Inheritance versus Parameterization

Erik Ernst
Aarhus University
Denmark
eernst@cs.au.dk

ABSTRACT
This position paper argues that inheritance and parameterization differ in their fundamental structure, even though they may emulate each other in many ways. Based on this, we claim that certain mechanisms, e.g., final classes, are in conflict with the nature of inheritance, and hence causes language designs to be less generalizable and more prone to semantic conflicts, and hence we recommend that these mechanisms should be optimized for playing different roles in language design.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features—inheritance; procedures, functions, and subroutines; polymorphism

General Terms
Languages, design

Keywords
Inheritance, parameterization, conceptual comparison

1. INTRODUCTION
This position paper discusses the relation between two broad classes of abstraction mechanisms in programming languages, namely inheritance and parameterization. Many technical details of the numerous variants of each of these mechanisms are insignificant for the discussion here and will not be addressed. The emphasis is on the fundamental properties which are shared among almost all of these variants, and the approach is informal and oriented toward the 'big picture'.

As a starting point, inheritance refers to the mechanism that enables classes and interfaces in object-oriented languages to be created by incremental specifications based on other classes and interfaces. Parameterization refers to the support for declaring and passing parameters, such as value parameters of methods and type parameters of methods and classes. Parameter passing creates a member of a set of instances of said entity (such as an invocation of a method respectively the creation of a class by passing actual type arguments to a parameterized class). Note that we consider a parameterized class C as one kind of instance, a class C<T> created by passing actual type arguments to C as a different kind of entity which is an instance of C, and an object o created as an instance of C<T> as a third kind of entity (typically involving yet another parameter passing and the method-like concept of a constructor). Hence, the instance relation may exist at multiple distinct levels.

We will at first discuss how inheritance and parameterization are similar in several ways, and then focus on their differences. Finally, we argue that good programming language design should make these two classes of mechanisms quite different because they are good at different things, rather than aiming for convergence.

2. SIMILARITIES
The classes of mechanisms known as inheritance and parameterization are similar in several ways. In particular, they enable entities such as classes and methods to be defined in such a way that they allow for variability, which is subsequently narrowed down to more specific choices elsewhere. For instance, a method may be declared to take several formal (ordinary or type) parameters, and each invocation of this method will narrow down the potential values of these parameters to one concrete choice of actual arguments.

```java
// a method like 'square' allows for many potential invocations, depending on the choice of argument values
int square(int x) { return x*x; }
// a concrete invocation narrows down the choice to one of them, in this case at runtime
int y = ... // some unknown computation
int z = square(y);
```

The choice of arguments will not always fully specify the instance, since, e.g., an object-oriented method like square may depend on entities reachable via names in scope, and these entities may even be mutable and dynamically allocated objects. Similarly, passing type arguments to a parameterized class will not fully determine the instance, e.g.,
because the class depends on types that are in scope rather than being passed as a parameter.

Turning to inheritance, a class may declare a set of features, such as methods, instance variables, and nested classes, and a subclass may narrow down the possible variations of this set of features, e.g., by adding more instance variables or defining features that were declared abstract. Here is an example in Java:

```java
// a class like 'C' makes a choice for the
// features named 'x' and 'foo'

class C {
    int x;
    void foo() { x=1; }
}

// inheritance may make a choice for features
// having other names, or it may redefine
// existing names

class D extends C {
    double y;
    int bar(char c) { return (int)c; }
    void foo() { super(); y=4.2; }
}
```

The set of features for which subclasses of class C may freely choose arbitrary properties is a superset of the corresponding set of features for class D, because restrictions have been added for bar. For D.boo it could be claimed that the semantics of the overriding version is a refinement of the semantics of the inherited version. Refinement of a feature is not the only possible effect of creating a subclass in mainstream object-oriented languages, but since we cannot add a double foo() to D, the core of the basic instantiation semantics is at play here: The creation of a subclass of a class C narrows down the possibilities to certain choices among all the possible choices that such a subclass could make, and the resulting subclass is a more specific entity than the superclass.

The core benefit derived from abstraction mechanisms like these is that the variability is both enabled and constrained by the original declaration, and all the usages (such as method call sites or subclasses) reuse a concretized version of the basic entities, without redundantly specifying their structure at each use site. At this point we might suspect that parameterization and inheritance are just different ways to narrow down a set of choices in an instantiation process, i.e., that these mechanisms are fundamentally very similar.

It was noted already in Mads Torgersen’s PhD thesis [7] and used in the description of vObj [6] that declaration and passing of value parameters may be emulated using inheritance, and it is well-known, for instance in the Scala community, that abstract type members may be used to emulate type parameters. Here is an example in Scala:

```scala
// we may emulate a method using a class

abstract class foo {
    val x: Int;
    val result = x*x;
}
```

Conversely, parameterization in a broad sense corresponds to the ability to replace selected elements in a composite entity in a different way for each invocation or instantiation of that entity, and it is not hard to think of parameter mechanisms that will enable us to extend a list of declarations, or concretize abstract declarations, or even replace an implemented method by a different one. For a hint of an example in a hypothetical Java-like language:

```java
// parameter passing amounts to subclassing,
// invocation amounts to object creation, and
// result delivery amounts to field lookup

val y: Int = ...
class foo_invoke_on_y extends foo { val x = y; }
val z = (new foo_invoke_on_y()).result;

// we may emulate a type parameter using an
// abstract type member

abstract class E {
    type T <: Any
}

// parameter passing amounts to subclassing

class E_applied_to_Int extends E { type T = Int }
```

A complete language design realizing this idea would need to contain a rich kind of types (here: SomeType) describing the admissible binding specifications (e.g., that we cannot include a binding of foo to a method with a different return type than void, but we can freely add a binding for bar). Alternatively, we could consider this mechanism to be purely untyped and insist that it only occurs at compile time. The point is that we can emulate inheritance using parameterization.

Hence, taking a broad perspective and skipping over the innumerable technical details, it is not unreasonable to claim that these two classes of mechanisms can emulate each other, and we could therefore strive to get rid of one of them and generalize the other one to do the jobs of both. Presumably
we would learn a lot about the nature of both by doing such a thing, possibly in both directions.

However, the point of this paper is to argue that it is more fruitful to note a particular type of difference among the two, and emphasize this distinction in order to obtain language designs based on concepts that are both natural and powerful.

3. A FUNDAMENTAL DICHOTOMY

The distinction which will be at the core of the following discussion is that inheritance works naturally as an end-function, i.e., a mapping with the same domain and codomain, whereas parameterization is in general concerned with mappings between different domains.

In particular, the vast majority of the inheritance mechanisms that we encounter maps classes to classes, possibly based on class bodies, mixins [1], or similar specifications of the operation. This is an endofunction because the kind of entity produced by the operation is the same as the kind of entity that was used as a starting point.

In contrast, parameterization of a method enables a different transformation which maps a method to a method invocation (based on a specific choice of actual arguments). It is evident that the domains are different, typically because a method invocation cannot be invoked—it is a fundamentally different kind of entity. In a context where function types are used we can see this on the type: If a given method \( x.m \) has type \( \text{int} \rightarrow \text{int} \) then \( x.m(2) \) has type \( \text{int} \) and cannot be invoked. Even with curried functions where \( x.n \) has type, say, \( \alpha \rightarrow \beta \rightarrow (\alpha \times \beta) \), \( x.n(2) \) has type \( \beta \rightarrow (\text{int} \times \beta) \), which is fundamentally different because it must be called in different ways than \( x.n \), and the outcome of type \( \text{int} \times \beta \) cannot be invoked (even if \( \beta \) is a function, we must project the pair to the second component before it can be invoked).

Similarly, a class that takes a list of type parameters is a different kind of entity than a class that has received actual type arguments. The former is actually a function from the type parameter list to classes, and the latter is a class. Currying could be used for type parameters as well, but the situation is similar and it remains true that parameterization maps an entity of one kind to an entity of a different kind.

This fundamental difference positions inheritance well as a mechanism for gradual concretization (or refinement, or specialization). In contrast, parameterization is well positioned for precisely keeping track of the degree to which the concretization (or invocation) has taken place. In other words, parameterization is inherently a mechanism that involves a fixed number of steps, whereas inheritance is inherently better suited for using a variable number of steps. We may therefore apply inheritance repeatedly (and potentially a number of times which is not fixed at compile-time) thus creating more and more elaborate instances (i.e., subclasses). Parameterization, in contrast, calls for strict static analysis (or even syntactically based constraints) to enforce conformance to the domains associated with the fixed number of steps. Henceforth we will abbreviate the two categories of abstraction mechanisms as variable-step and fixed-step, respectively.

We claim that it is worth striving for consistency in language designs with respect to this distinction, and also that both flavors of abstraction serves useful purposes and should be exploited as such. In particular, there is a non-trivial cost associated with the introduction of rules and mechanisms that contradict the nature of the two. Let us discuss inheritance and parameterization separately.

Assuming that we strive for a smooth and consistent variable-step mechanism, inheritance should be designed in such a way that it does not violate its nature as an end-function. For instance, there should not be a distinction between classes that may be subclassed and other classes that cannot, and there should not be mechanisms that prevent refinement with respect to individual features. So final methods are problematic, and so are final classes. Great care should be exercised in order to handle name clashes among a class and its superclasses gracefully. In general, inheritance ought to be available and safe as a runtime mechanism, because that is a natural extension of allowing a variable number of steps in any context.

Parameterization, on the other hand, is optimized for keeping strict accounts of the degree to which the parameterization has taken place. For instance, traditional function types allow us to call a curried function using a fixed number of application steps, and the types of the arguments and the kind of entity returned are fully determined at each step. It is well-established and natural to use static type checking and similar analyses to enforce conformance to the nature of these fixed steps. On the other hand, we claim that it would not be appropriate to attempt to enhance parameterization mechanisms to allow variable-step schemes. For instance, with method calls in an imperative object-oriented context, the dynamic semantics of a method invocation requires a very well-defined point in time for the actual execution of the method body—ambiguity in this area would make programs very hard to reason about. Similarly, generic classes have a clear transition associated with the provision of actual type arguments.

In order to maintain a high degree of variable-step flexibility for inheritance, and in order to preserve the high degree of static fixed-step analyzability of parameterization, it would be very useful if ways could be found to make parameterization take over the elements of the traditional inheritance semantics that restrict repeated refinement. This would affect the treatment and interpretation of final methods and classes, and in general any mechanism whereby a definite transformation from one domain to another takes place, e.g., a transformation from an abstract class to a concrete class.

In the opposite direction, virtual classes [4, 5, 2, 3] illustrate how an inheritance mechanism is able to provide a variable-step version of type parameterization. This enhances the flexibility in various ways, including support for inheritance that is applied to classes which are not known at compile-time, but the balance between dynamic flexibility and static safety is complex and requires careful consideration.

```plaintext
// a `gbeta` example shows how a virtual class `v` can be refined an unknown number of times; first in a "normal", simple, compile-time class hierarchy
// using another hierarchy `F` and `H`

C: { v:< F }; // introduce `v` as F
D: G { v: G }; // further-bind `v` to G
E: D { v: H }; // further-bind `v` to H

// ... then in a method `useC` taking a class as
```
an argument which must be 'C' or a subclass thereof, e.g., 'D' or 'E'; in this method 'c' is a local variable which denotes an instance of a subclass of 'myC' whose 'v' is refined to include an int field named 'y'

useC: (myC: #C){ c: myC { v:: { y: int }} .. }

The virtual class \( v \) in class \( C \) is similar to a type argument, but it may be refined a variable number of times. In the subclasses \( D \) and \( E \) refinement is fully specified at compile time, but in the method \( \text{useC} \) we create a subclass of the argument \( myC \) (which is actually a class, not an object which is an instance of \( C \)); if we invoke \( \text{useC} \) with \( C \) as the argument then \( myC \) (and hence \( c \)) will have a \( v \) with the value \( F\{ y: \text{int} \} \), with \( E \) as the argument it will have the value \( H\{ y: \text{int} \} \), etc. This illustrates how we may obtain a greater amount of flexibility than that which is associated with type parameterized classes, where variable-step gradual refinement of a type argument is very inconvenient to handle in a type safe manner.

4. CONCLUSION

We have discussed the notion of inheritance and the notion of parameterization informally, skipping over a large number of technical details and difficulties, in order to emphasize one particular fundamental trait which we consider crucial. This is the natural association between inheritance and refinement taken in a variable number of steps—mathematically connected with the notion of an endofunction, mapping between entities of the same kind—and the corresponding association between parameterization and definite changes of domain in a fixed number of steps—enabling a high degree of static analyzability. We argue that it is beneficial for the quality of a language design if this difference in the nature of the two classes of abstraction mechanisms is respected, such that the variable-step refinement flexibility of inheritance is allowed to unfold smoothly and as dynamically flexible as possible, and the fixed-step nature of parameterization is exploited to apply static analysis.

5. REFERENCES