

Simulating Inheritance using Genericity

Craig A. Rich

Computer Science Department
California State Polytechnic University
Pomona, CA 91768, USA

e-mail: `carich@csupomona.edu`

Technical Report 1993-02
January 1993

Abstract

Inheritance and genericity are programming language features which provide a concise and intuitive syntax for defining operations in terms of operations. They are perhaps the most distinguishing features of “object-oriented” programming languages. Several object-oriented languages contain or are being extended to contain both inheritance and genericity, e.g., C++, Ada 9X, and Eiffel. [Meyer 88] simulates genericity using inheritance, and conjectures that “the answer to the question—can inheritance be simulated with genericity?—is no.” In this paper, we apply a fixed-point semantics of inheritance [Cook and Palsberg 89] to obtain a concise and provably correct simulation of inheritance using genericity. To demonstrate the practicality of the simulation, we use Ada generic packages to simulate Smalltalk inheritance.

Inheritance

We introduce inheritance through the following example, which is written in pseudo-Smalltalk:

```
class Point (a,b)
  method x      = a
  method y      = b
  method distance =  $\sqrt{(\text{self.x})^2 + (\text{self.y})^2}$ 
  method closer (p) = (self.distance < p.distance)

class Circle (a,b,r) inherit Point (a,b)
  method radius = r
  method distance = super.distance - self.radius
```

The *class* Point is a collection of operations called *methods* parameterized by *instantiation parameters* a and b. A Point *object* is obtained by instantiating the class Point with actual coordinates a and b. A method m in a Point object p is invoked by the Point *reference* p.m. The Point references p.x and p.y return the coordinates of p; p.distance returns the distance to the origin from p; and p.closer(q) returns true iff p is closer to the origin than the object q (which may or may not be a Point object).

The *subclass* Circle is a collection of methods defined in terms of methods in its *superclass* Point. A Circle object is obtained by instantiating the class Circle with actual center coordinates a, b and radius r. A Circle object implicitly contains *inherited methods* x, y, and closer, and explicitly contains *overridden method* distance and *new method* radius. The Circle references c.x and c.y return the center coordinates of c; c.radius returns the radius of c; c.distance returns the distance to the origin from c; and c.closer(q) returns true iff c is closer to the origin than the object q.

The Circle reference c.distance is defined using the *super reference* super.distance, which returns the value of distance as defined in the superclass Point object, i.e., the distance to the origin from the center. Super references allow invocation of inherited methods which have been overridden. The Circle reference c.closer is defined using the *self reference* self.distance which returns the value of distance as defined in c itself, i.e., c.distance. Therefore, a single definition of closer can be used to correctly compare any pair of Point and/or Circle objects.

The Simulation

[Cook and Palsberg 89] give an elegant formal semantics of inheritance wherein classes are modeled as function generators, inheritance and overriding are modeled as an operation producing generators (subclasses) from generators (superclasses), and objects are obtained as least fixed-points of generators (classes). They prove formally that their model is equivalent to the operational semantics of Smalltalk inheritance. The following example, which is written in pseudo-Ada, shows how we implement their model using genericity:

```
generic
  a,b                : Float;
  with function self_x      : Float  is <>;
  with function self_y      : Float  is <>;
  with function self_distance : Float  is <>;
  with function self_closer (d: Float) : Boolean is <>;
package GeneratePoint is
  function x                : Float  is begin return a;                end;
  function y                : Float  is begin return b;                end;
  function distance         : Float  is begin return  $\sqrt{(\text{self\_x})^2 + (\text{self\_y})^2}$ ; end;
  function closer (d: Float) : Boolean is begin return (self_distance < d); end;
end GeneratePoint;

generic
  a,b,r              : Float;
  with function self_x      : Float  is <>;
  with function self_y      : Float  is <>;
  with function self_radius : Float  is <>;
  with function self_distance : Float  is <>;
  with function self_closer (d: Float) : Boolean is <>;
package GenerateCircle is
  package super                is new GeneratePoint (a,b);
  function x                    : Float  renames super.x;
  function y                    : Float  renames super.y;
  function radius               : Float  is begin return r;                end;
  function distance             : Float  is begin return super.distance - self_radius; end;
  function closer (d: Float)    : Boolean renames super.closer;
end GenerateCircle;
```

The class Point is a generic package of functions defining its methods. The instantiation parameters and self-referential methods appear as generic formal parameters. A Point object is a package obtained by iteratively instantiating the generic package GeneratePoint with actual coordinates a, b and the resulting defined methods (starting with undefined methods) until a fixed-point is reached. A method m in a Point object p is invoked by the qualified function call p.m.

The subclass Circle is a generic package of methods defined in terms of package instantiation super, representing the superclass Point object. Inherited methods rename methods in the package instantiation super; overridden and new methods are explicitly defined. A Circle object is a package obtained by iteratively instantiating the generic package GenerateCircle until a fixed-point is reached.

References

- [Cook and Palsberg 89] Cook, William and Palsberg, Jens. “A Denotational Semantics of Inheritance and its Correctness,” in *Proceedings ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, pp. 433–443, 1989.
- [Meyer 88] Meyer, Bertrand. *Object-Oriented Software Construction*, Prentice-Hall, New York, 1988.