Concepts behind the C# 3 language

Tomáš Petříček (tomas@tomasp.net)

Faculty of Mathematics and Physics,
Charles University in Prague

1 Introduction

The C# 3 (currently available in preliminary version) is part of the LINQ project [1]. The aim of this project is to integrate better support for working with data into main stream general purpose programming languages developed at Microsoft targeting the .NET platform* (C# and VB.Net).

Most of the ideas that are implemented in C# 3 are already available in (mostly) research and experimental languages that were previously developed at Microsoft Research†. Two most interesting and influential languages developed at Microsoft Research are Cω [2] and F# [3].

1.1 Contribution of F#

F# is language based on ML, which is an impure functional language (in fact, it is based on language called OCaml that adds several features to the standard ML). Functional languages are very different from imperative languages (most of the widely used languages like C++, Java and C# are imperative). The biggest contribution of F# is that it shows how functional languages can be compiled for the .NET runtime (CLR), because .NET CLR was initially designed for executing code written in imperative languages. Another aim of F# is the interoperability with other languages targeting the .NET platform. Part of the F# related research is also the ILX [6] project that shows how .NET runtime could be extended to provide better support for functional languages (like first-class functions).

Functional programming in general were big inspiration for some of the C# 3 features and the F# research language already showed how these features can be implemented for the .NET platform. C# 3 includes constructs that were inspired by type inference (ability to deduce the type of expression), tuples (data types that represent pair of values), first class functions (ability to take function as a parameter and return it as a result), lazy evaluation (ability to evaluate expression only when it is later needed) and meta-programming (ability to manipulate with program source code).

Most of these features that were added to C# 3 are very limited when compared with their implementation in F# and other functional languages, but it is very interesting to see how functional concepts are becoming more and more important and can benefit to the non-functional languages.

1.2 Contribution of Cω

Cω is a language based on C# and it extends it in two most important areas. First area is better support for working with structured data (XML)

---

* .NET is a platform for developing applications developed by Microsoft. It consists of Common Language Runtime (CLR) that provides managed execution environment and Base Class Library (BCL) which is a class library providing implementation of large range of programming tasks.

† Microsoft Research is independent computer science research group inside Microsoft Corporation. Microsoft Research is academic organization, which means that it focuses primary on research activities.

and relational data (databases). The language extends type system of C# to include support for several data types that are common in relational and structured data and it provides querying capabilities for working with these data structures.

The second area is support for concurrency constructs as a part of the language. In most of the widely used programming languages, support for concurrency is provided as a class library. By including these constructs in language, the program becomes more readable (intentions of programmer can be expressed better) and more work can be done automatically by the compiler.

C# 3 and LINQ project in general is inspired by the first area of extensions in Cω and the syntax extensions in C# 3 are very similar to the concepts developed in Cω. The biggest difference (aside from the fact that Cω is experimental language) is that C# 3 provides better extensibility allowing developers to provide mechanism for working with different data sources. This extensibility is provided as a language feature and not as a compiler feature in case of Cω. On the other side, some extensions in Cω were simplified in C# 3 (for example anonymous types that will be mentioned later), so where appropriate I will include examples where something possible in Cω can't be done in C# 3.

1.3 Overview of C# 3

The main goal of C# 3 is to simplify working with data and make it possible to access relational data sources (database) in much simpler way. In C# 3 this is implemented in extensible way, not by simply adding several special purpose constructs. Thanks to the new language constructs it is possible to write statements like following:

```
var query = from c in db.Customers
            where City == "London"
            orderby c.ContactName;
            select new { CustomerID, ContactName };
```

This statement looks like a SQL query. This is achieved thanks to query operators (*from*, *where*, *select*, *orderby* and some other) that were added to the language. These operators are syntactic sugar that simplify writing of queries, but are mapped to underlying functions that perform projection, filtering or sorting (these functions are called *Select*, *Where*, *OrderBy*).

To perform for example filtering, the *Where* function needs to take another function as a parameter. Passing functions as a parameter to other functions is simply possible using new language feature called lambda expressions (this is similar to the lambda functions known from many functional languages).

You can also see, that the query returns only *CustomerID* and *ContacctName* from the more complex Customer structure. It is not required to explicitly declare new class with only these two members, because C# 3 allows developers to use anonymous types. It is also not required to declare type of *query* variable, because type inference automatically deduces the type when *var* keyword is used.

1.4 Cω and integration of data access

The original idea of integrating data access into the general purpose programming language first appeared in the Cω research project at Microsoft Research. The data access possibilities integrated in Cω includes working with databases and structured XML data. The LINQ project is mostly based on Cω, however there are some differences.

The features that can be found in both C# 3 and Cω include anonymous types (these are not limited to local scope in Cω), local type inference and query operators. One concept that wasn't available in Cω is extensibility through expression trees. In Cω you couldn't write your own implementation of data source that would execute queries over other than in-memory data. The following demonstration shows how working with database in Cω looks (it is very similar to the previous example written in C# 3):

```
query = select CustomerID, ContactName
          from db.Customers
          where City == "London"
          order by ContactName;
```

The Cω project will be mentioned later in other sections, because some of the features that are available in C# and were originally implemented in Cω are more powerful in Cω and it is interesting to see this generalization.

2 First class functions

> *A programming language is said to support first class functions if functions can be created during the execution of a program, stored in data structures, passed as arguments to other functions, and returned as the values of other functions.*
>
> *(Source: Wikipedia.org)*

Support for first class functions is necessary for functional style of programming. For example functions that operate on lists in Haskell (*map*, *foldl*, *etc…*) are all higher order functions, which mean that they take function as a parameter.

The previous quotation summarizes what first class functions are and what are advantages of languages supporting this feature. More exact definition of what language has to support to treat functions as first class objects can be found in [5]. According to this book, language has first class functions if functions can be

1. declared within any scope
2. passed as arguments to other functions, and
3. returned as results of functions

The point 2 and 3 can be accomplished in many languages including C/C++ that allows passing pointer to function as argument. In first version of C#, it was possible to use delegates, which can be simply described as type-safe function pointers; however neither C/C++, nor first version of C# allowed declaration of function in any scope, particularly inside body of other function (or method).

2.1 First class functions in F#

I will first mention how first class functions are supported by F #, which is mixed imperative and functional language for .NET platform. I choose the F# language, because it shows that implementing functional language features under the .NET platform is possible. I will first show some features that are unique in F# (when compared with other .NET languages):

```
// Declaration of binding 'add' that is initialized
// to function using lambda expression
let add = (fun x y -> x + y);;

// Standard function declaration (shorter version,
// but the result is same as in previous example)
let add x y = x + y;;

// Currying – creates function that adds 10 to parameter
let add10 = add 10;;
```

The first what this example shows, is that in F#, functions are type like any other. This means that unlike in most of the languages where functions are something else than global variables, functions in F# are just ordinary global variables (or bindings, to be more precise, because by default all data in F# are immutable). This is demonstrated by the first example of *add* function, which is just a global variable whose value is function created using lambda expression. The second example shows simplified syntax for declaring global functions (but the only difference when compared with first version of *add* function is syntax!).

The next example shows currying, which is an operation that takes function and binds its first parameter with specified value. In this case, the parameters are function *add* (with type *int* -> *int* -> *int*) and constant 10. The result is function (assigned to *add10*) of type *int* -> *int* that adds 10 to specified parameter. The currying operation can be used in any language that has first class functions, but the syntax in F# makes it extremely easy to use. The following examples show some common usages of first class functions, that I will later write in C# as well:

```
// passing function to higher order functions
let words = ["Hello"; "world"] in
iter (fun str -> Console.WriteLine(str); ) words;

// Returning nested function
let compose f g =
  (fun x -> f (g x));;
```

The first example first declares list containing strings, and than uses function *iter* that calls function passed as first parameter for every element in the list (note that, F# is not purely functional language, which means that passed function can have side effects, like printing string on the console).

The second example is function that returns composition of two functions passed as arguments. This is a good example of returning function declared in local scope. The second interesting point in this example, are types of the functions. No types are declared explicitly so type inference algorithm infers that type of first parameter is *'b* -> *'c*, type of second parameter is *'a* -> *'b* and return value has type *'a* -> *'c* (where *'a*, *'b* and *'c* are type parameters). This means that *compose* function can be used for any two functions, where return type of the second function is the same as the type of the parameter of first function.

2.2 First class functions in C# 2 and C# 3

Since first version of .NET Framework and C#, it provides mechanism called delegates. Delegates are type-safe function pointers that can be passed as a parameter or return value from a function. As mentioned earlier, first version of C# didn't support declaration of functions inside function body (nested functions), so it was only possible to initialize delegate to globally declared method.

The following example shows how anonymous delegates in C# 2 can be used for creating method that returns "counter" delegate. The counter delegate is a function that adds numbers to specified init value and returns total:

```
CounterDelegate CreateCounter(int init) {
  int x = init;
  return new delegate(int n) {
      x += n;
      return x;
    }
}

CounterDelegate ct = CreateCounter(10);
ct(2); // Returns 12
ct(5); // Returns 17
```

This example shows one interesting problem that appears with the ability to declare functions inside body and the ability to return this function as a return value. You can see that the anonymous delegate uses local variable called *x* for storing total value, so there must be some mechanism for creating reference to local variables inside a method body. Another problem in this example is that when function *CreateCounter* returns, the activation record for the function (with the local variable *x*) would be destroyed. These issues are solved in functional languages using closures and the similar mechanism was added to C# 2 (closure captures the value of local variable and it preserves when the activation record is destroyed).

While anonymous delegates provide everything that is needed for first class functions, the syntax is quite verbose and inconvenient for use known from functional languages. The C# 3 comes with new language feature called lambda expressions, which provides more functional syntax, and the .NET base class library is extended to include some of more important functions known from functional languages (like *map*, *filter*, etc…). Use of these functions is illustrated in the following example:

```
IEnumerable<int> nums = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
IEnumerable<int> odd = nums.Where( (n) => (n%2) == 1 );
IEnumerable<int> oddSquares = odd.Select( (n) => n*n );
```

The first line of the example declares variable *nums*, which is array containing numbers from 1 to 10. In the second line the numbers are filtered and returned list *odd* contains only odd numbers. The *Where* function takes delegate as a parameter and the delegate is created using lambda expression syntax. In this syntax *(n)* specifies parameters of the delegate and the expression on the right side of "=>" token is the body of delegate. The *Select* function is later used to calculate squares of all odd numbers in the list. One more interesting point in this example is that you don't have to explicitly declare type of lambda expression parameter when the type can be deduced from the context (this will be discussed in following section).

Lambda expressions can be also used for accessing the data representation of expressions, but this will be described in more detail alter in the section about meta-programming.

3 Type inference

Type inference is the feature of strongly typed programming languages that refers to the ability to deduce the data type of an expression. This feature is currently available in many (mostly functional) programming languages including Haskell and ML (and it is also available in F#).

In languages that support type inference, most of the identifiers (like variables and functions) can be declared without explicit type declaration; however the language is still type-safe because the type is inferred automatically during the compilation.

3.1 Type inference in F#

F# type system (which is based on the ML type system) depends heavily on type inference. In fact, no declaration in F# expects name of the type unless some conflict or ambiguity occurs, in this case programmer can use type annotations to provide hints for the compiler. Let's start with simple example that demonstrates how F# type inference works:

```
// Some value bindings
let num = 42;;
let msg = "Hello world!";;

// Function that adds 42 to the parameter
let addNum x = x + num;;

// Identity function
let id x = x;;
let idInt (x:int) = x;;
```

In this example, no type is explicitly declared, but type of all identifiers is known at compile time. The type of value bindings (*num* and *msg*) can be inferred because the type of expression on the right side of binding is known (42 is integer and "Hello world!" is string). In the last example, type of *num* is known (it is integer) and the "+" operator takes two integer parameters; therefore the type of *x* parameter must be integer too. Because "+" applied to two integral parameters returns integer, the return type of the function is also integer.

The last two examples show how to declare identity function in F# (the function that returns passed parameter). The type inference algorithm figures out that the type of returned value is exactly the same as the type of parameter, but it doesn't know what type exactly it is. In this case, F# type system can use type parameters[‡], so the inferred type is *'a -> 'a* (where *'a* is type parameter).

If I want to define identity function only for integral values, I need to use type annotations, to provide hints for the compiler. This is shown in the last example where it is explicitly declared that the type of parameter *x* will be integer. Using this information, type inference algorithm can deduce that the return value of the function will be integer too.

3.2 Type inference in C# 3

The support for type inference in C# 3 is very limited when compared with type inference in F#, but it is very interesting, because it makes C# first main stream language that supports it (for no obvious reason, type inference was implemented only in functional languages so far).

In C# 3, type inference can be used only with lambda expressions or to infer type of the local variable that is initialized to some value with initialization expression. It is not possible to use type inference for neither method return value nor any class members. First, let's look at the type inference that can be used with lambda expressions:

```
IEnumerable<int> nums = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
IEnumerable<int> odd = nums.Where( (n) => (n%2) == 1 );
```

In this example the lambda expression *(n) => (n%2) == 1* is used, but the type of parameter *n* is not explicitly declared. In this case, the compiler knows that the type of parameter expected by the *Where* function is delegate that takes *int* as a parameter and returns *bool*. Based on this information, compiler deduces that the type of *n* must be an *int*. It also checks whether the return type of lambda expression matches with the expected return value of the delegate.

The second case where type inference is used in the C# 3 is the ability to deduce type of local variable using the *var* keyword:

```
// Some simple examples
var str = "Hello world!";
var num = 42;

// Declaration without using type inference
TypeWithLongName<AndWithTypeParameter> v =
  new TypeWithLongName<AndWithTypeParameter>();

// Same declaration with type inference
var v = new TypeWithLongName<AndWithTypeParameter>();
```

The first two examples shows how type inference can be used to infer type of primitive variables (first one is string and the second is integer). The next example shows that type inference can significantly simplify the source code, because it is not needed to repeat same type twice in variable declaration. There is however still intensive discussion, in which situations

[‡] The use of type parameters is concept similar to the templates known from C++ or to the generics implemented in .NET languages (like C#). In F# compilation it is implemented using .NET generics.

the *var* keyword should be used and when not, because when used improperly, the code becomes less readable.

There is one very important reason for including type inference in C#. This reason is anonymous types. Anonymous types will be discussed in more detail later, so I will show only a simple example:

```
var anon = new { Name = "Hello" };
Console.WriteLine(anon.Name);
```

In this example, the *anon* variable is initialized to instance of anonymous type, which has one member called *Name* with value "Hello". The compiler knows exactly the structure of the type, but there is no name that could be used for declaring variable of this type. In this situation the type inference and the *var* keyword becomes extremely useful.

4 Lazy evaluation

Lazy evaluation is one of the basic concepts in the Haskell programming language and thanks to lazy evaluation it is for example possible to declare infinite lists using list comprehensions. I will discuss two different uses for lazy evaluation in the C# language, first I will show how to emulate lazy evaluation for parameter passing in C# which is possible thanks to better support of first-class functions and in the second section I will show that writing infinite (lazy) lists in C# is almost as easy as in Haskell.

4.1 Lazy parameter passing

When calling function using lazy evaluation strategy, the value of parameter is evaluated only when needed. This is default behavior of Haskell and it can be also simulated in Scheme using *delay/force*. In C# similar behavior can be achieved using high-order functions (either anonymous delegates or better using lambda expressions). Typical example of lazy evaluation is function with two parameters that uses second parameter only when first matches some criteria:

```
// Function that may not need second parameter
int func(int first, int second)
{
  if ((first%2) == 0) return 0; else return second;
}

// Call to the 'func'
func(n, expensiveCalculation());
```

In this example, lazy evaluation may be very useful, because it can prevent *expensiveCalculation()* from being called when it is not needed. In the following code, I will use the *Func<T>* type (where *T* is type parameter). This type represents delegate (.NET type used for passing functions) with no parameters and with return value of type *T*.

```
// Now, the parameters are functions that
// can be used for calculating values
int func(Func<int> first, Func<int> second)
{
  if ((first()%2) == 0) return 0; else return second();
}
```

```
// Parameters are passed using lambda expressions
func(() => n, () => expensiveCalculation());
```

This function now behaves similarly to its equivalent written in Haskell, because when the second parameter is not needed, the *expensiveCalculation* function is never invoked. There is only one important difference – when value is needed twice, the function would be evaluated also twice, however this can be simply solved by passing wrapper class that contains delegate for calculating value and stores result of the call locally when it is evaluated.

4.2 Infinite lists in C# 3

The closest equivalent to infinite lists that are possible in Haskell thanks to lazy evaluation is concept of iterators known from object oriented programming. Iterators are used for traversing through elements of collection, usually using method that moves to next element in collection and method that returns current element. However writing iterators for creating number sequences is much more complicated than using of list comprehensions in Haskell.

Already mentioned language Cω introduced new data type called stream. Streams in Cω are collections homogenous collections of particular type like arrays, but unlike arrays, streams are lazy. The syntax for stream declaration is asterisk (*) appended to the name of element type. This language also introduced interesting syntax for generating streams that makes it possible to generate streams much easily than using iterators[§]:

```
// COmega – stream with fibonacci numbers
public static int* Fibs() {
  int tmp, v1=0, v2=1;
  while (true) {
    yield return v1;
    tmp = v1; v1 = v2; v2 = tmp + v2;
  }
}
```

The key concept here is the *yield return* statement that returns next element of the stream. The stream is lazy, so when reading elements from the stream, only the required number of iterations is done and the execution is stopped when no more elements are needed. The Cω language also provides in-line syntax for declaration of streams:

```
int* nums = { for(int i=0; ; i++) yield return i; };
```

When working with lists in Haskell, the most useful functions are map (that provides projections) and filter (that provides filtering). These functions are available in Cω in two (syntactically different) ways. First way makes it possible to use XPath-like operators and the second way enables SQL-like querying (the functionality is of course same). In the following

[§] Note that Cω was developed on .NET 1.0 and was introduced before C# 2, so the yield return statement known from C# 2 was first introduced in Cω.

example, I will show how to get squares of all even numbers from previously defined sequence of numbers (in both XPath-like and SQL-like ways):

```
// XPath-like filter and projection operators
int* squaresOfEven = nums[ (it%2)==0 ].{ it*it };

// SQL-like projection (select) and filter (where) operators
int* squaresOfEven = select it*it from nums where (it%2)==0;
```

This *yield return* statement was considered as very successful, so it was introduced in the next version of mainstream C# language (version 2.0). The main difference in C# is, that the returned data type is not a stream (*int**), but an iterator object (*IEnumerable<int>*). This introduces simple method for writing infinite lists in C#, however other features for working with streams (like apply-to-all statement) are not available in C# 2.

The main reason why C# 2 didn't contain functions known from functional programming (projection and filtering) was, that C# 2 contained only poor support for first passing functions as arguments. This is improved in C# 3, so thanks to lambda expressions, it is possible to declare methods that provide filtering and projection operations for sequences (*IEnumerable<T>*). The following example shows how to work with sequences in C# 3 in way that is similar to the Cω examples:

```
// Function GetNums returns all numbers using yield return
IEnumerable<int> nums = GetNums();

// Manipulation using Where and Select
IEnumerable<int> squaresOfEven =
  nums.Where((it) => (it%2)==0 ).Select((it) => it*it );

// Same operation written using query operators
IEnumerable<int> squaresOfEven =
  from it in nums where (it%2)==0 select it*it;
```

In the first example manipulation is done using *Where* and *Select* methods that take lambda expression to use for filtering, respectively projection. Exactly the same result can be achieved using special operators (*select*, *from*, *where* and some other) that are available in C# 3.

I still didn't show how to declare infinite lists in C# 3 in a way similar to the list comprehensions known from Haskell. Declaration of list in Haskell can consist of four parts. First (initialization) section declares initial values that can be later used for calculating the rest of the list. Second (generator) section declares list from which values in the newly created list can be calculated (this list can use the newly created list recursively). The last two sections are filtering and projection for manipulating elements from the generator list. Let's see how this can be written in Haskell and in C# 3 (following example defines infinite list of Fibonacci numbers):

```
-- Haskell
fibs = 0 : 1 : [a + b | (a, b) <- zip fibs (tail fibs)]

// C# 3
var fibs = new InfiniteList<int>((t) =>
  t.ZipWith(t.Tail()).Select((v) => v.First + v.Second),
  0, 1);
```

Semantics of both examples are same. The list of Fibonacci numbers is declared recursively using already known elements in the list. The generator section in C# - *t.ZipWith(t.Tail())* is equivalent to *zip fibs (tail fibs)* written in Haskell. In the C# version, it is not possible to reference *fibs* variable while it is being initialized, so the *t* is instance of the list passed to the generator to allow recursive declarations. The projection secion in C# - *Select((v) => v.First + v.Second)* is equivalent to *a + b | (a, b)* from Haskell, the difference here is that C# doesn't directly support tuples, so tuple is represented as structure with members *First* and *Second*. The filtering section is not used in this example and finally, the last two parameters (0, 1) define initial values in the list which is equal to declaration of first two values in Haskell (0 : 1 : ...).

4.3 Lazy evaluation summary

The lazy evaluation strategy can be simulated in many non-functional languages, because it is general concept, however in most of these languages this simulation would be difficult and inconvenient to use. Thanks to new features available in latest version of C# 3, some of concepts known from functional programming (including lazy evaluation and list comprehensions) can be naturally used in C#.

5 Anonymous types

Anonymous types are another example of useful feature known from functional programming that appeared in C# 3. Anonymous types are based on tuples known from Haskell (or any other functional programming language including F# developed at Microsoft Research). First object oriented language that implemented concept of tuples was already mentioned Cω.

Tuples (also called anonymous structs) in Cω can contain either named or anonymous fields. The type of tuple containing anonymous string and integer field called N is *struct{string; int N;}*. Anonymous fields of the tuple can be accessed using index expression. For example to get the string field from the previous tupe, you can use *t[0]*. The named fields can be either accessed using the same way as anonymous fields or using the name. In Cω tuple is regular type so it can be used in any variable or method declaration. The following example shows how the tuples in Cω can be used.

```
// Method that returns tuple type
public struct{string;int Year;} GetPerson() {
  return new { "Tomas", Year=1985 };
}

// Call to the GetPerson method
struct{string;int Year;} p = GetPerson();
Console.WriteLine("Name={0}, Year={1}", p[0], p.Year);
```

One of the situations where tuples are extremely useful is when using projection operation (map function in Haskell, projection operator or select operator in Cω, Select method in C# 3). The following example first creates stream with numbers and than uses projection operator to create stream of tuples where every element contain original number and its square.

```
// Get stream with numbers
int* nums = GetNumbersStream();
// Projection returns stream of tuples with number and its square
struct{int;int;} tuples = nums.{ new { it, it*it } };
```

The projection operation was also probably the main reason for including anonymous types in C# 3, because the C# 3 and the LINQ project focuses primary on simplifying data access and while writing database queries, projection is very important. The following example shows using of anonymous types in C# 3:

```
// Creating anonymous type
var anon = new { Name="Tomas Petricek", Year=1985 };

// Creating anonymous type in projection operator
from db.People select new { Name=p.FirstName+p.FamilyName, p.Year };
```

It is obvious that anonymous types in C# 3 are based on idea presented in Cω, however there are several differences. First difference is that in C# all fields of anonymous type are named. In the previous example, name of first field is specified explicitly (*Name*) and name of second field is inferred automatically (the name *Year* will be used). The second and probably most important difference is that in C# 3 it is possible to create instance of anonymous type, but the only way to declare variable of this type is to use the *var* keyword. This keyword infers type of the variable from expression, so type information isn't lost. This means, that there is no way to reference anonymous type in C# 3 which makes it impossible to use anonymous types as parameters of methods and anonymous types can therefore be used only in limited scope.

This limitation of scope in C# 3 is intentional, because misusing of anonymous types in object oriented programming could lead to less readable and less maintainable code, however there are some situations when passing anonymous types as return value from methods would be very useful. The Cω language shows that this limitation can be resolved and that tuples known from functional programming can be used without limitations in object oriented languages as well.

6 Meta-programming

6.1 Language oriented development

"Language oriented programming is a style of programming in which, the programmer creates domain-specific languages for the problem first and solves the problem in this language."

(Source: Wikipedia.org)

Mainstream languages like C++, C#, Java or functional Haskell are languages that belong to the category called general purpose languages (GPL). This category contains languages that are not intended for some specific use, but can be used for developing wide range of applications.

Opposite to these languages are so-called domain specific languages (DSL) that can be used only for one specific purpose. These languages are usually designed and optimized for their purpose, so they serve better than

GPL. These languages are in some sense similar to class libraries (in object oriented world) because class libraries are also designed for solving one specific purpose.

Good example of DSL is the SQL language, whose purpose is database querying. This language is demonstrates all characteristics of DSL – it has very limited domain of use, but in this domain it serves better than any general purpose languages.

In his article [4], Martin Fowler divides DSLs into external and internal. Internal (also called embedded) DSL are languages that extend and modify the host (general purpose) language, in which they are used. The example of language that allows developers to modify it (and it is necessary for bigger projects) is LISP that is itself very simple, but it provides ways for extending it, so in more complex projects, the language is first extended using macros and the problem can be than solved easily using the created LISP dialect. On the other side, external DSL are languages independent to the language in which they are used (for example SQL).

6.2 Meta-programming

"Meta-programming is the writing of programs that write or manipulate other programs (or themselves) as their data."
(Source: Wikipedia.org)

As described in this quotation, the principle of meta-programming is that code written by the developer works with some representation of program code (either in same or in different language). This code can be analyzed, interpreted or translated to some other language.

It is obvious, that meta-programming is used for creating domain specific languages; however creating of DSLs is not the only possible use of meta-programming. In the case of external DSL developer has to write compiler or some translator to other (usually general purpose) language. This option isn't used very often, because writing of compiler is not a trivial task.

The case of internal DSL is much more interesting, because the program manipulates with its own code at run-time. To enable this, the language must provide some way for accessing to data representation of its own code as well as some extensibility that would allows users to define their own sub-languages.

From what I already mentioned, you can see that in language that supports advanced form of meta-programming, it is possible to develop domain specific language similar to SQL. Using the meta-programming, it would be later possible to get representation of code written in this SQL-like sub-language, translate it to SQL command text and execute it!

In the rest of the text I will use the term meta-programming for manipulating with programs own code using the features provided by the programming language, because manipulation with external code can be written almost in any GPL language (only the support for reading and manipulation with text files is needed for this).

6.3 Meta-programming in C# 3

Support for meta-programming (however only very limited form) is one of the key features that enables the LINQ project and especially its implementation for working with databases called "LINQ to SQL" (previously DLINQ). When writing LINQ code that will be used for accessing to database, the expressions used for filtering and projection (as well as for other operators like joins) must be translated to SQL command and this couldn't be done without the possibility to get the data representation of those expressions (therefore without some form of meta-programming).

In C# 3, it is possible to get the data representation (called expression tree) only from lambda expression whose body is expression. I see the biggest limitation of C# 3 here, because it is not possible to get data representation of neither statement nor statement block. The lambda expression can be compiled in one of two ways. It can be either compiled to delegate (object that can be used to execute the function), or to the code that returns expression tree (the data representation) of lambda expression. The decision between these two options is based on the l-value of the code in which lambda expression appears. When it is assigned to the variable of type *Func*, it is compiled as delegate and when the type of variable is *Expression*, it is compiled as data.

The following example shows how the same lambda expression can be compiled to delegate as well as data representation. For demonstration purposes we'll use a function that takes integer as parameter, performs some calculations with it and returns true when the value is less than ten.

```
// Lambda expression as executable code
Func<int, bool> =
  x => DoSomeMath(x) < 10;

// Lambda expression as data representation
Expression<Func<int, bool>> =
  x => DoSomeMath(x) < 10;
```

This is exactly the principle that is used in the LINQ project in case when application works with database and so the query is translated to the SQL language. The following example demonstrates how two lambda expressions (one for filtering and second for projection) can be used when accessing database:

```
// Database query written using Where
// and Select extension methods
var q =
  db.Customers.
    Where(c => c.City == "London").
    Select(c => c.CompanyName);
```

From this example, you can see that expressions used for filtering and projection of data from *Customers* table are passed to methods *Where* and *Select* using lambda expressions. It depends on the concrete LINQ implementation whether type of the parameter passed to these methods will be *Func* or *Expression* and so it is possible to write implementations that accept delegate types and execute them (for example for filtering data that are stored in memory) as well as implementations that accept data representation of the code and translates it to different language (in case of LINQ to SQL implementation) or use some other way to execute the code.

This work well, however to make the data access even more intuitive, the C# 3 provides operators (from, select, where and some other) for this purpose. These operators are only syntactic sugar and code written using these operators is translated to the code shown in previous example. Using the LINQ operators you can write following:

```
// Database query that uses "syntactic sugar"
var q =
  from c in db.Customers
    where c.City == "London"
    select c.CompanyName;
```

If you think of these examples in terms of DSL and meta-programming, you can see that domain specific language used to represent database queries in C# 3 is composed from expressions that can be compiled to expression trees and from the query operators (that are built in the language) or to be more precise from the query methods (Select, Where, etc). The target language of translation (SQL in the case of LINQ to SQL) doesn't usually support all the expressions that can be written in C#, so the translator must check whether all constructs written by the programmer can be translated.

In general, any DSL language that can be created in C# 3 can use only expressions (or limited subset of C# expressions) and some way for composing these expressions together (for example using sequence of method invocations or using overloaded operators).

6.4 Meta-programming in F#

In the F# language, the support for meta-programming goes even further by making it possible to access (almost) any code written in F# by using special operators. This includes expressions (as in C# 3) but also statements and more complex structures. The following example illustrates how to get data representation of infinite loop writing the text "Hello" (The type of returned value is *expr* which represents F# code):

```
let e = <@
      while true do
        print_string "hello"
      done @>;;
```

As I already said, it is possible to write lambda expressions in F#, because lambda expressions are one of the basic features in any functional language. It is also possible to get *expr* representation of those lambda expressions, so F# quotations can be used in same way as lambda expressions in C# 3:

```
let e = <@
        fun x -> doSomeMath(x) > 10
      @>;;
```

Recently released version of F# contains example that shows how to extend this language with support for data querying constructions similar to the operators in the LINQ project. This shows the flexibility of F# language, because it was not needed to extend the language itself, because the key feature (meta-programming) was already supported and it is possible to define custom operators, so no syntactic sugar has to be added.

The implementation of data querying in F# internally transforms data representation of expression written in F# to the C# 3 expression tree and this expression tree is passed to the "LINQ to SQL" converter. In the following example, you can see how to write database query similar to the previously shown C# 3 example:

```
let query =
  db.active.Customers
  |> where <@ fun c -> c.City = "London" @>
  |> select <@ fun c -> c.ContactName @>;;
```

There is one more very interesting feature in F# that I didn't mention so far. I will demonstrate this on the example with lambda expression that calls *doSomeMath* function. F# allows you to expand all top-level definitions in the code representation with its actual value. This means that the call to *doSomeMath* function can be replaced with its actual implementation. This can be for example used, to allow the developer to use custom functions in data query expressions[**].

6.5 Meta-programming summary

From these examples, you can see that F# meta-programming is more flexible than lambda expressions available in C# 3. The data representation of code may look more complex than the representation used in LINQ, however it is easy to use thanks to its functional design and thanks to functional features in F#.

The most important features that aren't available in C# 3 are that in C# you can get data representation only for expression, while in F# it is possible to get code of statements (including loops and statement blocks) as well as function declarations. The second feature (that is not turned on in the compiler by default) is that in F# you can get data representation of any top-level definition (all declared functions) which makes it possible to expand function calls in expressions.

[**] This is very important feature that makes it possible to reuse part of the query across whole data-access layer. It is possible to achieve something similar in C# 3 as described here: http://tomasp.net/blog/linq-expand.aspx

References

[1] The LINQ Project. D. Box and A. Hejlsberg.
See http://msdn.microsoft.com/data/ref/linq/

[2] Cω language. E. Meijer, W. Schulte, G. Bierman, and others.
See http://research.microsoft.com/Comega/

[3] F# language. D. Syme.
See http://research.microsoft.com/fsharp/

[4] Language Workbenches: The Killer-App for Domain Specific Languages?
Marin Fowler.
See http://martinfowler.com/articles/languageWorkbench.html

[5] Concepts in Programming Languages. John C. Mitchell.
Cambridge University Press, Cambridge UK, 2003

[6] ILX project. D. Syme.
See http://research.microsoft.com/projects/ilx/ilx.aspx