Static Typing for Dynamic Messages

Susumu Nishimura

RIMS, Kyoto University

Sakyo-ku, Kyoto 606-01, JAPAN

nisimura@kurims.kyoto-u.ac.jp

Abstract

Dynamic messages are first-class messages dynamically bound to program variables. By dynamic messages, the methods to be invoked can be varied dynamically at run-time, which provides a powerful abstraction mechanism for object-oriented languages. Dynamic messages are critically needed for some programs, but it seems that there has been no proposal of static type systems for dynamic messages. This paper presents a static typing discipline for dynamic messages and formalizes it into a second order polymorphic type system. The type system satisfies the type soundness property and has a principal type inference algorithm. The type system therefore provides a foundation for a statically typed object-oriented language enriched with polymorphic dynamic messages.

1 Introduction

In object-oriented languages, method is invoked by passing a message to an object. If messages are treated as first-class values and they are bound to program variables, the method invoked by the messages can be varied dynamically. In this paper, we call first-class messages as *dynamic messages* to emphasize their dynamic nature. Most of object-oriented languages do not allow dynamic messages, but only a few languages support them. Smalltalk [GR89] and Objective-C [PW91], for example, provide a special method `` `perform:' `` for the purpose of passing dynamic messages to objects. Though dynamic messages are not so frequently used even in these languages, there are some significant programs whose use of dynamic messages is critical and cannot be easily replaced by statically specified messages.

One example of the critical use of dynamic messages is delegate objects. We can simply define a delegate object as an object that redirects all the passed messages to another object by using dynamic messages as follows

{ **redirect**(m) = o←m }

where { **redirect**(m) = ... } defines a delegate object consisting of a single method named **redirect**. The **redirect** method receives a dynamic message bound to the argument variable m and passes the message to an object o (the message passing is expressed by o←m), where o refers to the object to which messages are redirected. The delegation cannot be easily expressed without the dynamic message bound to m representing any message to be transmitted to the object o.

Dynamic messages also bring a powerful abstraction mechanism into object-oriented languages: a dynamic message bound to a program variable abstracts the method actually invoked by the message passed to an object, which is analogous to the abstraction mechanism of higher-order functions in functional languages. By means of this abstraction mechanism, we can describe a variety of methods that implement useful features similar to those provided by higher-order functions, in the framework of an object-oriented language. For example, a method **map** for passing a list of messages to an object, which is an object-oriented analogue of the higher-order map function, would be implemented as follows

```
method map(msglst) =
   case msglst of
        [] ⇒ []
      | head::tail ⇒ (self←head)::(self←map(tail))
   end
```

where [] stands for an empty list, *head*::*tail* is a list constructor which adds a new element *head* to the head of a list *tail*, and the identifier self refers to the object itself executing the method. This **map** method would be useful for programming some applications. In graphical application programs, for example, we can invoke a sequence of methods for window manipulation and picture drawing by a single message passing as follows

w←**map**([**raise**(), **clear**(), **box**(0,0,10,10), **line**(0,30,20,5)])

where w refers to a window object and the four methods **raise**, **clear**, **box**, and **line** are invoked one after another.

In addition to the ability of defining a variety of flexible methods, the abstraction mechanism provided by dynamic messages can contribute to the efficiency of distributed object-oriented computing, where objects are modeled as *network objects* [BNOW93] accepting network-transparent method invocations. In distributed computation, network latency is a major bottleneck and it is crucial for efficient computation to reduce network load by exploiting locality. We observe that dynamic messages can exploit locality by abstracting remote computation.

To see how locality is exploited by dynamic messages, we consider the execution of a Unix standard command `find` in a networked environment, where a file system is maintained by a network file server and the `find` command is executed on a remote client. The execution of a `find` command on a remote client would consist of recursive traversal of the directory tree, conditional checking for each file, and an operation on the searched files,

To appear in the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1998, San Diego, California.

each of which requires a communication between the server and the client. The execution therefore produces a heavy network load between the server and the remote client, which causes poor performance consequently. The network load can be reduced, however, if the locality of the file system is exploited, i.e., if the server executes the `find` command in behalf of the remote client.

This locality exploitation can be naturally expressed by means of dynamic messages. We can implement the network file server as a network object that provides network-transparent file access services including the service for executing the `find` command in behalf of remote clients. Remote clients are allowed to utilize the service by sending a **find** message of the following form to the server object

find(*dir*, *cond*, *fop*)

where *dir* is a directory object which is the top node of a directory hierarchy to be searched, and *cond* and *fop* are messages abstracting the method for conditional checking on the files and the method for file operation on the searched files, respectively. For example, to list all the names of such files that contain the string "ACM", we send the following message to the server object

find(dir,**grep**(``ACM''),**fname**())

where dir is the top directory to be searched, **grep** and **fname** are built-in methods for file object to check if the specified string is contained and to return the file name string, respectively.

The **find** method can be defined with dynamic messages in a simple way. First, the **find** method for the file server object is defined as

method FileServer.**find**(dir,cond,fop) =
  dir ← **find**(cond,fop)

where the **find** message is simply redirected to the directory object dir. The redirected **find** message is recursively passed to all descendant objects (files and directories) in the directory hierarchy. If a directory object receives a **find**, the received **find** message is redirected to all immediate descendant files and directories. We assume that every directory object has a method named **mapdir** that passes a given message to all immediate descendant objects: if a message **mapdir**(*msg*) is passed to a directory object, then the directory object redirects the message *msg* to all objects (files and directories) registered in the directory record. The **mapdir** can be defined similarly to the **map** method defined above. The **find** method for a directory object is defined by using the **mapdir** method as follows.

method Dir.**find**(cond,fop) =
  self ← **mapdir**(**find**(cond,fop))

If the **find** message reaches a file object, the search condition is first checked, and then the method for file operation is invoked if the condition is satisfied. The conditional checking and the method invocation for file operation is done by passing the messages contained in the **find** message to the file object itself, as show in the following program

method File.**find**(cond,fop) =
  if (self←cond) then [self←fop] else []

where $[e]$ stands for a singleton list whose only element is e. The method returns either a singleton list or an empty list depending on the result of conditional checking. The lists of results are collected and concatenated into a single list by the **mapdir** method.

In spite of their advantages, dynamic messages are not frequently used in object-oriented programming. It seems that one major reason for this infrequent use of dynamic messages is that there is no static typing discipline for dynamic messages. There are a few typed object-oriented languages supporting dynamic messages, but the type safety is not guaranteed by their type systems. Objective-C [PW91], for example, is a strongly typed object-oriented language that allows dynamically bound method invocations in a way similar to those of Smalltalk. In Objective-C, however, every dynamic message is given a single type called selector type, and therefore the object receiving a dynamic message may not implement the corresponding method, in which case a run-time error is signaled.

The aim of this paper is to develop a static typing discipline for dynamic messages to incorporate them into the framework of a typed object-oriented language. A static type system for dynamic messages would bring the fruitful benefit of statically typed languages into an object-oriented language enriched with the powerful dynamic method invocation mechanism. Static typing is an effective method to develop programs in a semantically verified way, where type inconsistency can be detected earlier by static type inference in advance of execution. Static typing for dynamic messages therefore would be useful for finding a message that attempts to invoke a method not implemented in an object, for detecting type mismatch of the arguments and the result of message passing, etc. To the author's knowledge, there seems to be no static type system that combines an objected-oriented language with the dynamic method invocation mechanism in a type safe manner. Several researchers have been elaborating foundational models for statically typed object-oriented languages [Car88, CCH+89, OB89, Mit90, Rém94, PT94, HP95, AC96, BSvG95, BPF97], but the dynamic method invocation mechanism is not incorporated into the proposed models.

In an object-oriented language embedded in λ-calculus, a dynamic message can be encoded by enclosing it in a function closure: a dynamic message $\textbf{msg}(a)$ is encodable as a function $f = \lambda x.x \leftarrow \textbf{msg}(a)$ and passing the dynamic message to an object o can be encoded as a function application fo. However, this encoding depends on the general abstraction mechanism of higher-order functions. It seems that the encoding adds an irrelevant indirection that every message is wrapped in a function closure. The concern of this paper is not in such a variant of λ-calculus enriched with object-oriented features, but in a language comprising of *purely* object-oriented components: objects, messages, and message passing. Such a pure object-oriented language can be a new calculus having enough power to encode some useful features: variants are just messages, pairs and records are encoded as objects, conditional expressions are special form of object definition, and λ-abstraction $\lambda x.M$ is encoded by an object $\{\mathsf{arg}(x) = M\}$ and application of the function to an argument N by $\{\mathsf{arg}(x) = M\} \leftarrow \mathsf{arg}(N)$. Furthermore, the pure object-oriented calculus suggests a calculus of objects and messages which interact with each other as the duals of one another, which seems to exhibit an elegance of object-oriented programming paradigm. A similar duality is formally presented in Barbanera and Berardi's symmetric lambda calculus [BB96] as the duality of pairs and sums. This duality is also generally known as that of records and labeled variants. For example, a labeled variant of type $\langle \ell_1(\tau_1), \ldots, \ell_n(\tau_n) \rangle$ is encoded by records as a function of type $\forall t.\{\!|\ell_1 : \tau_1 \to t, \ldots, \ell_n : \tau_n \to t|\!\} \to t$, following the type notation in Ohori's polymorphic record calculus [Oho95]. However, the type of the encoded variant indicates that application of the variant to a record forces the resulting types to be the same type t. This typing restriction is critical for object-oriented programming languages, since an object generally consists of methods yielding differently typed results. A type system for an object-oriented language should allow the same message to be used to invoke identically named methods yielding results of different types, but it seems that there is no such type system for object-oriented languages. In Aiken, Wimmers, and Lakshman's soft typing system [AWL94], conditional types allow a single case expression to yield differently typed results. However, their type system cannot be static, since their type system depends on the dynamic nature of programs, i.e., the control-flow.

In the rest of the section, we will show the difficulties in typing dynamic messages by some examples, and explain a general idea to overcome the difficulties. Consider the following simple but illustrative example.

method **foo**(o,m) = o←m

The method **foo** is such a method that receives an object o and a message m as arguments and sends the message to the object. There is no static information about the name of the method invoked by the message passing: o can be any object, m can be any message, and therefore the method name to be invoked is completely unknown. Nevertheless, to preserve the type safety, the type system must guarantee that the object has the corresponding method that will be invoked by the message. Furthermore, the method **foo** cannot be typed in the usual way, since the result type of the method cannot be determined uniquely. Consider the following two expressions

o←**foo**({**a**(x)=x+1,**b**(x)=(x=0)}, **a**(3))

o←**foo**({**a**(x)=x+1,**b**(x)=(x=0)}, **b**(1))

where we assume that o is some object that accepts the **foo** method. The first expression and the second expression should be typed as *int* and *bool*, respectively. This indicates that the results of invocations of an identical method **foo** can have completely different types depending on the types of the object and the message given as arguments. The type system for dynamic messages therefore must solve two problems: static typing without sufficient static method name information and method typing dependent on the argument types.

We solve these problems based on the following observation: Each dynamic message can be viewed as an abstraction of a set of method names that the dynamic message can invoke, and therefore each dynamic message should be statically typed by a set of method names that may be invoked by the message. For example, if a dynamic message has no static information about method names invoked by the message, the dynamic message is typed by an empty set of method names. If some static information about a method name is supplied later, the newly supplied method name can be included in the set of method names in a way that type consistency is preserved. Based on this observation, we develop our type system as a second order polymorphic type system, where the type information is expressed by kinding as is done in Ohori's type system for record calculus [Oho95]. Our type system, however, gives a solution for the two problems in typing dynamic messages: Every dynamic message can be typed by a set of method names, where only one method out of the set is invoked by the dynamic message, and the method typing depending on the argument types is achieved as a process of resolving type inconsistency. In the following sections of the paper, we will describe in detail how the type system solves the two problems and then give a formal definition of the type system.

The rest of the paper is organized as follows. Section 2 informally shows our typing strategy for dynamic messages. Section 3 presents a formal definition of an object-oriented language enriched with dynamic messages and gives its operational semantics and typing rules. In the section, we also show the type soundness and give a principal type inference algorithm. In Section 4, we give some examples of typing derivations and show how classes can be defined by means of recursive object definition. Finally, Section 5 concludes the paper and discusses future work.

We only sketched the proof of the type soundness theorem in Appendix. The proof of the other theorems is omitted from the present paper due to the page limitation. The author intends to publish the proof of those theorems elsewhere in a more detailed form.

2 Typing Dynamic Messages

We formalize the general idea proposed in the previous section as a second-order polymorphic type system. In the following, we conventionally write $t :: k$ to express a type variable t kinded by a kind k. Note that our type system has no type directly representing the type structure of objects and messages. Instead, the type information is expressed through kindings of the form $t :: k$, where t is a type variable indexing the type information of an object or a message and k is a kind representing the type information. The type information of objects and messages cannot be expressed by ordinary types, since the exact type structure of objects and messages cannot be known statically in the presence of dynamic messages. Even statically specified objects and messages cannot determine their exact type structure, since another dynamic message may add new type information. Furthermore, the type information must be shared among the types to preserve type consistency. This requires the types of objects and messages to be indexed by distinguished type variables.

Based on the above observation, we represent each dynamic message by a kind enumerating method names that may be invoked by the message with their corresponding arguments. A dynamic message is typed by a type variable kinded by a *message kind*, $s :: \langle\!\langle \ell_1(\tau_1), \ldots, \ell_n(\tau_n) \rangle\!\rangle$, where $\ell_1, \ldots, \ell_n$ are method names and each τ_i is the argument type of the corresponding method. Each object is then represented by a kind enumerating dynamic messages acceptable to the object with their corresponding result types. An object is expressed by a type variable kinded by an *object kind*, $t :: \{\!|t_1 \to \tau_1, \ldots, t_n \to \tau_n|\!\}$, where $t_1, \ldots, t_n$ are types of dynamic messages accepted by the object and each τ_i is the result type of the method invocation by the corresponding dynamic message. The type variables $t_1, \ldots, t_n$ therefore must be kinded by some message kinds. The object kind furthermore must respect a particular type consistency condition: identically named methods of an object must be invoked with the same argument type and the same return type. This condition is rephrased as follows: if $t_i :: \langle\!\langle \ell(\tau), \ldots \rangle\!\rangle$ and $t_j :: \langle\!\langle \ell(\tau'), \ldots \rangle\!\rangle$, then $\tau = \tau'$ and $\tau_i = \tau_j$. We call this condition the *well-formedness* condition for object kinds.

To see how types can be inferred by the above typing strategy, we first consider the typing for the **foo** method given in the previous section.

method **foo**(o,m) = o←m

We type this method in the following way. Let s be the type for the dynamic message m. Since no information about method names is given for the dynamic message, s is kinded by a null message kind, i.e., $s :: \langle\!\langle\rangle\!\rangle$. Then o is typed as an object that accepts the dynamic message m, i.e., the type of o is given as $t :: \{\!|s :: \langle\!\langle\rangle\!\rangle \to u|\!\}$ where the type variable u represents some result type not yet determined. The dynamic message kinded by a null message kind may seem useless since the kinding indicates that the message cannot invoke any method. However, the kinding makes sense when some static information about a method name is supplied later. Actually, the type variable s plays a role similar to that of the raw variables in Wand's type system [Wan87].

To see how we can consistently infer the result type of the **foo** method which depends on the type of the argument, we revisit the following example given in the previous section.

o←**foo**({**a**(x)=x+1,**b**(x)=(x=0)}, **a**(3))

In this message passing, the argument variable m is bound to the message **a**(3), whose method name is statically specified. We therefore augment the null message kind with the new static information, i.e., $s :: \langle\!\langle \mathbf{a}(int) \rangle\!\rangle$. Furthermore, since the program variable o is also bound to an object {**a**(x)=x+1,**b**(x)=(x=0)}, the object kind should be augmented to express that the methods **a** and **b** are implemented in the object. The object kind is therefore expanded to accept two more dynamic messages as follows

$$t :: \{\!| s :: \langle\!\langle \mathbf{a}(int) \rangle\!\rangle \rightarrow u, \\ s_1 :: \langle\!\langle \mathbf{a}(int) \rangle\!\rangle \rightarrow int, s_2 :: \langle\!\langle \mathbf{b}(int) \rangle\!\rangle \rightarrow bool |\!\},$$

where the new type variables s_1 and s_2 are types of dynamic messages to invoke the methods **a** and **b**, respectively. However, this object typing contains an apparent type inconsistency. The well-formedness condition is broken for the object kind: the kindings $s :: \langle\!\langle \mathbf{a}(int) \rangle\!\rangle$ and $s_1 :: \langle\!\langle \mathbf{a}(int) \rangle\!\rangle$ indicate that two differently typed dynamic messages can invoke an identical method **a** but the results have different types (u and int). We resolve this type inconsistency by unifying the two types u and int. The unification yields int as the expected result type of the above message passing. The typing procedure described above resolves the type dependency by a simple unification process respecting the well-formedness condition. If **b**(1) is given instead as the second argument of the **foo** method in the above example, then a similar inference process yields the expected type *bool* by unifying u into *bool*.

The typing strategy presented so far enables us to type dynamic messages by augmenting type information with the static method name information supplied during the type inference procedure. However, the unlimited augmentation of type information breaks the soundness of the type system. To see this, consider the following example.

method **bar**(m) = {**succ**(x)=x+1, **pos**(x)=(x>0)}←m

The method **bar** receives a dynamic message through the argument m and passes it to the object consisting of two methods **succ** and **pos**. Following the above typing strategy, we initially type the dynamic message m by $s :: \langle\!\langle\rangle\!\rangle$ and the object {**succ**(x)=(x+1), **pos**(x)=(x>0)} by

$$t :: \{\!| s :: \langle\!\langle\rangle\!\rangle \rightarrow u, \\ s_1 :: \langle\!\langle \mathbf{succ}(int) \rangle\!\rangle \rightarrow int, s_2 :: \langle\!\langle \mathbf{pos}(int) \rangle\!\rangle \rightarrow bool |\!\}.$$

This typing is valid as long as the method invoked by the dynamic message m is either **succ** or **pos**. However, if the method **bar** is used with a statically named message different from both **succ** and **pos**, we obtain an improper typing. Suppose a message **pred**(7) is given as the argument of the method **bar**. Then the message kind for the dynamic message m is augmented to include the method **pred**, i.e., $s :: \langle\!\langle \mathbf{pred}(int) \rangle\!\rangle$. This typing should be illegal, since the object implements only the two methods **succ** and **pos** but the kinding indicates that the object must have the method **pred**.

We observe that such an improper typing is caused by the typing strategy that ignores the static information about method names in objects. To keep trace those static information, we attach a set of method names to every kind as a constraint. The constrained message kind and object kind are expressed as $\langle\!\langle \ell_1(\tau_1), \ldots, \ell_n(\tau_n) \rangle\!\rangle_L$ and $\{\!| s_1 \rightarrow \tau_1, \ldots, s_n \rightarrow \tau_n |\!\}_L$, respectively, where L is some subset of method names. The set attached to a message kind indicates the maximal set of method names that can be enumerated in the message kind, i.e., any message kind $\langle\!\langle \ell_1(\tau_1), \ldots, \ell_n(\tau_n) \rangle\!\rangle_L$ must satisfy $\{\ell_1, \ldots, \ell_n\} \subseteq L$; The set attached to an object kind indicates the maximal set of method names the corresponding object can accept, i.e., for any object kind $\{\!| s_1 \rightarrow \tau_1, \ldots, s_n \rightarrow \tau_n |\!\}_L$, each s_i must be kinded by a message kind $\langle\!\langle \cdots \rangle\!\rangle_{L_i}$ such that $L_i \subseteq L$. According to this kinding strategy, we type the method **bar** with the constrained kinds as follows: The method receives a dynamic message of type $s :: \langle\!\langle\rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}}$ and passes it to the object of type

$$t :: \{\!| s :: \langle\!\langle\rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}} \rightarrow u, \\ s_1 :: \langle\!\langle \mathbf{succ}(int) \rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}} \rightarrow int, \\ s_2 :: \langle\!\langle \mathbf{pos}(int) \rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}} \rightarrow bool |\!\}_{\{\mathbf{succ},\mathbf{pos}\}}.$$

Due to the attached constraint, we can obtain a sound type system by rejecting any augmentation of type information that does not obey the constraint attached to kinds. For example, the message **pred**(7) cannot be used as a well-typed argument of the method **bar** any more, since the inclusion of the method **pred** into the type of the argument yields a type $s :: \langle\!\langle \mathbf{pred}(int) \rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}}$, which violates the constraint.

We formalize the idea presented above into a second order polymorphic type system in the following section. The resulting type system supports an ML-style let polymorphism and recursive objects, where the recursiveness of types is expressed by mutual reference between types and kinds. Though we have put the type consistency condition that disables the invocations of an identical method with different types, the let polymorphism allows identical methods to be used with different types. In this paper, we do not consider subtyping which is often regarded as a formal model for classes with inheritance. We will mention about the subtyping issue briefly in Section 5.

The general idea of our type system is based on Ohori's type system for polymorphic record calculus [Oho95], where each object is kinded by a record kind associating distinct labels to corresponding types. However, our type system is far more powerful than Ohori's in the sense that it achieves polymorphic typing for dynamic messages, whose counterpart in the record calculus is first-class labels that are not considered in Ohori's type system. In addition to this, as mentioned in Section 1, our object-oriented language can be viewed as a calculus of the duals, i.e., objects and messages, which is reflected in the type system as interaction between object kinds and message kinds. In Ohori's type system, variant kinds are considered, but they have no interaction with record kinds.

The typing mechanism of our type system is also similar to that of Rémy's type system for records and variants [Rém89]. His type system allows type inclusion for record and variant fields by attaching an attribute to each labeled field to tell if the field value must, may, or must not exist for the field. The kind restriction in our type system tells essentially the same constraint expressed by the attributes in Rémy's type system. However, first-class labels are not considered in the Rémy's work either.

3 An Object-Oriented Language with Dynamic Messages

In this section, we use the following mathematical notations. We denote set difference by $A \setminus B = \{x \mid x \in A \text{ and } x \notin B\}$. For every finite map f and g, $domain(f)$ denotes the domain of f, $f \circ g$ denotes functional composition, and $f + g$ defines a finite map by overriding the definition of f by g, i.e., $domain(f+g) = domain(f) \cup domain(g)$, $(f+g)(x) = g(x)$ for every $x \in domain(g)$, and $(f+g)(x) = f(x)$ for every

$$
\begin{array}{rcll}
M & ::= & c & \text{(constant)} \\
& | & x & \text{(identifier)} \\
& | & \{\ell_1(x_1) = M_1, \ldots, \ell_n(x_n) = M_n\} & \text{(object)} \\
& | & \ell(M) & \text{(message)} \\
& | & M \leftarrow M & \text{(message passing)} \\
& | & \mathsf{let}\; x = M \;\mathsf{in}\; M & \text{(let binding)} \\
& | & \mathsf{letobj}\; x = \{\ell_1(x_1) = M_1, \ldots, \ell_n(x_n) = M_n\} \;\mathsf{in}\; M & \text{(recursive object)}
\end{array}
$$

Figure 1: Syntax of the Language

$$
\frac{}{\rho \vdash c \Downarrow c}\;\textsc{Const} \qquad \frac{x \in \mathit{domain}(\rho)}{\rho \vdash x \Downarrow \rho(x)}\;\textsc{Var} \qquad \frac{\rho \vdash M \Downarrow v}{\rho \vdash \ell(M) \Downarrow \ell(v)}\;\textsc{Message}
$$

$$
\frac{}{\rho \vdash \{\ell_1(x_1) = M_1, \ldots, \ell_n(x_n) = M_n\} \Downarrow \{\ell_1(\_, x_1, M_1, \rho), \ldots, \ell_n(\_, x_n, M_n, \rho)\}}\;\textsc{Object}
$$

$$
\frac{\begin{array}{c}\rho \vdash M \Downarrow \{\ell(x, y, M', \rho'), \ldots\} \quad \rho \vdash N \Downarrow \ell(v') \\ \rho' + \{x \mapsto \{\ell(x, y, M', \rho'), \ldots\}, y \mapsto v'\} \vdash M' \Downarrow v\end{array}}{\rho \vdash M \leftarrow N \Downarrow v}\;\textsc{MsgPass}
$$

$$
\frac{\rho \vdash M \Downarrow v' \quad \rho + \{x \mapsto v'\} \vdash N \Downarrow v}{\rho \vdash \mathsf{let}\; x = M \;\mathsf{in}\; N \Downarrow v}\;\textsc{Let} \qquad \frac{\rho + \{x \mapsto \{\ell_1(x, x_1, M_1, \rho), \ldots, \ell_n(x, x_n, M_n, \rho)\}\} \vdash N \Downarrow v}{\rho \vdash \mathsf{letobj}\; x = \{\ell_1(x_1) = M_1, \ldots, \ell_n(x_n) = M_n\} \;\mathsf{in}\; N \Downarrow v}\;\textsc{LetObj}
$$

Figure 2: Operational Semantics

$x \in \mathit{domain}(f) \setminus \mathit{domain}(g)$. We conventionally write a finite map as a finite sequence, e.g. $\{x_1 : q_1, \ldots, x_n : q_n\}$. Finally, we assume that $\mathcal{L}$ denotes a denumerably infinite set of labels $\{\ell, \ell', \cdots\}$, and we use meta variables $L, L', \ldots$ to range over $\mathcal{L}$ and finite subsets of $\mathcal{L}$.

3.1 The Language Syntax

The language presented here is a minimal object-oriented language that provides only essential features for discussing dynamic messages. The language is a common core of languages that have been investigated in the series of studies of object-oriented programming [GM94], providing the following features: object creation, messages, method invocation, and polymorphic let binding. We give the language syntax in Figure 1. An object $\{\ell_1(x_1) = M_1, \ldots, \ell_n(x_n) = M_n\}$ is comprised of $n (n \geq 0)$ methods uniquely distinguished by labels $\ell_1, \ldots, \ell_n$, and each method ℓ_i receives an argument through x_i and executes the body M_i. A message is expressed as $\ell(M)$, a value tagged by a label ℓ. The expression for method invocation $M \leftarrow N$ passes the message N to the object M and executes the corresponding method of the object. N can be any expression, thus it can be a dynamic message. In addition to these, for the sake of recursive object definition, we included a special construct letobj. The letobj construct binds an object to an identifier and the identifier can be referred to the object itself in the object being defined; The identifier corresponds to the *self* variable which is used in some object-oriented languages to refer to the object being executed.

Although the language is small, it has enough power to encode some useful features: variants, pairs, records, conditional expressions, and λ-abstractions, as mentioned in Section 1. We can even encode class definitions by means of recursive object definitions, which will be discussed in Section 4.

3.2 Operational Semantics

We give an operational semantics of our language in the style of natural semantics [Gun92]. We first define semantic values, which are ranged over by meta-variable v, by the following grammar.

$$
\begin{array}{rcll}
v & ::= & c & \text{(constant)} \\
& | & \ell(v) & \text{(message)} \\
& | & \{\ell_1(x_1, y_1, M_1, \rho_1), \ldots, \ell_n(x_n, y_n, M_n, \rho_n)\} & \text{(object)}
\end{array}
$$

The meta-variable c ranges over the set of constant values such as integers. A message is a value tagged with a label, where the tag is the method name and the value is the argument. An object is a sequence of *method closures* of the form $\ell_i(x_i, y_i, M_i, \rho_i)$, where ℓ_i is the method name, x_i is an identifier referring to the record itself (like a *self* variable), y_i the argument of the method, M_i the method body, and ρ_i the run-time environment when the method was created. A run-time environment ρ is a finite map that assigns identifiers to values, written as $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$.

The operational semantics is given in Figure 2 by a set of rules that derive a formula of the form $\rho \vdash M \Downarrow r$, which indicates that the expression M evaluates to the result r under the run-time environment ρ, where r is a value or a special symbol *error* standing for a run-time error. Note that the complementary rules that yields *error* are omitted from the figure, whereas they are given as implicit rules: if either one of the subderivations yields *error* or none of the rules given in the figure cannot be applied to derive a formula, then the evaluation derivation yields *error*. Furthermore, if a method closure has the form $\ell(\_, y, M, \rho)$, $\_$ stands for a special anonymous identifier which is never bound in a run-time environment. That is, writing $\rho + \{\_ \mapsto v\}$ is equivalent to writing ρ. The closure of method defined in a non-recursive object has the anonymous identifier as its first component, since any non-recursive object does not have the reference to the object itself.

3.3 Type System

We present a polymorphic type system for our language. We first define types and kinds of our type system.

3.3.1 Types and Kinds

Types, ranged over by $\tau, \tau', \ldots$, are defined by the following grammar

$$\tau ::= b \mid t$$

where b stands for basic types, and t for type variables. As discussed in Section 2, we have no type directly representing objects and messages. The type of an object or a message is expressed by a type variable indexing a kind representing the corresponding type information, and the type information is shared through the type variables. For example, an object and dynamic messages accepted by the object share the type information so that the types of the dynamic messages respect the type consistency constraint induced by the well-formedness condition, which will be formally defined in the following.

Kinds, ranged over by $k, k', \ldots$, have distinguished three forms, as defined in the following grammar.

$$\begin{array}{rcll} k & ::= & U & \text{(universal kind)} \\ & \mid & \langle\!\langle \ell_1(\tau_1), \ldots, \ell_n(\tau_n) \rangle\!\rangle_L & \text{(message kind)} \\ & \mid & \{\!\!\{ t_1 \to \tau_1, \ldots, t_n \to \tau_n \}\!\!\}_L & \text{(object kind)} \end{array}$$

U is *universal kind*, the kind for any type. A *message kind* $\langle\!\langle \ell_1(\tau_1), \ldots, \ell_n(\tau_n) \rangle\!\rangle_L$ is a possibly empty sequence of types tagged with labels, subscripted by a set of labels L. The order of the sequence is insignificant. The message kind represents such a dynamic message that may invoke some method named ℓ_i with an argument of type τ_i. The subscripted L designates the upper bound of the set of labels enumerated in the sequence. An *object kind* $\{\!\!\{ t_1 \to \tau_1, \ldots, t_n \to \tau_n \}\!\!\}_L$ is a possibly empty sequence of pairs of a type variable and a type of the form $t \to \tau$, where t is the type of a dynamic message, thus t must be kinded by a message kind. The order of the sequence is insignificant. The object kind represents such an object that accepts n dynamic messages, whose types are given by $t_1, \ldots, t_n$, returning the results of types $\tau_1, \ldots, \tau_n$, respectively. The subscripted set of labels L designates the upper bound of the set of method names that can be accepted by the object. To give consistent typing for objects, we impose a particular condition on object kinds: identically named methods of an object must be invoked with the same argument type and the same return type. This type consistency for object kind is formally described in the following part of this section as the *well-formedness* condition.

In the rest of the paper, for every message kind $k = \langle\!\langle \ell_1(\tau_1), \ldots, \ell_n(\tau_n) \rangle\!\rangle_L$, we denote the set of labels $\{\ell_1, \ldots, \ell_n\}$ by $Labels(k)$, and L by $Labub(k)$. The set of *free type variables* occurring in a type τ is denoted by $FTV(\tau)$, and the set of free type variables in a kind k by $KFTV(k)$.

3.3.2 Kinding Rules and Kinded Substitution

Kinds are given to types relative to a *kind assignment*. A kind assignment $\mathcal{K}$ is a finite map from type variables to kinds, written as $\{t_1 :: k_1, \ldots, t_n :: k_n\}$. We extend the definition of $KFTV$ to kind assignments by $KFTV(\mathcal{K}) = \bigcup_{t \in domain(\mathcal{K})} KFTV(\mathcal{K}(t))$.

In our type system, kinds are given only for *well-formed types* under a *well-formed kind assignment*. We say a type τ is well-formed under a kind assignment $\mathcal{K}$ if $FTV(\tau) \subseteq domain(\mathcal{K})$. A kind assignment $\mathcal{K}$ is well-formed if $\mathcal{K}(t)$ is a *well-formed kind* for every $t \in domain(\mathcal{K})$, where the well-formedness of a kind k under a kind assignment $\mathcal{K}$ is defined by the following conditions.

- $KFTV(k) \subseteq domain(\mathcal{K})$.
- If $k = \langle\!\langle \ell_1(\tau_1), \ldots, \ell_n(\tau_n) \rangle\!\rangle_L$ then
  - $\ell_1, \ell_2, \ldots, \ell_n$ are distinct labels,
  - $\tau_1, \ldots, \tau_n$ are well-formed under $\mathcal{K}$, and
  - $\{\ell_1, \ell_2, \cdots, \ell_n\} \subseteq L$.
- If $k = \{\!\!\{ t_1 \to \tau_1, \ldots, t_m \to \tau_m \}\!\!\}_L$ then
  - $t_1, \ldots, t_n$ are distinct type variables,
  - $\tau_1, \ldots, \tau_n$ are well-formed under $\mathcal{K}$,
  - every $\mathcal{K}(t_i)$ is a message kind such that $Labub(\mathcal{K}(t_i)) \subseteq L$, and
  - $\mathcal{K}(t_i) = \langle\!\langle \ell(\tau), \cdots \rangle\!\rangle_{L_i}$ and $\mathcal{K}(t_j) = \langle\!\langle \ell(\tau'), \cdots \rangle\!\rangle_{L_j}$ implies $\tau = \tau'$ and $\tau_i' = \tau_j'$.

In the rest of the paper, we assume that every kind assignment is well-formed.

We say a type τ has a kind k under $\mathcal{K}$, if $\mathcal{K} \vdash \tau :: k$ is derived by applying one of the following rules.

- $\mathcal{K} \vdash \tau :: U$ for every well-formed type τ.
- $\mathcal{K} \vdash t :: \langle\!\langle \ell_1(\tau_1), \ldots, \ell_n(\tau_n) \rangle\!\rangle_L$ if $\mathcal{K}(t)$ is a well-formed message kind $\langle\!\langle \ell_1(\tau_1), \ldots, \ell_n(\tau_n), \cdots \rangle\!\rangle_{L'}$ such that $L' \subseteq L$.
- $\mathcal{K} \vdash t :: \{\!\!\{ t_1 \to \tau_1, \ldots, t_n \to \tau_n \}\!\!\}_L$ if $\mathcal{K}(t)$ is a well-formed object kind $\{\!\!\{ t_1 \to \tau_1, \ldots, t_n \to \tau_n, \cdots \}\!\!\}_{L'}$ such that $L' \subseteq L$.

A *type substitution*, or simply *substitution*, is a finite map from type variables to types, written as $[t_1 \backslash \tau_1, \ldots, t_n \backslash \tau_n]$. An empty substitution is in particular written as $[\,]$. Substitution for a type τ by a type substitution θ, written $\theta(\tau)$, is obtained by simultaneously replacing every type variable t in τ by $\theta(t)$. Substitution for a kind k is similarly defined except for the case k is an object kind. If k is an object kind, a simultaneous replacement may yield an object kind $\{\!\!\{ t_1 \to \tau_1, \ldots, t_n \to \tau_n \}\!\!\}_L$ which contains repeated occurrences of identical type descriptions $t \to \tau$. If there are such repeated occurrences, they are suppressed to a single occurrence.

In our type system, only substitutions that conform to a kind assignment are allowed. A *kinded substitution* is a pair $(\mathcal{K}, \theta)$ of a kind assignment $\mathcal{K}$ and a substitution θ. We say a kinded substitution $(\mathcal{K}, \theta)$ *respects* a kind assignment $\mathcal{K}'$, if $\mathcal{K} \vdash \theta(t) :: \theta(\mathcal{K}'(t))$ for every $t \in domain(\mathcal{K}')$.

Kinded substitution preserves the kinding relation.

Proposition 3.1 *Let $\mathcal{K}$ be a kind assignment, and $(\mathcal{K}', \theta)$ be a kinded substitution that respects $\mathcal{K}$. Then, for any τ and any kind k such that $\mathcal{K} \vdash \tau :: k$, we have $\mathcal{K}' \vdash \theta(\tau) :: \theta(k)$.*

3.3.3 Polytypes

Polymorphism is expressed in our type system by *polytypes*. A polytype σ has the form $\forall t_1 :: k_1, \cdots, t_n :: k_n.\tau$, where $n \geq 0$ and the order of quantification is insignificant. Note that not only the type variables $t_1, \ldots, t_n$ in τ but also those in $k_1, \ldots, k_n$ are quantified. The set of free type variables in a polytype is therefore defined by $FTV(\forall t_1 :: k_1, \cdots, t_n :: k_n.\tau) = (\bigcup_{i=1}^{n} KFTV(k_n) \cup$

$$\frac{}{\mathcal{K}, \Gamma \vdash c : b}\ \textsc{Const} \qquad \frac{x \in domain(\Gamma) \quad \tau \preceq_{\mathcal{K}} \Gamma(x)}{\mathcal{K}, \Gamma \vdash x : \tau}\ \textsc{Var}$$

$$\frac{\mathcal{K}, \Gamma \vdash M : \tau \quad \mathcal{K} \vdash t :: \langle\!\langle \ell(\tau) \rangle\!\rangle_{\mathcal{L}}}{\mathcal{K}, \Gamma \vdash \ell(M) : t}\ \textsc{Message}$$

$$\frac{\begin{array}{l}\mathcal{K}, \Gamma + \{x_i : \tau_i\} \vdash M_i : \tau'_i \quad \text{(for each } i = 1, \cdots, n) \\ \mathcal{K} \vdash t :: \{\!\!\{ t_1 \to \tau'_1, \ldots, t_n \to \tau'_n \}\!\!\}_{\{\ell_1, \ldots, \ell_n\}} \\ \mathcal{K} \vdash t_i :: \langle\!\langle \ell_i(\tau_i) \rangle\!\rangle_{\{\ell_1, \ldots, \ell_n\}} \quad \text{(for each } i = 1, \cdots, n)\end{array}}{\mathcal{K}, \Gamma \vdash \{\ell_1(x_1) = M_1, \ldots, \ell_n(x_n) = M_n\} : t}\ \textsc{Object}$$

$$\frac{\mathcal{K}, \Gamma \vdash M : t \quad \mathcal{K} \vdash t :: \{\!\!\{ s \to \tau \}\!\!\}_L \quad \mathcal{K}, \Gamma \vdash N : s}{\mathcal{K}, \Gamma \vdash M \Leftarrow N : \tau}\ \textsc{MsgPass}$$

$$\frac{\mathcal{K}', \Gamma \vdash M : \tau' \quad (\mathcal{K}, \sigma) = Clos(\mathcal{K}', \Gamma, \tau') \quad \mathcal{K}, \Gamma + \{x : \sigma\} \vdash N : \tau}{\mathcal{K}, \Gamma \vdash \textsf{let}\ x = M\ \textsf{in}\ N : \tau}\ \textsc{Let}$$

$$\frac{\begin{array}{c}\mathcal{K}', \Gamma + \{x : t\} \vdash \{\ell_1(x_1) = M_1, \ldots, \ell_n(x_n) = M_n\} : t \\ (\mathcal{K}, \sigma) = Clos(\mathcal{K}', \Gamma, t) \quad \mathcal{K}, \Gamma + \{x : \sigma\} \vdash N : \tau\end{array}}{\mathcal{K}, \Gamma \vdash \textsf{letobj}\ x = \{\ell_1(x_1) = M_1, \ldots, \ell_n(x_n) = M_n\}\ \textsf{in}\ N : \tau}\ \textsc{LetObj}$$

Figure 3: Typing Rules

$FTV(\tau)) \setminus \{t_1, \ldots, t_n\}$. We say a polytype σ is well-formed under $\mathcal{K}$, if $FTV(\sigma) \subseteq domain(\mathcal{K})$. Furthermore, we assume the *bound variable convention* on the quantified type variables by α-conversion. Under the bound variable convention, we define substitution for polytypes by $\theta(\forall t_1 :: k_1, \cdots, t_n :: k_n.\tau) = \forall t_1 :: \theta(k_1), \cdots, t_n :: \theta(k_n).\theta(\tau)$.

We define the notion of *generic instance* of polytypes. Let $\sigma = \forall t_1 :: k_1, \ldots, t_n :: k_n.\tau$ and $\sigma' = \forall s_1 :: k'_1, \ldots, s_m :: k'_m.\tau'$, be two polytypes, where we assume that $t_1, \ldots, t_n, s_1, \ldots, s_m$ are fresh type variables by the bound variable convention. We say a polytype σ is a generic instance of σ' with respect to a kind assignment $\mathcal{K}$, written $\sigma \preceq_{\mathcal{K}} \sigma'$, if there exists a substitution θ such that $domain(\theta) = \{s_1, \ldots, s_m\}$, kinded substitution $(\mathcal{K} + \{t_1 :: k_1, \ldots, t_n :: k_n\}, \theta)$ respects $\mathcal{K} + \{s_1 :: k'_1, \ldots, s_m :: k'_m\}$, and $\theta(\tau') = \tau$.

3.3.4 Typing Rules

Types are given for expressions relative to a *type assignment*, which is a finite map from identifiers to polytypes, written as $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$. We extend the notion of free type variables to type assignments by $FTV(\Gamma) = \bigcup_{x \in domain(\Gamma)} FTV(\Gamma(x))$. The notion of type substitution is also extended to type assignments, i.e., we write $\theta(\Gamma)$ for a type assignment such that $domain(\theta(\Gamma)) = domain(\Gamma)$ and $\theta(\Gamma)(x) = \theta(\Gamma(x))$ for every $x \in domain(\Gamma)$.

To type let and letobj expressions, we need to take closure of a type with respect to a type assignment. However, we must be careful taking type closure, since the type structure in our type system is expressed by the kinds attached to the type variables by means of the kind assignment. Furthermore, we need to take it into account that the types contained in an object kind may have dependency on one another due to the well-formedness condition. To take a proper set of free type variables, we introduce the notion of *essentially free type variables* that respects the type structure expressed by the kind assignment and the type dependency induced by the well-formedness condition, whose definition is similar to that of Ohori's type system [Oho95]. Given a kind assignment $\mathcal{K}$ and a polytype σ well-formed under $\mathcal{K}$, the set of essentially free type variables denoted by $EFTV(\mathcal{K}, \sigma)$, is defined as the smallest set of type variables satisfying the following conditions.

- $FTV(\sigma) \subseteq EFTV(\mathcal{K}, \sigma)$.
- If $t \in EFTV(\mathcal{K}, \sigma)$, then $KFTV(\mathcal{K}(t)) \subseteq EFTV(\mathcal{K}, \sigma)$.
- If $\mathcal{K}(t) = \{\!\!\{ s \to \tau', \ldots \}\!\!\}_L$ and $s \in EFTV(\mathcal{K}, \sigma)$, then $t \in EFTV(\mathcal{K}, \sigma)$.

The first two conditions say that the set of effectively free type variables contains all the type variables that can be reached by traversing the type structure expressed by kinds. The last condition tells that a type variable kinded by an object kind must be contained in the set of essentially free type variables to preserve the type dependency, if a type of a dynamic message accepted by the object belongs to the set of essentially free type variables. The definition of essentially free type variables is extended to type assignments by $EFTV(\mathcal{K}, \Gamma) = \bigcup_{t \in domain(\Gamma)} EFTV(\mathcal{K}, \Gamma(t))$.

A *type closure* of τ under a kind assignment $\mathcal{K}$ and a type assignment Γ, denoted by $Clos(\mathcal{K}, \Gamma, \tau)$, is a pair $(\mathcal{K}', \sigma)$ of a kind assignment $\mathcal{K}'$ and a polytype $\sigma = \forall t_1 :: \mathcal{K}(t_1), \cdots, t_n :: \mathcal{K}(t_n).\tau$ such that $\{t_1, \ldots, t_n\} = EFTV(\mathcal{K}, \tau) \setminus EFTV(\mathcal{K}, \Gamma)$, $domain(\mathcal{K}) \setminus \{t_1, \ldots, t_n\} = domain(\mathcal{K}')$, and $\mathcal{K}'(t) = \mathcal{K}(t)$ for all $t \in domain(\mathcal{K}')$.

Figure 3 shows the set of rules to derive a typing judgment of the form $\mathcal{K}, \Gamma \vdash M : \tau$, which asserts that an expression M has a type τ under a kind assignment $\mathcal{K}$ and a type assignment Γ. The rule Const gives a proper basic type for each constant. The rule Var gives an instance of the polytype assigned to the identifier. The rule Message gives a type kinded by a message kind $\langle\!\langle \ell(\tau) \rangle\!\rangle_{\mathcal{L}}$ for a message, where ℓ is the method name to be invoked and τ is the argument type. The subscripted set of labels $\mathcal{L}$ indicates that the typing rule does not care about the constraint on the upper

(i) $(E \cup \{(\tau, \tau)\}, \mathcal{K}, \theta) \Longrightarrow (E, \mathcal{K}, \theta)$

(ii) $(E \cup \{(t, \tau)\}, \mathcal{K} \cup \{(t, U)\}, \theta) \Longrightarrow ([t\backslash\tau](E), [t\backslash\tau](\mathcal{K}), [t\backslash\tau] \circ \theta)$
if $t \notin FTV(\tau)$.

(iii) $(E \cup \{(t, s)\}, \mathcal{K} \cup \{(t, \langle\!\langle F \rangle\!\rangle_L), (s, \langle\!\langle F' \rangle\!\rangle_{L'})\}, \theta)$
$\Longrightarrow$ ($[t\backslash s](E \cup \{(F(\ell), F'(\ell)) \mid \ell \in Dom(F) \cap Dom(F')\})$,
$[t\backslash s](\mathcal{K} \cup \{(s, \langle\!\langle F + F' \rangle\!\rangle_{L \cap L'})\})$,
$[t\backslash s] \circ \theta)$
if $Dom(F) \cup Dom(F') \subseteq L \cap L'$.

(iv) $(E, \mathcal{K} \cup \{(t, \{\!\!\{s \to \tau, s \to \tau', \ldots\}\!\!\}_L)\}, \theta) \Longrightarrow (E \cup \{(\tau, \tau')\}, \mathcal{K} \cup \{(t, \{\!\!\{s \to \tau, \ldots\}\!\!\}_L)\}, \theta)$

(v) $(E \cup \{(t, s)\}, \mathcal{K} \cup \{(t, \{\!\!\{R\}\!\!\}_L), (s, \{\!\!\{R'\}\!\!\}_{L'})\} \cup \{(u, \langle\!\langle F_u \rangle\!\rangle_{L_u}) \mid u \in Dom(R + R')\}, \theta)$
$\Longrightarrow$ ($[t\backslash s](E \cup \{(R(u), R'(u)) \mid u \in Dom(R) \cap Dom(R')\})$,
$[t\backslash s](\mathcal{K} \cup \{(s, \{\!\!\{R + R'\}\!\!\}_{L \cap L'})\} \cup \{(u, \langle\!\langle F_u \rangle\!\rangle_{L \cap L' \cap L_u}) \mid u \in Dom(R + R')\})$,
$[t\backslash s] \circ \theta)$
if $Dom(F_u) \subseteq L \cap L'$ for any $u \in Dom(R + R')$.

(vi) $(E, \mathcal{K} \cup \{(t, \{\!\!\{R\}\!\!\}_L), (s_1, \langle\!\langle F_1 \rangle\!\rangle_{L_1}), (s_2, \langle\!\langle F_2 \rangle\!\rangle_{L_2})\}, \theta)$
$\Longrightarrow$ ($E \cup \{(R(s_1), R(s_2))\} \cup \{(F_1(\ell), F_2(\ell)) \mid \ell \in Dom(F_1) \cap Dom(F_2)\}$,
$\mathcal{K} \cup \{(t, \{\!\!\{R\}\!\!\}_L), (s_1, \langle\!\langle F_1 \rangle\!\rangle_{L_1}), (s_2, \langle\!\langle F_2 \rangle\!\rangle_{L_2})\}$,
$\theta)$
if $s_1, s_2 \in Dom(R)$, $Dom(F_1) \cap Dom(F_2) \neq \emptyset$, and either $R(s_1) \neq R(s_2)$ or $F_1(\ell) \neq F_2(\ell)$ for some $\ell \in Dom(F_1) \cap Dom(F_2)$.

Rule (i)–(iv) are prior to (v); Rule (i)–(v) are prior to (vi).

Figure 4: Rewriting Rules for Kinded Unification

bound of the set of method names. The rule OBJECT gives a type kinded by an object kind $\{\!\!\{t_1 \to \tau'_1, \ldots, t_n \to \tau'_n\}\!\!\}_{\{\ell_1,\ldots,\ell_n\}}$. This kinding describes the object as an object accepting n dynamic messages whose types are given as $t_1 :: \langle\!\langle \ell_1(\tau_1) \rangle\!\rangle_{\{\ell_1,\ldots,\ell_n\}}, \ldots, t_n :: \langle\!\langle \ell_n(\tau_n) \rangle\!\rangle_{\{\ell_1,\ldots,\ell_n\}}$, respectively. For each method ℓ_i, the argument type τ_i and the result type τ' is derived by a typing derivation similar to that for functions in functional languages. The rule MSGPASS first derives the type of the object and the type of the message passed to the object. If the message has a type s and the object has such a type $t :: \{\!\!\{s \to \tau\}\!\!\}_L$ that accepts the dynamic message and returns the result of type τ, then the derivation yields τ as the type of the message passing. The rule LET introduces an ML-style let polymorphism by taking a type closure. The rule LETOBJ is a variant of the LET rule, which combines the recursive object typing with the polymorphic typing.

3.4 Type Soundness

We show that the type system is sound with respect to the operational semantics by following Leroy's method for proving the soundness in the presence of recursion [Ler92]. We first define types of values relative to a kind assignment. Value typing is given as the following three relations.

$\mathcal{K} \models v : \tau$	A value v has a type τ relative to $\mathcal{K}$.
$\mathcal{K} \models v : \sigma$	A value v has a polytype σ relative to $\mathcal{K}$.
$\mathcal{K} \models \rho : \Gamma$	A run-time environment ρ has a type assignment Γ relative to $\mathcal{K}$.

These relations are defined on the structure of v and ρ as follows.

- $\mathcal{K} \models c : b$ if c is a constant of base type b.
- $\mathcal{K} \models \ell(v) : t$ if $\mathcal{K}(t) = \langle\!\langle \ell(\tau), \ldots \rangle\!\rangle_L$ and $\mathcal{K} \models v : \tau$.
- $\mathcal{K} \models \{\ell_1(x_1, y_1, M_1, \rho_1), \ldots, \ell_n(x_n, y_n, M_n, \rho_n)\} : t$
  - if $\mathcal{K}(t) = \{\!\!\{t_1 \to \tau_1, \ldots, t_n \to \tau_n\}\!\!\}_L$ such that $\bigcup_i Labels(\mathcal{K}(t_i)) = L = \{\ell_1, \ldots, \ell_n\}$ and
  - for every i and j such that $\mathcal{K}(t_j) = \langle\!\langle \ell_i(\tau), \ldots \rangle\!\rangle_{L'}$, there exists a type assignment Γ such that $\mathcal{K} \models \rho_i : \Gamma$ and $\mathcal{K}, \Gamma + \{x_i : t, y_i : \tau\} \vdash M_i : \tau_i$.
- $\mathcal{K} \models v : \sigma$ if for any kinded substitution $(\mathcal{K}', \theta)$ that respects $\mathcal{K}$ and any type τ such that $\tau \preceq_{\mathcal{K}'} \theta(\sigma)$, $\mathcal{K}' \models v : \tau$.
- $\mathcal{K} \models \rho : \Gamma$ if $domain(\rho) = domain(\Gamma)$ and $\mathcal{K} \models \rho(x) : \Gamma(x)$ for all $x \in domain(\Gamma)$.

We can show that both typing derivation and value typing are preserved by any kinded substitution respecting the kind assignment.

Proposition 3.2 *Let $\mathcal{K}$ be a kind assignment, Γ be a type assignment M be an expression, and τ be a type. If $\mathcal{K}, \Gamma \vdash M : \tau$, then $\mathcal{K}', \theta(\Gamma) \models M : \theta(\tau)$ holds for every kinded substitution $(\mathcal{K}', \theta)$ that respects $\mathcal{K}$.*

Proposition 3.3 *Let $\mathcal{K}$ be a kind assignment, v be a value, and τ be a type. If $\mathcal{K} \models v : \tau$, then $\mathcal{K}' \models v : \theta(\tau)$ holds for every kinded substitution $(\mathcal{K}', \theta)$ that respects $\mathcal{K}$.*

Then we can show the desired property.

Theorem 3.4 (Type soundness) *Let $\mathcal{K}$ be a kind assignment, Γ be a type assignment, and ρ be a run-time environment satisfying $\mathcal{K} \models \rho : \Gamma$. If an expression M has a typing derivation $\mathcal{K}, \Gamma \vdash M : \tau$ and there is an evaluation derivation $\rho \vdash M \Downarrow r$, then we have $\mathcal{K} \models r : \tau$.*

$\mathcal{I}(\mathcal{K},\Gamma,c) = (\mathcal{K},[\,],b)$
where b is the base type of constant c.

$\mathcal{I}(\mathcal{K},\Gamma,x) =$
if $x \notin domain(\Gamma)$ then *failure*
else let $\forall t_1 :: k_1, \cdots, t_n :: k_n.\tau = \Gamma(x)$
where $t_1,\ldots,t_n$ are renamed to fresh type variables.
in $(\mathcal{K} + \{t_1 :: k_1, \cdots, t_n :: k_n\}, [\,], \tau)$

$\mathcal{I}(\mathcal{K},\Gamma,\ell(M)) =$
let $(\mathcal{K}',\theta,\tau) = \mathcal{I}(\mathcal{K},\Gamma,M)$
let t be a fresh type variable
in $(\mathcal{K}' + \{t : \langle\!\langle \ell(\tau)\rangle\!\rangle_{\mathcal{L}}\}, \theta, t)$

$\mathcal{I}(\mathcal{K},\Gamma,\{\ell_1(x_1) = M_1,\ldots,\ell_n(x_n) = M_n\}) =$
let $t, t_1,\ldots,t_n, s_1,\ldots,s_n, u_1,\ldots,u_n$ be fresh type variables
let $L = \{\ell_1,\ldots,\ell_n\}$
let $\mathcal{K}_0 = \mathcal{K} + \{\; t :: \{\!\!\{ t_1 \to u_1,\ldots,t_n \to u_n \}\!\!\}_L,$
$t_1 :: \langle\!\langle \ell_1(s_1)\rangle\!\rangle_L,\ldots,t_n :: \langle\!\langle \ell_n(s_n)\rangle\!\rangle_L,$
$s_1 :: U,\ldots,s_n :: U, u_1 :: U,\ldots,u_n :: U\}$
let $(\mathcal{K}_0',\theta_0',\tau_1) = \mathcal{I}(\mathcal{K}_0,\Gamma + \{x_1 : s_1\}, M_1)$
let $(\mathcal{K}_1,\theta_1) = \mathcal{U}(\mathcal{K}_0', \{(\tau_1,\theta_0'(u_1))\})$
$\vdots$
let $(\mathcal{K}_{n-1}',\theta_{n-1}',\tau_n) = \mathcal{I}(\mathcal{K}_{n-1},\theta_{n-1}\circ\cdots\circ\theta_1\circ\theta_0'(\Gamma + \{x_n : s_n\}), M_n)$
let $(\mathcal{K}_n,\theta_n) = \mathcal{U}(\mathcal{K}_{n-1}', \{(\tau_n,\theta_{n-1}'\circ\theta_{n-1}\circ\cdots\circ\theta_1\circ\theta_0'(u_n))\})$
in $(\mathcal{K}_n, \theta_n\circ\theta_{n-1}'\circ\cdots\circ\theta_1\circ\theta_0', \theta_n\circ\theta_{n-1}'\circ\cdots\circ\theta_1\circ\theta_0'(t))$

$\mathcal{I}(\mathcal{K},\Gamma,M \leftarrow N) =$
let $(\mathcal{K}_0,\theta_0,\tau_0) = \mathcal{I}(\mathcal{K},\Gamma,M)$
let $(\mathcal{K}_1,\theta_1,\tau_1) = \mathcal{I}(\mathcal{K},\theta_0(\Gamma),N)$
let t, s, u be fresh type variables
let $(\mathcal{K}_2,\theta_2) = \mathcal{U}(\mathcal{K}_1 + \{t :: \{\!\!\{ s \to u \}\!\!\}_{\mathcal{L}}, s :: \langle\!\langle\rangle\!\rangle_{\mathcal{L}}, u :: U\}, \{(t,\theta_1(\tau_0)),(s,\tau_1)\})$
in $(\mathcal{K}_2, \theta_2\circ\theta_1\circ\theta_0, \theta_2(u))$

$\mathcal{I}(\mathcal{K},\Gamma,\textsf{let}\; x = M \;\textsf{in}\; N) =$
let $(\mathcal{K}_0,\theta_0,\tau_0) = \mathcal{I}(\mathcal{K},\Gamma,M)$
let $\{t_1,\ldots,t_n\} = EFTV(\mathcal{K}_0,\tau_0) \setminus EFTV(\mathcal{K}_0,\theta_0(\Gamma))$
let $(\mathcal{K}_1,\theta_1,\tau_1) = \mathcal{I}(\mathcal{K}_0,\theta_0(\Gamma) + \{x : \forall t_1 :: \mathcal{K}_0(t_1),\cdots,t_n :: \mathcal{K}_0(t_n).\tau_0\}, N)$
in $(\mathcal{K}_1, \theta_1\circ\theta_0, \tau_1)$

$\mathcal{I}(\mathcal{K},\Gamma,\textsf{letobj}\; x = \{\ell_1(x_1) = M_1,\ldots,\ell_n(x_n) = M_n\} \;\textsf{in}\; N) =$
let t be a fresh type variable.
let $(\mathcal{K}_0,\theta_0,s) = \mathcal{I}(\mathcal{K} + \{t :: \{\!\!\{\}\!\!\}_{\mathcal{L}}\}, \Gamma + \{x : t\}, \{\ell_1(x_1) = M_1,\ldots,\ell_n(x_n) = M_n\})$
let $(\mathcal{K}_1,\theta_1) = \mathcal{U}(\mathcal{K}_0, \{(\theta_0(t), s)\})$
let $\{t_1,\ldots,t_n\} = EFTV(\mathcal{K}_1,\theta_1(s)) \setminus EFTV(\mathcal{K}_1,\theta_1\circ\theta_0(\Gamma))$
let $(\mathcal{K}_2,\theta_2,\tau) = \mathcal{I}(\mathcal{K}_0,\theta_1\circ\theta_0(\Gamma) + \{x : \forall t_1 :: \mathcal{K}_1(t_1),\cdots,t_n :: \mathcal{K}_1(t_n).\theta_1(s)\}, N)$
in $(\mathcal{K}_2, \theta_2\circ\theta_1\circ\theta_0, \tau)$

Figure 5: Type Inference Algorithm

3.5 Type Inference

We present a type inference algorithm for our language. The type inference algorithm needs an auxiliary procedure, *kinded unification*, that unifies types under a kinding restriction.

Let E be a set of tuples of types, where each tuple represents a type equation to be solved. A *kinded set of type equations* is a pair $(\mathcal{K}, E)$ of a kind assignment $\mathcal{K}$ and a set E of type equations, where each equation is an unordered pair of types and every type τ contained in E is well-formed under $\mathcal{K}$. A *unifier* for a kinded set of type equations $(\mathcal{K}, E)$ is a kinded substitution $(\mathcal{K}',\theta)$ respecting $\mathcal{K}$ such that $domain(\mathcal{K}') \cap domain(\theta) = \emptyset$ and $\theta(\tau) = \theta(\tau')$ for every $(\tau,\tau') \in E$. We say a unifier $(\mathcal{K}',\theta)$ is a *most general unifier* for a kinded type equation $(\mathcal{K}, E)$, if for any unifier $(\mathcal{K}'',\theta')$, there exists a substitution θ'' such that $(\mathcal{K}'',\theta'')$ respects $\mathcal{K}'$ and $\theta' = \theta''\circ\theta$.

We define our kinded unification algorithm as a rewriting procedure that unifies a kinded set of type equations, in the spirit of the transformational method for E-unification by Gallier and Sny-

$$\Pi_0 = \dfrac{\begin{array}{c} \dfrac{\vdots}{\mathcal{K}_0, \{\mathsf{m} : s, \mathsf{x} : \mathit{int}\} \vdash \mathsf{x+1} : \mathit{int}} \qquad \dfrac{\vdots}{\mathcal{K}_0, \{\mathsf{m} : s, \mathsf{x} : \mathit{int}\} \vdash \mathsf{x>0} : \mathit{bool}} \\ \mathcal{K}_0 \vdash s_1 :: \langle\!\langle \mathbf{succ}(\mathit{int}) \rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}} \quad \mathcal{K}_0 \vdash s_2 :: \langle\!\langle \mathbf{pos}(\mathit{int}) \rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}} \quad \mathcal{K}_0 \vdash t_1 :: \{\!\!\{ s_1 \to \mathit{int}, s_2 \to \mathit{bool} \}\!\!\}_{\{\mathbf{succ},\mathbf{pos}\}} \end{array}}{\mathcal{K}_0, \{\mathsf{m} : s\} \vdash \{\mathbf{succ}(\mathsf{x})\mathsf{=x+1}, \mathbf{pos}(\mathsf{x})\mathsf{=(x>0)}\} : t_1}$$

$$\dfrac{\begin{array}{c} \mathcal{K}_0 \vdash s_0 :: \langle\!\langle \mathbf{bar}(s) \rangle\!\rangle_{\{\mathbf{bar}\}} \\ \mathcal{K}_0 \vdash t_0 :: \{\!\!\{ s_0 \to u \}\!\!\}_{\{\mathbf{bar}\}} \end{array} \quad \dfrac{\begin{array}{c}\Pi_0 \\ \vdots \\ \mathcal{K}_0, \{\mathsf{m} : s\} \vdash \{\mathbf{succ}(\mathsf{x})\mathsf{=x+1}, \mathbf{pos}(\mathsf{x})\mathsf{=(x>0)}\} : t_1 \end{array} \quad \mathcal{K}_0 \vdash t_1 :: \{\!\!\{ s \to u \}\!\!\}_{\{\mathbf{succ},\mathbf{pos}\}} \quad \mathcal{K}_0, \{\mathsf{m} : s\} \vdash \mathsf{m} : s}{\mathcal{K}_0, \{\mathsf{m} : s\} \vdash \{\mathbf{succ}(\mathsf{x})\mathsf{=x+1}, \mathbf{pos}(\mathsf{x})\mathsf{=(x>0)}\} \leftarrow \mathsf{m} : u}}{\mathcal{K}_0, \{\} \vdash \{\mathbf{bar}(\mathsf{m}) \mathsf{=} \{\mathbf{succ}(\mathsf{x})\mathsf{=x+1}, \mathbf{pos}(\mathsf{x})\mathsf{=(x>0)}\} \leftarrow \mathsf{m}\} : t_0}$$

where $\mathcal{K}_0 = \{\, t_0 :: \{\!\!\{ s_0 \to u \}\!\!\}_{\{\mathbf{bar}\}}, t_1 :: \{\!\!\{ s \to u, s_1 \to \mathit{int}, s_2 \to \mathit{bool} \}\!\!\}_{\{\mathbf{succ},\mathbf{pos}\}}, u :: U,$
$s :: \langle\!\langle\rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}}, s_0 :: \langle\!\langle \mathbf{bar}(s) \rangle\!\rangle_{\{\mathbf{bar}\}}, s_1 :: \langle\!\langle \mathbf{succ}(\mathit{int}) \rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}}, s_2 :: \langle\!\langle \mathbf{pos}(\mathit{int}) \rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}} \,\}.$

(a) Object definition

$$\dfrac{\dfrac{\vdots}{\mathcal{K}_1, \{\} \vdash \{\mathbf{bar}(\mathsf{m}) \text{ = ...}\} : t_0} \quad \mathcal{K}_1 \vdash t_0 :: \{\!\!\{ s_0 \to \mathit{int} \}\!\!\}_{\{\mathbf{bar}\}} \quad \dfrac{\dfrac{\mathcal{K}_1, \{\} \vdash \mathsf{3} : \mathit{int} \quad \mathcal{K}_1 \vdash s :: \langle\!\langle \mathbf{succ}(\mathit{int}) \rangle\!\rangle_{\mathcal{L}}}{\mathcal{K}_1, \{\} \vdash \mathbf{succ}(\mathsf{3}) : s} \quad \mathcal{K}_1 \vdash s_0 :: \langle\!\langle \mathbf{bar}(s) \rangle\!\rangle_{\mathcal{L}}}{\mathcal{K}_1, \{\} \vdash \mathbf{bar}(\mathbf{succ}(\mathsf{3})) : s_0}}{\mathcal{K}_1, \{\} \vdash \{\mathbf{bar}(\mathsf{m}) \mathsf{=} \{\mathbf{succ}(\mathsf{x})\mathsf{=x+1}, \mathbf{pos}(\mathsf{x})\mathsf{=(x>0)}\} \leftarrow \mathsf{m}\} \leftarrow \mathbf{bar}(\mathbf{succ}(\mathsf{3})) : \mathit{int}}$$

where $\mathcal{K}_1 = \{\, t_0 :: \{\!\!\{ s_0 \to \mathit{int} \}\!\!\}_{\{\mathbf{bar}\}}, t_1 :: \{\!\!\{ s \to \mathit{int}, s_1 \to \mathit{int}, s_2 \to \mathit{bool} \}\!\!\}_{\{\mathbf{succ},\mathbf{pos}\}},$
$s :: \langle\!\langle \mathbf{succ}(\mathit{int}) \rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}}, s_0 :: \langle\!\langle \mathbf{bar}(s) \rangle\!\rangle_{\{\mathbf{bar}\}}, s_1 :: \langle\!\langle \mathbf{succ}(\mathit{int}) \rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}}, s_2 :: \langle\!\langle \mathbf{pos}(\mathit{int}) \rangle\!\rangle_{\{\mathbf{succ},\mathbf{pos}\}} \,\}.$

(b) Message passing

Figure 6: Examples of Typing Derivation

der [GS89]. The rewriting procedure is given in Figure 4 by a set of rules for rewriting triple $(E, \mathcal{K}, \theta)$, where $(\mathcal{K}, E)$ is a kinded set of type equations to be solved and θ is a type substitution for unifying the type equations that have been solved so far. We impose some priority between the rewriting rules; we say a rewriting rule A is prior to B, if the rewriting rule B can be applied only when the rule A cannot. In each rewriting rule, a kind assignment $\mathcal{K}$ is expressed by a set of ordered pairs of type variable and its kind, i.e., $\{(t, \mathcal{K}(t)) \mid t \in \mathit{domain}(\mathcal{K})\}$. We define substitution to a set E of type equations by $\theta(E) = \{(\theta(\tau_1), \theta(\tau_2)) \mid (\tau_1, \tau_2) \in E\}$. Furthermore, we use the following notations. We write a message kind as $\langle\!\langle F \rangle\!\rangle_L$, where F is a finite map from a set of labels $\ell_1, \ldots, \ell_n$ to the corresponding argument types representing a sequence $\ell_1(F(\ell_1)), \ldots, \ell_n(F(\ell_n))$. Similarly, we write an object kind as $\{\!\!\{R\}\!\!\}_L$, where R is a finite map from type variables to types representing a sequence $t_1 \to R(\tau_1), \ldots, t_n \to R(\tau_n)$ such that $t_1, \ldots, t_n$ are distinct.

Given a kinded set of type equations $(\mathcal{K}, E)$, suppose that triple $(E, \mathcal{K}, [\,])$ is rewritten to a triple $(\emptyset, \mathcal{K}', \theta)$ by the rewriting rules and that no rewriting rule cannot be applied any more. Then, we define the result of kinded unification algorithm $\mathcal{U}$ by $\mathcal{U}(\mathcal{K}, E) = (\mathcal{K}', \theta)$; If there is no such rewriting, we define instead $\mathcal{U}(\mathcal{K}, E) = \mathit{fail}$. The following property holds for the kinded unification algorithm $\mathcal{U}$.

Theorem 3.5 *Let $(\mathcal{K}, E)$ be a kinded set of type equations. If $(\mathcal{K}, E)$ has a unifier, the unification algorithm $\mathcal{U}$ computes a most general unifier $(\mathcal{K}', \theta)$; otherwise the algorithm reports fail.*

We give a type inference algorithm $\mathcal{I}$, a variant of Damas-Milner's ML type inference algorithm W [DM82], in Figure 5. The algorithm fails if and only if either the algorithm returns *failure* or the subsidiary kinded unification algorithm $\mathcal{U}$ fails in unification; Otherwise, $\mathcal{I}(\mathcal{K}, \Gamma, M)$ returns a triple $(\mathcal{K}', \theta, \tau)$. We say a triple $(\mathcal{K}', \theta, \tau)$ is a *typing* of an expression M under a kind assignment $\mathcal{K}$ and a type assignment Γ, if there is a typing derivation to yield $\mathcal{K}', \theta(\Gamma) \vdash M : \tau$ and $(\mathcal{K}', \theta)$ respects $\mathcal{K}$. We say $(\mathcal{K}', \theta, \tau)$ is a *most principal typing*, if for any typing $(\mathcal{K}'', \theta', \tau')$ of M under $\mathcal{K}$ and Γ there exists a substitution ψ such that $(\mathcal{K}'', \psi)$ respects $\mathcal{K}'$, $\psi(\tau) = \tau'$, and $\psi \circ \theta = \theta'$. The following theorem shows that the type inference algorithm $\mathcal{I}$ infers a most principal typing.

Theorem 3.6 (Principality of type inference algorithm) *Let $\mathcal{K}$ be a kind assignment, Γ be a type assignment, and M be an expression. $\mathcal{I}(\mathcal{K}, \Gamma, M)$ successfully returns a most principal typing if and only if M has a typing under $\mathcal{K}$ and Γ.*

4 Typing Examples and Class Definition

We show some examples of typing derivations. We first consider the following expression defining an object

{**bar**(m) = {**succ**(x)=x+1, **pos**(x)=(x>0)}←m}

whose only method is the **bar** method defined in Section 2. The derivation tree is given in Figure 6 (a), where Π_0 is a sub-derivation tree for the inner object definition, {**succ**(x)=..., **pos**(x)=...}. The **bar** method is typed as a method which receives a message, whose type is $s :: \langle\!\langle\rangle\!\rangle_{\{\textbf{succ},\textbf{pos}\}}$, and returns the result of passing the message to the inner object. The type of the result of the message passing is expressed by $u :: U$, whose proper type is determined when a message is given as the argument of the **bar** method, as shown in the following expression.

{**bar**(m) =
 {**succ**(x)=x+1, **pos**(x)=(x>0)}←m}←**bar**(**succ**(3))

The typing derivation for this expression is given in Figure 6 (b). To type this expression, we need to alter the kinding for the dynamic message m to include the method **succ**, i.e., $s :: \langle\!\langle \textbf{succ}(int)\rangle\!\rangle_{\{\textbf{succ},\textbf{pos}\}}$. Furthermore, we need to fix the kinding for the inner object so that it conforms the well-formedness condition by unifying u and int. Due to this unification, we have the expected result type int, for the above message passing.

Our language can define recursive objects by using letobj, and the type of recursive objects is expressed in our type system as recursively kinded types, where the recursiveness is expressed by mutual reference between types and kinds. As an example of such recursive typing, we consider a class definition, which is encoded as an object with a parameterized method for creating instances as described in [CHC90]. The following example defines a simple point class.

letobj point =
 {**new**(x) = {**get**()=x, **move**(y)=point←**new**(x+y)}}

where the method **new** returns an instance of the point class. The instance consists of two methods, **get** for returning the initial location given as the parameter when the instance is created, and **move** for creating a new instance whose location is moved according to the given argument. By a typing derivation similar to the above example, the point class is typed as an object whose only method **new** receives an integer parameter and returns an instance of the following type

$$t :: \{\!|s_1 :: \langle\!\langle \textbf{get}(unit)\rangle\!\rangle_L \to int, s_2 :: \langle\!\langle \textbf{move}(int)\rangle\!\rangle_L \to t|\!\}_L,$$

where $L = \{\textbf{get}, \textbf{move}\}$. The type of the instance represents a recursive object, where the result type of the method **move** refers to the type of the object itself.

5 Conclusion and Future Work

We have presented a second order polymorphic type system for an object-oriented language with dynamic messages. The type system offers a powerful static typing discipline that allows typing dynamic messages with insufficient static information about method names. The type system takes an approach to determine types lazily when sufficient static information is supplied, which allows the result type of the method to vary depending on the argument type. The language can encode classes by recursive objects, where the types of recursive objects are expressed as recursively kinded types in the type system. The type system is sound with respect to the operational semantics of the language, and the types of programs can be reconstructed by a type inference algorithm.

The language presented in the paper is a small language that provides no class inheritance mechanism. To support inheritance based on the class encoding described in Section 4, we need to extend the language with an object concatenation operator. However, it seems difficult to type the object concatenation operator in the framework of the presented type system, since the object concatenation requires negative information to tell that some methods must not be contained in an object. To avoid the complexity of treating negative information, we would have to put some syntactic restriction on class definitions so that the set of method names can be determined statically.

Subtyping is another possible way to support class inheritance. A possible strategy to support subtyping in our type system would be to define a *subkinding* relation $\sqsubseteq$ under a kind assignment $\mathcal{K}$ as follows.

- $\langle\!\langle \ell_1(\tau_1), \ldots, \ell_n(\tau_n), \ldots\rangle\!\rangle_L \sqsubseteq \langle\!\langle \ell_1(\tau_1'), \ldots, \ell_n(\tau_n')\rangle\!\rangle_{L'}$ if $L \subseteq L'$ and $\tau_i \sqsubseteq \tau_i'$ for every $i = 1, \ldots, n$.
- $\{\!|t_1 \to \tau_1, \ldots, t_n \to \tau_n, \ldots|\!\}_L \sqsubseteq \{\!|t_1' \to \tau_1', \ldots, t_n' \to \tau_n'|\!\}_{L'}$ if $L \subseteq L'$ and also $\mathcal{K}(t_i') \sqsubseteq \mathcal{K}(t_i)$ and $\tau_i \sqsubseteq \tau_i'$ for every $i = 1, \ldots, n$.

However, it seems difficult to incorporate this subkinding relation into a sound type system. We need further careful investigation on this topic.

As mentioned in Section 1, the object-oriented language enriched with dynamic messages suggests a calculus of objects and messages, which are interacting with each other as the duals of one another, and therefore our type system provides a foundation for a typed calculus of the duals. The calculus has an interesting typing property that identically named methods invoked by the same message can yield differently typed results. We might be able to apply this property to other type systems. Besides, in a distributed environment, the calculus utilizes the distributed resources in a different way from that of λ-calculus. The example of the network file server given in Section 1 indicates a difference between the abstraction mechanisms of higher-order functions and dynamic messages. Suppose that dynamic messages are encoded by functions. In the case, function closures packaging the local computational resources, namely the run-time environment and the program code, are passed over the network, and the function closures are executed on the server machine. This corresponds to the conventional *migration* mechanism in distributed computing. In contrast to the migration based approach, dynamic messages passed to a remote server transfer only a small amount of local resources, namely arguments, to the remote server and mainly utilize remote resources. Dynamic messages seem to have the advantage over function closures of producing a smaller amount of network traffic than function closures. Further investigation on this topic would shed some light on the utilization of distributed resources in distributed computing.

Acknowledgment

I would like to thank anonymous referees whose comments are very valuable for improving the paper. I am also grateful to Atsushi Ohori for his advice and encouragement.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[AWL94] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[BB96] F. Barbanera and S. Berardi. A symmetric lambda calculus for classical program extraction. *Information and Computation*, 125:103–117, 1996.

[BNOW93] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 217–230, 1993.

[BPF97] K. B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97 Proceedings*, volume 1242 of *LNCS*, pages 104–127, 1997.

[BSvG95] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *ECOOP '95 Proceedings*, volume 952 of *LNCS*, pages 27–51, 1995.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2–3):138–164, 1988.

[CCH+89] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of ACM Conference on Functional Programming and Computer Architecture*, pages 273–280, 1989.

[CHC90] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

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

[GM94] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming* Types, Semantics, and Language Design. The MIT Press, 1994.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80 The Language*. Addison-Wesley, 1989.

[GS89] J. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67(2):203–260, 1989.

[Gun92] C. A. Gunter. *Semantics of Programming Languages*. The MIT Press, 1992.

[HP95] M. Hofmann and B. C. Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, 1995.

[Ler92] Xavier Leroy. Polymorphic typing of an algorithmic language. Ph.D. thesis RR-1778, INRIA, 1992.

[Mit90] J. C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 109–124, 1990.

[OB89] A. Ohori and P. Buneman. Static type inference for parametric classes. In *Proceedings of ACM OOPSLA conference*, pages 121–148, 1989.

[Oho95] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995.

[PT94] B. C. Pierce and D. N. Turner. Simple type theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[PW91] L. J. Pinson and R. S. Wiener. *Objective-C: object-oriented programming techniques*. Addison-Wesley, 1991.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 77–87, 1989.

[Rém94] Didier Rémy. Programming objects with ML-ART, an extension to ML with abstract and record types. In *Theoretical Aspects of Computer Software*, pages 321–346, 1994.

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

Appendix: Summary of Type Soundness Proof

Let $\Phi_{\mathcal{K}}$ be a function defined as follows

$$\begin{aligned}\Phi_{\mathcal{K}}(S) = \; & S \\ & \cup \bigcup_{t \in S} KFTV(\mathcal{K}(t)) \\ & \cup \{t \in domain(\mathcal{K}) \mid \mathcal{K}(t) = \{\!\!\{ s \to \tau, \ldots \}\!\!\}_L \text{ and } s \in S\},\end{aligned}$$

where $S \subseteq domain(\mathcal{K})$. The set of essentially free type variables can be defined by means of iteration of this function, i.e.,

$$\begin{aligned} EFTV(\mathcal{K}, \sigma) &= \Phi_{\mathcal{K}}^{\infty}(FTV(\sigma)) = \textstyle\bigcup_{n=0}^{\infty} \Phi_{\mathcal{K}}^{n}(FTV(\sigma)) \\ EFTV(\mathcal{K}, \Gamma) &= \Phi_{\mathcal{K}}^{\infty}(FTV(\Gamma)) = \textstyle\bigcup_{n=0}^{\infty} \Phi_{\mathcal{K}}^{n}(FTV(\Gamma)). \end{aligned}$$

The following lemma holds.

Lemma A.1 *Let $\mathcal{K}, \mathcal{K}'$ be kind assignments, θ be a type substitution such that $(\mathcal{K}', \theta)$ respects $\mathcal{K}$.*

(1) *For any $t \in domain(\mathcal{K})$ and any kind k such that $\mathcal{K} \vdash t :: k$, we have $KFTV(\mathcal{K}(t)) \supseteq KFTV(k)$.*

(2) *For any $t \in domain(\mathcal{K})$,*
$$KFTV(\theta(\mathcal{K}(t))) \subseteq \bigcup_{s \in FTV(\theta(t))} KFTV(\mathcal{K}'(s)).$$

(3) $$\bigcup_{t \in \Phi_{\mathcal{K}}^{n}(S)} FTV(\theta(t)) \subseteq \Phi_{\mathcal{K}'}^{n}\Big(\bigcup_{t \in S} FTV(\theta(t))\Big)$$
As a corollary,
$$\bigcup_{t \in EFTV(\mathcal{K}, \sigma)} FTV(\theta(t)) \subseteq EFTV(\mathcal{K}', \theta(\sigma))$$
$$\bigcup_{t \in EFTV(\mathcal{K}, \Gamma)} FTV(\theta(t)) \subseteq EFTV(\mathcal{K}', \theta(\Gamma)).$$

Proof. Property 1 is immediately from the definition of the kinding relation. Property 2 is proved by case analysis on $\mathcal{K}(t)$. Property 3 is proved by induction on n, using the property 2. □

The proof of Proposition 3.2 is given below.

Proof. Proof is by induction on the height of typing derivation. We prove the induction step by case analysis on the last rule used to derive the typing derivation.

We only consider the most involving case that the last rule is LET. The typing derivation is as follows.

$$\frac{\mathcal{K}_0, \Gamma \vdash M : \tau' \quad (\mathcal{K}, \sigma) = Clos(\mathcal{K}_0, \Gamma, \tau') \quad \mathcal{K}, \Gamma + \{x : \sigma\} \vdash N : \tau}{\mathcal{K}, \Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \tau}$$

By the definition of type closure, $\sigma = \forall t_1 :: \mathcal{K}_0(t_1), \cdots, t_m :: \mathcal{K}_0(t_m).\tau'$ such that $\{t_1, \ldots, t_m\} = EFTV(\mathcal{K}_0, \tau') \setminus EFTV(\mathcal{K}_0, \Gamma)$, $domain(\mathcal{K}_0) = \{t_1, \ldots, t_m\} \cup domain(\mathcal{K})$, and $\mathcal{K}_0(t) = \mathcal{K}(t)$ for all $t \in domain(\mathcal{K})$. We can assume, by the bound variable convention, $t_1, \ldots, t_m$ are fresh variables that appear neither in $\mathcal{K}$ nor in $\mathcal{K}'$. Let $\mathcal{K}_1 = \mathcal{K}' + \{t_1 :: \theta(\mathcal{K}_0(t_1)), \ldots, t_m :: \theta(\mathcal{K}_0(t_m))\}$. It is easy to show that $(\mathcal{K}_1, \theta)$ respects $\mathcal{K}_0$.

By applying the induction hypothesis to the two premises, we have typing derivations $\mathcal{K}_1, \theta(\Gamma) \vdash M : \theta(\tau')$ and $\mathcal{K}', \theta(\Gamma) + \{x : \theta(\sigma)\} \vdash N : \theta(\tau)$. We will show $Clos(\mathcal{K}_1, \theta(\Gamma), \theta(\tau')) = (\mathcal{K}', \theta(Clos(\mathcal{K}, \Gamma, \tau')))$. First, we show $EFTV(\mathcal{K}_1, \theta(\tau')) \setminus EFTV(\mathcal{K}_1, \theta(\Gamma)) = \{t_1, \ldots, t_m\}$. We have $\{t_1, \ldots, t_m\} \subseteq EFTV(\mathcal{K}_1, \theta(\tau'))$, by lemma A.1 (3). Then we prove $\{t_1, \ldots, t_m\} \cap EFTV(\mathcal{K}_1, \theta(\Gamma)) = \emptyset$, i.e. $\{t_1, \ldots, t_m\} \cap \Phi^n_{\mathcal{K}_1}(FTV(\theta(\Gamma))) = \emptyset$. Proof is by induction on n. If $n = 0$, $t_i \notin \Phi^n_{\mathcal{K}_1}(FTV(\theta(\Gamma))) = FTV(\theta(\Gamma)) \subseteq domain(\mathcal{K}')$ for every i. Suppose that the equation holds for n and that $t_i \in \Phi^{n+1}_{\mathcal{K}_1}(FTV(\theta(\Gamma)))$ for some i. By the definition of $\Phi_{\mathcal{K}_1}$ and the induction hypothesis, either $t_i \in \bigcup_{t \in \Phi^n_{\mathcal{K}_1}(FTV(\theta(\Gamma)))} KFTV(\mathcal{K}_1(t))$ or there exists $s \in \Phi^n_{\mathcal{K}_1}(FTV(\theta(\Gamma)))$ such that $\mathcal{K}_1(t_i) = \{\!\{s \to \tau, \ldots\}\!\}_L$. However, since t_i is assumed to be a fresh type variable, either of these conditions cannot be satisfied due to the definition of $EFTV$. Therefore, the inclusion $\{t_1, \ldots, t_m\} \subseteq EFTV(\mathcal{K}_1, \theta(\tau')) \setminus EFTV(\mathcal{K}_1, \theta(\Gamma))$ holds. To show the converse inclusion, we prove the following property: for any $t \in \Phi^n_{\mathcal{K}_1}(FTV(\theta(\tau'))) \setminus EFTV(\mathcal{K}_1, \theta(\Gamma))$, $t = t_i$ for some i for some n. Proof is by induction on n by refutation. If $n = 0$ and there exists $t \in FTV(\theta(\tau')) \setminus EFTV(\mathcal{K}_1, \theta(\Gamma))$ such that t is not any of $t_1, \ldots, t_m$, then there exists $s \in FTV(\tau')$ such that s is not any of $t_1, \ldots, t_m$ (hence, $s \in EFTV(\mathcal{K}_0, \Gamma)$) and $t \in FTV(\theta(s))$. Therefore, by lemma A.1 (3), we have $t \in FTV(\theta(s)) \subseteq EFTV(\mathcal{K}_1, \theta(\Gamma))$. This contradicts to $t \notin EFTV(\mathcal{K}_1, \theta(\Gamma))$. Suppose the property holds for n and there exists t such that $t \in \Phi^{n+1}_{\mathcal{K}_1}(FTV(\theta(\tau'))) \setminus EFTV(\mathcal{K}_1, \theta(\Gamma))$ and t is not any of $t_1, \ldots, t_m$. By the definition of $\Phi_{\mathcal{K}_1}$, we need to check three cases. (a) If $t \in \Phi^n_{\mathcal{K}_1}(FTV(\theta(t)))$, the proof has finished by the induction hypothesis. (b) Suppose there exists $t' \in \Phi^n_{\mathcal{K}_1}(FTV(\theta(\tau')))$ such that $t \in KFTV(\mathcal{K}_1(t'))$. By the induction hypothesis, either $t' = t_i$ for some i or $t' \in EFTV(\mathcal{K}_1, \theta(\Gamma))$. If $t' \in EFTV(\mathcal{K}_1, \theta(\Gamma))$, then $t \in EFTV(\mathcal{K}_1, \theta(\Gamma))$ by the definition of $EFTV$, which contradicts to $t \notin EFTV(\mathcal{K}_1, \theta(\Gamma))$. Suppose $t' = t_i$ for some i. Then, we have $t \in KFTV(\mathcal{K}_1(t')) = KFTV(\theta(\mathcal{K}_0(t_i)))$, and therefore there exits $s \in KFTV(\mathcal{K}_0(t_i))$ such that $t \in FTV(\theta(s))$. Notice that $KFTV(\mathcal{K}_0(t_i)) \subseteq \{t_1, \ldots, t_m\} \cup EFTV(\mathcal{K}_0, \Gamma)$. If $s = t_j$ for some j, then $t = t_j$, contradicting to the assumption. If $s \in EFTV(\mathcal{K}_0, \Gamma)$ otherwise, by lemma A.1 (3), we have $t \in FTV(\theta(s)) \subseteq EFTV(\mathcal{K}_1, \theta(\Gamma))$, which contradicts to $t \in EFTV(\mathcal{K}_1, \theta(\Gamma))$. (c) Suppose there exists $s \in \Phi^n_{\mathcal{K}_1}(FTV(\theta(\tau')))$ such that $\mathcal{K}_1(t) = \{\!\{s \to \tau, \ldots\}\!\}_L$. Since $domain(\mathcal{K}_1) = \{t_1, \ldots, t_n\} \cup domain(\mathcal{K}')$ and t is not any of $t_1, \ldots, t_n$, $t \in domain(\mathcal{K}')$. Therefore s is not any of $t_1, \ldots, t_n$, and by the induction hypothesis, $s \in EFTV(\mathcal{K}_1, \theta(\Gamma))$. By the definition of $EFTV$, this implies $t \in EFTV(\mathcal{K}_1, \theta(\Gamma))$, which contradicts to $t \notin EFTV(\mathcal{K}_1, \theta(\Gamma))$.

By the discussion above, we can conclude $EFTV(\mathcal{K}_1, \theta(\tau')) \setminus EFTV(\mathcal{K}_1, \theta(\Gamma)) = \{t_1, \ldots, t_m\}$ and we have $Clos(\mathcal{K}', \theta(\Gamma), \theta(\tau')) = (\mathcal{K}_1, \forall t_1 :: \theta(\mathcal{K}(t_1)), \cdots, t_m :: \theta(\mathcal{K}(t_m)).\theta(\tau')) = (\mathcal{K}_1, \theta(\sigma))$. Therefore, we can get the typing derivation $\mathcal{K}', \theta(\Gamma) \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \theta(\tau)$ by applying the LET rule. □

Proposition 3.3 is proved by routine induction on the structure of value v, by using proposition 3.2. The proposition leads the following corollary.

Corollary A.2 *If* $\mathcal{K} + \{t_1 :: k_1, \ldots, t_n :: k_n\} \models v : \tau$ *and* $domain(\mathcal{K}) \cap \{t_1, \ldots, t_n\} = \emptyset$, *then* $\mathcal{K} \models v : \forall t_1 :: k_1, \cdots, t_n :: k_n.\tau$.

Proof. By the bound variable convention and the definition of generic instance. □

The type soundness theorem (theorem 3.4) is proved as follows.

Proof. Proof is by induction on the height of evaluation derivation. We argue each induction step by case analysis on the last rule used to derive the derivation. Only a few critical cases are examined in the following.

Case VAR *rule.* $M = x$ and the only possible typing is

$$\frac{x \in domain(\Gamma) \quad \tau \preceq_{\mathcal{K}} \Gamma(x)}{\mathcal{K}, \Gamma \vdash x : \tau}$$

Since $x \in domain(\Gamma)$, $x \in domain(\rho)$ and $r = \rho(x)$. We have $\mathcal{K} \models \rho(x) : \Gamma(x)$ by the hypothesis $\mathcal{K} \models \rho : \Gamma$. Since $\tau \preceq_{\mathcal{K}} \Gamma(x)$, by the definition of relation $\models$, we conclude $\mathcal{K} \models \rho(x) : \tau$.

Case MSGPASS *rule.* The only possible typing is

$$\frac{\mathcal{K}, \Gamma \vdash M : t \quad \mathcal{K} \vdash t :: \{\!\{s \to \tau\}\!\}_L \quad \mathcal{K}, \Gamma \vdash N : s}{\mathcal{K}, \Gamma \vdash M \leftarrow N : \tau}$$

By the induction hypothesis, we have the following four relations.

$$\begin{array}{ll} \rho \vdash M \Downarrow \{\ell_1(x_1, y_1, M_1, \rho_1), \ldots\}, & \rho \vdash M \Downarrow \ell(v'), \\ \mathcal{K} \models \{\ell_1(x_1, y_1, M_1, \rho_1), \ldots\} : t, & \text{and } \mathcal{K} \models \ell(v') : s. \end{array}$$

Since $\mathcal{K} \vdash t :: \{\!\{s : \tau\}\!\}_L$, there exists i such that $\ell_i = \ell$ and $\mathcal{K}(s) = \langle\!\langle \ell_i(\tau'), \ldots \rangle\!\rangle_{L'}$. Hence, there exists a type assignment Γ such that $\mathcal{K} \vdash \rho_i : \Gamma$ and $\mathcal{K}, \Gamma + \{x_i : t, y_i : \tau'\} \vdash M_i : \tau$. Since $\mathcal{K} \models \rho_i + \{x_i \mapsto \{\ell(x_i, y_i, M_i, \rho_i), \ldots\}, y_i \mapsto v'\} : \Gamma + \{x_i : t, y_i : \tau'\}$, by applying the induction hypothesis, we have $\rho_i + \{x_i \mapsto \{\ell_i(x_i, y_i, M_i, \rho_i), \ldots\}, y_i \mapsto v'\} \vdash M_i \Downarrow v$ and $\mathcal{K} \models v : \tau$.

Case LET *rule.* The only possible typing is

$$\frac{\mathcal{K}', \Gamma \vdash M : \tau' \quad (\mathcal{K}, \sigma) = Clos(\mathcal{K}', \Gamma, \tau') \quad \mathcal{K}, \Gamma + \{x : \sigma\} \vdash N : \tau}{\mathcal{K}, \Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \tau}$$

By the induction hypothesis, we have $\rho \vdash M \Downarrow v'$ and $\mathcal{K} \models v' : \tau'$. By corollary A.2, $\mathcal{K} \models v' : \sigma$ and $\mathcal{K} \models \rho + \{x : v'\} : \Gamma + \{x : \sigma\}$. By the induction hypothesis, we have $\rho + \{x : v'\} \vdash N \Downarrow v$ and $\mathcal{K} \models v : \tau$. □