Advances in Programming Languages
APL11: Heterogeneous Metaprogramming in F#

Ian Stark

School of Informatics
The University of Edinburgh

Thursday 21 February 2008
Semester 2 Week 7
This is the second 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
This is the second 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

Outline

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

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

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

#define BEGIN {
#define END }

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

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

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

```c++
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.
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

```javascript
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

\[
\text{eval} \ (\ + \ 3 \ 4) \quad ; \ Result \ is \ 7
\]

\[
\text{eval} \ (+ \ ,x \ ,x \ ,x)) \quad ; \ Result \ is \ 3\cdot x, \ whatever \ x \ is
\]

\[
\text{eval} \rightarrow \text{after} \rightarrow \text{load} \ "\text{bibtex}" \\
\quad \text{(define} \rightarrow \text{key} \ \text{bibtex} \rightarrow \text{mode} \rightarrow \text{map} \\
\quad \quad \quad \quad \text{[(meta} \ \text{backspace}]) \ \text{'}\text{backward} \rightarrow \text{kill} \rightarrow \text{word})
\]

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 ,( ... ).
Arbitrary OCaml code can be quoted `.< >.`., antiquoted `~.`. and executed `!`. 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.
Various research projects have implemented multi-stage versions of (at least) Scheme, Standard ML and Java/C#.
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

1. Metaprogramming

2. F#

3. Examples of metaprogramming in F# with LINQ
**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#

```fsharp
defib n = match n with 0 | 1 -> 1 | n -> fib (n-1) + fib (n-2)

defbuild first last = System.String.Join( ", " 

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

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

1. Metaprogramming

2. F#

3. Examples of metaprogramming in F# with LINQ
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:

```csharp
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

```fsharp
> <@ "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 clearly the same.
An expression of function type includes details of the function body. Here `x#39844.4` is a variable name chosen by the expression printer.
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.
The query function will inspect an in-memory datastructure `db.Employees`, filtering those working in London and projecting out their name and address.
Query via SQL

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

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

Quoting the internals now gives a query function that will inspect an external database instead.
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 = "London"
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.
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
The `metapower` function computes $x^n$ as an expression rather than a value.
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.
Powers of \( x \)

\[
\text{> metapower4 5}
\]

\[
\text{val it : Expr<int> = <@}
\]

\[
\begin{align*}
\text{(App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))} \\
\text{(App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))} \\
\text{(App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))} \\
\text{(App (App (Microsoft.FSharp.Core.Operators.op_Multiply) (5))} \\
\text{((Int32 1))))))) \text{ @>}
\end{align*}
\]

LINQ provides lightweight code generation: at runtime the code is built, JIT compiled, run, and then garbage collected away.
let matrix f = Array2.init x y f  // Fixed dimensions 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) =
    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) ;;
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 array operations
let Or (a:FPA) (b:FPA) = FPA.Max (a, b)
...
let Rotate (a:FPA) i j = a.Rotate([| i;j |])
...
let nextGenerationGPU (a:FPA) =
  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);;
Instead of writing the array code for this particular application, we can write a general translator that does this for expressions:

```
val accelerate : ('a [,] -> 'a[,] as expr) Expr -> 'a[,] as expr
```

Caveat: The semantic mismatches are now more serious — actual floating-point arithmetic on GPU and CPU is not bit-identical.
Instead of writing the array code for this particular application, we can write a general translator that does this for expressions:

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

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

```fsharp
let nextGenerationGPU' = accelerate <$> nextGeneration @@>
```
Instead of writing the array code for this particular application, we can write a general translator that does this for expressions:

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

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

```fsharp
let nextGenerationGPU' = accelerate <@ nextGeneration @>  
```

Caveat: The semantic mismatches are now more serious — actual floating-point arithmetic on GPU and CPU is not bit-identical.
Polymorphic typing of units of measure in F#  
Andrew Kennedy  
Microsoft Research Cambridge  

also

Compiling with Continuations, Continued  
Seminar  
Laboratory for Foundations of Computer Science  
Room 2511, James Clerk Maxwell Building  
4pm Monday 25 February 2008
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.