TYPE SYSTEMS OF OBJECT-ORIENTED PROGRAMMING LANGUAGES

David W. Sandberg
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

87-60-2

Type Systems of Object-Oriented Programming Languages

David W. Sandberg
Oregon State University

Abstract

Three different approaches to type checking have been taken in object-oriented programming languages. Smalltalk-80 uses run-time type checking. C++ uses subtypes. A third alternative is to use parameterized types. We examine the difficulties of programming in an object-oriented fashion with compile-time type checking and argue that parameterized types are better than subtypes in a language with compile-time type checking.

1 Introduction

Smalltalk-80[1] has become the prototypical object-oriented programming language. Smalltalk uses run-time type checking and run-time binding. Object-oriented techniques are hard to incorporate into conventional languages such as Ada and Modula-2[13], mainly because of the limitations of the compile-time type system used. Some recent languages like C++[11], and Trellis/Owl[9] have extended the type system to include subtypes to overcome these limitations. In this paper, we show why object-oriented programming is difficult in conventional languages and how subtyping solves some of these problems. We then develop a parameterized type system that provides an alternative solution to this problem.

Compile-time typing has several advantages. Many errors can be caught at compile time, and this produces more reliable code, and reduces development time. The code is easier to read because the types of variables are explicitly mentioned, which gives more information to the reader. More optimizations can be performed because more information is available at compile time. The language can be compiled into more efficient code because the type checking is done at compile-time instead delaying it to run-time. Programmers give up these advantages in a language typed at run-time in favor of added flexibility.

2 A Programming Problem

A Smalltalk-80 style windowing interface will be used to illustrate object-oriented programming techniques. In such a user interface, the screen is divided into regions called windows. This arrangement can be viewed as modeling papers on a desk top, where the screen is the desk and the windows are the papers. The details of this basic model vary from system to system. Some implementations allow windows to overlap and some do not. Some direct input to one designated window while others direct the input to whichever window contains the cursor. Some implementations have the display change as the cursor is moved from pane to pane whereas in other implementations the display remains relatively unchanged. In this paper we are not concerned with these design decisions but with the impact of the programming language on the implementation of a windowing scheme.

An important design criterion of a windowing system is that an application programmer be able to easily add his own types of windows to the existing window scheme without modifying the system code. There is usually one segment of code that is responsible for implementing such operations as creating new windows, destroying windows, and relocating windows. We will call this segment of the code the window manager. The window manager is usually buried deep in the system code and is often difficult or impossible for the user to modify. On the other hand, this code must know some information about user-defined windows such as how to display them and how to deal with the keyboard input sent to them. This information varies from one type of window to another. Hence, the window manger must refer to code that is not yet written.

Next we examine how this problem is solved in several languages. In Smalltalk-80, where methods are bound to messages at run-time depending on the data type, the code for the window manager can refer to the message names and defer the binding of the messages to user-defined methods until run-time. For a robust system, care must be taken so that bugs in the user code do not cause the whole system to crash.

In LISP, there is run-time binding of procedure names to procedures, but the binding does not depend upon the data type except for some primitive types. Some technique must be used to include the data dependences in the look-up. Three possible solutions are:

- If an object-oriented extension to Lisp is available such as the Flavors package[6], then a solution similar to the Smalltalk-80 technique can be used.
- An atom can be associated with each type of window. The user-defined procedures are put on the property list of this atom. The atom that represents the type of the window is included in the data structure for each window. The window manager can find the procedure to use by looking on the property list of this atom.
- A structure can be defined that not only includes the data structure for the window but the functions that implement the operations on the window. The window manager can then call the functions in the structure to perform operations on the user-defined windows. When there are many windows of the same type, this scheme can be optimized to use less space by creating a structure that contains just the functions and share this record among all windows of the same type.

This final solution can be modified to work in languages that are typed at compile time and allow procedures to be stored in structures, but the strong typing must still be avoided. Languages that fall into this category are Cedar Mesa[12], Modula-2, and C. Consider how this would be implemented in Modula-2.

```
window=POINTER TO RECORD END; (* Dummy type *)
window_description=POINTER TO RECORD
                  data_structure:window
                  control: PROCEDURE(window,char);
                  display: PROCEDURE(window,rectangle);
                 END;

PROCEDURE install(mywindow: ADDRESS;place:rectangle)
(* Procedure to tell the window manager about a new window. *)
VAR p : window_description;
BEGIN
 p:=window;
 ... (* code to install window *)
 p.display(p.data_structure,place);  (* display the window *)
END install;
```

The difficulty here is that a specific type for the window must be included, but the code should allow any type to be used in place of window. To allow this, the strong typing is broken by using the pre-defined type ADDRESS which matches any pointer type. The programmer is then responsible to declare a window description like the one above with the type window replaced by his own window type and pass it to the procedure install. The programmer would write code like:

```
mywindow=POINTER TO RECORD ... END;

my_description=POINTER TO RECORD
          data_structure: mywindow;
          control: PROCEDURE(mywindow,char);
          display: PROCEDURE(mywindow,rectangle);
         END;

PROCEDURE mycontrol(w:mywindow; ch: char);
BEGIN ... END mycontrol;

PROCEDURE mydisplay(w:mywindow; r: rectangle);
BEGIN ... END mydisplay;

PROCEDURE test;
VAR w : my_description;
    t : my_window;
    r : rectangle;
BEGIN
  new(w); new(t);
  w.data_structure=t;
  w.control=mycontrol;
  w.display=mydislay
  ... -- initialize t and r
  install(w,r);
END test;
```

C++, Eiffel[3,4], and Trellis/Owl solve the typing mismatch in the above Modula-2 example by introducing subtyping. All user-defined windows are made subtypes of window. A subtype of window can be assigned to the type window but a window can not be assigned to a subtype of window. This allows a user-defined window to be passed as a parameter to install without breaking the type system.

The above techniques will not work for either Ada or Pascal because they do not allow procedures to be stored in records. In such a language the window manager needs to know every type of window it will ever deal with.

3 A Parameterized Type System

Instead of subtyping, parameterization of types can be used to design an object-oriented programming language. We will first describe a parameterized type system and then give several examples before we return to window managers. The term *type* will be used when the type system is being emphasized. The term *class* will be used in a broader sense to describe the type, the operations that are defined on the type, and the implementation.

Types will be represented by strings. Types without parameters are represent by the name, for example, *real*, *int*, and *char*. Types that have parameters are represented using the same notation as that of a function call; for example, *array(int)*, *pair(array(int),bool)*, and *array(array(int))*. Only types will be allowed as parameters to types. Two types are equal if and only if the strings are the same.

Two mechanisms are used to define sets of types. To represent an infinite set of types a parameter is declared to be free. A free parameter varies over all types. For example, the set of types described by *array(f) where f is free* is the set {array(f) | f is a type}.

Predicates are used to restrict the range of free parameters to types that have certain characteristics. Predicates are boolean-valued functions whose parameters are types. The first letter of the name of a predicate will be capitalized. For example, let Collection be a predicate that is defined by

```
Collection(g,f)= true    when g=list(t) and f=t for some t
               = true    when g=array(int) and f=int
               = false   otherwise
```

The expression *proc(array(t),g) where t is free and Collection(t,g) is required* then represents the union of {*array(list(t),t)* | f is a type} and {*proc(array(array(int)),int)*}.

Our first example of polymorphic code in a strongly typed language declares a data type pair with the types of the elements of the pair left unspecified. These types will be specified sometime before run-time. Each of the procedures below will work on an infinite set of types, yet the programmer declares the procedures only once, and the compiler maps all calls of a procedure to the same code.

```
type
 pair(f,g)=class first:f; second:g; end;
 --In this declaration both f and g are free parameters.
 --A class declaration can be viewed as declaring a pointer to a record.

procedure first(p:pair(f,g)) returns f;
where f and g are free;
begin return (p.first); end first;

procedure second(p:pair(f,g)) returns g;
where f and g are free;
begin return (p.second); end second;

procedure new_pair(a:f; b:g) returns pair(f,g);
where f and g are free;
var  p : pair(f,g);
begin new(p); p.first:=a; p.second:=b; return(p); end new_pair;
```

When a procedure is compiled the free type parameters in the body are replaced by unique types not used elsewhere. This prevents b from being assigned to verb+p.first+ because the types will not be the same.

The only operation possible on the objects that are passed as free parameters is to pass around references to them. Often some other operation is needed. For example, when a pair is printed, operations are needed to print the elements of the pair. The availability of such operations can be specified by the use of predicates. Three steps are needed to use predicates. First, the set of operations available when the predicate is true is defined. Second, the types for which the predicate is true is defined and actual procedures are bound to the operations specified by the predicate. The predicate is assumed to be false everywhere else. Finally, procedures are written that use the predicate. The order in which the second and third steps are done is not important.

Let us define a predicate Printable that represents the set of classes that have an operation to print its instances.

```
Predicate Printable(f) is  print:proc(f); end;

procedure printpair(p:pair(f,g));
where f and g are free;
Printabale(f) and Printable(g) are required;
begin
 printstr("("); print(first(p)); printstr(","); print(second(p)); printstr(")");
end printpair;
```

Integers are declared to be Printable by specifying the binding of procedures on integers to the operations needed for a class to be Printable. The following declaration also declares strings and pairs to be printable:

```
define Printable(int) is print=printint end;

define Printable(string) is print=printstr end;

define Printable(pair(f,g))
where f and g are free;
Printabale(f) and Printable(g) are required;
is print=printpair end;
```

This last declaration defines a relationship between types. It declares that if f and g are Printable then so is pair(f,g). Here is a procedure that uses the above procedures:

```
procedure test
var p:pair(int,pair(int,int));
begin
  p:=new_pair(3,new_pair(4,5));
  printpair(p); --will print: (3,(4,5))
  printpair(new_pair("a","b")); --will print: (a,b)
end test;
```

The second printpair uses the type *pair(string,string)* which has never been explicitly declared. Furthermore there are no remaining free type parameters in this code. In this example, the identifier print could have been overloaded with `printstr`, `printpair` and `printint` but was not in order to make the code easier to understand.

4 A Menu Example

The following example uses procedures as first class objects to define a menu. Each item on the menu consists of a string to be displayed and an action to take when the item is selected. Operations are provided to create a new menu, add selections to the menu and to have the user select an item, and then perform the corresponding action. The type parameter allows the same code to be used in many places. Without the parameter this approach to defining a menu abstract data type would not be very useful.

```
menu(f)=class
      k:array(pair(string,proc(f)))
     end;

procedure new_menu returns menu(f)
free f;
begin new(m); m.k:=new_array; end new_menu;

procedure additemto(m:menu; text:string; action: proc(f))
free f;
begin add(k,new_pair(text,action)); end additemto;

procedure select_item(m:menu(f); data:f)
begin
    ... --display menu and set i to user selected item.
    m.k[i](data)
end select_item;
```

Often it is useful to form a new procedure from an already existing procedure. To illustrate this, three examples are given that transform a procedure so that it can be used as an action in a menu. The first example replaces a parameter by a constant which reduces the number of parameters by one. This changes the type of the procedure from *proc(data,int)* to *proc(data)*. The operation of `fillparameter` is to take the procedure `accept` and to fill in the second parameter with the contents of the variable i.

```
procedure accept(d:data;i:int);
(* procedure that is to be used as an action.*)
begin ...  end accept;

procedure example(i:int);
var menu: menu(data);
      p: proc(data);
begin
 m:=new_menu;
 p:=fillparameter(accept,2,i);
 additemto(m,"accept",p);
 ...
end example;
```

The second example will add a parameter to the procedure `accept`. A helper function is used to do this:

```
procedure accept(); begin ...  end accept;

procedure add(p:proc();t:f)
where f is free;
begin p(); end add;
```

```
procedure example;
var menu: menu(data);
       p: proc(data);
begin
 m:=new_menu;
 p:=fillparameter(add,1,accept);
 additemto(m,"accept",p);
 ...
end example;
```

The final example changes the type of a parameter of a polymorphic procedure. The type of a procedure is changed from *proc(f)* to type *proc(pair(int,f))*.

```
procedure fix(p:proc(f));a:pair(int,f));
where f is free;
begin p(second(a)); end fix;

procedure additem(m:menu(pair(int,f)); str:string; p:proc(f)));
where f is free;
var p2: proc(pair(int,f))
begin
 p2:=fillparameter(fix,1,p);
 additemto(m,str,p2);
end additem;
```

5 Representing L-Values

Parameterized types also allow l-values to be represented easily. The l-value of a variable can be represented as Name(t) where t is the type of the variable. The r-value of the variable is represented as t. Procedures to perform assignments and to dereference a variable can then be written:

```
procedure assign(left:Name(f),right:f)
where f is free;
begin left:=right; end assign;

procedure dereference(left:Name(f) return f
where f is free;
begin return(left) end dereference;
```

Read-only access or read-write access to a field of a class can be provided by returning the r-value or l-value of the field. This is very useful in defining abstract data types.

```
class example is a:int; b:example end;

procedure first(e:example) return int;
$Only read access is provided.
begin  return(e.a); end first;

procedure second(e:example) return name(example);
$Both read and write access is provided.
```

```
begin return(e.b); end second;

procedure test;
var e: example;
begin
 new(e); second(e):=e; --value is changed
 first(e):=1; -- type mismatch will be detected by the compiler.
 second(e):=second(e); --value is fetched and then stored.
end test;
```

6 The Window Manager Revisited

Now we are ready to show how the window manager can be written. First a Predicate is declared to describe classes that can be used as windows:

```
predicate Pane(f) is  display: proc(f, rectangle);
                      control: proc(f, char);
                  end;

procedure install(window:f; location:rectangle);
where f is free;
Pane(f) is required;
-- Procedure that tells the window manager about a new window.
begin
 ...
 display(window,location); --display the window.
end install;
```

When a user-defined class is to be treated as a window, two procedures are declared by the user to describe how to display the window and how the window should handle input. The class of the window is declared to be a Pane and a procedure written to create a new window:

```
procedure my_display(w:my_window; loc:rectangle);
begin ... end display;

procedure my_control(w:my_window; ch:char);
begin ...  end control;

define Pane(my_window) is
                    display=my_display;
                    control=my_control
                 end;

procedure new_my_window;
--procedure to create a new window.
var
 w:my_window; r: rectangle;
begin
 new(w);
 ...
 install(w,r);
end new_my_window;
```

Another problem arises in writing the install procedure that was not mentioned above. The window manager must keep a data structure that contains all the windows on the screen. This data structure stores instances of any class that is a Pane. Thus it must store instances of different types. One solution is to break the strong typing. This can be isolated in the system code for the window manger so that the application programmmer never needs to know that it was broken. In the Modula-2 window manager example, the application programmer was force to deal with breaking the strong typing. Another solution is to resort to run-time typing and binding.

A better solution is to design the language so that an associated class is formed for each Predicate with a single argument. For example, the declaration Pane would create a class anyPane. The class anyPane itself would be a Pane. In addition, the system would create an operation `convert` that would take an instance of any class that is a Pane and make an instance of the class anyPane. The instances of anyPane would be represented as a vector of operations and the instance passed to `convert`. The window manager would store instances of anyPane in its data structures. Hence, the install procedure would become:

```
var all_windows : list(any_Pane); --This is a global variable in
--which to store a list of all the windows.

procedure install(window:f; location:rectangle);
where f is free;
Pane(f) is required;
-- Procedure that tells the window manager about a new window.
var any : any_Pane;
begin
 ...
 any:=convert(window);    --convert the instance.
 addtolist(all_windows,any); --add window to list.
 display(window,location); --display the window.
end install;
```

7 Forming One Window Out of Two.

A useful operation in building windows is to take two instance of Pane and combine them so that they behave as one window. The follow code describes how to treat a pair of panes as a single pane:

```
procedure merge_display(p:pair(f,g);r:rectangle);
where f and g are free;
pane(f) and pane(g) are required;
var
 r1,r2:rectangle;
begin
 ... --divide r into two rectangles r1 and r2
 display(first(p),r1); display(second(p),r2);
end merge_display

procedure merge_control(p:pair(f,g);ch:char);
where f and g are free;
```

```
Pane(f) and Pane(g) are required;
var
 r1,r2:rectangle;
begin
 ... --divide r into two rectangles r1 and r2
 if cursor is in r1 then
  control(first(p),ch)
 else
  control(second(p),ch);
 end;
end merge_control;

define Pane(pair(f,g))
where f and g are free;
Pane(f) and Pane(g) are required;
is
   display=merge_display;
   control=merge_control;
end;
```

Now if p1 and p2 are instances of Pane, then `install(new_pair(p1,p2),rect)` will install them as a single window.

8 Discussion

Many of the examples given for parameterized types do not have equivalent formulations using only subtypes. Subtyping loses type information about an object in parameter transmission when the type of the actual parameter is a subtype of the formal parameter. Type information is not lost for parameterized classes or if the typing is done at run-time.

Other languages have used parameterized types. CLU[2] is one of the best examples. CLU does not have any notion of a predicate as given here, although CLU does contain a mechanism for specifying that a type parameter requires some operations. The predicate notion fits better with Alphard[10] form and the way it is used to write generic algorithms. The proof techniques developed for Alphard should carry over without much modification. Object-oriented techniques have been used in Mesa and Cedar-Mesa[12], but without much assistance from the language.

The parameterized type system described above has been incorporated into a programming language called X2[7,8]. Predicates are implementated as implicit parameters to procedures that use them. The implicit parameters are vectors of operations that describe the predicate. Our pratical experience with X2 has confirmed that parameterization does allow polymorphic code to be written easily in a language that is typed at compile time.

9 References

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

[2] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

[3] B. Meyer. Genericity Versus Inheritance. In [5].

[4] B. Meyer. Eiffel: Programming for Reusability and Extendibility. *SIGPLAN Notices* 22(2), Feburary 1987, 85-94.

[5] N. Meyrowitz, ed. *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings.* Portland, Oregon, 1986. Published as *SIGPLAN Notices* 21(11), November 1986.

[6] D. Moon. Object-Oriented Programming with Flavors. In [5].

[7] D. Sandberg. An Alternative to Subclassing. In [5].

[8] D. Sandberg. *Preliminary X2 Reference Manual.* Oregon State University. Technical Report 85-1-1, 1985.

[9] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An Introduction To Trellis/Owl. In [5].

[10] M. Shaw, ed. Part VIII – An Alphard Specification of a Correct and Efficient Transformation on Data Structures. *ALPHARD: Form and Content.* Springer-Verlag, 1981.

[11] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[12] D. Sinehart, P. Zellweger, R. Beach, and R. Hagmann. A Structural View of The Cedar Programming Environment. TOPLAS 8(4), October 1986.

[13] N. Wirth. *Programming in MODULA-2.* Springer-Verlag, 1983.