A Typeful Approach to Object-Oriented Programming with Multiple Inheritance*

Chiyan Chen, Rui Shi, and Hongwei Xi

Computer Science Department

Boston University

{chiyan,shearer,hwxi}@cs.bu.edu

Abstract. The wide practice of objected oriented programming (OOP) in current software practice is evident. Despite extensive studies on typing programming objects, it is still undeniably a challenging research task to design a type system that can satisfactorily account for a variety of features (e.g., binary methods and multiple inheritance) in OOP. In this paper, we present a typeful approach to implementing objects that makes use of a recently introduced notion of guarded datatypes. In particular, we demonstrate how the feature of multiple inheritance can be supported with this approach, presenting a simple and general account for multiple inheritance in a typeful manner.

1 Introduction

The popularity of object-oriented programming (OOP) in current software practice is evident. While this popularity may result in part from the tendency of programmers to chase after the latest "fads" in programing languages, there is undeniably some real substance in the growing use of OOP. For instance, the inheritance mechanism in OOP offers a highly effective approach to facilitating code reuse. There are in general two common forms of inheritance in OOP: single inheritance and multiple inheritance. In object-oriented languages such as Smalltalk and Java, only single inheritance is allowed, that is, a (sub)class can inherit from at most one (super)class. On the other hand, in object-oriented languages such as C++ and Eiffel, multiple inheritance, which allows a (sub)class to inherit from more than one (super)classes, is supported. We have previously outlined an approach to implementing objects through the use of guarded recursive datatypes [XCC03]. While it addresses many difficult issues in OOP (e.g., parametric polymorphism, binary methods, the self type, etc.) in a simple and natural manner, it is unclear, *a priori* whether this approach is able to cope with multiple inheritance. In this paper, we are to make some significant adjustment to this approach so that the issue of multiple inheritance can also be properly dealt with in a typeful manner, and we believe that such a typeful treatment of multiple inheritance is entirely novel in the literature.

* Partially supported by NSF grants no. CCR-0224244 and no. CCR-0229480

We take a view of objects in the spirit of Smalltalk [GR83,Liu96]; we suggest to conceptualize an object as a little intelligent being capable of performing actions according to the messages it receives; we suggest not to think of an object as a record of fields and methods in this paper. More concretely, we are to implement an object as a function that interprets messages received by the object. We first present a brief outline of this idea. Let *MSG* be a guarded recursive datatype constructor that takes a type τ to form a message type $MSG(\tau)$. We require that *MSG* be extensible (like the exception type in ML). Intuitively, an object is expected to return a value of type τ after receiving a message of type $MSG(\tau)$. Therefore, we assign an object the following type *OBJ*:

$$OBJ = \forall\alpha.MSG(\alpha) \rightarrow \alpha$$

Suppose that we have declared through some syntax that *MSGgetfst*, *MSGgetsnd*, *MSGsetfst* and *MSGsetsnd* are message constructors of the following types,

$$\begin{array}{rcl@{\qquad}rcl} MSGgetfst & : & MSG(int) & MSGsetfst & : & int \rightarrow MSG(\mathbf{1}) \\ MSGgetsnd & : & MSG(int) & MSGsetsnd & : & int \rightarrow MSG(\mathbf{1}) \end{array}$$

where **1** stands for the unit type.[1] We can now implement integer pairs as follows in a message-passing style:

```
fun newIntPair x y = let
  val xref = ref x and yref = ref y
  fun dispatch msg =
    case msg of
     | MSGgetfst => !xref
     | MSGgetsnd => !yref
     | MSGsetfst x' => (xref := x')
     | MSGsetsnd y' => (yref := y')
     | _ => raise UnknownMessage
in dispatch end
withtype int -> int -> OBJ
```

The above program is written in the syntax of ATS, a functional programming language we are developing that is equipped with a type system rooted in the framework *Applied Type System* [Xi03,Xi04]. The syntax should be easily accessible for those who are familiar with the syntax of Standard ML [MTHM97]. The `withtype` clause in the program is a type annotation that assigns the type $int \rightarrow int \rightarrow OBJ$ to the defined function *newIntPair*.[2] Given integers x and y, we can form an integer pair object *anIntPair* by calling $newIntPair(x)(y)$; we can then send the message *MSGgetfst* to the object to obtain its first component: $anIntPair(MSGgetfst)$; we can also reset its first component to x' by sending the message $MSGsetfst(x')$ to the object: $anIntPair(MSGsetfst(x'))$; operations on

[1] Note that it is solely for illustration purpose that we use the prefix *MSG* in the name of each message constructor.

[2] The reason for *newIntPair* being well-typed can be found in our work on guarded recursive datatypes [XCC03].

the second component of the object can be performed similarly. Note that an exception is raised at run-time if the object *anIntPair* cannot interpret a message sent to it.

Obviously, there exist some serious problems with the above approach to implementing objects. Since every object is currently assigned the type *OBJ*, we cannot use types to differentiate objects. For instance, suppose that *MSGfoo* is some declared message constructor of the type $MSG(\mathbf{1})$; then $anIntPair(MSGfoo)$ is well-typed, but its execution leads to an uncaught exception *UnknownMessage* at run-time. This is clearly undesirable: $anIntPair(MSGfoo)$ should have been rejected at compile-time as an ill-typed expression. We address this problem by providing the type constructor *MSG* with additional parameter. Given a type τ and a class C, $MSG(C, \tau)$ is now a type; the intuition is that a message of the type $MSG(C, \tau)$ should only be sent to objects in the class C, to which we assign the type $OBJ(C)$ defined as follows:

$$OBJ(C) = \forall\alpha.MSG(C, \alpha) \to \alpha$$

First and foremost, we emphasize that a class is *not* a type; it is really a tag used to differentiate messages and objects. For instance, we may declare a class *ip* and associate it with the following message constructors of the given types:

$MSGgetfst$	:	$MSG(ip, int)$	$MSGgetsnd$	:	$MSG(ip, int)$
$MSGsetfst$	:	$int \to MSG(ip, \mathbf{1})$	$MSGsetsnd$	:	$int \to MSG(ip, \mathbf{1})$

The type $int \to int \to OBJ(ip)$ can now be assigned to the function *newIntPair*; then *anIntPair* has the type $OBJ(ip)$, and therefore $anIntPair(MSGfoo)$ becomes ill-typed if *MSGfoo* has a type $MSG(C, \mathbf{1})$ for some class C that is different from *ip*.

We refer the reader to [XCC03,Xi02] for further details on this typeful approach to OOP (with single inheritance). The treatment of multiple inheritance in this paper bears a great deal of similarity to the treatment of single inheritance in [XCC03,Xi02], though there are also some substantial differences involved.

In order to handle multiple inheritance, we are to treat a path from a (super)class to a (sub)class as a first-class value, and we use $PTH(C_1, C_2)$ as the type for paths from the (super)class C_1 to the (sub)class C_2. The message type constructor *MSG* is to take three parameters C_0, C and τ to form a type $MSG(C_0, C, \tau)$; an object tagged by C is expected to return a value of type τ after receiving a message of type $MSG(C_0, C, \tau)$, where C_0 indicates the original class in which the message constructor that constructs this message is declared. With this view, the type $OBJ(C)$ for objects tagged by C can now be defined as follows:

$$OBJ(C) = \forall c_0 : cls.\forall\alpha : type.PTH(c_0, C) \to MSG(c_0, C, \alpha) \to \alpha$$

where *cls* is the sort for class tags.

In the rest of the paper, we are to describe an implementation of programming objects based on the above idea that supports (a form of) multiple inheritance. The primary contribution of the paper lies in a simple and general account for multiple inheritance in a typeful setting, which we claim to be entirely novel.

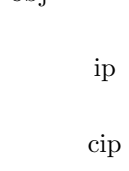
We emphasize that the approach to implementing multiple inheritance as is presented in this paper is intended to serve as a reference for more realistic implementations in future and thus should not be judged in terms of efficiency.

2 Implementing Multiple Inheritance

In this section, we describe a typeful approach to implementing objects that supports multiple inheritance. The presented code is written in the concrete syntax of ATS, which is rather close to the syntax of SML [MTHM97]. We think that the most effective strategy to understand this implementation is to first understand the various types involved in it. Therefore, our description of this implementation is largely guided by these types.

2.1 Run-Time Class Tags

As mentioned previously, there is a sort *cls* for class tags. Let us assume that we have class tags *obj* (for object class), *eq* (for equality class), *ip* (for integer pair class), *cip* (for colored integer pair class), etc. Let us further assume that *cip* is an immediate subclass of *ip*, which is an immediate subclass of both *obj* and *eq*. The following diagram illustrates this class structure:



The class tags here should really be called compile-time or static class tags as they only exist in the type system (of ATS) and are not available at run-time. To address the need for accessing class tags at run-time, we declare a (guarded) datatype *CLS* as follows:

```
datatype ClS (cls) =
  | CLSobj (obj) | CLSeq (eq) | CLSip (ip) | CLScip (cip) | ...
```

Intuitively, for each static class tag C, $CLS(C)$ is a singleton type that contains a value $\underline{C}$ corresponding to C. The above declaration simple means that for C to be *obj*, *eq*, *ip*, *cip*, $\underline{C}$ are *CLSobj*, *CLSeq*, *CLSip*, *CLScip*, respectively. We use ... in the declaration of *CLS* to indicate that *CLS* is extensible (like the exception type *exn* in SML).

2.2 Paths

We need to treat paths from (super)classes to (sub)classes as first-class values. For this need, we declare a (guarded) datatype *PTH* as follows, which takes two (static) class tags C_1 and C_2 to form a type $PTH(C_1, C_2)$.

```
datatype PTH (cls, cls) =
  | {c:cls} PTHend (c, c) of CLS (c)
  | {c1:cls,c2:cls,c3:cls}
      PTHcons (c1, c3) of (ClS (c1), PTH (c2, c3))
```

The syntax indicates that there are two value constructors *PTHend* and *PTHcons* associated with *PTH*, which are given the following types:

$$\begin{array}{rcl} \mathit{PTHend} & : & \forall c : \mathit{cls}.\mathit{CLS}(c) \rightarrow \mathit{PTH}(c, c) \\ \mathit{PTHcons} & : & \forall c_1 : \mathit{cls}.\forall c_2 : \mathit{cls}.\forall c_3 : \mathit{cls}.(\mathit{CLS}(c_1), \mathit{PTH}(c_2, c_3)) \rightarrow \mathit{PTH}(c_1, c_3) \end{array}$$

Given $\underline{C}_1, \ldots, \underline{C}_n$ for $n \geq 1$, we write $[\underline{C}_1, \ldots, \underline{C}_n]$ for $\mathit{PTHcons}(\underline{C}_1, [\underline{C}_2, \ldots, \underline{C}_n])$ if $n \geq 2$, or for $\mathit{PTHend}[\underline{C}_n]$ if $n = 1$. As an example, $[\mathit{CLSobj}, \mathit{CLSip}, \mathit{CLScip}]$, which stands for $\mathit{PTHcons}(\mathit{CLSobj}, \mathit{PTHcons}(\mathit{CLSip}, \mathit{PTHend}(\mathit{CLScip})))$, is a path from class *obj* to *cip*.

Clearly, one may also form a value like $[\mathit{CLScip}, \mathit{CLSip}, \mathit{CLSobj}]$, which does not correspond to any legal path. It is possible to declare the datatype constructor *PTH* in a more involved manner so that only values representing legal paths can be formed. However, such a declaration would significantly complicate the presentation of the paper and is thus not pursued here. In the following presentation, we simply assume that only values representing legal paths are ever to be formed. It will soon be clear that the main use of a path is to direct method lookup when an object tries to interpret a received message. In practice, we anticipate that a overwhelming majority of paths in a program can be automatically constructed by a compiler. However, the construction of a path may need certain interaction from a programmer when there is some ambiguity involved, i.e., when there are more than one paths from a given (super)class to a (sub)class.

2.3 Regular Objects and Temporary Objects

We are to encounter two forms of objects: regular objects (or just objects) and temporary objects. Given a static class tag C, $\mathit{OBJ}(C)$ is the type for regular objects in class C. In the concrete syntax of ATS, *OBJ* is defined as follows,

```
typedef OBJ (c:cls) =
  {c0:cls,a:type} PTH (c0,c) -> MSG(c0,c,a) -> a
```

which means that $\mathit{OBJ}(C)$ is just a shorthand for the following type:

$$\forall c_0 : \mathit{cls}.\forall \alpha : \mathit{type}.\mathit{PTH}(c_0, C) \rightarrow \mathit{MSG}(c_0, C, \alpha) \rightarrow \alpha$$

Therefore, a regular object o in class C takes a path from C_0 to C for some class tag C_0 and a message of type $\mathit{MSG}(C_0, C, \tau)$ for some type τ, and is then expected to return a value of type τ.

There is another form of objects that are only constructed during run-time, and we use the name *temporary objects* for them. Given a static class tag C, we use $\mathit{OBJ}_0(C)$ as the type for temporary objects in class C, where OBJ_0 is defined as follows,

```
typedef OBJ0 (c:cls) = {c0:cls,a:type} MSG(c0,c,a) -> a
```

i.e., $OBJ_0(C)$ stands for the type $\forall c_0 : cls.\forall \alpha : type.MSG(c_0, C, \alpha) \rightarrow \alpha$. Given a temporary object o in class C, it takes a message of type $MSG(C_0, C, \tau)$ for some static class tag C_0 and type τ, and then is expected to return a value of type τ. A temporary object always does method lookup in a fixed manner, and one may think that a path is already built into a temporary object in some special manner. However, we emphasize that a temporary object is in general *not* constructed by applying a regular object to a given path.

2.4 Wrapper Functions

The notion of wrapper functions naturally occurs in the process of implementing a mechanism to support inheritance. A wrapper function (or just a wrapper, for short) for a class C is assigned the type $WRP(C)$, where WRP is defined as follows,

```
typedef WRP(c:cls) = OBJ(c) -> OBJ0(c)
```

i.e., $WRP(C)$ stands for the type $OBJ(C) \rightarrow OBJ_0(C)$ for each static class tag C. Therefore, a wrapper is a function that turns a regular object in class C into a temporary object in class C. The typical scenario in which a wrapper function is called can be described as follows: Given a regular object o, a path pth and a message msg, let us apply o to pth and msg; if the object o could not interpret the message msg directly, a wrapper function wrp is to be constructed according to the path pth and then be applied to the object o to form a temporary object o', to which the message msg is then subsequently passed.

2.5 Super Functions

As in the case of single inheritance [XCC03], the notion of super functions also plays a key role in implementing multiple inheritance. For each class C, there is a super function associated with C. In the following presentation, we use $SUPERobj$, $SUPEReq$, $SUPERip$ and $SUPERcip$ to name the super functions associated with the classes obj, eq, ip and cip, respectively. Given a static class tag C_0, the super function associated with C_0 is assigned the type $SUPER(C_0)$, which is a shorthand for the following type:

$$\forall c : cls.PTH(C_0, c) \rightarrow WRP(c) \rightarrow WRP(c)$$

In Figure 1, the super functions $SUPERobj$, $SUPEReq$, $SUPERip$ and $SUPERcip$ are implemented, where the involved message constructors are of the following types:

$$\begin{array}{rcl}
MSGcopy & : & \forall c : cls.MSG(obj, c, OBJ(c)) \\
MSGeq & : & \forall c : cls.OBJ(c) \rightarrow MSG(eq, c, bool) \\
MSGneq & : & \forall c : cls.OBJ(c) \rightarrow MSG(neq, c, bool) \\
MSGgetfst & : & \forall c : cls.MSG(ip, c, int) \\
MSGgetsnd & : & \forall c : cls.MSG(ip, c, int) \\
MSGsetfst & : & \forall c : cls.int \rightarrow MSG(ip, c, \mathbf{1}) \\
MSGsetsnd & : & \forall c : cls.int \rightarrow MSG(ip, c, \mathbf{1}) \\
MSGswap & : & \forall c : cls.MSG(ip, c, \mathbf{1})
\end{array}$$

```
fun SUPERobj pth wrp obj =
  let
     fun dispatch msg =
       case msg of
         | MSGcopy => obj
         | _ => wrp obj msg
     withtype {c0:cls,c:cls,a:type} MSG (c0,c,a) -> a
  in
     dispatch (* a temporary object *)
  end
withtype {c:cls} PTH (obj,c) -> WRP(c) -> WRP (c)

fun SUPEReq pth wrp obj =
  let
     fun dispatch msg =
       case msg of
         | MSGeq (obj') => not (obj pth (MSGneq (obj')))
         | MSGneq (obj') => not (obj pth (MSGeq (obj')))
         | _ => wrp obj msg
   in
     dispatch (* a temporary object *)
   end
withtype {c:cls} PTH (eq,c) -> WRP(c) -> WRP (c)

fun SUPERip pth wrp obj =
  let
     fun dispatch msg =
       case msg of
         | MSGswap =>
             let
                val fst = obj pth MSGgetfst
                and snd = obj pth MSGgetsnd
             in
                (obj pth (MSGsetfst snd); obj pth (MSGsetsnd fst))
             end

         | MSGeq (other) =>
           if !xref = other [CLSip] (MSGgetfst) then
              if !yref = other [CLScip] (MSGgetsnd) then true
              else false
           else false

         | _ => wrp obj msg
  in
     dispatch (* a temporary object *)
  end
withtype {c:cls} PTH (ip,c) -> WRP(c) -> WRP (c)

fun SUPERcip pth wrp obj = wrp obj
withtype {c:cls} PTH (cip,c) -> WRP(c) -> WRP (c)
```

Fig. 1. The definition of some super functions

We have previously already explained the meaning of these message constructors except for *MSGneq* and *MSGswap*; sending $MSGneq(o')$ to an object o means to compare whether o and o' are not equal (according to some specific interpretation of the message $MSGneq(o)$ by o'); sending *MSGswap* to an (integer pair) object o is to swap the first and the second components in o.

The use of super functions in implementing inheritance is somewhat subtle, and we present below a rather informal explanation on this point. Let $super_C$ be the super function associated with some class C. Suppose o is an object of type $OBJ(C_1)$ for some class tag C_1, m a message of type $MSG(C_0, C_1, \tau)$ for some class tag C_0 and type τ, and p a path from C_0 to C_1 of the form $p_a{+\!\!+}[C]{+\!\!+}p_b$, i.e., p_a is a prefix of p, p_b is a suffix of p and C is on the path p. Then the call $o(p)(m)$ is essentially evaluated as follows: A method in o for interpreting m is invoked if it is implemented; Otherwise, the process to look for a method to interpret m is first done along the path p_b; now suppose this process of method lookup fails to find a proper method to interpret m; at this point, the super function $super_C$ associated with the class C is called on the path p and some wrapper function w (determined by the path p_a) to return another wrapper function, which is then applied to o to form a temporary object to interpret the message m; if the temporary object cannot interpret m, then w is applied to o to form yet another temporary object to interpret m. The picture is to become more clear later once we introduce an example.

2.6 Chaining Super Functions Together

Let *super* be the function that takes a run-time class tag $\underline{C}$ to return the super function associated with C. Therefore, *super* can be assigned the following type:

$$\forall c : cls.CLS(c) \rightarrow SUPER(C)$$

The function *path2wrapper* is implemented in Figure 2, which turns a path into a wrapper.

```
fun nullWrapper (obj) = lam msg => raise UnknownMessage
withtype {c:cls} OBJ (c) -> OBJ0 (c)

fun path2wrapper pth = let
  fun aux pth wrp =
    case pth of
      | PTHend (c) => super c pth wrp
      | PTHcons (c, pth') => aux pth' (super c pth wrp)
  withtype {c0:cls,c:cls} PTH (c0, c) -> WRP (c) -> WRP (c)
in aux pth nullWrapper end
withtype {c0:cls,c:cls} PTH (c0, c) -> WRP (c)
```

Fig. 2. A function for chaining super functions together

Let $pth = [\underline{C}_1, \ldots, \underline{C}_n]$ be a path from (super)class C_1 to (sub)class C_n, and $pth_i = [\underline{C}_i, \ldots, \underline{C}_n]$ for $i = 1, \ldots, n$, and $wrp_1 = super(\underline{C}_1)(pth_1)(nullWrapper)$

and $wrp_{i+1} = super(\underline{C}_{i+1})(pth_{i+1})(wrp_i)$ for $1 \leq i < n$. Then $path2wrapper(pth)$ returns the wrapper function wrp_n. For instance, we have

$$
\begin{array}{l}
path2wrapper\ [CLSobj, CLSip, CLScip] = \\
\quad SUPERcip\ [CLScip] \\
\quad\quad (SUPERip\ [CLSip, CLScip] \\
\quad\quad\quad (SUPERobj\ [CLSobj, CLSip, CLScip]\ nullWrapper))
\end{array}
$$

2.7 Constructing Objects

We now present a function *newIntPair* in Figure 3, which takes two integers to create an integer pair object.

```
fun newIntPair x y = let
  val xref = ref x and yref = ref y
  fun dispatch pth msg =
    case msg of
      | MSGgetfst => !xref
      | MSGgetsnd => !yref
      | MSGsetfst x' => (xref := x')
      | MSGsetsnd y' => (yref := y')
      | MSGcopy => newIntPair (!xref) (!yref)
      | _ => path2wrapper pth dispatch msg
in dispatch end
withtype int -> int -> OBJ (ip)
```

Fig. 3. A function for constructing integer pair objects

Given two integer pair objects o_1 and o_2, we now explain how method lookup is handled after o_1 receives the message $MSGneq(o_2)$. Formally, we need to evaluate $o_1\ [CLSeq, CLSip]\ (MSGneq(o_2))$ as the message constructor originates with the class *eq*. By inspecting the body of the function *newIntPair*, we see the need for evaluating the following expression

$$path2wrapper\ [CLSeq, CLSip]\ (o_1)\ (MSGneq(o_2))$$

as there is no code directly implemented for interpreting the message $MSGneq(o_2)$. Let us assume:

$$
\begin{array}{l}
wrp_1 = SUPEReq\ [CLSeq, CLSip]\ nullWrapper \\
wrp_2 = SUPERip\ [CLSip]\ wrp_1
\end{array}
$$

and we have $path2wrapper\ [CLSeq, CLSip] = wrp_2$. So we are to evaluate the expression $wrp_2\ o_1\ (MSGneq(o_2))$. By inspecting the body of the function *SUPERip*, we need to evaluate the following expression,

$$wrp_1\ o_1\ (MSGneq(o_2))$$

and by inspecting the body of the function *SUPEReq*, we then need to evaluate the following expression:

$$not(o_1\ [CLSeq, CLSip]\ (MSGeq(o_2)))$$

There is no code in the body of *newIntPair* for handling the $MSGeq(o_2)$ directly; instead, the message is finally to be handled by some relevant code in the body of *SUPERip*.

2.8 Syntactic Support for OOP

We now outline some syntax specially designed to facilitate object-oriented programming in ATS. We use the following syntax:

```
class obj {
  superclass: /* none */
  message MSGcopy (OBJ (myclass))
  method MSGcopy = myself
} // end of class obj
```

to introduce a class tag *obj* and a message constructor *MSGcopy* of the type $\forall c : cls.MSG(obj, c, OBJ(c))$. Please note the special use of *myclass* and *myself*: the former is a class tag and the latter is an object of the type $OBJ(myclass)$ that is supposed to receive the message. In general, a line as follows:

$$message\ MSGfoo\ (\tau)\ of\ (\tau_1, \ldots, \tau_n)$$

in the declaration of some class C introduces a message constructor *MSGfoo* of the type $\forall myclass : cls.(\tau_1, \ldots, \tau_n) \rightarrow MSG(C, myclass, \tau)$.

The super function associated with the class tag *obj*, which we refer to as *SUPERobj*, is also introduced automatically through the above syntax: the line *method MSGcopy = myself* translates into the clause $MSGcopy \Rightarrow obj$ in the definition of *SUPERobj* in Figure 1

The code in Figure 4 declares a class *ip* and some message constructors associated with the class *ip*, and then implements a function *newIntPair* for creating objects in the class *ip*. We write *obj* @ *msg* to mean sending the message *msg* to the object *obj*, which translates into *obj* (*pth*) (*msg*) for some path *pth* to be constructed by the compiler.[3] It should be straightforward to relate the code in Figure 4 to the code for the super function *SUPERip* in Figure 1 and the code for the function *newIntPair* in Figure 3.

2.9 Parametric Polymorphism

There is an immediate need for classes that parametrize over types. For instance, we may want to generalize the monomorphic function *newIntPair* to a polymorphic function *newPair* that can take values x and y of any types to create an

[3] We plan to require the programmer to provide adequate information if there is ambiguity in constructing such a path.

```
class ip {
  superclass: obj, eq // ip is a subclass of both obj and eq

  message MSGgetfst (int)
  message MSGsetfst (unit) of int
  message MSGgetsnd (int)
  message MSGsetsnd (unit) of int
  message MSGswap (unit)

  method MSGswap: unit =
    let
        val x = myself @ MSGgetfst and y = myself @ MSGgetsnd
    in
        myself @ (MSGsetfst y); myself @ (MSGsetsnd x)
    end

  method MSGeq (other): bool =
    if myself @ MSGgetfst = other @ MSGgetfst then
      if myself @ MSGgetsnd = other @ MSGgetsnd then else false
    else false

} // end: class ip

// newIntPair: int -> int -> OBJ (ip)

object newIntPair (x: int) (y: int): ip = {

  val xref = ref x and yref = ref y

  method MSGgetfst = !xref
  method MSGsetfst (x') = (xref := x)
  method MSGgetsnd = !yref
  method MSGsetfst (y') = (yref := y)
  method MSGcopy = newIntPair (!xref) (!yref)

} // end: object newIntPair
```

Fig. 4. Some code written in the special syntax for OOP

object representing the pair whose first and second components are x and y, respectively. To do this, we first introduce a constant *pair* that takes two types τ_1 and τ_2 to form a class tag $pair(\tau_1, \tau_2)$, and then introduce a constructor *CLSpair* assigned the given type:

$$CLSpair \quad : \quad \forall\alpha : type.\forall\beta : type.CLS(pair(\alpha, \beta))$$

and then assume the message constructors *MSGgetfst*, *MSGsetfst*, *MSGgetsnd*, *MSGsetsnd* are given the following types:

$$\begin{array}{rcl} MSGgetfst & : & \forall\alpha : type.\forall\beta : type.\forall c : cls.MSG(pair(\alpha, \beta), c, \alpha) \\ MSGsetfst & : & \forall\alpha : type.\forall\beta : type.\forall c : cls.\alpha \to MSG(pair(\alpha, \beta), c, \mathbf{1}) \\ MSGgetsnd & : & \forall\alpha : type.\forall\beta : type.\forall c : cls.MSG(pair(\alpha, \beta), c, \beta) \\ MSGsetsnd & : & \forall\alpha : type.\forall\beta : type.\forall c : cls.\beta \to MSG(pair(\alpha, \beta), c, \mathbf{1}) \end{array}$$

All of this is handled by the following syntax:

```
class pair (a:type, b:type) = {
  superclass: obj

  message MSGgetfst (a)
  message MSGsetfst (unit) of a
  message MSGgetfst (b)
  message MSGsetfst (unit) of b
}
```

A function *newPair* for creating pair objects can then be properly implemented, which is assigned the type $\forall\alpha : type.\forall\beta : type.\alpha \to \beta \to OBJ(pair(\alpha, \beta))$:

```
object newPair{a:type, b:type} (x: a) (y: b): pair (a, b) = {
  ...
}
```

3 Facilitating Code Sharing

There is a great deal of code redundancy in the libraries of functional languages such as SML and Objective Caml as there is little code sharing across difference data structures. For instance, functions such as *map*, *foldLeft*, *foldRight* are defined repeatedly for lists and arrays. This issue is already studied in the context of generic programming [Hin00] and polytypic programming [JJ97], but the proposed solutions are not applicable to data structures that are given abstract types.

We now use a (contrived) example to outline an approach that can effectively address the issue of code sharing across difference data structures even when the data structures are given abstract types. Suppose that we declare a parameterized class *IsList* as follows:

```
class IsList (elt:type, lst:type) {
  superclass: ...
  message nil (lst)
  message cons ((elt, lst) -> lst)
  message uncons (lst -> (elt, lst))
  message isEmpty (lst -> bool)

  message foreach ((elt -> unit, lst) -> unit)
  method foreach = ...
    /* can be defined in terms of isEmpty and uncons */
  ...
}
```

Intuitively, an object of type $OBJ(IsList(\tau_1, \tau_2))$ can be thought of as a term that proves a value of type τ_2 can be treated as a list in which each element is of type τ_1. Now let us construct an object *intIsList1* of type $OBJ(IsList(unit, int))$ as follows:

```
object intIsList1: IsList (unit, int) = {
  method nil = 0
  method isEmpty (n) = (n == 0)
  method cons (_, n) = n + 1
  method uncons (n) =
    if n > 0 then ((), n - 1) else raise EmptyList
}
```

Then *intIsList1* @ *foreach* returns a function of type $(unit \to unit, int)$; applying this function to f and n means executing $f()$ for n times, where f is assumed to be a function of type $unit \to unit$ and n a natural number. Now let us construct another object *intIsList2* of type $OBJ(IsList(int, int))$ as follows:

```
object intIsList2: IsList (int, int) = {
  method nil = 0
  method isEmpty (n) = (n == 0)
  method cons (_, n) = n + 1
  method uncons (n) =
    if n > 0 then (n, n - 1) else raise EmptyList
}
```

Then *intIsList2* @ *foreach* returns a function of type $(int \to unit, int)$; applying this function to f and n means executing $f(n), f(n-1), \ldots, f(1)$, where f is assumed to be a function of type $int \to unit$ and n a natural number. Now let us construct another object *arrayIsList* as follows,

```
object arrayIsList{elt:type} (A: array(elt)): IsList (elt,int) = {
  method nil = 0
  method isEmpty (n) = (n == 0)
  method cons (x, n) = (update (A, n, x); n + 1)
  method uncons (n) = (sub (A, n - 1), n - 1)
}
```

where *sub* and *update* are the usual subscripting and updating functions on arrays. Let A be an array of type $array(\tau)$. Then $arrayIsList(A)$ @ *foreach* returns a function of type $(\tau \to unit, int)$; applying this function to f and n means executing $f(v_{n-1}), f(v_{n-2}), \ldots, f(v_0)$, where we assume that f is a function of type $\tau \to unit$, n is a natural number less than or equal to the size of A, and $v_0, \ldots, v_{n-1}$ are the values stored in A, from cell 0 to cell $n-1$.

Though this is an oversimplified example, the point made is clear: The code for *foreach* in the class *IsList* is reused repeatedly. Actually, the code for all the functions implemented in the class *IsList* in terms of *nil*, *isEmpty*, *cons*, and *uncons* can be reused repeatedly. Note that it is difficult to make this approach to code sharing available in OOP languages such as Java as it requires some essential use of parametric polymorphism. On the other hand, the approach bears some resemblance to the notion of type classes in Haskell [HHJW96,P+99]. One may argue that what we have achieved here can also be achieved by using functors in SML. This, however, is not the case. First, functors are not first-class values and thus are not available at run-time. But more importantly, functors simply do

not support code inheritance, a vital component in OOP. For the sake of space limitation, we could not show the use of inheritance in the above example, but the need for inheritance in practice is ubiquitous in practice. Please see some on-line examples [SX04].

4 Related Work and Conclusion

Multiple inheritance is supported in many object-oriented programming languages such as Eiffel and C++. In Eiffel, a straightforward approach is taken to resolve method dispatching conflicts that may occur due to multiple inheritance: If a class has multiple superclasses, each method in the class must determine statically at compile-time from which superclass it should inherit code. This approach, though simple, makes multiple inheritance in Eiffel rather limited. For instance, it cannot accommodate a scenario in which a method needs to be inherited from different superclasses according to where the method is actually called. When compared to Eiffel, C++ offers a more flexible approach to resolving method dispatching conflicts as the programmer can supply explicit annotation at method invocation sites to indicate how such conflicts should be resolved. However, it is still required in C++ that method dispatching conflicts be resolve statically at compile-time. With paths being first-class values, our approach to multiple inheritance can actually address the need for resolving method dispatching conflicts at run-time.

In some early studies on multiple inheritance in a typed setting [Wan89,Car88], the essential idea is to model inheritance relation by a subtyping relation on record types and multiple inheritance then corresponds to the situation where one record extends multiple records. However, this idea is unable to address the crucial issue of dynamic method dispatching, which is indispensable if abstract methods are to be supported. In [CP96], an approach to encoding objects is presented that supports both dynamic method dispatching and a restricted form of multiple inheritance (like that in Eiffel). This rather involved approach is based on higher-order intersection types and its interaction with other type features such as recursive types and parametric polymorphism remains unclear.

In the literature, most of existing approaches to typed OOP take the view of *objects as records*. They are often centered around the type system $F^{\omega}_{\leq}$ or its variants and use structural subtyping to support inheritance [BCP99,Bru02]. On the other hand, the realistic object-oriented programming languages that we know all rely on nominal subtyping. In this paper, we have developed a typeful approach to OOP that supports a form of multiple inheritance. This approach, which is based on the notion of guarded recursive datatypes [XCC03], does not use structural subtyping to model inheritance and is largely in line with the current practice of OOP.

References

[BCP99] Kim B. Bruce, Luca Cardelli, and Benjamin Pierce. Comparing Object Encodings. *Information and Computation*, 155:108–133, 1999.

[Bru02] Kim B. Bruce. *Foundations of Object-Oriented Languages*. The MIT Press, Cambridge, MA, 2002. xx+384 pp.

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

[CP96] Adriana B. Compagnoni and Benjamin C. Pierce. Higher-Order Intersection Types and Multiple Inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[GR83] A. Goldenberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison Wesley, 1983.

[HHJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[Hin00] Ralf Hinze. A New Approach to Generic Functional Programming. In *Proceedings of 27th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '00)*, pages 119–132. Boston, 2000.

[JJ97] P Jansson and J. Jeuring. PolyP - Polytypic programming language extension. In *Proceedings of 24th ACM Symposium on Principles of Programming Languages (POPL '97)*, pages 470–482. Paris, France, 1997.

[Liu96] Chamond Liu. *Smalltalk, Objects, and Design*. Manning Publications Co., Greenwich, CT 06830, 1996. ISBN 1-884777-27-9 (hc). x+289 pp.

[MTHM97] Robin Milner, Mads Tofte, Robert W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997. ISBN 0-262-63181-4.

[P+99] Simon Peyton Jones et al. Haskell 98 – A non-strict, purely functional language. Available at `http://www.haskell.org/onlinereport/`, February 1999.

[SX04] Rui Shi and Hongwei Xi. Some Examples of Structuring Libraries with Parametrized Classes, February 2004. Available at: `http://www.cs.bu.edu/~hwxi/ATS/lib`.

[Wan89] Mitchell Wand. Type Inference for Record Concatenation and Multiple Inheritance. In *Proceedings of Fourth IEEE Symposium on Logic in Computer Science*, pages 92–97. Pacific Grove, California, 1989.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235. New Orleans, January 2003.

[Xi02] Hongwei Xi. Unifying Object-Oriented Programming with Typed Functional Programming. In *Proceedings of ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (ASIA-PEPM '02)*, pages 117–125. Aizu-Wakamatsu, Japan, September 2002.

[Xi03] Hongwei Xi. Applied Type System, July 2003. Available at: `http://www.cs.bu.edu/~hwxi/ATS`.

[Xi04] Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*. Springer-Verlag LNCS, February 2004. (to appear).