Meta-programming & you

Robin Message*

April 6, 2010

1 What is meta-programming?

Meta-programming is a term used in several different ways. Definitions include:

- Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime.[1]
- A metaprogram is a program that manipulates other programs (or itself) as its data. The canonical example is a compiler.[2]
- Meta-programming, by which is meant that one constructs an interpreter for a language close to the problem and then writes problem solutions using the primitives of this language.[3]

I'm going to define meta-programming as "code that creates, manipulates or influences other code."

2 Why do meta-programming?

There are three main reasons to do meta-programming. Firstly, to generate code that is specialised to a particular problem or environment, usually as a performance *optimisation*. Secondly, to add *abstraction* to programs where the abstraction cannot be expressed by a primary abstraction mechanism in the language the program is written in. Thirdly, to enable programs to be written in a language that better *expresses* the intentions of the author.

*Thanks to Dominic Orchard, Boris Feigin, Max Bolingbroke and Stephen Kell for their helpful comments. This pdf is available at `http://www.cl.cam.ac.uk/~rhm31/mplecture/handout.pdf`

[1]`http://en.wikipedia.org/wiki/Metaprogramming`

[2]`http://c2.com/cgi/wiki?MetaProgramming`

[3]Alan Mycroft *On integration of programming paradigms* ACM Computing Surveys 28.2 (1996) `http://doi.acm.org/10.1145/234528.234735`

2.1 Optimisation

By meta-programming, we can do some of the work of the compiler within our program. We can also do analyses and optimisations that the compiler cannot do, either because it was not designed to do them, or because they depend on domain-specific knowledge.

For example, the fast fourier transform is easiest to code up for power-of-two sized instances. It can also be done for non-power-of-two instances, but this is tricky and each one is different. However, it is also algorithmic, so a machine can easily do it if we write a program to do so. A program writing a program – sounds like meta-programming to me! It is also possible to unroll small FFT instances (which larger ones are made from) to produce faster code. Again, there is no need for this to be done by hand, and it is nice to be able to integrate small unrolled transformations into automatically generated code.[4]

2.2 Abstraction

Most programming languages provide one or more mechanisms for abstraction. For example, Java provides subclassing and generics, and ML provides higher-order functions, polymorphism and a module system. However, these abstraction mechanisms are always insufficient. Whatever abstraction mechanism you have, there will always be parts of your program that it doesn't work for[citation needed].

By using meta-programming we can add new abstraction mechanisms that are not built into the language. For example, aspect-orientated programming[5] is a technique for expressing *cross-cutting concerns*. A *concern* is a particular behaviour or set of behaviours in a system. Normally, the system is decomposed into parts based on some particular set of *concerns*. For example, a medical records application might be divided up into different sections for different users, and subdivided into screens for doing different things to records for each user. A *cross-cutting concern* would then be a *concern* that affect multiple other *concerns* and can't be slotted neatly into the dominant decomposition. In the medical records example, the logging of who has viewed and edited each record anywhere in the system is such a *concern*, and aspect-orientated programming gives a way to express such *cross-cutting concerns*.

Aspect-orientated programming adds advice, which is program code adding a new behaviour or mediating another behaviour, and join points, which are ways of expressing where advice should be applied to other parts

[4]These things are all implemented by FFTW, Fastest Fourier Transform in the West. `http://www.fftw.org/`

[5]Kiczales et al. *Aspect-Oriented Programming* In Proc. of ECOOP '97 `http://www.cs.ubc.ca/~gregor/papers.html`

of the program. Advice and join points are weaved with a program to provide a program that implements all the behaviours in the original program as well as the behaviours in the *cross-cutting concerns*.

A popular aspect-orientated system is Aspect/J[6] which adds aspect-orientated to the object-orientation of Java. Aspect/J runs by preprocessing the Java source code with the aspect source code to produce Java source code. Hence, Aspect/J is a form of meta-programming.

Another simple example of meta-programming for abstraction is the new-style for loop added in Java 1.5. This just abstracts the pattern:

```
for(Iterator i=collection.iterator();i.hasNext();) {
    Element e=i.next();
    //code
}
```

into

```
for(Element e:collection) {
    //code
}
```

This new control structure abstraction is elegant and useful – why did we have to wait to get it? Why can't we describe new syntax and control structures in some kind of meta-Java?

This is also a good point to note that there is no actual difference between abstraction and the subject of the next section, expressiveness. It is just a question of degree. However, one handy rule of thumb seems to be abstraction generally involves only rewriting based on context-free grammars, whereas expressiveness requires computation.

2.3 Expressiveness

Meta-programming can also be used to express programs that not only can't be properly abstracted in your current programming language, but can't be represented at all.

A good example is a Prolog interpreter. A prolog program is nothing like a program in another programming language. Nor is it anything like machine code. Therefore, we can see a prolog interpreter or compiler as a meta-program that turns a prolog program into a program in another language that is expressed in a completely different way.

Note a library could add a feature like a prolog interpreter to a language without such a facility. However, libraries do not appear integrated into the language in the same way as what we are trying to achieve.

The remainder of these notes will concentrate on meta-programming for expressiveness, under the title of domain-specific languages (DSLs).

[6] `http://www.eclipse.org/aspectj/`

3 Domain Specific Languages

In many ways, these are the most useful and common meta-programs. Configuration files, embedded scripting languages, Javascript+DOM in the browser, Excel – these are all domain specific languages. Generally, domain specific languages are more human readable, more structured, and less expressive than general purpose programming languages.

From our perspective, the most useful DSLs are those embedded into a general purpose programming language. This is for two reasons: *a*) so there are no limitations on the power of the embedded language – existing abstraction mechanisms are available; and *b*) to ease implementation – no need for a separate compiler.

By embedded, we mean that the DSL is valid code in the language it is embedded in, as well as expressing some domain specific concepts. This enables it to be parsed by our existing compiler/interpreter, and allows us to use the full power of the general purpose language in the DSL. The other aspect of embedding is we use the same type system in the DSL as the language we are embedded into. This is simple, but can cause us problems in two ways: *a*) It might be difficult or impossible to give a type to DSL elements, which causes problems for languages with a static type checker; and *b*) if we have type checking, we'd like it to check the DSL too, but concepts in the DSL might not be expressible in the type system From here on in, DSL will refer to an embedded DSL.

4 Let's get on with the code!

We'll now look at embedded DSLs in a number of languages and consider how amenable that language is to implementing a DSL. As a running example we will try to make a little language of mathematical expressions. Such a language could be useful in, for example, a user interface for layout constraints. The expression we will try to capture is: $area = (width + 8) * (height + 4)$

4.1 Java

Back in the days of fanfold, there was a type of programmer who would only put five or ten lines of code on a page, preceded by twenty lines of elaborately formatted comments. Object-oriented programming is like crack for these people: it lets you incorporate all this scaffolding right into your source code.[7]*.*

— Paul Graham

[7]`http://www.paulgraham.com/noop.html`

Java is a pretty standard Bondage and Discipline language[8], so we are not going to achieve any miracles with the syntax. The most useful technique for writing DSLs in Java is method chaining. This means returning an object from all of our DSL methods, usually *this*. Sometimes it makes more sense to return another object or a new object that allows you to do some kind of scoped actions. Note that the JVM is flexible and powerful, and with features like dynamic-classloading and reflection we can do almost anything on top of Java (for example, Clojure and Scala both run on top of the JVM and could be expressed in Java) but I'd argue that these things are not really Java.

```java
abstract class Expression {
    abstract double calculate(Map<String,double> env);
    Expression add(Expression b) {
        return new Add(this,b);
    }
    Expression mul(Expression b) {
        return new Mul(this,b);
    }
}
class Add extends Expression {
    private Expression a,b;
    Add(Expression _a,Expression _b) {
        a=_a;b=_b;
    }
    double calculate(Map<String,double> env) {
        return a.calculate(env)+b.calculate(env);
    }
}
class Mul extends Expression {
    private Expression a,b;
    Mul(Expression _a,Expression _b) {
        a=_a;b=_b;
    }
    double calculate(Map<String,double> env) {
        return a.calculate(env)*b.calculate(env);
    }
}
class Var extends Expression {
    private String name;
    Var(String _n) {
        name=_n;
    }
    double calculate(Map<String,double> env) {
        return env.get(name);
    }
}
class Num extends Expression {
    private double num;
    Num(double _n) {
        num=_n;
    }
    double calculate(Map<String,double> env) {
        return num;
    }
}
```

[8] `http://c2.com/cgi-bin/wiki?BondageAndDisciplineLanguage`

```
   static Expression add(Expression a,Expression b){return new Add(a,b);}
   static Expression variable(String name){return new Var(name);}
   static Expression number(double n){return new Num(n);}

50 Expression area=add(variable("width"),number(8)).mul(variable("height"
       ).add(number(4)));

   area.calculate(env);
```

There are several points to note in this example. Firstly, we generate expressions either using static functions on lines 46–48, or the method chaining methods in the Expression class on lines 3 and 6. This causes duplication, which is ugly, but unfortunately unavoidable.

Secondly, we are having to wrap variable names and numbers into classes. We could get around this by overloading all the mul and add functions, but that would quickly get tedious.

Thirdly, we have the classic expression problem here, which makes it difficult to add new processing methods. It is also impossible to add methods to chain any new binary operators like we have done for add and mul.

However, we do have a tree structure which we could do manipulations on. Overall, this isn't much of an embedded DSL since we have explicit wrapping of values and are just constructing a tree in memory.

4.2 ML & Haskell

Since ML is called Meta-Language, we would hope it is reasonably amenable to meta-programming! And it is amenable to meta-programming, but with the caveat the syntax is quite restrictive. If we are willing to write our DSL as data constructors forming trees, then we have an excellent system for building a compiler or interpreter, but not an embedded DSL. ML was designed for expressing tactics for theorem proving, not designing embedded DSLs.

Haskell has more lighter weight syntax, operator overloading, and lazy evaluation make it quite usable for embedded DSLs. Dominic Orchard's lecture, Mathematically Structuring Programming Languages, will look at construction a DSL in Haskell.

4.3 Ruby

> *Ruby is simple in appearance, but is very complex inside, just like our human body*[9].
>
> — Yukihiro "matz" Matsumoto

Ruby is a dynamically-typed, object-orientated scripting language[10]. Particular features that make ruby useful for creating DSLs are its lightweight

[9] http://www.ruby-lang.org/en/about/

[10] http://www.ruby-lang.org/

syntax, code blocks, symbols, and its overloading and dymanic method dispatch features, in particular *method_missing*.

```
class Expression
  def initialize b
    @block=b
  end
  def calculate environment
    @env=environment
    instance_eval &@block
  end
  def method_missing name,*args
    if args.length==0
      @env[name]
    else
      super name,*args
    end
  end
end

def Expression &block
  Expression.new block
end

area=Expression {
  (width+8) * (height+4)
}

area.calculate :width => 100, :height => 5
```

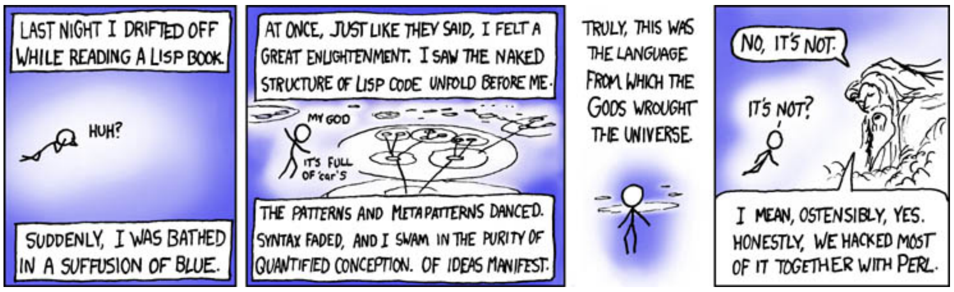
Here we create an Expression class that will hold our expressions, and three functions within it. The first just saves a parameter in an instance variable "@block". Instance variables are always written with an @ in front of them, and are the same as member variables in Java. The second calculates the result of an expression in an environment by saving the environment in an instance variable and evaluating the expression inside the current object, by calling "instance_eval". The third function handles any method calls that aren't recognised. This is a very handy Ruby meta-programming feature. We use it here to fetch parameters in the expression from the environment.

After that, we have a top-level function to make creating Expression instances neater. The "area" variable is initialised to an Expression that does our calculation. In this case, we are writing our expression in Ruby itself, so the only bit of meta-programming is being able to look up any variables in the expression correctly.

Finally, we evaluate the expression in an environment. It should be noted that this DSL is perhaps more fragile than the Java example. For example, if one of your variables was called "true", this code would not function correctly. More generally, there is a trade-off in shallow embedding of DSLs – it is easy to confuse DSL features with language features. In some cases, this is deliberate, for example to generate part of a DSL using the full power of the language, but as in the example above can lead to errors.

Ruby is good for embedded DSLs, as long as the DSL does not attempt to express something very different to what can be expressed in Ruby. Features like blocks, method_missing and eval make it very powerful for expressing code-like DSLs very succinctly. For example, the Rails web toolkit is very powerful partly due to its extensive use of Ruby meta-programming for things like ORM, routing and HTML generation.

4.4 Lisp


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.
AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF 'car'S
THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.
TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.
NO, IT'S NOT.
IT'S NOT?
I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

Copyright Randall Munroe[11]

What discussion of meta-programming would be complete without Lisp? Lisp is more of a family of languages than one in particular. There are two main ecosystems – Common Lisp and Scheme. Common Lisp seems more traditional and has an advanced object system. Scheme is smaller, more functional and has continuations and a hygienic macro system. If you are interested in running this code, it runs in the "Pretty Big" mode of DrScheme[12]. Lisp has never really broken through as a popular programming language, but it has always been influential. Particular features that make it good for meta-programming and DSLs are its macro system, lightweight syntax and very small core language.

Anyway, on with the code! We wish to represent mathematical expressions like $(width + 8) * (height + 4)$, so lets see the Lisp code to process expressions in that form:

```
(define (deinfix exp )
  (cond
    ((and (list? exp) (equal? 3 (length exp))) (list (cadr exp) (
        deinfix (car exp)) (deinfix (caddr exp))))
    (else exp)
) )

(define (calculate exp env)
  (map (lambda (v) (eval `(define ,(car v) ,(cdr v)))) env)
  (eval exp)
)

(define area '((width + 8) * (height + 4)))
(calculate area '((width . 100) (height . 50)))
```

[11]http://xkcd.com/224/
[12]http://www.plt-scheme.org/

This code defines a function “deinfix” that converts infix expressions into the prefix style Lisp uses, by recursively examining the expressions and turning any three part lists like $(a * b)$ into (* a b), and doing the same to a and b. Next, we define a calculate function. This function takes a Lisp expression and an environment, and evaluates the expression in that environment. Finally, we see how our area calculation can be expressed and how the environment is specified.

Hopefully, despite the lack of clear syntax, you can see how the program works. Now we'd like to fix some bugs, the first being the fact we only accept well-bracketed operators and we'd rather accept general mathematics. The second problem is our syntax for starting expressions and processing them is a bit ugly.

```
(define ^ expt)

(define (munge exp)
  (cond ((> (length exp) 4) (munge-prec (car exp) (cadr exp) (caddr
      exp) (cadddr exp) (cddddr exp)))
        ((equal? 1 (length exp)) (car exp))
        (else exp)
) )
(define (prec op)
  (case op
    ('+ 1) ('- 1) ('* 2) ('/ 2) ('^ 3) (else 0)
) )
(define (rassoc op)
  (eq? op '^)
)
(define (munge-prec a op1 b op2 rest)
  (let ((p1 (prec op1)) (p2 (prec op2)))
    (cond
      ((< p1 p2) (munge (cons a (cons op1 (munge (list* b op2 rest))))
          ))
      ((and (eq? p1 p2) (rassoc op1)) (list a op1 (munge (list* b op2
          rest))))
      (else (munge (list* (list a op1 b) op2 rest)))
) ) )

(define-syntax expression
  (syntax-rules ()
    ((_ exp ...)
     (deinfix (munge '(exp ...)))
) ) )

(calculate (expression (width + 8) * (height + 4)) '((width . 100) (
    height . 50)))
```

In this section of code, first we define ^ to be the exponentiation operator. Next, we define the “munge” function, which implements an operator precedence parser with precedences defined by “prec” and right-associativity by “rassoc”[13].

The “define-syntax” part of the program declares that any expression

[13]There is a small bug in the operator precedence parser – $(x+y*z)$ goes into an infinte loop. A prize to the best solution that implements operator precedence parsing correctly.

beginning (*expression...*) should be parsed with the “munge” function, followed by the “deinfix” function from earlier. This means we can express our expressions naturally within the programming language. The next thing to note is, since our expressions are structured data, we can process them. Let’s demonstrate this by differentiating our expressions.

```scheme
(define (diff exp var)
  (cond
    ((symbol? exp) (if (eq? exp var) 1 0))
    ((number? exp) 0)
    ((equal? 3 (length exp)) (diff-binop (car exp) (cadr exp) (caddr
       exp) var))
    (else (raise (string-append "Not an expression" exp)))
) )
(define (diff-binop op a b var)
  (cond ((equal? '+ op) `(+ ,(diff a var) ,(diff b var)))
        ((equal? '- op) `(- ,(diff a var) ,(diff b var)))
        ((equal? '* op) `(+ (* ,(diff a var) ,b) (* ,a ,(diff b var)))
          )
        ((equal? '^ op) (if (number? b)
                            (if (eq? b 0) 0 (diff-binop '* a `(^ ,a
                               ,(- b 1)) var))
                            (raise "^ not a non-number")
                        ))
        (else (raise (string-append ``Can't differentiate '' (symbol->
            string op))))
)  )
```

These are some simple rules of differentiation. The “diff” function handles symbols and numbers, and the “diff-binop” function handles operators. However, the output of “diff” is full of multiplications by 1 and 0, and additions of 0. Let’s simplify the expressions it returns.

```scheme
(require scheme/match)
(define (simplify exp)
  (cond
    ((and (list? exp) (equal? 3 (length exp))) (simplify-binop (car
       exp) (cadr exp) (caddr exp)))
    (else exp)
) )
(define (simplify-binop op ina inb)
  (define a (simplify ina))
  (define b (simplify inb))
  (match (list op a b)
    ((list '+ a 0) a)
    ((list '- a 0) a)
    ((list '+ 0 b) b)
    ((list '+ a a) `(* 2 ,a))
    ((list '- a a) 0)
    ((list '* a 0) 0)
    ((list '* 0 b) 0)
    ((list '* a 1) a)
    ((list '* 1 b) b)
    ((list '^ a 0) 1)
    ((list '^ a 1) a)
    (else (list op a b))
) )

(simplify (diff (expression x ^ 2 - 2 * x + 1) 'x))
```

Here we pattern match on the binary operations, simplifying them where possible. The “match” syntax allows us to do powerful pattern matching like ML and unification like Prolog – but note that it is just a library in Lisp.

I hope this has demonstrated some of the power of Lisp for DSLs and for programming in general.

> *Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.*
>
> — Eric S. Raymond, “How to Become a Hacker”