Position paper
1991 ECOOP Workshop on Types, Inheritance, and Assignment

Norman C. Hutchinson
Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada

May 23, 1991

Background

Historically, types have been required to serve two purposes:

- Classification of the entities involved in a computation, and
- Providing "representation independence"; ensuring that the meaning of a program is not dependent on the representations chosen for its values.

If we throw away all of the baggage that the phases "object-oriented" and "object-based" have accumulated over the last decade, we can see that the fundamental advantage that systems that support objects have over systems that do not is encapsulation. That is, a system that supports objects requires the grouping of data and operations and guarantees that only those operations defined with the data will be allowed access to the data. The encapsulation of objects provides exactly the "representation independence" mentioned above; it ensures that only code that understands the representation used for data will be allowed access to that data.

Objects and types

Accepting the object-oriented philosophy allows us to rethink the question of what we want from our type systems. We already have a mechanism for enforcing encapsulation, what we need is a mechanism for the classification of objects. There are two major forms of classification that we might desire:

- Classification based on implementation. The class systems that have evolved since Simula address this need very nicely. One can define a subclass of an existing class as a refinement: either extending or modifying the behaviour of the superclass.

  Such a classification scheme is of interest to the programmer of a collection of classes because it allows her to reuse code, ensure that objects behave in a consistent way, etc. It is also of interest to the compiler writer because the information about how objects are implemented can be exploited to generate smaller objects and faster code.

- Classification based on the abstract invocation protocol implemented by the object. By this I mean that each "client" of an object expects the object to implement a particular collection of operations, and any supplied object that implements all of the required operations meets (at least syntactically) the requirements imposed by that client.[1]

[1]We could strengthen this form of classification by requiring that the object's *semantics* appropriately satisfy the demands of the client. While this is obviously desireable, I believe this to be outside of the scope of type systems.

Example of such requirements abound. A window manager expects a particular protocol from each window under its control (move, resize, refresh, terminate). A file system expects its directories to implement add, lookup, delete, and list.

One can simulate this in a traditional object-oriented system by creating abstract superclasses that define "dummy" implementations of the operations and then subclassing to get each of the various implementations. There are at least two important problems with this approach:

– You must have the insight to do this in advance of the need, since adding superclasses to existing objects is not generally possible.

– I believe that this kind of classification is fundamental, and we must directly address the need rather than simulating it using mechanisms that were designed to solve a different problem.

Position

I believe that in order to fulfill their full potential, object-oriented systems must address both of these forms of classification. I therefore believe that we need to be talking about two notions of typing for object-oriented languages. I therefore believe that *class*, which has historically referred to classification based on implementation should continue to address this need, and that *type* should be used for classification at the abstract level, separate from implementation.

There are a number of issues that must be addressed by further research.

Subclass vs. subtype

Does creating a subclass imply that it must be or should be a subtype? Without additional restriction, a subclass may not be a subtype since the subclass may redefine the types of arguments or results to an operation. Whether languages should force a subclass to also be a subtype is not so clear.

Type inference

Type inference can be done at both levels, for different purposes. Type inference at the abstract level can free the programmer from the tedium of specifying all the type information. Type inference at the concrete level provides the compiler with additional information to aid in optimization.

Implementation

If typing is done at an abstract level, then the compiler gets no information (in general) about the implementations of the objects that are being manipulated. How can one efficiently implement method lookup under these circumstances? The methods used in untyped languages can surely be applied, but can one approach the efficiency of the single level of indirection achievable in languages where typing is based on classes?

The Emerald programming language has been exploring these notions for the past several years, and has partial answers to some of the questions, but much more work needs to be done.