STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 1, 2008

INTRODUCING A NEW FORM OF PARAMETRIC POLYMORPHISM IN OBJECT ORIENTED PROGRAMMING LANGUAGES

IANCU MIHAI CĂPUTĂ AND SIMONA MOTOGNA

Abstract. Nowadays software development tools have to provide effective means of data manipulation with minimal development time. As types represent the meaning of raw data, this paper focuses upon taking types to another level in an object oriented dynamically type-safe programming language, in order to increase language flexibility and productivity.

Key words: Object-oriented programming, parametric polymorphism

1. Introduction

Mainstream object oriented programming languages, such as .NET languages or Java, support both kind of type-checks (static and dynamic).

Static type checking or static typing means that the typechecker, which may or may not be part of the compiler, performs an analysis over the source-code at compile time to ensure that certain type-constraints are not being violated.

In C#, for instance, static type checks are made when resolving the overloading of methods, or when performing an upcast (casting from derived type into base type). Dynamic type checking (also known as runtime type checking) characterizes a dynamically typed programming language which is one where type constraints are being checked at runtime. In C#, dynamic type checks are made when a downcast (casting from base type into a derivate type) is performed, in order to ensure that the backing object, that is cast into the derivate type, is an instance of the derivate type or of another type that derives from that derivate type. This check can only be done at runtime, except some scenarios in which a smart compiler can figure out that the cast is possible.

By taking the dynamically-typing even further, productivity can be increased considerably. This can be achieved by introducing the concept of

Received by the editors: May 5, 2008.

2000 *Mathematics Subject Classification.* 68N15, 68N19.

1998 *CR Categories and Descriptors.* D.3.3 [**Language Constructs and Features**]: Data types and structures, Polymorphism.

Type-unbound variables which induces a new form of parametric polymorphism. We propose a programming language prototype, called X Language or shortly X, that will incorporate this new form of parametric polymorphism with clear benefits in productivity. It targets the .NET framework 2.0 and the syntax is almost identical to C# 2.0s. This language is under development. We will often make references to C#, which is very popular amongst developers, but these references are traceable in most of the OOPL on the mainstream. (Java, Delphi.NET, C++.NET etc.)

This paper is organized as follows: The second paragraph is introducing the notion of type unbound variables, and then in paragraph three we discuss its relation with parametric polymorphism, describing in several examples the role of type unbound variables in implementing paramatric polymorphism. Section 4 discusses different aspects that should be taken into consideration when introducing this feature in a programming language. Paragraph 5 refers to some details regarding the implementation of type unbound variables, targeting a virtual machine, and in the end some ideas for our future work on this langauge.

2. Introducing type unbound variables

In C#, when you declare a variable, you must specify its type. This is a hint for the compiler so that it will be able to enforce type-safety. That variable will be bound to its type throughout its entire scope. This means that you wont be able to change its type at runtie. Please note that eventhough a variable of type B can have a backing object of type D, which is a derivate of B, conceptually speaking that variable is of type B (from a compilers point of view). Type unbound variables refer to variables that arent bound to a certain type. At the moment i of execution the variable has the type T_i and at the moment $i+1$ it can have any other type, T_{i+1} which is not necessarily ad-hoc polymorphic with T_i.

In almost every object oriented programming language, there are system classes used to manipulate the concept of Type, such that information about types (system or user defined ones) is accessible at runtime throughout instances of this classes. In C# there are mechanisms (such as reflection [4]) for creating objects based on information held by these instances.

This is the common way to create instances of a variable type at runtime. Sometimes this turns to be quite tedious and the usage of `System.Type` class in order to do that does not intervene to the programmer in a natural manner. This class hints us more to a class schema or an object runtime inspector rather than to a usable type.

In the **X** language, which implements the concept of type unbound variables, the behavior of `System.Type` class instances is similar to the one of a type identifier. We can use an instance of the `System.Type` class (which

holds meta-informations about a type) just as if it was a type, as shown in the following example:

Example 1: Usage of a type unbound variable (a)

```
System.Type T;
T = int;
T a = 10; //a is a valid integer with the value of 10
```

3. Type unbound variables and parametric polymorphism

Polymorphism is a programming language feature that allows values of different data types to be handled using a uniform interface. Christopher Strachey identified in 1967, two fundamentally different kinds of polymorphism: ad-hoc and parametric [3].

Ad-hoc polymorphism is when the range of actual types that can be used is finite, and the combinations must be specified individually prior to use. In object-oriented programming this is implemented through class-inheritance (objects of different types can be handled uniformly through an interface or a common base class called superclass).

Parametric polymorphism enables a function or a data type to be written generically so that it can handle values identically without depending on their type[1], such that it increases the expresiviness of the language.

Firstly introduced in ML (1967), and then inherited in several other langauges, parametric polymorphisim still remains a desirable feature in a programming langauge, due to its benefits. Recently, Java and C# introduced "generics" as a form of parametric polymorphism.

Cardelli and Wegner [1] introduced in 1985 "bounded parametric polyorphism", which imposes some bounds on the parameters, such as to be subtypes of a given type.

For example, in C++ parametric polymorphism is implemented with templates or in C# with generics, as presented in Example 2.

Example 2: C# Generics

```
List<int> myList = new List<int>();
myList.Add(1);
myList.Add(2);
myList.Add(3);
```

This kind of parametric polymorphism is resolved statically at compile time: "In .NET 2.0, generics have native support in IL (intermediate language) and the CLR itself. When you compile generic C# server-side code, the compiler compiles it into IL, just like any other type. However, the IL only contains parameters or place holders for the actual specific types. In addition, the metadata of the generic server contains generic information. The client-side compiler uses that generic metadata to support type safety. When the client provides a specific type instead of a generic type parameter, the client's

compiler substitutes the generic type parameter in the server metadata with the specified type argument. This provides the client's compiler with type-specific definition of the server, as if generics were never involved. This way the client compiler can enforce correct method parameters, type-safety checks, and even type-specific IntelliSense" [2].

In the following, we propose some use cases and present some examples that show how type unbound variables are introduced and what improvements they have on the written code.

Use case 1: In C# the abstract data type List is available as a generic type. When you use the class List to specify the type of a variable, or to derive from it, you have to specialize it by telling the compiler what kind of elements this list will handle. Parametric polymorphism comes inherently in X Language when using an instance of `System.Type` class as a type identifier for a methods return type, a methods argument or class member, as illustrated below.

Example 3:

```
class GenericList
{ protected System.Type itemType;
  public ItemType
  {
    get { return this.itemType;}
    set {this.itemType = value;}
  }
  public GenericList(System.Type itemType)
  {
    ItemType = itemType;
  }
  public void Add(ItemType item)
  {//... }
  public void Remove(ItemType item)
  { //...}
  public ItemType operator [](int index)
  {//...}
  . . .
  }
```

In the above example notice that the property `ItemType`, that represents the type of one item from the `GenericList`, is used in `Add, Remove` method declaration, and in [] operator declaration. Parametric polymorphism comes from the fact that by changing the `ItemType` of the `GenericList` instance, it will handle different lists of items, without having to change any code.

When specializing this kind of genericity (by supplying a valid `System.Type` instance for each `Type variable` in the class), you will not create a new type,

but a new behavior. This approach makes the definition of generic lists more naturally.

Use case 2: Another situation in which type unbound variables can be useful is the following: Suppose that we have a class, named `MyClass`, written by a third party, leaving out of posibilities of modifying this class. We want to create a proxy for `MyClass`, named `MyClassProxy` which delegates all the methods calls to a Remote method call server (this illustrates the design pattern Proxy, and is often used in RMI - Remote Method Invocation, and RPC - Remote Procedure Call). `MyClassProxy` looks identical in terms of method signatures to `MyClass`, but those types are not ad-hoc polymorphic since instances of these types cannot be treated uniformly through an interface or a base class that exposes their methods. Suppose that we have to write down code that dynamically decides whether it uses objects of `MyClass` or objects of `MyClassProxy` and does a certain task. In order to achieve that, in C# 2.0, well either have to write the code that does the job, twice, firstly for `MyClass`, and secondly for `MyClassProxy`, or as an alternative we will have to extract that code into a generic method, but that is a bit intrusive, and sometimes it is not quite straight-forward.

In **X** programming language the solution comes from the usage of the parametric polymorphism in the form of type-unbound variables. We declare a variable of type `System.Type`, and we fill it with `MyClass` or `MyClassProxy` accordingly. Then we use that variable to declare the instances of the class `MyClass/MyClassProxy`, and operate with them just as if we dont have to decide whether to use `MyClass` or `MyClassProxy`. Example 4 shows how this approach is taken in **X**.

Example 4:

```
System.Type T;
if (bUseProxy) //we need to use the proxy class
{
   T = MyClassProxy;
}
else
{
   T = MyClass;
}
T obj; // crete an instance of MyClass or MyClassProxy
             depending on the previous decision
 // use obj no matter what type underlies it
```

Example 5: Dynamic casting - Suppose that we have a variable T of type Type. A cast will be performed at runtime from int to the value of the T variable (which is a type).

```
int a;
```

```
Type T;
T b;
//read a value for a;
if (a>0)
{
   b = float;
}
else
{
   b = double;
}
b = (T)a; //a dynamic cast is performed from int to float or
             // double, depending on the runtime value of T.
```

4. Implications of type unbound variables

When implementing this new form of parametric polymorphism several aspects must be kept in mind regarding: type safety, strong typing, threads safety and limitations of type unbound variables.

4.1. **Type safety.** For each variable that is about to be used, at the moment of execution, the underlying type must be known, otherwise a runtime exception will be thrown.

4.2. **Strong Typing.**

- Whenever a method argument is a type unbound variable, that methods overloading resolution must be done at runtime
- Member access of a type unbound variable is resolved at runtime hence all the validations upon that member must be done at runtime
- Method calls of type unbound variables are late bound
- Each operator is subject to all the constraints the methods are subject to
- When changing the underlying type of a type-unbound variable, a policy regarding the current value of the variable needs to be adopted: Since every type has a default value, the value of the variable will be reset to this default value.

4.3. **Limitations.** We have identified three restrictions that must be imposed:

- Variable types (variables of type `System.Type`) cannot be used in class hierarchies definition (cannot be used as base types)
- Variable types cannot be used to define delegate types (function pointer types), as pointed out below.

```
class D {...}
class A
```

```
{
   Type typeVar = D;
   class C:typeVar { ...} //this is not allowed
   delegate typeVar myDelegate(D b, A a) ;// not allowed
   delegate int myDelegate(typeVar b, A a);//not allowed
}
```

- Type instances cannot be used to define entities (variables/members) of a larger domain of visibility, as in the following sequence

```
class C
{
   private Type memberType;
   public memberType member1; //not allowed:  member1
      // has a larger domain of visibility than memberType
   protected memberType member2; //not allowed:  member2
      //has s larger domain of visibility than memberType
}
```

4.4. **Thread safety.** `System.Type` instances can be regarded as shared resources once they were used to create instances of a type. The problem of thread safety arises here.

Suppose that a type-unbound variable V is used by thread Th1, and thread Th2 wants to change the underlying type of V. The system should expose a mechanism through which the programmer could be able to ensure thread-safety.

In order to ensure this, the `System.Type` class from **X**, exposes a variable counter, which indicates the number of type unbound variables of that type. When the variable counter is 0, the type can be changed without causing any havoc. Please note that this variable counter is different than the reference counter which indicates how many instances of that type are being referenced at a moment of execution.

In a compiler implementation, the type-variables could be copied into the thread local storage (with a compiler directive the programmer is able to modify this implicit behaviour) so the programmer wont have to interogate the variable counter before changing the value of the type variable.

4.5. **Lifetime and variable storage.** The lifetime of a type-unbound variable is not determined by the lifetime of the type instance that was used to define it. Consider the sequence:

```
{
   Type T;
   T a = new T();
   StartThread(a); //start a thread and pass 'a' as parameter
}
```

When T runs out of scope, the instance that was passed to the thread doesnt get garbage collected. In the storage of variable a, a reference to the value of T will be held. When variable T runs out of scope, its reference counter gets decremented, but it will not reach down to 0 because the instance referenced by a, will be passed to a thread, hence incremented.

The type-unbound variable storage is subjected to all the policies, that the type-bound variables (the regular ones) are subjected to.

5. Implementation details

When implementing such a feature in a programming language, several aspects should be taken into consideration. Firstly, type-unbound variables are not syntactic sugar. In order to implement this feature several extensions must be supported by the core of the virtual machine and covered by the design of the compiler.

There are plenty of ways to implement such a feature, and many performance-related policies can be applied. These implementation details aim a virtual machine that will support type-unbound variables.

The compiler will not be able to fully handle the usage of variables, because their type might be unknown at compile time. This implies that the virtual machine will provide mechanisms to handle variables (such as member function call, member access, etc) as instructions built within its core. In the sequence of code below, well try to exemplify how one can implement this feature. The code is written in C++, and covers only the surface of the concept: how to implement instructions in the virtual processor, that provide method invoking and member access of type-unbound variables.

The idea behind the implementation can be understood from the definitions of the structures and the functions.

The structure *Method* holds metainfo about a method. These kind of metainfo are also usefull for reflecting upon a method. The structure *Type* holds metainfo about a type, in which the methods and the type contains can be organized as a dictionary that maps a method hash to a *Method*. The structure *Instance* contains information about an instance of a class. If the type member of this structure can be chagned, we are talking about a type unbound variable.

The function *call()* represents the call instruction supported by the virtual processor. The function *OverloadingResolution()* resolves the collision of methodes that have same *methodHash*. Two methodes have same *methodHash* if they have same name.

The function *xcall* represents the extended call instruction suported by the virtual processor. It perfoms a dynamic call.This means that the method cannot be determined at compile time, and it has to be searched through the methods of the underlying type, by the *methodHash*. This hash is determined

at compile time by applying some sort of a hash function on the method name. *methodAddress* represents the method address that needs to be determined. If the type does not contain a definiton for that method, then an exception will be thrown. If the method is virtual, then we need to lookup its address into the virtual function table, still by its hash, and if the method is not virtual then its address is the one held by the *methodInfo* structure returned by the method overloading.

The *xMemberAccess* function provides access to a member returning its address from a specific variable storage. It will perform a look-up by the *memberHash*, into the *Type* of *Instance*, to find the address of the member into the *Instance* storage. This access is type checked, which means that if the *Type* does not contain a definiton for the specified member, an exception will be thrown.

```
struct Method
{
   void* address; //address of method, determined at load-time
   bool isVirtual; //whether or not the method is virtual
   //...  //any other metainfo
};
struct Type
{
   Method** methodes;
};
struct Instance
{
   Type* type; //the type of the instance
   void* vft; //virtual function table
   void* storage; //data storage
   //...//other related info
};
void call(void* method) { }
Method* OverloadingResolution(Type* type, unsigned long methodHash,
   Instance** params)
{ }
void xcall(Instance* _this, unsigned long methodHash,
   Instance** params, unsigned int paramCount)
{
   void* methodAddress = 0;
   Method* methodInfo = OverloadingResolution(_this→type,
   methodHash, params);
   if (methodInfo==0)
      throw new Exception();
   if (methodInfo→IsVirtual)
   {
```

```
        methodAddress = _this→vft[methodInfo];
    }
    else
    {
        methodAddress = methodInfo→address;
    }
    push(_this);
    for (int i=0;i<paramCount;++i)
    {
        push(params[i]);
    }
    call(methodAddress);
}
void* xMemberAccess(Instance* _this, unsigned long memberHash) { }
```

6. Future work

The fully development of the **X** Language, will illustrate this concept.

We also intend to describe the complete definition of type constraints that will allow the static type-checker to resolve some validations which are currently done by the runtime type-checker.

Also the intellisense of the development environment **X Language** will integrate in, would be developed so that it will work in the case of type-unbound variables and variable types.

7. References

[1] L. Cardelli and P. Wagner: *On Understanding Types, Data Abstraction and Polymorphism*, Technical Report CS-85-14, Brown University, Department of Computer Science, 1985

[2] Juval Lowy: *An Introduction to C# Generics*, Visual Studio 2005 Technical Articles, 2005, http://msdn2.microsoft.com/en-us/library/ms379564(vs.80).aspx

[3] C. Strachey, Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967

[4] T.L. Thai, Hoang Lam - *.NET Framework Essentials*, O'Reilly Programming Series, 2001

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, 1 M. Kogălniceanu, Cluj-Napoca 400084, Romania

E-mail address: ci29836@scs.ubbcluj.ro

E-mail address: motogna@cs.ubbcluj.ro