Baltic J. Modern Computing, Vol. 1 (2013), No. 1, 77-100

Strongly typed metadata access in object oriented programming languages with reflection support

Mikus VANAGS [1,2], Arturs LICIS [2], Janis JUSTS [2]

[1] Latvia University of Agriculture, 2 Liela Street, Jelgava, LV-3001, Latvia
[2] Logics Research Centre, Sterstu street 7-6, Riga, LV-1004, Latvia

mikus.vanags@logicsresearchcentre.com,
arturs.licis@logicsresearchcentre.com,
janis.justs@logicsresearchcentre.com

Abstract. Type safety is important property of any type system. Modern programming languages support different mechanisms to work in type safe manner, e.g., properties, methods, events, attributes (annotations) and other structures, but none of the existing, general purpose, programming languages which support reflection provide type safe type (class/structure) member metadata access. Existing solutions provide no or limited type safety which are complex and processed at runtime which by definition is not built-in type-safe metadata access, but only more or less type safe workarounds called "best practices". Problem can be solved by introducing methods for type safe type member metadata access.

Keywords: programming language syntax, type safety, metadata, reflection.

1. Background

Reflection is a powerful mechanism provided by some major object oriented languages and frameworks that allows to access information about classes and their members at metadata level and use it in different scenarios. A few use cases are: detecting what kind of methods or fields does the class have, detecting the specific field data type at runtime, dynamically invoking methods with their names unknown at the compile time. These are non-standard cases of classical object oriented programming, but they are significant to contemporary object oriented design solutions and frameworks where modularity and extensibility are key values (Demers et.al., 1995).

The way reflection is designed, metadata access is not straightforward and type-safe for distinct members. The usual scenarios of using reflection allows: (a) traversing class fields or members, and doing the processing operations on each iteration; (b) checking if a specific member (field, method, constructor, event, property, etc.) with a specified name (and possible additional signature information for the latter) exists, and then processing it. This approach does not allow direct and type-safe access of distinct fields or methods the programmer is aware of (WEB, c), (Forman et.al., 2004), (WEB, d), (Skeet, 2011).

The following C# example demonstrates getting metadata information about the field. The example class 'Person' has an instance level field named 'FullName' and a static field 'TotalPersons':

```
//C#
public class Person
{
    public string FullName;
    public static int TotalPersons;
    public Person(string fullName)
    {
        this.FullName = fullName;
    }
    public void DoSomething() { }
    public void DoSomething(string arg) { }
    public int DoSomething(string arg1, double arg2)
    {
        return 0;
    }
    public event EventHandler<EventArgs> SomeEvent;
}
//Accessing type metadata:
Type personType = typeof (Person);
//Accessing instance field metadata:
FieldInfo instanceMemberMetadata = personType.GetField("FullName");
//Accessing static field metadata:
FieldInfo staticMemberMetadata =
    personType.GetField("TotalPersons");
//Accessing metadata for method with one string parameter:
MethodInfo methodMetadata = personType.GetMethod("DoSomething",
    new Type[] { typeof(string) });
//Accessing metadata for type constructor with one string parameter:
ConstructorInfo constructorMetadata = personType.GetConstructor(
    new Type[] { typeof(string) });
//Accessing metadata for event:
EventInfo eventMetadata = personType.GetEvent("SomeEvent");
```

The following code snippet below demonstrates literally the same example in Java:

```
public class Person {
      public String fullName;
      public static int totalPersons;

      public Person(String fullName) {
            this.fullName = fullName;
      }

      public void doSomething() { }
      public void doSomething(String arg) { }
      public int doSomething(String arg1, double arg2) {
            return 0;
      }
}
```

Java[1] also allows accessing both private and public fields. In case of Java there is a special case of retrieving class metadata by using "class" keyword as a member of class (consider Person.class example):

```
Class<Person> personClass = Person.class;
//Accessing instance field metadata:
Field instanceMemberMetadata =
personClass.getField("fullName");
//Accessing static field metadata:
Field staticMemberMetadata =
personClass.getField("totalPersons");
//Accessing metadata for method with one string parameter:
Method doSomethingArg =
personClass.getMethod("doSomething", String.class);
//Accessing metadata for type constructor with one string
//parameter:
Constructor<Person> constructorMetadata =
personClass.getConstructor(String.class);
```

Previously demonstrated examples show the existing technique of accessing metadata in two major general purpose programming languages: C# and Java. Providing type member name as string instances to access type member metadata is not type safe. It means that code is not reliable for maintenance (refactoring) and also that a mistake, if there is any, will be noticed only at runtime.

The most obvious benefit of static type-checking is that it allows early detection of some programming errors. Errors that are detected early can be fixed immediately, rather than lurking in the code to be discovered much later, when the programmer is in the middle of something else or even after the program has been deployed. Most of the general purpose object-oriented programming languages are strongly typed, but none of them provide fully type safe metadata access mechanism – they lack type safe type (class/structure) member metadata access. Not all programming languages support metadata access mechanisms like reflection, but for languages which support metadata access, type safety in this field is considered to be a property of the particular computer program rather than of the programming language used in the respective program. In such cases programmer is responsible for type safety, namely, correct metadata representation in basic data types, usually strings.

---

[1] Please note that Java has some differences in metadata retrieval and metadata structure (e.g., Constructor has a single generic parameter referencing to a constructor holder class), but conceptual approach is almost identical to that of C#.

2. Existing techniques for type safe metadata access

The simplest way of demonstrating importance of type safe metadata access is to try different approaches in implementing MVVM design pattern (Smith, 2009). In this chapter we will implement only ViewModel part of MVVM and focus on member metadata access issues.

2.1. ViewModel example without type safety

Here is ViewModel declaration example typing member name in string data type – unsafe way[2] (Smith, 2009):

```
//C#
//base class is simple class without generic parameter
public class CustomerViewModel: ViewModelBase
{
    private readonly Customer _model;

    public CustomerViewModel(Customer model)
    {
        _model = model;
    }

    public string FullName
    {
        get { return _model.FullName; }
        set
        {
            if (_model.FullName!= value)
            {
                _model.FullName = value;
                //Not type safe form,
                //property processed at compile time
                OnPropertyChanged("FullName");
            }
        }
    }
}
```

Member metadata access is not type safe, but it is performed at compile time and performs fast.

[2] This example is intended to compare it with improvements added in next chapters.

2.2. Metadata access at runtime using expression trees

The best programmer can do if programming language does not have strongly typed type member metadata access is checking type member names (metadata) at runtime which only partly solves type safety issues, but on downside makes code more complex, forcing programmer to use redundant type expressions and leading to performance slowdown. Microsoft provides best practices to access metadata in mentioned type safe way using lambda expressions (Rusina, 2010), (Migliore, a).

```
//C#
//Base class generic parameter specification contains redundant
//information. In ideal case this information should be known from
//context
public class CustomerViewModel : ViewModelBase<CustomerViewModel>
{
    private readonly Customer _model;
    public CustomerViewModel(Customer model)
    {
        _model = model;
    }
    public string FullName
    {
        get { return _model.FullName; }
        set
        {
            if (_model.FullName != value)
            {
                _model.FullName = value;
                //type safe, but confusing syntax and expression
                //tree processing could take significant time
                OnPropertyChanged(x => x.FullName);
            }
        }
    }
}
```

Unfortunately, the practice described above does not guarantee 100% type safety and desired result. In case if programmer provides lambda expression without member access expression, program execution will fail.

2.3. Metadata access at compile time using CallerInfo attributes

From C# 5, it is possible to tag optional parameters with one of three caller info attributes (WEB, b), (Albahari et.al., 2012):

[CallerMemberName] applies the caller's member name;

[CallerFilePath] applies the path to caller's source code file;

[CallerLineNumber] applies the line number in caller's source code file.

By using CallerInfo attributes, it is possible to obtain information about the caller to a method. You can obtain file path of the source code, the line number in the source code, and the member name of the caller. CallerInfo attributes instruct the compiler to

feed information obtained from the caller's source code into the parameter's default value: This information is helpful for tracing, debugging, and creating diagnostic tools.

Here is a modified C# example from chapter 2.1 by using CallerInfo attributes:

```
public class CustomerViewModel : INotifyPropertyChanged
{
    private readonly Customer _model;

    public event PropertyChangedEventHandler PropertyChanged =
delegate { };

    private void RaisePropertyChanged(
        [CallerMemberName] string propertyName = null)
    {
        Console.WriteLine(propertyName);
        PropertyChanged(this,
            new PropertyChangedEventArgs(propertyName));
    }

    public CustomerViewModel(Customer model)
    {
        _model = model;
    }

    public string FullName
    {
        get { return _model.FullName; }
        set
        {
            if (_model.FullName != value)
            {
                _model.FullName = value;
                //Type safe form, processed at compile time
                RaisePropertyChanged();
                // The compiler converts the above line to:
                // RaisePropertyChanged ("FullName");
            }
        }
    }
}
```

CallerInfo attributes makes metadata access more type safe, metadata access is performed at compile time and it works much faster than processing expression trees at runtime. However, CallerInfo attributes are not a generic solution. CallerInfo attributes are suitable for narrow use cases, generally to safely access property name from inside the property. CallerInfo attributes are not applicable for data querying and metadata processing in other use cases.

All problems mentioned in chapter 2 can be solved by introducing language improvements to support type safe metadata access which would make compile-time checks possible (WEB, e).

3. Type safe member metadata access

Type metadata gathering operator typeof (C#) and ".class" call (Java) returns metadata about specified type (class, structure, interface), but there is no type safe way to access class member metadata. For example, programming languages can be extended with operator named 'memberof' so that memberof(classField) returns field metadata instance FieldInfo (C#) or Field (Java) instances instead of field value instances. Microsoft Corporation was first who introduced idea of such conception for member metadata access and they called their member metadata access operator: 'infoof' (Lippert, 2009). If Microsoft could rename 'typeof' to 'infoof' then name 'infoof' would be better choice for such operator[3]. As this paper focuses on the type safe metadata access idea rather than readability problems, further we will prefer using 'memberof'.

Microsoft has been thinking about operator infoof as similar operator to operator typeof which allows type safe metadata access:

```
Type info = typeof (int);
```

Operator typeof accepts parameter which is type instead of instance. Following example is invalid according to C# specification:

```
Type x = typeof(6);
```

Use cases of member metadata access operator, that does not accept instance expressions as operator parameter, are too specific and that is not enough for fully type safe member metadata access.

Operator memberof in C#

Here is example of instance creation needed for further examples:

```
var myFriend = new Person("Oscar");
```

Instance member metadata could be accessed using object instance:

```
FieldInfo instanceMemberMetadata = memberof(myFriend.FullName);
```

Static member metadata could be accessed using type information:

```
FieldInfo staticMemberMetadata = memberof(Person.TotalPersons);
```

Operator memberof in Java

```
Person myFriend = new Person("Oscar");
```

Instance member metadata could be accessed using object instance:

```
Field instanceMemberMetadata = memberof(myFriend.fullName);
```

Static member metadata could be accessed using class:

```
Field instanceMemberMetadata = memberof(Person.totalPersons);
```

---

[3] It would be essential to reduce number of standard keywords in programming language. Otherwise overloaded member metadata access operators require at least 5 or even more new keywords which significantly rises complexity of programming language.

The most important aspect of operator ‘memberof’ is that member access or member call expressions provided to operator ‘memberof’ are not processed as member access operations or member call operations, but instead the metadata instance of supplied member (field, method, property, constructor, event) is created and returned. This means that code from previous example - memberof(myFriend.FullName) - is not reducible to memberof(“Oscar”) as it would be if field myFriend.FullName would be interpreted as field access operation.

In Frameworks operator ‘memberof‘ could be overloaded with following versions:

.NET

- memberof(field) which should return FieldInfo instance;
- memberof(method(parameter type list – optional)) which should return MethodInfo instance;
- memberof(property) which should return PropertyInfo instance;
- memberof(class(parameter type list – optional)) which should return ConstructorInfo instance;
- memberof(event) which should return EventInfo instance.

In .NET class MemberInfo is base class for classes: FieldInfo, MethodInfo, PropertyInfo, ConstructorInfo, EventInfo. Therefore in cases when only type safe member name determination is needed, the use of MemberInfo instance would be more appropriate. Similar metadata type system architecture is used in programming language Java, only metadata class names and usage syntax differs:

Java

- memberof(field_path) – returns Field instance for a given field (expressed with a full path, e.g. myFriend.fullName or Person.fullName);
- memberof(method_path(Class<?> … parameterTypes)) – returns Method instance;
- memberof(class_name(Class<?> … parameterType)) – returns Constructor instance representing constructor information for the given class.

C# examples

```
//accessing member when having source object
var somebody = new Person("Anonymous");
MemberInfo memberMetadata = memberof(somebody.FullName);
//or any other member

//accessing field without having source object
FieldInfo fieldMetadata = memberof(Person.FullName);

//accessing property without having source object
PropertyInfo propertyMetadata =
    memberof(CustomerViewModel.FullName);
```

```
//accessing parameter less constructor metadata
ConstructorInfo constructorMetadata = memberof(CustomerViewModel());

//accessing metadata from constructor with one string parameter
ConstructorInfo constructorMetadata = memberof(Person(string));

//accessing parameterless method metadata without having source
//object
MethodInfo methodMetadata = memberof(Person.DoSomething());

//accessing metadata from method with two parameters: string, double
MethodInfo methodMetadata
    = memberof(Person.DoSomething(string, double));

//accessing metadata from event
EventInfo eventMetadata = memberof(Person.SomeEvent);
```

Java examples[4]

```
Person somebody = new Person("Anonymous");
//AccessibleObject - superclass for Constructor, Method,
Field AccessibleObject memberMetadata
    = memberof(somebody.fullName);

Field fieldMetadata = memberof(Person.fullName);

Constructor<CustomerViewModel> constructorMetadata
    = memberof(CustomerViewModel());

Constructor<Person> constructorMetadata
    = memberof(Person(String.class));

Method methodMetadata = memberof(Person.doSomething());

Method methodMetadata
= memberof(Person.doSomething(String.class, double.class));
```

[4] In order to maintain consistency with existing Java language specification, meta-data class Constructor provides a single generic parameter which refers to a parent class of a constructor (just as described in examples above when metadata was retrieved using standard solution).

Use case of operator ‘memberof’

One of the most valuable use cases of operator ‘memberof’ could be in design pattern MVVM ViewModel declarations. Here is more type safe version of class CustomerViewModel that was introduced in section 2.1.[5]:

```
//C#
//base class is simple class without generic parameter
public class CustomerViewModel : ViewModelBase
{
    private readonly Customer _model;
    public CustomerViewModel(Customer model)
    {
        _model = model;
    }
    public string FullName
    {
        get { return _model.FullName; }
        set
        {
            if (_model.FullName != value)
            {
                _model.FullName = value;
                //optimal type safety, but member access code
                //doesn’t use available context information
                MemberInfo info = memberof(this.FullName);
                OnPropertyChanged(info.Name);
            }
        }
    }
}
```

Context dependent type member metadata access

Many programming languages have operator ‘this’ which points to current instance context, but none of programming languages have operator which could point to current instance member context. Such operator could be operator named ‘member’:

```
//C#
//base class is simple class without generic parameter
public class CustomerViewModel : ViewModelBase
{
    private readonly Customer _model;
    public CustomerViewModel(Customer model)
    {
        _model = model;
    }
```

[5] Type safety is achieved by using operator ‘memberof’.

```
    public string CustomerName
    {
        get { return _model.FullName; }
        set
        {
            if (_model.FullName != value)
            {
                _model.FullName = value;
                //optimal type safety,
                //context dependent member metadata access,
                //clean code,
                //'member' returns PropertyInfo instance
                MemberInfo info = member;
                OnPropertyChanged(info.Name);
            }
        }
    }
}
```

Operator ‘member’ depends on usage context. When used in constructor code block it should return constructor metadata instance, when used in method code block it should return method metadata instance, when used in property code blocks it should return property metadata instance. The difference from previous example is that now we are using context dependent operator which will be handled by compiler, enabling easier refactoring. For example, in renaming property changes inside property declaration (including body) are required to be made only in one place – in property name.

4. Detecting type of member from member access expressions

Sometimes it is required not only to access type member metadata, but also to process type member taking into consideration a(some) parameter(s) whose type should be compatible with initial member type. For example, in database querying method ‘FilterByEquality’ declared as follows could be useful:

```
public IEnumerable<object> FilterByEquality (MemberInfo
memberMetaData, object constrainedValue) {…}
```

An example described above is not type safe because type of parameter ‘constrainedValue’ may not be compatible with type of member to which parameter ‘memberMetaData’ indirectly points to. To solve such problems we propose extending metadata types to generic versions:

C#

MemberInfo<T> should replace MemberInfo
FieldInfo<T> should replace FieldInfo
MethodInfo<T> should replace MethodInfo
PropertyInfo<T> should replace PropertyInfo
ConstructorInfo<T> should replace ConstructorInfo
EventInfo<T> should replace EventInfo

Java[6]

AccessibleObject<T> should replace AccessibleObject
Field<T> should replace Field
Method<T> should replace Method
Constructor<C, T> should replace Constructor<C>
Now operator 'memberof' should return generic versions of metadata instances.

NET

- memberof(suppliedField) which should return FieldInfo<T> instance and T is type of field 'suppliedField';
- memberof(suppliedMethod(parameter type list – optional)) which should return MethodInfo<T> instance and type T is type container for all 'suppliedMethod 'parameters. In .NET this type container could be standard delegate: Func<…> or Action<…>. Which type exactly will function as type container depends on 'suppliedMethod' returning type – for methods with returning value type container will be Func<…> and for methods without returning value (void) type container will be Action<…>. It is possible to use custom type container types, but it would be safer to choose well-known standard classes, this way it would be possible to use implicit type declarations (keyword 'var');
- memberof(suppliedProperty) which should return PropertyInfo<T> instance and T is type of property 'suppliedProperty';
- memberof(suppliedClass(parameter type list – optional)) which should return ConstructorInfo<T> instance and type T is type container for all 'suppliedClass' constructor parameters. In .NET this type container could be standard delegate Action<…>;
- memberof(event) which should return EventInfo<T> instance where type T should specify event argument (in should be class EventArgs or class which inherits from EventArgs);

C# examples

```
//accessing member when having source object
var somebody = new Person("Anonymous");
MemberInfo<string> memberMetadata = memberof(somebody.FullName);

//accessing field without having source object
FieldInfo<string> fieldMetadata = memberof(Person.FullName);

//accessing property without having source object
PropertyInfo<string> propertyMetadata
```

[6] Java metadata class Constructor already had a single type-parameter (referencing to a constructor's holder class), and thus we are extending it by an additional generic parameter.

```
    = memberof(CustomerViewModel.FullName);

//accessing parameter less constructor metadata
ConstructorInfo<Action> constructorMetadata
    = memberof(CustomerViewModel());

//accessing metadata from constructor with one string parameter
ConstructorInfo<Action<String>> constructorMetadata
    = memberof(Person(string));

//accessing parameterless method metadata without having source
//object
MethodInfo<Action> methodMetadata = memberof(Person.DoSomething());

//accessing metadata from method with one parameter of type string
MethodInfo<Action<string>> methodMetadata
    = memberof(Person.DoSomething(string));

/*accessing metadata from method with two parameters: first -
string, second - double and returning value of type - int */
MethodInfo<Func<string, double, int>> methodMetadata
    = memberof(Person.DoSomething(string, double));

//accessing metadata from event
EventInfo<EventArgs> eventMetadata = memberof(Person.SomeEvent);
```

Java examples

```
Person somebody = new Person("Anonymous");
//AccessibleObject - superclass for Constructor, Method,
Field AccessibleObject<String> memberMetadata
    = memberof(somebody.fullName);

//accessing field without having source object
Field<String> fieldMetadata = memberof(Person.fullName);

//accessing parameter less constructor metadata
Constructor<CustomerViewModel, Action> constructorMetadata
    = memberof(CustomerViewModel());

//accessing metadata from constructor with one string
//parameter
Constructor<Person, Action1<String>> constructorMetadata
    = memberof(Person(String.class));


//accessing parameterless method metadata without having
//source object
```

```
Method<Action> methodMetadata
    = memberof(Person.doSomething());

//accessing metadata from method with one parameter of type
//string
Method<Action1<string>> methodMetadata
    = memberof(Person.doSomething(String));

/*accessing metadata from method with two parameters: first
- string, second - double and returning value of type - int
*/
Method<Func2<String, double, int> methodMetadata =
memberof(Person.doSomething(String.class, double.class));
```

In programming languages which do not support delegates, programmer needs to take care of designing type containers for method parameters. Type 'Action' variations are supposed to function as method parameter type containers for methods which do not have returning value (void methods). Type 'Action' is supposed to describe fact that method does not have parameters; Action1<T1> is supposed to describe fact that method has one parameter which type should equal to generic parameter T1; Action2<T1, T2> is supposed to describe fact that method has two parameters which types should be equal to generic parameters T1 and T2 accordingly etc.

Variations of 'Func' are created similarly.. 'Func' acts as method parameter type container for methods which return value. Func<R> is supposed to describe fact that method does not have parameters and type of returning value should be equal to generic parameter R. Func1<T1, R> is supposed to describe fact that method has one parameter of type which should be equal to generic parameter T1 and type of returning value should be equal to generic parameter R. Func2<T1, T2, R> is supposed to describe fact that method has two parameters of types which should be equal to generic parameters T1 and T2 accordingly and type of returning value should be equal to generic parameter R.

It should be noted that the final names of 'Action' and 'Func' type containers can be changed depending on a target framework. For example C# already provides such types along with delegates, and it allows using the same name across different versions (different generic parameters' count designates different types). However, in case of Java generics are processed differently, and different type names are required even if generics declaration differs. The Action, Action1, Action2, … and Func, Func1, Func2, … could be introduced in Java as a metadata parameter holders.

Taking into consideration all previously proposed ideas, method 'FilterByEquality' example can be improved as follows:

```
public IEnumerable<object> FilterByEquality <T>(MemberInfo<T>
memberMetaData, T constrainedValue) {…}
```

```
//C# usage example
string personName = "John Doe";
var memberMetadata = memberof(Person.FullName);
IEnumerable<object> wantedPersons
    = FilterByEquality(memberMetadata, personName);
```

Important part is included in expression: memberof(Person.FullName) which returns FieldInfo<T> instance where type T is determined as string. Compiler automatically detects type of variable 'memberMetadata' from 'memberof' operator call context and in example this type is MemberInfo<string>. Demonstrated example of 'memberof' call is equivalent to following code where returning type is explicitly declared:

```
MemberInfo<string> memberMetadata = memberof(Person.FullName);
```

5. Detecting member type and member containing type from member access expressions

Method's 'FilterByEquality' example still is not fully type safe, because returning collection items type is not detected from provided metadata. Problem can be solved by extending metadata containing types with one more generic parameter which will hold member containing object's type information.

This means: MemberInfo<T> extension to MemberInfo<TObject, TMember> where TMember refers to member's type and TObject refers to members containing object type:

C#

MemberInfo<TObject, TMember> should replace MemberInfo
FieldInfo<TObject, TMember> should replace FieldInfo
MethodInfo<TObject, TMember> should replace MethodInfo
PropertyInfo<TObject, TMember> should replace PropertyInfo
ConstructorInfo<TObject, TMember> should replace ConstructorInfo
EventInfo<TObject, TMember> should replace EventInfo

Java

Member< TObject, TMember> should replace Member
Field< TObject, TMember> should replace Field
Method< TObject, TMember> should replace Method
Constructor< TObject, TMember> should replace Constructor

C# examples

```
//accessing member when having source object
var somebody = new Person("Anonymous");
MemberInfo<Person, string> memberMetadata
    = memberof(somebody.FullName);
```

```
//accessing field without having source object
FieldInfo<Person, string> fieldMetadata = memberof(Person.FullName);

//accessing property without having source object
PropertyInfo<CustomerViewModel, string> propertyMetadata
    = memberof(CustomerViewModel.FullName);

//accessing parameter less constructor metadata
ConstructorInfo<CustomerViewModel, Action> constructorMetadata
    = memberof(CustomerViewModel());

//accessing metadata from constructor with one string parameter
ConstructorInfo<Person, Action<String>> constructorMetadata
    = memberof(Person(string));

//accessing parameter less method metadata without having source
//object
MethodInfo<Person, Action> methodMetadata
    = memberof(Person.DoSomething());

//accessing metadata from method with one parameter of type string
MethodInfo<Person, Action<string>> methodMetadata
    = memberof(Person.DoSomething(string));

/*accessing metadata from method with two parameters: first -
string, second - double and returning value of type - int */
MethodInfo<Person, Func<string, double, int>> methodMetadata
    = memberof(Person.DoSomething(string, double));

//accessing metadata from event
EventInfo<Person, EventArgs> eventMetadata
    = memberof(Person.SomeEvent);
```

Java examples

```
Person somebody = new Person("Anonymous");
//AccessibleObject-superclass for Constructor,Method,Field
AccessibleObject<Person, String> memberMetadata
    = memberof(somebody.fullName);

//accessing field without having source object
Field<Person, String> fieldMetadata
    = memberof(Person.fullName);

//accessing parameterless constructor metadata
Constructor<CustomerViewModel, Action> constructorMetadata
    = memberof(CustomerViewModel());

//accessing metadata from constructor with one string
```

```
//parameter
Constructor<Person, Action1<String>> constructorMetadata
    = memberof(Person(String.class));

//accessing parameterless method metadata without having
//source object
Method<Person, Action> methodMetadata
    = memberof(Person.doSomething());

//accessing metadata from method with one parameter of type
//string
Method<Person, Action1<string>> methodMetadata
    = memberof(Person.doSomething(String));

/*accessing metadata from method with two parameters: first
- string, second - double and returning value of type - int
*/
Method<Person, Func2<String, double, int> methodMetadata
= memberof(Person.doSomething(String.class, double.class));
```

Taking into consideration previously described improvements to operator 'memberof', example with method 'FilterByEquality' can be declared as follows:

```
public IEnumerable<TObject> FilterByEquality<TObject, TMember>(
    MemberInfo<TObject, TMember> memberMetaData,
    TMember constrainedValue) {…}

//C# usage example
string personName = "John Doe";
MemberInfo<Person, string> memberMetadata
    = memberof(Person.FullName);
IEnumerable<Person> wantedPersons
    = FilterByEquality(memberMetadata, personName);
```

Last code line from previous example can be rewritten to use implicit type declaration:

```
var wantedPersons = FilterByEquality(memberMetadata, personName);
```

6. Multiple level member access expressions

It is possible that member access expression is invoked from existing member access expression. Consider example class declarations:

```
class Person
{
    public Address HomeAddress;
}
class Address
{
    public string Street;
}
```

Example of Multiple level member access expression with two level member accesses will look like this:

```
var instance = new Person();
instance.HomeAddress = new Address();
instance.HomeAddress.Street = "My street number 6";
var memberMetadata = memberof(instance.HomeAddress.Street);
```

In case of two level member access expression (in example: instance.HomeAddress.Street) type of operator 'memberof' returning value should be member containing type from first member access expression. In example described above first member access expression is 'HomeAddress' member access expression and its containing type is Person, so previous example can be rewritten without implicit type declaration as follows:

```
MemberInfo<Person, string> memberMetadata
    = memberof(instance.HomeAddress.Street);
```

For multiple member access expressions to be useful as metadata, compiler should maintain whole chain of member access expressions. In previous example it means that variable 'memberMetadata' represents 'Street' member access expression and contains information that member 'Street' was accessed from 'HomeAddress' which is another member access expression. Member 'HomeAddress' was accessed from instance (not from another member access expressions), therefore here stops member access chain backtracking.

If 'Address' is needed as returning type and we have only 'Person' instance, then multiple member access expression should be separated as follows:

```
var homeAddress = instance.HomeAddress;
MemberInfo<Address, string> memberMetadata
    = memberof(homeAddress.Street);
```

Such multiple member access level behaviour of 'memberof' operator would be useful in defining queries.

7. Passing metadata to methods

Metadata can be gathered and then passed to methods, like in the following example:

```
//method declaration
void TestMetadata<T, TProp>(MemberInfo<T, TProp>
    memberAccessExpression) {…}
//method call
TestMetadata(memberof(Person.FullName));
```

In cases when metadata needs to be passed to method as parameter, 'memberof' operator syntax can be transformed into another syntax using method parameter modifier called 'meta':

```
//method call
TestMetadata(meta Person.FullName);
```

Method parameter modifier 'meta' forces compiler to interpret method actual parameter as metadata access expression instead of value access expression what is default behaviour in method parameter interpretation.

Reference parameters and output parameters change not only how method accepts parameter, but also the way how method processes parameters. For this reason reference parameters and output parameters require parameter modifier usage at method declaration. Method parameter modifier 'meta' makes changes only in actual value passed to method call. Parameter modifier 'meta' does not impact method execution, so method parameter modifier 'meta' specifying at method declaration is not necessary.

Method parameter modifier 'meta' has 5 different forms:

1) parameter modifier for field metadata access, example:

```
//TestMetadata1 method declaration
void TestMetadata1<T, TField>(FieldInfo<T, TField>
memberAccessExpr) {…}
//TestMetadata1 method call providing field from class Person
TestMetadata1(meta Person.FullName);
```

2) parameter modifier for property metadata access, example:

```
//TestMetadata2 method declaration
void TestMetadata2<T, TProp>(PropertyInfo<T, TProp>
memberAccessExpr) {…}
//TestMetadata2 method call providing property from class
//CustomerViewModel
TestMetadata3 (meta CustomerViewModel.FullName);
```

3) parameter modifier for method metadata access, example:

```
//TestMetadata3 method declaration
void TestMetadata3<T, TMet>(MethodInfo<T, TMet>
memberAccessExpr) {…}
//TestMetadata3 method call providing other method from class
//Person
TestMetadata3 (meta Person.DoSomething(string, double));
```

4) `TestMetadata4 (meta Person(string));`parameter modifier for constructor metadata access, example:

```
//TestMetadata4 method declaration
void TestMetadata4<T, TCon>(ConstructorInfo<T, TCon>
memberAccessExpr) {…}
//TestMetadata4 method call providing constructor of class
//Person
```

5) parameter modifier for event metadata access, example:

```
//TestMetadata5 method declaration
void TestMetadata5<T, TEventArgs>(EventInfo<T, TEventArgs>
memberAccessExpr) {…}
//TestMetadata5 method call providing event declared in class
//Person
TestMetadata5 (meta Person.SomeEvent);
```

Most benefits from method parameter modifier ‘meta’ usage can be gained in frameworks where reflection is used as architectural discipline, especially in frameworks supporting MVC architectural pattern where views usually are linked with models using binding mechanism which uses reflection. The following example demonstrates View designed in ASP.NET MVC Razor View Engine (Palermo et.al., 2012); HTML helper ‘TextBox’ accepts metadata in type unsafe way:

```
@using (Html.BeginForm())
{
    <p>Your name: @Html.TextBox("FullName")</p>
    <input type="submit" value="Go" />
}
```

Best that is possible without operator ‘memberof’ invention is usage of lambda expressions:

```
model LrcSite.Models.Person

@using (Html.BeginForm())
{
    <p>Your name: @Html.TextBoxFor(model => model.FullName)</p>
    <input type="submit" value="Go" />
}
```

Example view is defined as strongly typed, namely, variable ‘Html’ is of type HtmlHelper<Person> and that is why HTML helper ‘TextBoxFor’ can accept member ‘Person.FullName’ metadata in strongly typed way. But, as lambda expressions are processed at runtime, they are not fully type safe. Besides lambda expressions syntax in HTML helper case requires declaration of formal parameter (in the previous example it is parameter named ‘model’) which is unnecessary from syntax perspective and should be removed to simplify syntax.

The following example demonstrates simple HTML helper ‘TextBoxFor’ accepting member metadata instance:

```
public static MvcHtmlString TextBoxFor<T, TProp>(this HtmlHelper<T>
    html, MemberInfo<T, TProp> memberAccess)
{
    var tag = new TagBuilder("input");
    tag.MergeAttribute("name", memberAccess.Name);
    tag.MergeAttribute("type", "text");

    ModelState modelState;
    html.ViewData.ModelState.TryGetValue(memberAccess.Name,
        out modelState);
    var value = modelState != null && modelState.Value != null
        ? modelState.Value.ConvertTo(typeof(TProp)): default(TProp);

    tag.MergeAttribute("value", Convert.ToString(value));
    return MvcHtmlString.Create(tag.ToString());
}
```

Example demonstrates how metadata from 'memberAccess' expression is gathered during compile time and syntax does not contain any unnecessary or redundant parts. Improved HTML helper calling code is demonstrated in following ASP.NET MVC Razor view example:

```
@model LrcSite.Models.Person

@using (Html.BeginForm())
{
    <p>Your name: @Html.TextBoxFor(meta Person.FullName)</p>
    <input type="submit" value="Go" />
}
```

However, it still can be simplified. In case of strongly typed view, member containing type specification in member access expression is redundant. Here is simplified, but equivalent code sample to previously declared ASP.NET MVC Razor view example:

```
//declaration of variable Html
public HtmlHelper<Person> Html;
…
//HTML helper TextBoxFor usage with method parameter modifier 'meta'
Html.TextBoxFor(meta Person.FullName);
```

HTML helper usage example can be rewritten specifying generic parameters explicitly as follows:

```
Html.TextBoxFor<Person, string>(meta Person.FullName);
```

Now can be seen, that generic parameter 'T' (type 'Person') in HTML helper 'TextBoxFor' call is used in 3 places: in 'Html' variable declaration, in 'TextBoxFor' method call and in member access expression. Compiler uses type inference to detect unknown generic types and for compiler it is sufficient to supply type for generic

parameter only in one place instead of all tree places. In example, place where generic parameter T type is specified is in variable 'Html' declaration, so further generic parameter T specifications are not necessary. In similar way compiler is capable to infer type of generic parameter 'TProp' from member access expression 'Person.FullName', so the shortest syntax of HTML helper 'TextBoxFor' usage would be as follows:

```
Html.TextBoxFor(meta FullName);
```

Finally, type member metadata access and usage syntax in all aspects are short, expressive and fully type safe. Here comes example of shortest syntax form for method modifier 'meta' demonstrating how metadata should be provided to HTML helpers in ASP.NET MVC Razor views engine:

```
@model LrcSite.Models.Person

@using (Html.BeginForm())
{
    <p>Your name: @Html.TextBoxFor(meta FullName)</p>
    <input type="submit" value="Go" />
}
```

The only part that is not yet covered is method modifier 'meta' for whole types (member containers). If method parameter modifier 'meta' works with type members, it should work with types as well. The following example demonstrates method parameter 'meta' usage syntax with types:

```
//method declaration
void DoSomething(Type someTypeFormalParameter) {...}
//method call
DoSomething(meta Person);
```

Such practice is equivalent to following code usage pattern:

```
//method declaration
void DoSomething(Type someTypeFormalParameter) {...}
//method call
DoSomething(typeof(Person));
```

But in case of method modifier 'meta' syntax is much simpler and nicer.

8. Summary

Current versions of general purpose programming languages provide poor type safety solutions. There are several workarounds, but they still need to be improved. Several years ago Microsoft Corporation was close to idea about member metadata access in form of 'infoof' operator, but in latest .NET releases they have chosen to implement expression trees and later CallerInfo attributes which do not offer 100% type safety in all use cases where metadata can be involved. In this paper idea about type safe member

metadata access is extended to cover different forms of new operator: 'memberof'[7], generic forms of operator 'memberof', context dependent operator 'member' and method parameter modifier 'meta' which forces compiler to interpret method actual parameter as metadata access expression instead of value access expression. Introduced operators require changes in programming frameworks like .NET, Java and others, as well as propose improvements in syntax of general purpose programming languages resulting in fully type safe metadata access in field of programming languages.

References

Albahari J., Albahari B. (2012). C# 5.0 in a Nutshell, 5th Edition, The Definitive Reference. O'Reilly Media

Demers F.-N., Malenfant J (1995). Reflection in logic, functional and object-oriented programming: a Short Comparative Study. In:Proc. of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI. Montreal (1995), 29–38.

Forman I.R., Forman N. (2004). Java Reflection in Action. Manning Publications

Lippert E. (2009). In Foof We Trust: A Dialogue,
http://blogs.msdn.com/b/ericlippert/archive/2009/05/21/in-foof-we-trust-a-dialogue.aspx

Migliore M. (a). How to implement MVVM (Model-View-ViewModel) in TDD (Test Driven Development),
http://code.msdn.microsoft.com/How-to-implement-MVVM-71a65441

Palermo J., Bogard J, Hexter E., Hinze M., Skinner J. (2012). ASP.NET MVC 4 in Action. Manning, New York

Rusina A. (2010). Getting Information About Objects, Types, and Members with Expression Trees,
http://blogs.msdn.com/b/csharpfaq/archive/2010/01/06/getting-information-about-objects-types-and-members-with-expression-trees.aspx

Skeet J. (2011). C# in Depth Second Edition, Manning, Stamford

Smith J. (2009). WPF Apps With The Model-View-ViewModel Design Pattern,
http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

WEB (b). Caller Information (C# and Visual Basic),
http://msdn.microsoft.com/en-us/library/hh534540.aspx

WEB (c). C# Language Specification,
http://go.microsoft.com/fwlink/?LinkId=199552

WEB (d). Java Language and Virtual Machine Specifications,
http://docs.oracle.com/javase/specs/

WEB (e). Strongly typed metadata access,
http://logicsresearchcentre.com/MetadataAccess

[7] For idea description 'memberof' is more appropriate than 'infoof'.

Authors' information

M. Vanags is PhD student at the Faculty of Information Technologies, Latvia University of Agriculture, Latvia. He is cofounder of Logics Research Centre SIA, where he works as CTO. Most of his working time he spends to develop future programming technologies.

A. Licis, M.Sc., is an enthusiastic software engineer. His areas of interest include scalable systems and databases.

J. Justs is a PhD student at Institute of Materials and Structures, Faculty of Civil Engineering at Riga Technical University. He is cofounder of Logics Research Centre SIA. His areas of interest include information technologies and databases.

Received March 14, 2013, revised July 13, 2013, accepted July 30, 2013