# 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

## Motivating example

```
(defmacro or (x y)
  '(let ((t ,x))
     (if t t ,y)))

(or 42 "strange")
```

One of the appealing features of macros is their ability to control evaluation order. For example, it's possible to write a short-circuiting or macro.

In this example, however, I would like to highlight the temporary variable that had to be introduced in order to prevent double evaluation.

The implementation is very simple. Yet it has two mistakes that can lead to subtle bugs. Let's take a closer look.

# Mistake #1

```
(defmacro or (x y)
  `(let ((t ,x))
     (if t t ,y)))

(let ((t #t))
  (or #f t))
```

The first bug manifests itself when there's a variable named t at the call site, and it's used in the second argument of the or macro. We can say that a variable introduced during macro expansion shadows a binding from the macro use site.

# Hygiene

*Hygiene Condition for Macro Expansion.*

*Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step.*

—Eugene Kohlbecker
PhD Thesis

## Breaking hygiene

```
(define-syntax forever
  (syntax-rules ()
    ((forever body ...)
     (call/cc (lambda (abort)
                (let loop () body ... (loop)))))))

(forever (print 4) (print 2) (abort))
```

At times, though, it's useful to break hygiene and meld lexical contexts from different transcription steps.

We've seen one such example, when studying the using macro written in Nemerle. Here's another one that features a DSL for writing loops. Whenever we're inside a loop introduced by our DSL, we should be able to use abort to quit.

For this snippet to work, the binding created by forever must be non-hygienic, since abort is supposed to be accessible from body.

# Mistake #2

```
(defmacro or (x y)
  '(let ((t ,x))
      (if t t ,y)))

(let ((let print))
  (or #f #t))
```

The second bug is, in a sense, dual to the first one. Previously, a binding introduced by a macro interfered with a binding at the macro use site. Now a binding introduced at the call site destroys a binding at the macro definition site.

# Referential transparency

*Macros defined in the high-level specification language are
referentially transparent in the sense that a macro-introduced
identifier refers to the binding lexically visible where the macro
definition appears rather than to the top-level binding or to the
binding visible where the macro call appears.*

—Kent Dybvig et al.
Syntactic Abstractions in Scheme

# State of the art

- ▶ Typical approaches to fixing the aforementioned problems are alpha-renaming and gensymming (from `gensym`).

- ▶ There are frameworks that provide hygiene and referential transparency automatically. Scheme is especially famous for pushing this research.

- ▶ These automatic frameworks usually have enough flexibility to allow for the use cases that require non-hygienic bindings.

# Outline

## printf

```
printf :: String -> Expr
printf s = gen (parse s) [| "" |]

gen :: [Format] -> Expr -> Expr
gen [] x = x
gen (D : xs) x = [| \n -> $(gen xs [| $x ++ show n |]) |]
gen (S : xs) x = [| \s -> $(gen xs [| $x ++ s |]) |]
gen (L s : xs) x = gen xs [| $x ++ $(lift s) |]

$(printf "Error: %s on line %d") msg line

(\s0 -> \n1 ->
  "Error: " ++ s0 ++ " on line " ++ show n1)
```

In Template Haskell macros don't need a special declaration form (unlike in Lisp, they are first-class), at a cost of requiring an explicit trigger for macro expansion.

# printf

```
printf :: String -> Expr
printf s = gen (parse s) [| "" |]

gen :: [Format] -> Expr -> Expr
gen [] x = x
gen (D : xs) x = [| \n -> $(gen xs [| $x ++ show n |]) |]
gen (S : xs) x = [| \s -> $(gen xs [| $x ++ s |]) |]
gen (L s : xs) x = gen xs [| $x ++ $(lift s) |]

$(printf "Error: %s on line %d") msg line

(\s0 -> \n1 ->
  "Error: " ++ s0 ++ " on line " ++ show n1)
```

Template Haskell has the notion of quasiquoting and unquoting (called *splicing* in authors' terms). Splicing outside quasiquotes (or an explicit call to splice) triggers macro expansion.

# printf

```
printf :: String -> Expr
printf s = gen (parse s) [| "" |]

gen :: [Format] -> Expr -> Expr
gen [] x = x
gen (D : xs) x = [| \n -> $(gen xs [| $x ++ show n |]) |]
gen (S : xs) x = [| \s -> $(gen xs [| $x ++ s |]) |]
gen (L s : xs) x = gen xs [| $x ++ $(lift s) |]

$(printf "Error: %s on line %d") msg line

(\s0 -> \n1 ->
  "Error: " ++ s0 ++ " on line " ++ show n1)
```

Quasiquotes automatically alpha-rename introduced identifiers to ensure
hygiene. Template Haskell also solves the referential transparency problem
by implementing cross-stage persistence for static symbols.

## printf

```
printf :: String -> Expr
printf s = gen (parse s) [| "" |]

gen :: [Format] -> Expr -> Expr
gen [] x = x
gen (D : xs) x = [| \n -> $(gen xs [| $x ++ show n |]) |]
gen (S : xs) x = [| \s -> $(gen xs [| $x ++ s |]) |]
gen (L s : xs) x = gen xs [| $x ++ $(lift s) |]

$(printf "Error: %s on line %d") msg line

(\s0 -> \n1 ->
  "Error: " ++ s0 ++ " on line " ++ show n1)
```

For non-static symbols, general case of cross-stage persistence is
undecidable. However if the programmer knows how to reproduce a certain
datum at runtime, he implements the Lift typeclass and calls lift.

## Bindings

```
cross2a :: Expr -> Expr -> Expr
cross2a f g = [| \(x,y) -> ($f x, $g y) |]

cross2c :: Expr -> Expr -> Expr
cross2c f g =
  do { x <- gensym "x"
     ; y <- gensym "y"
     ; ft <- f
     ; gt <- g
     ; return (Lam [Ptup [Pvar x, Pvar y]]
                   (Tup  [App ft (Var x)
                         ,App gt (Var y)]))
     }
```

At the low level of Template Haskell lie plain ADTs that represent program fragments. Quotation monad can make low-level ASTs hygienic (but also may ignore hygiene). High level of syntactic abstraction is represented by quasiquotes, which translate to the hygienic quotation monad.

# Typechecking

▶ Quasiquotes are expanded in *re*namer, because later in the pipeline bindings are frozen.

▶ Every quotation is then sanity-checked in typer to reject nonsense like [| "hello" && True |]. Having passed the check, quotations pass the type check and are assigned the Q Exp type.

▶ After quotations are typechecked, macros are expanded, and the compilation goes on just as if the programmer had written the expanded program in the first place.

A retrospective write-up from TH developers indicates that:

▶ There is a real need for strongly typed quotes TExp a (for more robustness).

▶ Normal quotes shouldn't be typechecked at all (to allow for declaration-introducing splices inside quasiquotes).

# Outline

# Outline

# Outline