CQLINQ SYNTAX

Table of contents

1. Introduction

This document assumes that you are familiar with the C# LINQ syntax, and exposes the CQLinq syntax peculiarities.

CQLinq, Code Query LINQ, is a feature proposed by the tool CppDepend since the version 3, to query C\C++ code through LINQ queries.

CQLinq syntax peculiarities are:

- CQLinq Query edition
- Predefined domains
- Defining the code base view just-my-code with notmycode prefix
- CQLinq code rules
- The Query operator and Query expression syntaxes
- CQLinq query result formatting
- Matching code elements by name string
- Defining query targets
- Defining range variables with let
- Beginning a query with let
- Defining a procedure in a query
- Types usables in a CQLinq query

2. CQLinq Query Edition

A CQLinq query can be edited live in the CppDepend UI (standalone or in Visual Studio). The query is executed automatically as soon as it compiles.

Notice in the screenshot below the 1ms at the top right, that indicates the execution duration of the query. CQLinq is fast and is designed to run hundreds of queries per seconds against a large real-world code base.

CQLinq edition comes also with code completion/intellisense, and also tooltip documentation on mouse hovering the query body.

3. Predefined domains

CQLinq defines a few predefined domains to query on including:

- Types,
- Methods,
- Fields,
- Namespaces,
- Assemblies.

These domains can be seen as variables of type

- IEnumerable<IType>,
- IEnumerable<IMethod>,
- IEnumerable<IField>,
- IEnumerable<INamespace>,
- IEnumerable<IProject>.

These domains enumerate not only all code elements of the code base queried, but also all third-party code elements used by the code base (like for example the type string and all methods and fields of the type string that are used by the code base).

The syntax is as simple as:

```
from m in Methods where m.NbLinesOfCode > 30 select m
```

A CQLinq query can rely on one or several domains. Notice in the query above how the domain word **Methods** is highlighted differently.

There is a predefined domain named **context** of type ICQLinqExecutionContext that is the root of all others predefined domains. For example the domain Methods is actually converted by the CQLinq compiler to the expression **context**.CodeBase.Methods:

```
from m in context.CodeBase.Methods where m.NbLinesOfCode > 30 selec
t m
```

The type of the domain **context** is reserved and thanks to the other predefined domains, there is no need to use **context**.

There is also a predefined domain named codeBase of type ICodeBase that is converted to **context**.CodeBase:

```
from m in codeBase.Methods where m.NbLinesOfCode > 30 select m
```

There are two convenient predefined domains that are used often: **Application** and **ThirdParty**.

As their name suggest, these domains are useful to enumerate code elements defined only in application assemblies, or only defined in third-party assemblies (*like mscorlib.dll, System.dll or NHibernate.dll*) and used by the application code. These two domains are of type ICodeBaseView and represent each a partial view of the entire code base.

```
from m in Application.Methods where m.NbLinesOfCode > 30 select m
```

The domains **Application** and **ThirdParty** are converted to **context**.CodeBase.Application and **context**.CodeBase.ThirdParty.

Notice how the interface ICodeBase extends the interface ICodeBaseView, since the code base can be seen as a total view on itself.

Thanks to the richness of the CppDepend.CodeModel namespace, it is easy to refine these predefined domains.

For example the query below matches large methods defined only in the namespace ProductName.FeatureA and its child namespaces:

```
from m in Application.Namespaces.WithNameLike("ProductName.FeatureA")
.ChildMethods()
where m.CyclomaticComplexity > 10 select m
```

4. Defining the code base view JustMyCode with notMyCode prefix

There is another convenient predefined domain named **JustMyCode** of type ICodeBaseView. The domain **JustMyCode** is converted to **context**.JustMyCode.

The domain **JustMyCode** represents a facility of CQLinq to eliminate generated code elements from CQLinq query results.

For example the following query will only match large methods that are not generated by a tool (like a UI designer):

```
from m in JustMyCode.Methods where m.NbLinesOfCode > 30 select m
```

The set of generated code elements is defined by CQLinq queries prefixed with the CQLinq keyword **notmycode**.

For example the query below matches methods defined in source files whose name ends up with designer.cs. These are file generated by some UI designer like the Windows Form designer:

```
notmycode from m in Methods where
  m.SourceFileDeclAvailable &&
  m.SourceDecls.First().SourceFile.FileName.ToLower().EndsWith(".designer.cs")
select m
```

The CQLinq queries runner executes all notmycode queries before queries relying on **JustMyCode**, hence the domain **JustMyCode** is defined once for all. Obviously the CQLinq compiler emits an error if a **notmycode** query relies on the **JustMyCode** domain.

5. CQLinq Code Rules

A CQLinq query can be easily transformed into a rule by prefixing it with a condition defined with the two CQLinq keywords **warnif count**.

The keyword **count** is an unsigned integer that is equal to the number of code elements matched by the query.

For example the following query warns if some large methods are matched in the code base application methods:

```
// <Name>Avoid too large methods</Name>
warnif count > 0
from m in Application.Methods
where m.NbLinesOfCode > 30
select m
```

CQLinq code rules are useful to define which bad practices the team wants to avoid in the code base.

The team can see code rules violation warning in the CppDepend UI (standalone or in Visual Studio), or in the report.

The team has also the possibility to define some rules as critical rules.

Here is the list of default CQLinq Rules and here is the documentation about validating code rules inside Visual Studio.

6. The query operator and query expression syntaxes

Since the CQLinq syntax is based on the C# LINQ syntax, both the query operator syntax and the query expression syntax are allowed. The query operator syntax is the one with direct calls to System.Linq.Enumerable extension methods like Where() and Select()…

```
Methods.Where(m => m.NbLinesOfCode > 30)
```

The query expressions syntax is the one with special C# LINQ keywords like where and select…

```
from m in Methods where m.NbLinesOfCode > 30 select m
```

Often you'll find convenient to mix both syntaxes in one query.

- The query operator syntax is convenient to define sub-set and sub-domains.
- The query expression syntax is convenient to define operations on these sub-sets and sub-domains.

For example the query below defines with the query operator syntax the sub-set of methods defined in static types, and use the query expression syntax to filter and project the large methods from this sub-set.

```
from m in Application.Types.Where(t => t.IsStatic).ChildMethods()
where m.NbLinesOfCode > 30 select m
```

7. CQLinq query result formating

The CQLinq query result formatting is constrained. A query result can be a simple numeric scalar value like in the query above:

```
Methods.Where(m => m.NbLinesOfCode > 30).Count()
```

Or the query result can be an anonymous type, whose first property is of type **IType**, **IMethod**, **IField**, **INamespace** or **IProject**,
and additional properties (if any) are of type:

- **IType**, **IMethod**, **IField**, **INamespace** or **IProject** or
- **IEnumerable<IType>**, **IEnumerable<IMethod>**, **IEnumerable<IField>**, **IEnumerable<INamespace>** or **IEnumerable<IProject>** or
- A numeric scalar value or a numeric scalar value nullable (like int, double? or Nullable<decimal>),
- A boolean or a boolean nullable (bool or bool?)
- A string
- A value of the enumeration CppDepend.CodeModel.Visibility.

For example the following query result is an enumerable of an anonymous type with 6 properties. Notice that a maximum of 16 properties is accepted for any anonymous type used in a CQLinq query.

```
from m in Application.Methods
select new {
   m,                // First property of type NDepend.CodeModel.IMethod
   m.Visibility,     // of type NDepend.CodeModel.Visibility
   parentTypeName = m.ParentType.Name + m.Name, // of type string
   m.MethodsCalled, // of type IEnumerable<NDepend.CodeModel.IMethod>
   m.IsStatic,       // of type bool
   m.NbLinesOfCode   // of type Nullable<uint> ( uint? )
 }
```

8. Machine code elements by name string

The table below summarizes the three different properties involved in naming.

Kind of code element	IMember.FullName	ICodeElement.Name	ISimpleNamed.SimpleName
Type	System.String System.Collections.Generic.List<T> System.Environment+SpecialFolder	String List<T> Environment+SpecialFolder	String List SpecialFolder
Method	System.Collections.Generic.List<T>.Add(T) System.Collections.Generic.List<T>.ConvertAll<TOutput>(Converter<T,TOutput>)	Add(T) ConvertAll<TOutput>(Converter<T,TOutput>)	Add ConvertAll
Field	System.String.Empty	Empty	-
Assembly	-	mscorlib System	-
Namespace	-	System.Collections.Generic System	Generic System

Notice that the interfaces INamespace and IProject don't implement IMember hence there is no full naming of namespace and assemblies.

Also the interfaces IField and IProject don't implement ISimpleNamed hence there is no simple naming of fields and assemblies.

With this naming system, it is easy to write CQLinq queries to match some code elements by name:

```
from t in ThirdParty.Types where
 t.FullName == "System.Collections.Generic.List<T>" &&
 t.Name == "List<T>" &&
 t.SimpleName == "List"
select t
```

Notice that the static class ExtensionMethodsNaming presents some convenient extension methods like WithFullNameIn(), that make code elements matching by name even easier:

```
from t in Types.WithFullNameIn("System.IDisposable", "System.String"
, "System.Object") select t
```

Some properties of CppDepend API and some of the ExtensionMethodsNaming extension methods are specialized in name matching through regular expression. Notice here the CQLinq compiler magic that occurs to make sure that the regular expression passed in argument, is just compiled once for all code elements listed. Notice as well the regular expression suffix \i for ignore case like regular expressions. To use \i the string constant must be verbatim (i.e prefixed with the character@).

```
from t in Types.WithNameLike("ist") where
  t.FullNameLike("ist") ||
  t.NameLike(@"Lis\i") ||
  t.SimpleNameLike("is")
select t
```

Finally, some of the ExtensionMethodsNaming extension methods are specialized in name matching with simple wildcard patterns, with the star * wildcard character:

```
from t in Types.WithFullNameWildcardMatch("System.I*") select t
```

9. Defining query targets

There are special API methods that take a code element name or full name string as argument. These methods are extension methods defined in the type ExtensionMethodsCQLinqDependency.

These methods are convenient to write elegant code queries, where no extra-characters are consumed to first match a code element by name and then search for its usage. For example the query below naturally matches disposable types:

```
from t in Types
where t.Implement("System.IDisposable")
select t
```

Notice that there is some CQLinq compiler magic to make sure that the interface System.IDisposable is searched just once before executing the query.

Such code element is named a **query target** and is listed in the query execution result pane.

In such situation, if no code element is matched by name, a query compilation error occurs. The reason can be a misspelling error, but it can be as well that, for example, the code base doesn't make any usage of the interface System.IDisposable. Only third-party code elements used by the application code are listed in the third-party code.

Hence, to write generic queries that virtually compile and run on any code base (no matter if the interface System.IDisposable is used or not), you can use the stringextension method AllowNoMatch(), that will prevent compilation error, and make the Implement() expression always returns false if the code elements cannot be matched by name:

```
from t in Types where t.Implement("System.IDisposable".AllowNoMatch()
) select t
```

If a query target name matches several code elements of the same kind (like several methods or several types) all these code elements are considered as query target.

If a query target name matches several code elements of different kinds, like for example the "System" string can match both the System assembly and the Systemnamespace, a query compilation error occurs. To resolve such situation, there are special extension methods like MatchAssembly() that forces the matching to occurs only with a particular kind of code elements:

```
from t in Types where t.IsUsing("System".MatchAssembly()) select t
```

Finally, as the side screenshot shows, notice that the star * wildcard character can be used to match several query target code elementes at once. Interestingly enough, in the screenshot the type System.IO.File is matched, where one would expect only interfaces.

If the wildcard syntax is used, the extension methods usage will have a any behavior (using any, implements any, with any return types...)

10. Defining range variable with let

The C# LINQ syntax present the facility to define range variables with the C# keyword let. In this section, we wanted to underline this possibility because using the letkeyword is a common practice when writing CQLinq queries.

For example, the following default rule define a custom code metrics thanks to several range variables:

```
// <Name>C.R.A.P method code metric</Name>
// Change Risk Analyzer and Predictor (i.e. CRAP) code metric
// This code metric helps in pinpointing overly complex and unteste
d code.
// Reference: http://www.artima.com/weblogs/viewpost.jsp?thread=21589
9
// Formula:   CRAP(m) = comp(m)^2 * (1 - cov(m)/100)^3 + comp(m)
warnif count > 0
from m in JustMyCode.Methods

// Don't match too short methods
where m.NbLinesOfCode > 10

let CC = m.CyclomaticComplexity
let uncov = (100 - m.PercentageCoverage) / 100f
let CRAP = (CC * CC * uncov * uncov * uncov) + CC
where CRAP != null && CRAP > 30
orderby CRAP descending, m.NbLinesOfCode descending
select new { m, CRAP, CC, uncoveredPercentage = uncov*100, m.NbLinesO
fCode }
```

Notice that using many let clauses in the main query loop can significantly decrease performance of the query execution.

11. Beginning a query with let

The CQLinq compiler extends the usage of the C# LINQ let keyword, because with CQLinq, the let keyword can be used to define a variable at the beginning of a CQLinq query.

For example, the default CQLinq rule below, first tries to match the System.IDisposable types, and if found, second let the query be executed.

```
// <Name>Types with disposable instance fields must be disposable</Na
me>
warnif count > 0

let iDisposable = ThirdParty.Types.WithFullName("System.IDisposable")
.FirstOrDefault()
where iDisposable != null // iDisposable can be null if the code bas
e doesn't use at all System.IDisposable

from t in Application.Types where
   !t.Implement(iDisposable) &&
   !t.IsGeneratedByCompiler

let instanceFieldsDisposable =
    t.InstanceFields.Where(f => f.FieldType != null &&
                                f.FieldType.Implement(iDisposable))

where instanceFieldsDisposable.Count() > 0
select new { t, instanceFieldsDisposable }
```

For some others CQLinq rules, it can be convenient to define multiple sub-sets through several let keyword expressions, before executing the query itself.

For example, the default CQLinq rules below; first define the sub-sets uiTypes and dbTypes before using them in the query code.

```
// <Name>UI layer shouldn't use directly DB types</Name>
warnif count > 0

// UI layer is made of types in namespaces using a UI framework
let uiTypes = Application.Namespaces.UsingAny(
                  Assemblies.WithNameIn("PresentationFramework", "Syst
```

```
em.Windows",
                                        "System.Windows.Forms", "Syste
m.Web")
                ).ChildTypes()

// You can easily customize this line to define what are DB types.
let dbTypes = ThirdParty.Assemblies.WithNameIn("System.Data", "Entity
Framework", "NHibernate").ChildTypes()
                .Except(ThirdParty.Types.WithNameIn("DataSet", "DataTab
le", "DataRow"))

from uiType in uiTypes.UsingAny(dbTypes)
let dbTypesUsed = dbTypes.Intersect(uiType.TypesUsed)
select new { uiType, dbTypesUsed }
```

12. Defining a procedure in a query

With the LINQ syntax it is possible to create a procedure in a query. This is useful if you wish to invoke such procedure from different locations in the query.

This possibility is illustrated in the default rule below where the procedure to check if a type can be considered as a dead type needs to be invoked from two different locations:

```
// <Name>Potentially dead Types</Name>
warnif count > 0
// Filter procedure for types that should'nt be considered as dead
let canTypeBeConsideredAsDeadProc = new Func<IType, bool>(
   t => !t.IsPublic && //   Public types might be used by client appl
ications of your assemblies.
         t.Name != "Program" &&
        !t.IsGeneratedByCompiler &&
        !t.HasAttribute("NDepend.Attributes.IsNotDeadCodeAttribute".A
llowNoMatch()))
         // If you don't want to link NDepend.API.dll,
         // you can use your own IsNotDeadCodeAttribute and adapt thi
s rule.

// Select types unused
let typesUnused =
   from t in JustMyCode.Types where
   t.NbTypesUsingMe == 0 && canTypeBeConsideredAsDeadProc(t)
   select t

// Dead types = types used only by unused types (recursive)
let deadTypesMetric = typesUnused.FillIterative(
types => from t in codeBase.Application.Types.UsedByAny(types).Except
(types)
         where canTypeBeConsideredAsDeadProc(t) &&
               t.TypesUsingMe.Intersect(types).Count() == t.NbTypesUs
ingMe
         select t)

from t in deadTypesMetric.DefinitionDomain
select new { t, t.TypesUsingMe, depth = deadTypesMetric[t] }
```

13. Types usable in a CQLinq query

Not all types are usable in a CQLinq query because internally, the CQLinq compiler and runner are optimized to work with a defined set of types. You'll find in this list all types needed to query a code base:

CppDepend.CodeModel	ExtensionMethodsHelpers ExtensionMethodsNaming ExtensionMethodsProjection ExtensionMethodsSequenceUsage IProject ICodeBase ICodeBaseView ICodeContainer ICodeElement ICodeElementParent ICodeMetric`2 ICodeMetricValue`2 ICodeNode`1 ICompareContext IField IMember IMethod INamespace ISimpleNamed ISourceFile ISourceFileLine IType IUsed IUser

	SourceFileLanguage Visibility
CppDepend.CodeQuery	RecordBase Record`1, Record`2 Record`3, Record`4 Record`5, Record`6 Record`7, Record`8 Record`9, Record`10 Record`11, Record`12 Record`13, Record`14 Record`15, Record`16
CppDepend.Helpers	ExtensionMethodsEnumerable ExtensionMethodsSet ExtensionMethodsString
CppDepend.Reserved.CQLinq	ExtensionMethodsCQLinqCompare ExtensionMethodsCQLinqDependency ExtensionMethodsCQLinqNaming ICQLinqExecutionContext
CppDepend.Path	IPath IDirectoryPath IFilePath IAbsoluteDirectoryPath IAbsoluteFilePath IRelativeDirectoryPath IRelativeFilePath PathHelpers
System.Linq	Enumerable: IGrouping`2 ILookup`2 IOrderedEnumerable`1

	IQueryable`1 Queryable
System.Collections.Generic	Dictionary`2, HashSet`1, ICollection`1, IDictionary`2, IEnumerable`1, IEnumerator`1, IList`1, KeyValuePair`2, List`1
System	Array, Boolean, Byte, Char, Decimal, Delegate, Double, Func`1, Func`2, Func`3, Func`4, Func`5 Int16, Int32, Int64 Math, Nullable`1, Object, Predicate`1, SByte, Single, String, UInt16, UInt32, UInt64, Void