Metaprogramming with Macros

Eugene Burmako

École Polytechnique Fédérale de Lausanne
http://scalamacros.org/

10 September 2012

The essence of macros

Macros are programmable code transformers

Example

```
(defmacro let args
  (cons
   (cons 'lambda
         (cons (map car (car args))
               (cdr args)))
   (map cadr (car args))))
```

Here we declare *let*, a Lisp function. Since it is declared as a macro, it's automatically plugged into the Lisp evaluator.

We can say that the evaluator installs a macro transformer implemented by the body of the macro into a slot named *let*.

Example

```
(defmacro let args
  (cons
   (cons 'lambda
         (cons (map car (car args))
               (cdr args)))
   (map cadr (car args))))

(let ((x 40) (y 2)) (print (+ x y)))
```

When the evaluator encounters a form that is an application of *let*, it yields control to the corresponding macro transformer, passing it the tail of the form.

Example

```
(defmacro let args
  (cons
   (cons 'lambda
         (cons (map car (car args))
               (cdr args)))
   (map cadr (car args))))

~~(let ((x 40) (y 2)) (print (+ x y)))~~

((lambda (x y) (print (+ x y))) (40 2))
```

The macro transformer takes the forms passed by the evaluator, and computes a resulting form (this is called *macro expansion*).

After that the evaluator proceeds with the form produced by macro expansion. The value of the new form is returned as the value of the original form.

Use cases

- Deeply embedded DSLs (database access, testing)
- Optimization (programmable inlining, fusion)
- Analysis (integrated proof-checker)
- Effects (effect containment and propagation)

Use cases

- Deeply embedded DSLs (database access, testing)
- Optimization (programmable inlining, fusion)
- Analysis (integrated proof-checker)
- Effects (effect containment and propagation)

Actually these use cases come from our experience with macros in Scala, which we developed this year.

All the aforementioned scenarios are either already supported by Scala macros or will be supported in vNext!

Setting the stage

In this talk we'll be looking into macros for compiled programming languages, i.e. macros as extensions to compilers.

Different combinations of junction points (function applications, code annotations, custom grammar rules, etc) and tightness of the integration (parser, namer, typer, etc) produce different challenges.

Today we're going to focus on a particular challenge in macrology: tackling syntactic abstractions. These slides discuss the ways to represent, analyze and generate code.

Outline

Revisiting *let*

```
(defmacro let args
  (cons
   (cons 'lambda
         (cons (map car (car args))
               (cdr args)))
   (map cadr (car args))))

(let ((x 40) (y 2)) (print (+ x y)))

((lambda (x y) (print (+ x y))) (40 2))
```

The code here is quite impenetrable. Without color coding it would be hard to understand the supposed structure of input and output.

Improving *let*

```
(defmacro let (decls body)
  `((lambda ,(map car decls) ,body) ,@(map cadr decls)))

(let ((x 40) (y 2)) (print (+ x y)))

((lambda (x y) (print (+ x y))) (40 2))
```

This snippet represents a shift towards declarativeness. Arguments of let are now pattern-matched in the macro signature. The shape of resulting form is encoded in a template.

Backquote, the template-building operator, is called *quasiquote*. By default code inside the quasiquote is not evaluated and is copied into the output verbatim (note that we no longer have to quote `lambda`).

Comma and comma-at are called *unquote* operators. They temporarily cancel the effect of quasiquoting, demarcating "holes" in templates.

Pushing the envelope

```
(define-syntax let
  (syntax-rules ()
    ((let ((name expr) ...) body ...)
     ((lambda (name ...) body ...) expr ...))))

(let ((x 40) (y 2)) (print (+ x y)))

((lambda (x y) (print (+ x y))) (40 2))
```

This amazing notation comes from Scheme and is dubbed *macro by example* (MBE). With the ellipsis operator it naturally captures repetitions in the input form and translates them into the output.

`syntax-rules` is a shortcut that eliminates the syntactic overhead of quasiquoting at a cost of supporting unquotes only for variables bound in a match. However MBE per se doesn't prevent arbitrary unquoting. `syntax-case`, a low-level implementation of MBE, supports all the functionality provided by quasiquotes.

Typechecking quasiquotes: strict

```
-| fun plus x = <fn y => ~x + y>;
val plus = Fn : <int> -> <int -> int>

-| val w = <fn y => ~(plus <y>)>;
val w = <(fn a => (fn b => a %+ b))> : <int -> int -> int>
```

MetaML is a statically typechecked language with quasiquotes. Its typesystem guarantees that if a meta-program is well-typed then all possibly produced object-programs will be well-typed as well.

Unlike in lisps, quasiquotes in MetaML cannot have unbound free variables, which effectively means that meta-programs cannot introduce new bindings (e.g. it's not possible to express *let* as a macro). This is a sacrifice to the altar of strong type safety guarantees.

Typechecking quasiquotes: strict

```
mac (let seq x = e1 in e2 end) =
   $(let val x = ?e1 in ?e2 end)
```

MacroML is a macro system built on top of MetaML. It inherits typing discipline of its parents, but introduces a clever way to create certain bindings without compomising type safety.

In the example above `$` stands for "delay" and `?` stands for "force". Macro `seq` implements (wtf does it implement?).

To typecheck the fluent binding to `x`, MacroML tracks the name of the bindee parameter in the type of the body parameters such as ei.

Typechecking quasiquotes: lenient

```
macro using(name, expr, body) {
  <[
    def $name = $expr;
    try { $body } finally { $name.Dispose() }
  ]>
}

using(db, Database("localhost"), db.LoadData())
```

Nemerle allows quasiquotes to contain free variables, it even permits splicing into binding positions (like in `def $name = $expr`).

This makes it impossible to typecheck quasiquotes at their declaration site, but the result of macro expansion that uses such quasiquotes is typechecked anyways, so errors are still caught at compile-time.

Relaxed typechecking rules like in Nemerle have proven to be essential for a practical metaprogramming system.

Typechecking quasiquotes: the best of both worlds

The ideal typechecking discipline would allow programmers to freely mix typed and untyped quasiquotes.

In F# both quasiquoting (`<@...@>`) and splicing (%) operators have untyped versions (`<@@...@@>` and %% respectively).

Unfortunately they are only untyped in a sense that untyped splicing allows `Expr` instead of `Expr<'a>`. Untyped quasiquotes do perform typechecking using type inference to determine type bounds for holes, which are verified at runtime. As a result, `<@@ (%%x).Name @@>` won't typecheck, whereas `<@@ (%%x).ToString() @@>` will fail at runtime.

In a 15 year old paper, Mark Shields, Tim Sheard and Simon Peyton Jones propose a type system that can defer type inference to runtime. Their findings can probably be adapted to the task at hand.

Outline

Outline

Outline

Outline

Outline