAge method of class C. The third and fourth members specify the ancestor classes of C, i.e. java.lang.Object and C itself. A constructor of class C is declared here as an external function by an external fun statement.

These declarations allows us to manipulate Java objects safely within the polymorphic type system. Moreover, we can treat those Java objects in the same way as built-in records of ML as seen in the following.

```
fun getName x = #Name x;
getName {Name="Murata"};
getName (ObjOf(C "Togo" 58));
```

Through this feature, we can enjoy the benefits of both ML-style polymorphism and object-oriented programming. For example, we can manipulate Java objects using ML-style higher-order functions as seen below.

```
val s = C "Saigo" 78;
val t = C "Togo" 58;
map (getName o ObjOf) [s,t];
```

The resulting system is to some extent more flexible than the original Java type system in manipulating Java objects due to ML's polymorphic functions with record polymorphism. For example, suppose we have another class definition of the form

```
class D{
  public String Name;
}
```

Since D and C has no common superclass having Name field, it is impossible in Java type system to define a function (method) for extracting Name attribute that is applicable to both C and D objects. By contrast, in our system the above getName can be applied to C and D objects.

```
getName (ObjOf(C "Togo" 58));
getName (ObjOf(D "Yamamoto" 21));
```

In this way, we can define various useful polymorphic functions manipulating Java objects by exploiting parametric polymorphism extended to Java objects.

7. CONCLUSIONS

We have defined a polymorphic calculus that is capable of interfacing with an object-oriented language, and have developed an ML-style type inference algorithm. The calculus enjoys the features of higher-order programming with ML-style polymorphism and class-based object-oriented programming with dynamic method dispatch. Method invocation is type checked in the polymorphic type system of the calculus. Its operational semantics is to call an external object-oriented language for dynamic method invocation. We have shown that the type system is sound with respect to an operational semantics which faithfully models linking to an external object-oriented language. These results have been implemented in our prototype interoperable language, which can access Java class files and other external resources.

This is our first step toward developing a statically typed interoperable languages, and there are a number of issues to be investigated. We only mention two of them below.

1. Accessing functions from an object-oriented language. We have only considered the problem of accessing external classes from a polymorphic higher-order language. We would also like to develop a mechanism for accessing functions in a polymorphic language from an object-oriented language. It should not be so hard to access first-order functions from an object-oriented language as a native static method. More challenging issues is to combine and use foreign higher-order functions directly in an object-oriented language.

2. Memory management in interoperable languages.
   In our formalism and in our implementation, values in the polymorphic calculus may contain pointers to the heap of the external object-oriented language, but not vise versa. Due to this simple structure, garbage collection in the object calculus can be done by including the set of exported pointers in the root set. However, if we extend the formalism to allow bi-directional interoperability as discussed above, then the heaps

of participating languages may contain global cycles, and garbage collection could be a serious problem. Garbage collection method in distributed computing may give us some hint.

We are now investigating these and other issues using our experimental prototype system.

Acknowledgments

We would like to thank anonymous reviewers for their helpful comments, which have been very useful for improving the presentation of this paper.

8. REFERENCES

[1] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In N. Benton and A. Kennedy, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

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

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

[4] M. Corporation and D. Corporation. The component object model specification, 1995.

[5] S. Finne, D. Leijen, E. Meijer, and S. L. P. Jones. Calling hell from heaven and heaven from hell. In *International Conference on Functional Programming*, pages 114–125, 1999.

[6] S. Finne, D. Leijen, E. Meijer, and S. P. Jones. *H/Direct*: A binary foreign language interface for Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 153–162. ACM, June 1999.

[7] K. Fisher and J. H. Reppy. The design of a class mechanism for Moby. In *SIGPLAN Conference on Programming Language Design and Implementation (PDLI)*, pages 37–49, 1999.

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

[9] C. Gunter and J. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[10] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, Mar. 1996.

[11] X. Leroy. *Polymorphic typing of an algorithmic language*. PhD thesis, University of Paris VII, 1992.

[12] X. Leroy. The Objective Caml system: Documentation and user's manual, 2000. (with Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon Available from `http://caml.inria.fr`.)

[13] T. Lindholm and F. Yellin. *The Java virtual machine specification*. Addison Wesley, second edition edition, 1999.

[14] E. Meijer and K. Claessen. The design and implementation of Mondrian. In *Haskell Workshop*. ACM, June 1997.

[15] E. Meijer and S. Finne. Lambada, Haskell as a better java. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.

[16] J. Mitchell. Type inference and type containment. In *Semantics of Data Types, Lecture Notes in Computer Science 173*, pages 257–277. Springer-Verlag, 1984.

[17] J. Nordlander. Pragmatic subtyping in polymorphic languages. In *International Conference on Functional Programming (ICFP)*, 1998.

[18] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995. A preliminary summary appeared at ACM POPL, 1992 under the title "A compilation method for ML-style polymorphic record calculi".

[19] 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.

[20] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. Summary in Proc. ACM POPL Symposium, 1997.

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