Advances in Programming Languages

APL12: Heterogeneous Metaprogramming in F#

Ian Stark

School of Informatics
The University of Edinburgh

Thursday 19 February 2008
Semester 2 Week 6

Topic: Domain-Specific vs. General-Purpose Languages

This is the third of three lectures on integrating domain-specific languages with general-purpose programming languages. In particular, SQL for database queries.

- Using SQL from Java
- LINQ: .NET Language Integrated Query
- Language integration for F# metaprogramming

Topic: Domain-Specific vs. General-Purpose Languages

This is the third of three lectures on integrating domain-specific languages with general-purpose programming languages. In particular, SQL for database queries.

- Using SQL from Java
- LINQ: .NET Language Integrated Query
- Language integration for F# metaprogramming

Don Syme. Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution.

In *Proceedings of the 2006 ACM SIGPLAN Workshop on ML*, Sep. 2006.

Outline

1. Metaprogramming
2. F#
3. Examples of metaprogramming in F# with LINQ

Outline

Metaprogramming

The term *metaprogramming* covers almost any situation where a program manipulates code, either its own or that of some other program. This may happen in many ways, including for example:

- Textual manipulation of code as strings
- Code as a concrete datatype
- Code as an abstract datatype
- Code generation at compile time or run time
- Self-modifying code
- Staged computation

Although this would also include any compiler or interpreter, the idea of metaprogramming usually indicates specific language features, or especially close integration between the subject and object programs.

Metaprogramming Examples

Macros

```
#define geometric_mean(x,y) sqrt(x*y)

#define BEGIN {
#define END }

#define LOOP(var,low,high) \
    for (int var=low; var<high; var++) BEGIN

int i, total = 0; LOOP(i,1,10) total=total+i; END
```

Here geometric_mean is an inlined function; while the *non-syntactic* LOOP macro is building code at compile time.

Metaprogramming Examples

C++ Templates

```
template<int n>
Vector<n> add(Vector<n> lhs, Vector<n> rhs)
{
  Vector<n> result = new Vector<n>;
  for (int i = 0; i < n; ++i)
    result.value[i] = lhs.value[i] + rhs.value[i];
  return(result);
}
```

This template describes a general routine for adding vectors of arbitrary dimension. Compile-time specialisation can give custom code for fixed dimensions if required. The C++ Standard Template Library does a lot of this kind of thing.

Metaprogramming Examples

Java reflection

```
import java.io.*;
import java.lang.reflect.*;

Class c = Class.forName("java.lang.System");   // Fetch System class
Field f = c.getField("out");                   // Get static field
Object p = f.get(null);                        // Extract output stream
Class cc  = p.getClass();                      // Get its class
Class types[] = new Class[] { String.class };  // Identify argument types
Method m  = cc.getMethod("println", types);    // Get desired method
Object a[]  = new Object[] { "Hello, world" }; // Build argument array
m.invoke(p,a);                                 // Invoke method
```

Reflection of this kind in Java and many other languages allows for programs to indulge in runtime *introspection*. This is heavily used, for example, by toolkits that manipulate Java *beans*.

Metaprogramming Examples

Javascript eval

```
eval("3+4");                 // Returns 7

a = "5-"; b = "2";
eval(a+b);                   // Returns 3, result of 5-2

eval(b+a);                   // Runtime syntax error

b = "1";
c = "a+a+b";
eval(eval(c));               // Returns 3, result of 5-5-1
```

Any language offering this has to include at least a parser and interpreter within its runtime.

Metaprogramming Examples

Lisp eval

```
(eval '(+ 3 4))         ; Result is 7

(eval `(+ ,x ,x ,x)))   ; Result is 3*x, whatever x is

(eval-after-load "bibtex"
  '(define-key bibtex-mode-map
               [(meta backspace)] 'backward-kill-word))
```

Unlike Javascript eval, code here is structured data, built using quote '(...) The backquote or *quasiquote* `(...) allows computed values to be inserted using the *antiquotation* comma ,(...).

Metaprogramming Examples

MetaOCaml

```
# let x = .< 4+2 >. ;;
val x : int code = .< 4+2 >.

# let y = .< .~x + .~x >. ;;
val y : int code = .< (4+2)+(4+2) >.

# let z = .! y ;;
val z : int = 12
```

Arbitrary OCaml code can be quoted `.< >.`, antiquoted with `.~` and executed with `.!`. All these can be nested, giving a *multi-stage* programming language with detailed control over exactly what parts are evaluated when in the chain from source to execution.

Metaprogramming Examples

MetaOCaml

```
# let x = .< 4+2 >. ;;
val x : int code = .< 4+2 >.

# let y = .< .~x + .~x >. ;;
val y : int code = .< (4+2)+(4+2) >.

# let z = .! y ;;
val z : int = 12
```

Various research projects have implemented multi-stage versions of Scheme, Standard ML, Java/C# and so on.

Metaprogramming Examples

MetaOCaml

```
# let x = .< 4+2 >. ;;
val x : int code = .< 4+2 >.

# let y = .< .~x + .~x >. ;;
val y : int code = .< (4+2)+(4+2) >.

# let z = .! y ;;
val z : int = 12
```

This is *homogeneous* metaprogramming: the language at all stages is OCaml. There is a version of MetaOCaml that supports *heterogeneous* metaprogramming, with final execution of the code *offshored* into C.

(pun)

Outline

F#

F# is a version of ML for the .NET platform. It is not unique in this: there is also SML.NET, implementing Standard ML, which itself grew from the MLj compiler for the Java virtual machine.

Easy F#

```
let rec fib n = match n with 0 | 1 -> 1 | n -> fib (n-1) + fib (n-2)

let build first last = System.String.Join( " ", [|first;last|] )

let name = build "Joe" "Smith"
```

To a (poor) first approximation, F# is OCaml syntax with .NET libraries.

F# Sales Pitch

F#

A succinct, type-inferred, expressive, efficient functional and object-oriented language for the .NET platform.

F# developed as a research programming language to provide the much sought-after combination of type safety, succinctness, performance, expressivity and scripting, with all the advantages of running on a high-quality, well-supported modern runtime system. This combination has been so successful that the language is now being transitioned towards a fully supported language on the .NET platform

http://research.microsoft.com/fsharp, 2009-02-18

F#

Interoperability with the .NET framework and other .NET languages is central to F#.

- Core syntax is OCaml: with higher-order functions, lists, tuples, arrays, records, ...
- Objects are nominal: with classes, inheritance, dot notation for field and method selection, ...

  (So no structural subtyping for objects, nor any row polymorphism)

- .NET toys: extensive libraries, concurrent garbage collector, install-time/run-time (JIT) compilation, debuggers, profilers, ...
- Creates and consumes .NET/C# types and values; can call and be called from other .NET languages.
- Generates and consumes .NET code: can exchange functions with other languages, and polymorphic expressions are exported with generic types.

F# Timeline

- Developed by Don Syme at Microsoft Research Cambridge (MSR).
- Started as Caml.NET, with a first preview release of F# compiler in 2002/2003.
- V1.0 release from MSR, with basic Visual Studio integration, in 2005.
- Official Microsoft *Community Technology Preview* (CTP) release in September 2008.
- Visual Studio 2010 will ship with F#

F# Timeline

- Developed by Don Syme at Microsoft Research Cambridge (MSR).
- Started as Caml.NET, with a first preview release of F# compiler in 2002/2003.
- V1.0 release from MSR, with basic Visual Studio integration, in 2005.
- Official Microsoft *Community Technology Preview* (CTP) release in September 2008.
- Visual Studio 2010 will ship with F#

> "This is one of the best things that has happened at Microsoft ever since we created Microsoft Research over 15 years ago"
>
> S. Somasegar, Head of Microsoft Developer Division

A Monad for Exhaustively Searching Infinite Sets in Finite Time

Martín Escardo
University of Birmingham

Room 4.31/4.33, Informatics Forum
4pm Thursday, 19th of February, 2009

Laboratory for Foundations of Computer Science
Seminar Series 2008–2009
http://www.lfcs.ed.ac.uk/seminar

Outline

LINQ Metaprogramming in C#

Recall from the last lecture that LINQ→SQL passes on the information needed to evaluate a query as an *expression tree*. By analyzing this, a complex expression combining several query operations might be executed in a single SQL call to the database.

Expression trees are built as required, and may include details of C# source code. For example:

```
Expression<Func<int,bool>> test = (id => (id<max));
```

Now `test` is not an executable function, but a data structure representing the given lambda expression.

This is quotation, but implicit: rather than having syntax to mark quotation of `(id => (id<max))`, the compiler deduces this from its `Expression` type.

Quotations in F#

Simple quote

```
> open Microsoft.FSharp.Quotations.Typed

- let a = <@ 3 @>;;
val a : Expr<int>

> a;;
val it : Expr<int> = <@ (Int32 3) @>
```

F# provides explicit quotation markers. Here the interactive response exposes the internal structure of an expression.

Quotations in F#

Larger quote

```
> <@ "Hello " + "World" @>;;
val it : Expr<string>
= <@
    (App (App (Microsoft.FSharp.Core.Operators.op_Addition)
              ((String "Hello")))
          ((String "World")))
  @>
```

A more complex quotation gives a more complex expression. Although verbose, the structure is exactly that of the original expression.

Quotations in F#

Function quote

```
> <@ fun x -> x+1 @>;;
val it : Expr<(int -> int)>
= <@
 fun x#39844.4 ->
   (App
      (App (Microsoft.FSharp.Core.Operators.op_Addition) x#39844.4)
      ((Int32 1)))
   @>
```

An expression of function type includes details of the function body. Here x#39844.4 is a variable name chosen by the expression printer.

Quotations Templates

Quote with hole

```
> let f = <@ 5 + _ @>;;
val f : (Expr<int> -> Expr<int>)

> f a;;                    // Remember that a is <@ 3 @>
val it : Expr<int>
= <@
(App (App (Microsoft.FSharp.Core.Operators.op_Addition) ((Int32 5)))
     ((Int32 3)))
   @>
```

A quotation with one or more holes gives a function mapping expressions to expressions: building large expressions from smaller ones. The operation `lift : 'a -> Expr<'a>` allows antiquotation, plugging in runtime values.

Application: F# to SQL by LINQ

Query in memory

```
val ( |> ) : 'a -> ('a -> 'b) -> 'b // Pipeline operator

let query =
      fun db ->
        db.Employees
        |> where (fun e -> e.City = "Edinburgh" )
        |> select (fun e -> (e.Name,e.Address))
```

The query function will inspect an in-memory datastructure db.Employees, filtering those working in Edinburgh and projecting out their name and address.

Here where and select are versions of filter and map for the db.Employees data type.

Application: F# to SQL by LINQ

Query via SQL

```
val ( |> ) : 'a -> ('a -> 'b) -> 'b

let query = SQL
  <@ fun db ->
      db.Employees
      |> where  (fun e -> e.City = "Edinburgh" )
      |> select  (fun e -> (e.Name,e.Address)) @>
```

Quoting the internals now gives a query function that will inspect an external database instead.

Application: F# to SQL by LINQ

Query via SQL

```
val ( |> ) : 'a -> ('a -> 'b) -> 'b

let query = SQL
    <@ fun db ->
        db.Employees
        |> where  (fun e -> e.City = "Edinburgh" )
        |> select  (fun e -> (e.Name,e.Address)) @>
```

The SQL function takes a quoted expression and passes it to LINQ; which compiles it to SQL and then hands it off to the database engine as:

SELECT Name, Address **FROM** Employees **WHERE** City = "Edinburgh"

Application: F# to SQL by LINQ

Query via SQL

```
val ( |> ) : 'a -> ('a -> 'b) -> 'b

let query = SQL
  <@ fun db ->
      db.Employees
      |> where  (fun e -> e.City = "Edinburgh" )
      |> select  (fun e -> (e.Name,e.Address)) @>
```

This heterogeneous metaprogramming leads to some mismatches between F# and SQL semantics: for example, SQL date/time is rounded to 3msec, less precise than .NET, and the definition of Math.Round is different.

Application: F# Runtime Code Generation

Powers of x

```
> let rec power (n,x) = if n = 0 then 1 else x*power(n-1,x);;
val power : int * int -> int

> let power4 = fun x -> power (4,x);;
val power4 : int -> int

> power4 5;;
val it : int = 625
```

Application: F# Runtime Code Generation

Powers of x

```
> let rec metapower (n,x) =
-   if n = 0
-     then  <@ 1 @>
-     else  <@ _ * _ @> (lift x) (metapower(n-1,x)) ;;
val metapower : int * int -> Expr<int>

> let metapower4 = fun x -> metapower (4,x) ;;
val metapower4 : int -> Expr<int>
```

The metapower function computes x^n as an expression rather than a value.

Application: F# Runtime Code Generation

Powers of x

```
> metapower4 5
- ;;
val it : Expr<int>
= <@
  (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
     (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
        (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
           (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
              ((Int32 1))))))) @>
```

The metapower4 function computes x^4 as an expression rather than a value. Like the database expression, this too can be passed to LINQ.

Application: F# Runtime Code Generation

Powers of x

```
> metapower4 5
- ;;
val it : Expr<int>
= <@
  (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
    (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
      (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
        (App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))
          ((Int32 1)))))) @>
```

LINQ provides lightweight code generation: at runtime the code is built, JIT compiled, run, and then garbage collected away.

Application: Accelerating F# by Outsourcing

```
let matrix f = Array2.init x y f    // Build x*y array filled with f x y
...
let neg a = matrix (fun i j -> - a.(i,j))
let  (.+)   a b = matrix (fun i j -> a.(i,j)  +  b.(i,j))
let (.&&)  a b = matrix (fun i j -> a.(i,j) && b.(i,j))
..
let rotate a dx dy = matrix (fun i j -> a.((i+dx)%x,(j+dy)%y))
let count a = matrix (fun i j -> int_of_bool a.(i,j))

let nextGeneration(a) =            // Take one step in Conway's Life
  let N dx dy = rotate (count a) dx dy in
  let sum = N (-1) (-1)  .+ N (-1) 0   .+ N (-1) 1
        .+ N   0  (-1)                 .+ N   0   1
        .+ N   1  (-1)  .+ N   1   0   .+ N   1   1  in
     (sum .= three) .|| (sum .= two) .&& a);;
```

Application: Accelerating F# by Outsourcing

```
open Microsoft.Research.DataParallelArrays // Use e.g. GPU pixel shader
let shape = [| x; y |]                     // Fixed dimensions x,y
..
let And (a:FPA) (b:FPA) = FPA.Min (a, b)  // Built−in operations on
let Or  (a:FPA) (b:FPA) = FPA.Max (a, b)  // floating−point arrays
..
let Rotate (a:FPA) i j = a.Rotate([| i;j |])
..
let nextGenerationGPU (a:FPA) =          // Take one step in Conway's Life
  let N dx dy = Rotate a dx dy in
  let sum = N (−1) (−1)  .+ N (−1) 0   .+ N (−1) 1
        .+ N   0  (−1)                 .+ N   0  1
        .+ N   1  (−1)  .+ N   1  0   .+ N   1  1  in
    Or (Equals sum three) (And (Equals sum two) a);;
```

Application: Accelerating F# by Outsourcing

Using the dataparallel library to drive an alternative computing engine is neat, but we did have to rewrite the code.

Application: Accelerating F# by Outsourcing

Using the dataparallel library to drive an alternative computing engine is neat, but we did have to rewrite the code.

As an alternative to writing new code for this particular application, we can write a general GPU translator that works over any expression:

```
val accelerateGPU : ('a[,] -> 'a[,]) expr -> 'a[,] -> 'a[,]
```

Application: Accelerating F# by Outsourcing

Using the dataparallel library to drive an alternative computing engine is neat, but we did have to rewrite the code.

As an alternative to writing new code for this particular application, we can write a general GPU translator that works over any expression:

```
val accelerateGPU : ('a[,] -> 'a[,]) expr -> 'a[,] -> 'a[,]
```

All we need do to run life on the GPU is then:

```
let nextGenerationGPU' = accelerateGPU <@ nextGeneration @>
```

Application: Accelerating F# by Outsourcing

Using the dataparallel library to drive an alternative computing engine is neat, but we did have to rewrite the code.

As an alternative to writing new code for this particular application, we can write a general GPU translator that works over any expression:

```
val accelerateGPU : ('a[,] -> 'a[,]) expr -> 'a[,] -> 'a[,]
```

All we need do to run life on the GPU is then:

```
let nextGenerationGPU' = accelerateGPU <@ nextGeneration @>
```

Caveat: The semantic mismatches are now more serious — actual floating-point arithmetic on GPU and CPU is not bit-identical.

Next Lecture

Database Programming Without Tiers

Sam Lindley
Laboratory for Foundations of Computer Science
The University of Edinburgh

9am Monday, 23rd of February, 2009

Links: Linking Theory to Practice for the Web
http://groups.inf.ed.ac.uk/links/

Summary

- Metaprogramming ranges from syntactic expansion through hygienic macros to staged computation and runtime code generation.
- F# is an ML for .NET, with an emphasis on interlanguage working.
- Quotations and templates bring metaprogramming to F#.
- F# can use LINQ to generate SQL . . .
- . . . or native code at runtime . . .
- . . . or to outsource execution wherever seems best.