Computer and Information Science; Vol. 10, No. 2; 2017
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

Type Safe Metadata Combining

Mikus Vanags[1,2] & Rudite Ceveree[2]

[1] sia Logics Research Centre, Riga, Latvia

[2] Latvia University of Agriculture, Jelgava, Latvia

Correspondence: Mikus Vanags, sia Logics Research Centre, Sterstu street 7-6, Riga, Latvia, LV 1004. Tel: 371-2667-3860. E-mail: mikus.vanags@logicsresearchcentre.com

Received: October 11, 2016 Accepted: January 21, 2017 Online Published: April 30, 2017

doi:10.5539/cis.v10n2p97 URL: http://doi.org/10.5539/cis.v10n2p97

Abstract

Type safety is an 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. Some programming languages allow access to metadata: type information, type member information and information about applied attributes. But none of the existing mainstream programming languages which support reflection provides fully type safe metadata combining mechanism built in the programming language. Combining of metadata means a class member metadata combining with data, type metadata and constraints. Existing solutions provide no or limited type safe metadata combining mechanism; they are complex and processed at runtime, which by definition is not built-in type-safe metadata combining. Problem can be solved by introducing syntax and methods for type safe metadata combining so that, metadata could be processed at compile time in a fully type safe way. Common metadata combining use cases are data abstraction layer creation and database querying.

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

1. Introduction

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 modern object oriented programming languages are strongly typed and many of them support reflection – mechanism to access metadata.

The following C# example demonstrates reflection (Hazzard & Bock, 2012) usage in getting metadata information about the field:

```
public class Person {
    public string FullName;
    ...
}
//Accessing type metadata:
Type personType = typeof(Person);
//Accessing instance field metadata using reflection in type unsafe way:
FieldInfo instanceMemberMetadata = personType.GetField("FullName");
```

Reflection is not type safe way to work with metadata, because a lot of configuration is done using strings instead of programming language constructions: types, fields, properties, methods, etc...

Programming language C# 6.0 supports operator 'nameof' (J. Albahari & B. Albahari, 2015) which returns construction metadata as string object. There are 2 fundamental problems with operator 'nameof':

It does not accept generic parameters constraining the accepting parameters and returning value types. Therefore, it does not fully provide a type safe way to work with metadata.

It returns only type name which is a small part of the available metadata that could be returned.

Both of mentioned problems have a common cause: operator ‘nameof’ returning type is ‘string’ instead of some specialized metadata class.

Object oriented programming languages which supports reflection, can be extended to support type safe metadata access using operator ‘memberof’ (Vanags at. al., 2013) which is an improved version of operator ‘nameof’:

```
//Accessing static field metadata using operator ‘memberof’ in type safe way:
FieldInfo<Person, string> memberMetadata = memberof(somebody.FullName);
```

To achieve a more concise syntax, the metadata accessing operator ‘memberof’ can be renamed to “meta”. Then, metadata accessing example could look as follows:

```
//Accessing static field metadata using operator ‘meta’ in type safe way:
FieldInfo<Person, string> memberMetadata = meta(somebody.FullName);
```

In addition to type safe metadata access, sometimes programmers need to organize metadata in some predefined way or combine metadata with data. For example, to store data in some relational database, it is needed to know a table name and the column names (in object oriented world they could be mapped to some class fields) and values. Thinking more generally, it would be useful to introduce property-value (key-value) abstraction which could contain necessary data to recognize table column (for relational database (James et.al., 2009)) or constructions used in NoSQL data stores (Harrison, 2015). Such property-value abstraction can be called property-constraint (Vanags at. al., 2013).

Going further, in relational databases data is stored in tables, thus querying result would be set of rows. Similarly in NoSQL data stores, data (key-value pairs) are stored in a structured way. For example, in objects, groups, and so on… Consequently, the general abstraction for querying data from some data source could be object set (the same as row set) abstraction which is called meta-set (Vanags at. al., 2013).

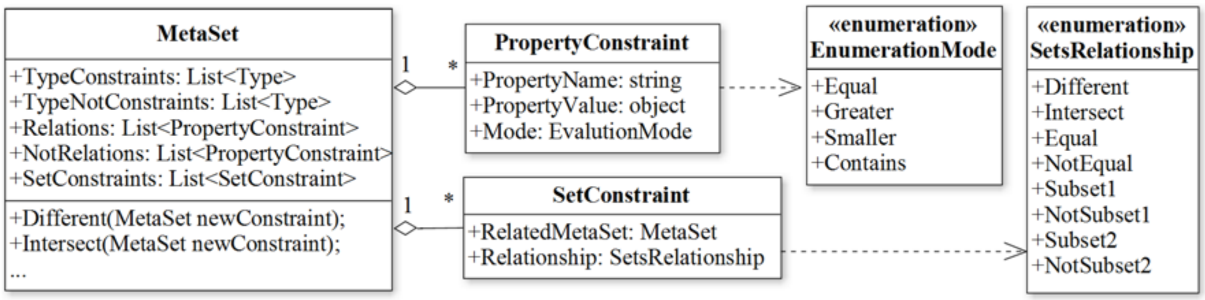

Figure 1. Meta-set physical structure represented in UML class diagram

Meta-sets can be used to support data querying. Ability to define meta-sets in general purpose programming languages means bringing abstract data querying language integration in general purpose programming languages. The aim of this paper is to cover data and metadata combining abstractions and to define them in type safe manner.

2. Metadata and Data Organization in Abstractions

Working with data requires information about data structure – metadata (NISO, 2010). Querying data from some data store requires not only metadata, but also constraint values (James et.al., 2009). Therefore, data querying combines metadata with data.

2.1 Property-Constraint Abstraction

Working with relational databases requires knowledge about needed tables and column names. Working with object data stores requires knowledge about types (classes) and class fields. Similar requirements exist for manipulations of other kind of NoSQL data. Thus, indivisible data element in data store querying should be a property-value (key-value) abstraction which is called “property-constraint”. In case of relational databases, the property part should point to appropriate column and the value part should constrain returning values from that column, but in case of NoSQL databases, property part should point to appropriate keys (in object database it would be appropriate class fields) and value part should constrain the value range which are expected in result.

Class member metadata can be interpreted as property part in the property-constraint data structure. For example, in .NET Framework, such class member metadata type is MemberInfo class. To form property-constraint instances in more concise way, the frameworks should support relational operators <,>,<>,==, != overriding in class member metadata type. Such relational operators as input can accept metadata instance (MemberInfo) and some value of corresponding type. Result of such relational operator is property-constraint instance. Here are some proposed signatures for operator overloading and new method introduction:

```
//C#
public partial class MemberInfo<TContainer, TMember> {
        public static PropertyConstraint operator ==
                (MemberInfo<TContainer, TMember> property, TMember value) {
                if (property != null) {
                        return new PropertyConstraint(property.Name, value);
                }
                return null;
        }
        public static PropertyConstraint operator !=
                (MemberInfo<TContainer, TMember> property, TMember value) {...}
        public static PropertyConstraint operator >
                (MemberInfo<TContainer, TMember> property, TMember value) {...}
        public static PropertyConstraint operator <
                (MemberInfo<TContainer, TMember> property, TMember value) {...}
        public static PropertyConstraint operator >=
                (MemberInfo<TContainer, TMember> property, TMember value) {...}
        public static PropertyConstraint operator <=
                (MemberInfo<TContainer, TMember> property, TMember value) {...}

        public PropertyConstraint Contains(TMember value) {...}
}
```

Having improved member metadata type, the type safe way to define the property-constraint structure is as follows:

```
MemberInfo<Person, int> ageProperty = meta(Person.Age);
PropertyConstraint youngerThan30 = ageProperty < 30;
```

Property-constraint can be defined in one line and it can take advantage of implicitly typed local variable ‘var’ as demonstrated in the following example:

```
var youngerThan30 = meta(Person.Age) < 30;
```

2.2 Meta-Sets Abstraction

Combining together one or more property-constraints with metadata - basic information about data structures (table name for relational databases, type name for object databases, etc.) leads to new abstraction definition – data query abstraction called “meta-set”. Meta-set is an object set abstraction – it describes a set of data records (rows, objects, etc., depending on data store architecture). Meta-set does not contain real objects, but it contains all necessary information to generate query to data store. Therefore, meta-set can be interpreted also as data query abstraction.

Meta-set physical structure (Vanags at. al., 2013) is used as basis for type safe meta-set declaration syntax:

```
var metaSetVariable = metaset
    [Type1 Selector1, Selector2,..., Type2 Selector3, Selector4,...]
    [not TypeNot1, TypeNot2...]
    [where constr1, constr2...]
    [where not constrNot1, constrNot2...]
    [different metaDifferent1, metaDifferent2...]
    [intersect metaIntersect1, metaIntersect2...]
    [equal metaEqual1, metaEqual2...]
    [subset metaSubset1, metaSubset2...]
    [isSubsetOf metaIsSubsetOf1, metaIsSubsetOf2...]
    [notEqual metaNotEqual1, metaNotEqual2...]
    [notSubset metaSubset1, metaSubset2...]
    [isNotSubsetOf metaNotSubsetOf1, metaNotSubsetOf2...]
```

Content included in square parenthesis are optional.

2.3 Type-Constraints

In defining meta-set, type constraints can be specified first, then, field selectors (optional part). In case of relational databases, type-constraints point to specified table. In case of object database, type-constraints constrain object type. Type constraints can be declared in 2 different lists:

1) Type constraints without applied operator NOT:

```
var metaCats = metaset Cat;
```

2) Meta-set 'metaCats' points to the set consisting of all the Cats.

Type constraints with applied operator NOT – are declared after keyword 'not':

```
var metaAnimalNotDogCat = metaset Animal not Dog, Cat;
```

Meta-set 'metaAnimalNotDogCat' points to set of all Animals who are not Dogs and not Cats. In object oriented world, it means instances of all class Animal subclasses which are not Dog and not Cat.

Type constraints list should contain types which are expected types of resulting object set. Type constraints can also contain interfaces.

2.4 Property-Constraints

Property-constraints specify value range limitation for the properties. In defining the meta-set, the "property-constraints" should be listed after keyword "where". Metaset defining the female persons younger than 33 years can be defined as follows:

```
var metaYoungFemalePersons = metaset Person
    where Age < 33, Sex == Gender.Female;
```

Relational database query generated from meta-set instance 'metaYoungFemalePersons' will be as follows:

```
SELECT * FROM Persons WHERE Age < 33 AND Gender = 'female'
```

Meta-set 'metaYoungFemalePersons' construction can be explained by defining property-constraints using operator 'meta' as follows:

```
var metaYoungPersons = MetaSet of Person
where meta(Person.Age) < 33,
      meta(Person.Sex) == Gender.Female;
```

This is not a type-safe meta-set declaration example, because it allows declaring property constraints using types that may not be contained in type-constraints list. But it is good for explaining how meta-sets are constructed. In this example, type Person is used 3 times and all 3 times meaning is the same, because we are accessing members of the same type. Therefore, meta-set definition syntax allows property-constraint declaration omitting type information. In such cases, type information will be kept only in type constraints list.

Type constraint list can contain more than one type constraint. If property-constraint definition is unambiguous then type name in the property-constraint definitions still can be omitted as demonstrated in the following example:

```
var ievasWithBigSalaries = MetaSet of IPerson, IEmployee
    where Name=="Ieva", Salary > 500;
```

Previous example is equivalent to the following type-unsafe example:

```
var ievasWithBigSalaries = MetaSet of IPerson, IEmployee
    where meta(IPerson.Name) =="Ieva",
    meta(IEmployee.Salary) > 500;
```

Property-constraints with applied logical NOT operator follows operator “wherenot”. Following example demonstrates usage of “wherenot” operator:

```
var catFemaleNot6YearsOld = MetaSet of Dog
    where Sex == Gender.Female
    where not Age == 6;
```

2.5 Set-Constraints

Operators: different, intersect, equal, subset, isSubsetOf, notEqual, notSubset and isNotSubsetOf are set-constraints that allows to define relations with other meta-sets. Example of set-constraint ‘different’ usage is shown in the following example:

```
var metaYoungCatsDifferentThanBlackCats = metaset Cat where Age < 2
    different metaset Cat where Color == Colors.Black;
```

When meta-set containing set-constraints will be processed, it will involve set-operations. Equivalent SQL query is following:

```
SELECT * FROM Cats WHERE Age < 2
EXCEPT
SELECT * FROM Cats WHERE Color = Colors.Black;
```

The same result can be achieved using property-constraints:

```
var metaYoungCatsDifferentThanBlackCats = metaset Cat where Age < 2, Color != Colors.Black;
```

Set-constraint ‘different’ will expect that the intersection of the meta-set declared before operator ‘different’ with the meta-set(s) declared immediately after operator ’different’, is empty set. If the intersection is not empty set, then from the first meta-set will be extracted all meta-sets following immediately after operator ‘different’.

Relationships of object sets are always defined between two sets represented by meta-sets: the first is meta-set containing object set constraints list (context meta-set) and the second is meta-set used as an object set constraint. In cases where it is necessary to define relationships between more than two meta-sets, it is possible to add more object set constraints.

SetsRelationship can be one of 8 supported meta-set relationship forms: Different, Intersect, Equal, NotEqual, Subset1, NotSubset1, Subset2, NotSubset2 (Vanags at. al., 2013).

Another way on how to add meta-set constraints is by using Meta-Set instance methods as demonstrated in following example:

```
var metaYoungCats = metaset Cat where Age < 2;
var metaBlackCats = metaset Cat where Color == Colors.Black;
var metaYoungCatsDifferentThanBlackCats = metaYoungCats.Different(metaBlackCats);
```

2.6 Selector Support

Efficient querries do not request data which will not be used in the software. Selectors are the way how to specify which columns data should be loaded.

The example of selector usage is following:

```
var youngCat = metaset Cat {Name, Age}
    where Age < 2
```

In static object oriented languages, the selector can be described as one of 2 options:

a) When working with types specified in type-constraints, but from database loading only fields specified in selector declaration. If selector is not specified, then loading all data.

b) When automatic type generation with subset of properties specified in selector. Using the complex property-constraint there might be difficulties to compare seemingly equal types with different selectors.

Responsibility to handle type generations should be on ORM side and not on querying language side. Option (a) is the simplest way to implement. Option (b) needs more research to understand how to implement it better. One solution could be multiple class inheritance, but many of modern mainstream programming languages do not allow multiple class inheritance.

3. Advanced Examples

Comparison operators cannot represent aggregation functions or value containment check. MemberInfo instance level methods can be used to represent value containment check and aggregation functions. Metadata declaration using value containment check in type safe way is demonstrated in the following example, and support for aggregation functions can be added similarly:

```
var metaPersonsWhoLikeNumber6 = metaset Person
    where FovouriteNumbers.Contains(6);
```

The same can be expressed in more verbose, but type-unsafe way as follows:

```
var metaPersonsWhoLikeNumber6 = MetaSet of Person
    where meta(Person.FovouriteNumbers).Contains(6);
```

Property-value constraints can be not only simple value constraints, but also complex constraints indicating the type of the property and values of its properties. These complex constraints can be built as another meta-set containing desired type constraint, property-value constraints and even set-constraints. The next example shows how to declare complex meta-set which represents set of persons having (meaning: property Pets contains) at least one trained dog. Example of such complex property-constraint is demonstrated in following example:

```
var trainedDogOwners = metaset Person
    where Pets.Contains(metaset Dog where Trained == true);
```

Summary

Existing programming languages allow creation of complex data structures by combining smaller pieces of data. Some programming languages allow the gathering of metadata, which can be fully or partially combined into more complex data structures. But the problem is that none of the existing programming languages offer language built in support for combining metadata and data in a type safe way.

The result of combining metadata and data is abstraction of property-constraint. The result of combining property-constraints with metadata is meta-set – abstraction of data query. Meta-sets contain neither real objects, nor references to real objects; meta-sets can contain only constraints which can be used to generate database query and retrieve set of objects. Therefore, first class meta-set support is a feature which can bring type safe querying language into other programming languages.

Further, meta-sets can be used in rule based engines increasing reuse of existing meta-sets and transforming data query building process into rule declaration process. It means transformation from query building process to expert system and transformation from 4th generation programming languages to 5th generation programming languages (Vanags at. al., 2013).

References

Albahari, J., & Albahari, B. (2015). C# 6.0 in a Nutshell: The Definitive Reference. O'Reilly Media.

Harrison, G. (2015). Next Generation Databases: NoSQLand Big Data, Apress.

Hazzard, K., & Bock, J. (2012). Metaprogramming in .NET. Manning. Manning Publications Co.

James, R. G., Paul, N. W., & Andrew, J. O. (2009). SQL: The Complete Reference, 3rd Edition, McGraw-Hill

Education.

NISO Press. (2010). Understanding Metadata. Retrieved from http://www.niso.org/publications/press/UnderstandingMetadata.pdf

Vanags, M., Licis, A., & Justs, J. (2013). Meta-set calculus as mathematical basis for creating abstract, structured data store querying technology. 20th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2013), 2013. p. 299-313. Retrieved from https://www.dcc.fc.up.pt/~ricroc/homepage/publications/2013-INAP.pdf

Vanags, M., Licis, A., & Justs, J. (2013). Strongly typed metadata access in object oriented programming languages with reflection support, Baltic J. Modern Computing, Vol. 1 (2013), No. 1, 77-100. Retrieved from http://www.bjmc.lu.lv/fileadmin/user_upload/lu_portal/projekti/bjmc/Contents/1_1-2_6_Vanags.pdf