Advantages of dynamic method-oriented mechanism in a statically typed object-oriented programming language Z_0

Sašo Greiner, Janez Brest, Viljem Žumer
University of Maribor, Faculty of Electrical Engineering and Computer Science,
Smetanova 17, 2000 Maribor, Slovenia
saso.greiner@uni-mb.si

Abstract

Z_0 is a simple class-based pure object-oriented programming language. It was basically designed as an experimental language that would provide a static yet expressive type system, method dynamics and pure object abstraction philosophy. Classes define their state explicitly and exclusively through method abstractions. There are no instance variables because Z_0 aims to achieve a clean and strict method-based modification mechanism for objects.
Dynamic features that enable method-based calculation between objects have been incorporated in a way that conforms to the language's strong static type system. This has been done with a method update mechanism that is fully checkable at compile-time and requires no runtime overhead in invocation.

Keywords: language design, compilation, type systems

1 Introduction to the language

It is a well known fact that certain language mechanisms attributable to dynamic languages such as Smalltalk [5] or Self [12] are hardly ever found in languages with a static type system. This is not surprising as dynamic mechanisms such as dynamic typing, runtime code optimisation, method and/or class modification tend to introduce a demanding task for the otherwise robust static type checker. Another problem arises when the language is not interpreted on parse tree level but is compiled into an intermediate (eg. bytecode for JVM or CLR) or actual code for a target processor. It is one thing to replace superclass in symbol table of a parse tree interpreter and a whole different scenario when the code is compiled into instructions and running in a virtual machine environment.
The language Z_0 [6] was designed to be a safe and efficient alternative to interpreted dynamically typed programming languages. Z_0 enables safe static typing with dynamic extensions through type `Self` (extensively described in [2]). Some of the most popular industrial strength languages, Java and C Sharp among them, provide merely invariant type systems even though we know for a fact that much more flexibility can be achieved within static typing systems. Z_0 provides covariant changes in return types and contravariant changes in argument types. This is as far as it gets with type changes within the class hierarchy. One of the language features is a pure reference-based object model which implies that all values, primitive and other are objects. The implementation of methods for primitive classes is, however, quite different. All method invocations upon values of primitive types (such as plus method for integral types) are compiled into virtual instructions which realise that method's functionality. This means these (static, "hard-wired") methods execute quickly, without any overhead of virtual invocation. The execution environment of Z_0 consists of objects whether they represent integral values or first-class program control constructs such as blocks, methods, or loops which are of course storable values as well. Object model of Z_0 has been designed to be concise and straight-

forward. Classes are used for the description of runtime objects. The entire class hierarchy is situated on one principal class – `Object`. Every class implicitly inherits functionality from `Object`. Z_0 is certainly unorthodox in comparison with most class-based object-oriented languages because the classes declare no explicit instance variables to represent object's state at runtime. In contrast, classes declare only methods. This implies that the object's state is given exclusively through methods. However, Z_0 *is* an imperative language and variables are provided in methods. Because class instance variables have been promoted to methods a mechanism for dynamic method update is crucial. This approach has mainly been inspired by the idea of a seemingly "stateless" philosophy for runtime objects in a compiled statically typed programming language. Methods have the ability to be modified and updated as straightforwardly as local variables. Methods too are first-class values which means they can be passed as arguments and returned from functions.

Z_0 implements a descendant-driven multiple inheritance model with uncomplicated rules for overriding. Because execution of byte code is much more efficient than interpretation, the language is compiled. The underlying execution platform of Z_0 is a fully implemented virtual machine environment similar to that of Java (JVM [10]). Empirical results show that execution time is comparable to JVM. Section 2 describes a strict method-oriented idea of an object and the advantages of such representation. Two variations of method modification are presented – one with explicit method update mechanism built into the language and the other which is feasible through reflection mechanics of the virtual machine. The first variation is presented in section 2, and the second in section 3. Section 4 gives a practical example of how runtime information may be incorporated into method optimisation. Section 5 provides a few notes on implementation details of method update facility. Related work and ideas are presented in section 6 and section 7 finally concludes the paper.

2 A strict method-oriented view of an object and method-update mechanism

A class in Z_0 declares only methods, not variables as is the case with Java, C sharp or C++. This exhibits a cleaner approach to object-orientation because the only operation upon object is method invocation. Class fields (instance variables) and methods are unified into a single state abstraction mechanisms. This philosophy yields a higher degree of abstraction because we do not require any knowledge of how object's state is represented. We only need to know the protocol (interface) to which object conforms in its messages. Unification of methods and fields serves as a considerate basis for a simpler and better suited formal model of language because there is no ontological distinction between the two. Consider a code snippet bellow which shows a distinction of a 2D point class written in Java (`JavaPoint`) and in Z_0 (`ZPoint`). Java defines state explicitly through instance variables `x` and `y` which are modified via methods `get` and `set`. Z_0 on the other hand defines only methods `get` and `set` and thus blurs the state into method abstractions.

```
class JavaPoint{
  public int getX(){
    return x;
  }
  public int getY(){
    return y;
  }
  public void setX(int _x){
    x = _x;
  }
  public void setY(int _y){
    y = _y;
  }
  private int x, y;
}

class ZPoint{
  restricted getX: Integer{
    return 0;
  }
  restricted getY: Integer{
    return 0;
  }
  restricted setX(Integer _x){
```

```
    method getX(Integer x)={
      return _x;
    };  // method
  }
  restricted setY(Integer _y){
    method getY(Integer y)={
      return _y;
    };  // method
  }
}
```

As can be seen, Z_0 defines object's state as reflected actively by the `getX` and `getY` methods which are modified to return the new value upon each invocation. As is the case in Java, all methods are virtual unless explicitly declared static. Method is updated using the `method` keyword which expects method name and the new definition. Right-hand side is a method block or method name from another class. Method update mechanism works for member methods and those declared in other classes. Another fact that should be explained briefly is the `restricted` access modifier not found in Java or C++. A restricted method is in fact a publicly accessible method that cannot be modified.

In Z_0 programming philosophy we tend to think about objects as dynamic entities that perform action rather than concern ourselves with what they contain. Methods are a more general concept than fields so fields may be promoted to methods. Because methods are the only way to reflect object's state there must be a way to modify object's state by modifying its methods. An implemented method update mechanism is able to update (replace) a method of an object at runtime in a type-safe way so that the consistency of method signature remains intact. Meta architecture will also allow replacing methods on class level which will be type safe because only method implementations will be modified. As is the case with instance level replacement, method signatures will have to remain compatible so that class structure remains consistent. It becomes obvious that such a mechanism exhibits a lot of advantages over "fixed" (immutable) methods. Take for example a method which has been constructed by the programmer to provide some functionality. As it often happens, program designers cannot take into account every possibility that can occur at runtime. Frequently, certain parameters and knowledge that could not be ascertained at compilation time only become available when the program is already running. A thought often repeated is then: if I had known that, I would have written the code differently. In Z_0 one can take runtime parameters, whatever those may be, into account and tailor a method as needed. Another aspect that benefits from this ability is runtime code specialisation. By employing runtime information and actual data, a code may be specialised for some rare or extreme case which could not be predicted at compile time. Z_0 allows not only the complete replacement of a method at runtime but also more fine-grained modifications. Method consists of an outer block and subblocks which are all first-class values. This means we can modify any block inside a method. To harness the full functionality of this idea, a meta-level architecture that will reify a meta object protocol (MOP) for Z_0 is being implemented at the time of this writing. Clearly, "only" introspective (readonly) reflection as is the case in Java and C Sharp does not suffice. Mechanisms that can alter runtime information are needed.

Runtime code specialisation also benefits from the ability of altering a method's functionality. Methods are no longer class specific but bound to runtime objects. Method descriptors (pointers) must be stored inside object because two objects of the same class may have their methods altered independently. The most important fact about method update mechanism is that it can be fully type checked at compile-time. Method updating is thus a dynamic yet safe extension to the static type system of the language.

3 Method modification via first-class language constructs

As has been said earlier, in addition to replacing entire methods they may also be modified with a more fine-grain control. Z_0 includes builtin types for control structures such as blocks, iterative statements, and `if` statement. All language control constructs, which have been made very similar to those in C++ and Java, are used in a straightforward manner. Since all constructs are objects, they may be stored, passed as arguments, and returned from functions. A method is in fact just a named block. A method has signature consisting of name and parameter types and one block which corresponds to method body. Method block contains normal language expressions and statements as well as other nested blocks. Blocks are represented under the notion of closures. A closure is simply a piece of code with its own environment. But since block (closure) is an object, it makes sense to make blocks modifiable and replaceable. Block is the smallest modifiable unit of execution in Z_0. You cannot replace an expression in a block (at least not at the time of this writing) but you can replace any nested block inside parent block. This gives the possibility of modifying methods at block level which offers much finer control than changing entire methods. Because of Z_0's pure reference model, control structures such as blocks and loops contain references to other blocks. A `while` loop for example contains reference to conditional block and the body block. Meta level computation for `while` loop will therefore allow to manipulate both conditional as well as body block of the loop. Objectisation of control constructs is vital in dynamic manipulation of program control flow. It should be noted that meta architecture plays a crucial role here as it serves as an abstraction interface layer between high level programming environment and virtual execution environment. The Z_0 meta architecture, when fully implemented, will allow almost unlimited manipulation of runtime object environment. It should be clear why we decided to make methods modifiable in two different ways. Direct modification/replacement is needed because objects exhibit their state exclusively through methods. Such modification is therefore more natural, cleaner, and quicker. But if we want specific, fine-tuned modification, we resort to reflection mechanisms. Method closures are dealt with through the `Method` metaclass. The functionality and the strength of expressibility of this class directly reflect the capabilities of Z_0 virtual machine. Let us sum up a part of functionality of `Method` metaclass currently supported by the virtual machine:

```
class Method{
  public Method;
  restricted getName: String;
  restricted getReturnType : Class;
  restricted getParameterTypes: Vector;
  restricted  getClosure: Closure;
  restricted isFunction: Boolean;
 ......
}
```

Instance of class `Method` represents a runtime method abstraction. As seen from the class definition, the method specific to modification mechanism is `getClosure` which returns an instance of method's block. `Closure` class then defines the methods such as `getNestedClosures` which returns a vector of all closures nested within this block and `setClosure` which replaces a specific nested block.

A drawback is that the architecture of a compiled language is susceptible to certain limitations imposed by the fact that the code is compiled and executed in a virtual machine environment. For an in-depth description of those limitations the reader is referred to [6].

4 A practical example

To show the advantages of method update facility we shall resort to a practical example of a real world application. Consider a common data structure – a binary tree which is used in solving many actual problems. One such problem is representing a parse tree of a program structure during the compilation process. The compiler reads program input and builds parse tree for every recognised structure. Let us look at basic arithmetic expressions for adding, subtracting, dividing,

and multiplying two values. It makes prefect sense that parse tree for all such expressions is represented equally: an operator and two expression branches. Evaluating such tree typically consists of checking operator type and then performing required operation on operands. The more operations we have the longer it takes to check the operation type. With method update facility in Z_0 this may be written the following way:

```
class ParseTree {

  public ParseTree(Integer a, Integer b,
                        Character op)
  {
     if{ op == '-'; } then {
          method evaluate = {
              return a - b;
          };
     } else if{ op == '+'; } then {
          method evaluate = {
              return a + b;
          };
     } else if{ op == '*'; } then {
          method evaluate = {
              return a * b;
          };
     } else {
          method evaluate = {
              return a / b;
          };
     };
  }
   restricted evaluate : Integer
  {
      return 0;
  }
}
```

At compile time we do not have the knowledge what type a particular parse tree node will be, so we have to consider all possible types. But when the tree node is constructed, the constructor sets the evaluation method properly. Note that this is done only once, and no matter how many times the `evaluate` method gets called, it will execute only the method that properly evaluates expression. No more checking for operation type is needed because general knowledge has been supplemented by runtime information. This is a very simple example but it demonstrates how functionality that has been predetermined by compilation can be optimised using runtime information. In C++ this is feasible with function pointers but without the necessary safety, ability of fine tuning, and elegance.

5 A note on implementation

Methods cannot be left to classes if they have the ability of modification at instance (object) level. This necessitates the need for method table to be present in every object. Such presentation enables alteration of objects, but as one might imagine, it also exhibits a space efficiency drawback. Obviously an object requires more space when it carries its method table with it. An optimisation of this remains a challenge for future. A method in Z_0 requires two receivers. One is needed to find a method and the other to execute invocation. This is because object's method may be replaced with method of another object. Method table entry contains memory address of method, actual receiver (self), and environment pointer. The latter is needed to successfully describe a closure which is basically what every method is.

6 Related work

The notion of closures is found in many highly respected and established programming languages but Common Lisp Object System [8] was our main inspiration. Strict and pure object-oriented model is drawn mainly from from Smalltalk, Strongtalk [1] and other languages that advocate pure objectisation. Variant typing system, its flexibility and applicability in solving practical problems have been observed in Eiffel [9] and Polytoil [3]. A similar type system has been implemented by Sather [11]. An in-depth discussion with theoretical background has been given in [4].

7 Conclusions and future work

We have designed an experimental object-oriented programming language Z_0 to demonstrate and, before all, to research the effects of a strict method-based model

applied to a static type system in a compiled language architecture. Our goal was to demonstrate advantages and practical applicability to real-life programming problems. Z_0 conforms to the philosophy of pure object-orientation which achieves a clean programming model and enables functionality that is natural to pure object system. Z_0 implements a "full" runtime method replacement mechanism which is type safe, efficient, and is of high practical importance. With meta architecture extension (that is currently being implemented) Z_0 will be able to construct and manipulate entire programs at runtime.

In the time when vast amounts of memory are available at a reasonable price we do not consider per-object method table to be particularly space inefficient even though object with this representation do eat up a lot of memory. Classes declare only methods and when speaking in the spirit of abstracting this is an advantage, not a drawback. It gives us a novel opportunity to think about objects and their behaviour rather than how they are represented.

In a programming language design future challenges always exist. In Z_0 we are focused on extending the static type system with parametrised polymorphism and bounded polymorphism. Dynamic compilation and code generation are another issues to be considered. Currently Z_0 does not support aspect-oriented programming [7] but with pure object-oriented model and reflection dynamics via MOP, implementation of aspects is not far from reality. And then of course there are optimisation issues on the virtual machine level, internal object representation, and efficiency of compilation.

References

[1] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 215–230, 1993.

[2] Kim B. Bruce. *Foundations of Object-Oriented Languages, Types and Semantics*. The MIT Press, Cambridge, Massachusetts, 2002.

[3] Kim B. Bruce, Angela Schuett and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. *Lecture Notes in Computer Science*, 952:27–51, 1995.

[4] Luca Cardelli and Martin Abadi. *A Theory of Objects*. Springer-Verlag New York, Inc., 1996.

[5] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[6] Sašo Greiner, Damijan Rebernak, Janez Brest and Viljem Žumer. Z0 - a tiny experimental language. *Sigplan notices*, 40(8):19–28, 2005.

[7] Gregor Kiczales et al. Aspect-oriented programming. *Proceedings European Conference on Object-Oriented Programming*, 1241:220–242, 1997.

[8] Jo A. Lawless and Molly M. Miller. *Understanding CLOS: The Common LISP Object System*. Digital Press, 1991.

[9] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 1988.

[10] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly, 1997.

[11] Stephen M. Omohundro. The sather programming language. *Dr. Dobb's Journal*, 18:42–48, 1993.

[12] David Ungar and Randall B. Smith. Self: The power of simplicity. *OOPSLA'87*, 4(8):227–242, 1987.