Metaprogramming with Macros

Eugene Burmako

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

10 September 2012

Macros

Macros

Macros realize the notion of textual abstraction.

Textual abstraction:

- Recognize pieces of text that match a specification
- Replace them according to a procedure

Why macros?

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

Outline

`if` in Lisp

```
(if (calculate)
  (print "success")
  (error "does not compute"))
```

Anaphoric `if`

```
(aif (calculate)
  (print it)
  (error "does not compute"))
```

```
(let* ((temp (calculate))
       (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

The aif macro

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro aif args
```

```
(let* ((temp (calculate))
       (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Quasiquoting: static template

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro aif args
       `(let*              ((temp .............)
                            (it  temp))
          (if  temp
             ..............
             ..............)))

(let* ((temp (calculate))
       (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Quasiquoting: dynamic holes

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro aif args
       `(let*              ((temp ,(car args))
                            (it  temp))
          (if  temp
             ,(cadr args)
             ,(caddr args))))

(let* ((temp (calculate))
       (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Macro by example (MBE)

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro+ aif
  (aif cond then else)
  (let* ((temp cond)
         (it temp))
    (if temp
        then
        else)))

(let* ((temp (calculate))
       (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Interlude

- Macros are regular functions that happen to work with syntax objects
- Quasiquotes = static templates + dynamic holes

Outline

Anaphoric `if`

```
(defmacro+ aif
  (aif cond then else)
  (let* ((temp cond)
         (it temp))
    (if temp then else)))
```

- So far macros are simple: define a function, recognize pieces of text and replace them with a template
- This is so immediately useful, that we could wrap up right now

But actually

The `aif` macro has two bugs

What's wrong?

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro+ aif
  (aif cond then else)
  (let* ((temp cond)
         (it temp))
    (if temp then else)))
```

What's wrong?

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro+ aif
  (aif cond then else)
  (let* ((temp cond)
         (it temp))
    (if temp then else)))
```

What's wrong?

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(defmacro+ aif
  (aif cond then else)
  (let* ((temp cond)
         (it temp))
    (if temp then else)))

(let* ((temp (calculate))
       (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Bug #1: Violation of hygiene

```
(let ((temp 451°F))
  (aif (calculate)
    (print it)
    (print temp)))

(defmacro+ aif
  (aif cond then else)
  (let* ((temp cond)
         (it temp))
    (if temp then else)))

(let ((temp 451°F))
  (let* ((temp (calculate))
         (it temp))
    (if temp
      (print it)
      (print temp))))
```

Bug #2: Violation of referential transparency

```
(let ((if hijacked))
  (aif (calculate)
    (print it)
    (error "does not compute")))

(defmacro+ aif
  (aif cond then else)
  (let* ((temp cond)
         (it temp))
    (if temp then else)))

(let ((if hijacked))
  (let* ((temp (calculate))
         (it temp))
    (if temp
      (print it)
      (error "does not compute"))))
```

Interlude

- Cross-pollination of scopes can lead to inadvertent variable capture
- Violation of hygiene = def site harms call site
- Violation of referential transparency = call site harms def site

Outline

Template Haskell

```
$(aif [| calculate |]
   [| putStrLn (show it) |]
   [| error "does not compute" |])

aif :: Q Exp -> Q Exp -> Q Exp -> Q Exp
aif cond then' else' =
   [| let temp = $cond
          it = temp
      in if temp /= 0 then $then' else $else' |]
```

Template Haskell

```
$(aif [| calculate |]
   [| putStrLn (show it) |]
   [| error "does not compute" |])

aif :: Q Exp -> Q Exp -> Q Exp -> Q Exp
aif cond then' else' =
   [| let temp = $cond
          it = temp
      in if temp /= 0 then $then' else $else' |]

let temp_a1mx = calculate
    it_a1my = temp_a1mx
in if (temp_a1mx /= 0)
   then putStrLn (show it)
   else error "does not compute"

Not in scope: 'it'
```

The Q monad

```
aif cond then' else' =
  [| let temp = $cond
         it = temp
     in if temp /= 0 then $then' else $else' |]
```

The Q monad

```
aif cond then' else' =
  [| let temp = $cond
         it = temp
     in if temp /= 0 then $then' else $else' |]

aif :: Q Exp -> Q Exp -> Q Exp -> Q Exp
aif cond' then'' else'' =
    do { temp <- newName "temp"
       ; it <- newName "it"
       ; cond <- cond'
       ; then' <- then''
       ; else' <- else''
       ; let notEq = mkNameG_v "ghc-prim" "GHC.Classes" "/="
         in return
          (LetE [ValD (VarP temp) (NormalB cond) [],
                 ValD (VarP it) (NormalB (VarE temp)) []]
                (CondE (... (VarE notEq) ...) then' else'))
       }
```

Breaking hygiene

```
$(aif [| calculate |]
   [| putStrLn (show $(dyn "it")) |]
   [| error "does not compute" |])

aif :: Q Exp -> Q Exp -> Q Exp -> Q Exp
aif cond then' else' =
   [| let temp = $cond
          it = temp
      in if temp /= 0 then $then' else $else' |]

let temp_a1mx = calculate
    it_a1my = temp_a1mx
in if (temp_a1mx /= 0)
   then putStrLn (show it_a1my)
   else error "does not compute"
```

Summary: Template Haskell

- Quasiquotes in Template Haskell are automatically hygienic and referentially transparent
- That's because they are lifted into the Q monad, which takes care of names (fresh names for locals, fully qualified names for globals)
- Sometimes we need to break hygiene

Nemerle

```
aif(calculate,
  WriteLine(it),
  throw Exception("does not compute"))

macro aif(cond, then, else_) {
  <[
    def temp = $cond;
    def it = temp;
    if (temp != 0) $then else $else_
  ]>
}
```

Nemerle

```
aif(calculate,
  WriteLine(it),
  throw Exception("does not compute"))

macro aif(cond, then, else_) {
  <[
    def temp = $cond;
    def it = temp;
    if (temp != 0) $then else $else_
  ]>
}

def calculate = 42;
def temp_1087 = calculate;
def it_1088 = temp_1087;
if (temp_1087 != 0) WriteLine(it) else throw Exception("...")

error: unbound name ‘it’
```

Coloring algorithm

```
def calculate = 42;
aif(calculate,
  WriteLine(it),
  throw Exception("does not compute"))

macro aif(cond, then, else_) {
  <[
    def temp = $cond;
    def it = temp;
    if (temp != 0) $then else $else_
  ]>
}

def calculate = 42;
def temp = calculate;
def it = temp;
if (temp != 0) WriteLine(it) else throw Exception("...")
```

Coloring algorithm: normal code gets a vanilla color

```
def calculate = 42;                          // vanilla color
aif(calculate,
  WriteLine(it),
  throw Exception("does not compute"))

macro aif(cond, then, else_) {
  <[
    def temp = $cond;
    def it = temp;
    if (temp != 0) $then else $else_
  ]>
}

def calculate = 42;
def temp = calculate;
def it = temp;
if (temp != 0) WriteLine(it) else throw Exception("...")
```

Coloring algorithm: each expansion gets unique colors

```
def calculate = 42;                          // vanilla color
aif(calculate,
  WriteLine(it),
  throw Exception("does not compute"))

macro aif(cond, then, else_) {               // expansion color
  <[
    def temp = $cond;
    def it = temp;
    if (temp != 0) $then else $else_
  ]>
}

def calculate = 42;
def temp = calculate;
def it = temp;
if (temp != 0) WriteLine(it) else throw Exception("...")
```

Coloring algorithm: at the end of the day

```
def calculate = 42;                          // vanilla color
aif(calculate,
  WriteLine(it),
  throw Exception("does not compute"))

macro aif(cond, then, else_) {               // expansion color
  <[
    def temp = $cond;
    def it = temp;
    if (temp != 0) $then else $else_
  ]>
}

def calculate = 42;                          // bind using colors
def temp = calculate;
def it = temp;
if (temp != 0) WriteLine(it) else throw Exception("...")
```

Coloring algorithm: inherit use site

```
def calculate = 42;                          // vanilla color
aif(calculate,
  WriteLine(it),
  throw Exception("does not compute"))

macro aif(cond, then, else_) {               // expansion color
  <[
    def temp = $cond;
    def $("it": usesite) = temp;
    if (temp != 0) $then else $else_
  ]>
}

def calculate = 42;                          // bind using colors
def temp = calculate;
def it = temp;
if (temp != 0) WriteLine(it) else throw Exception("...")
```

Summary: Nemerle

- Nemerle takes care of hygiene with a coloring algorithm of impressive simplicity and power
- No complex translation algorithms are necessary
- As another bonus programmer can fine-tune colors with `MacroColors`
- Referential transparency works as well

Racket

A Lisp, descendent from Scheme

25 years of hygienic macros, a bunch of macro systems

Language features written using macros (classes, modules, etc)

Q: "How to turn macros into proper abstractions?"

Racket

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(define-syntax (aif stx)
  (syntax-case stx ()
    ((aif cond then else)

      #'(let ((temp cond)
              (it temp)))
          (if temp then else)))))
```

Racket

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(define-syntax (aif stx)
  (syntax-case stx ()
    ((aif cond then else)

      #'(let ((temp cond)
              (it temp)))
          (if temp then else)))))

(let* ((temp (calculate))
       (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Racket

```
(aif (calculate)
  (print it)
  (error "does not compute"))

(define-syntax (aif stx)
  (syntax-case stx ()
    ((aif cond then else)
     (with-syntax ((it (datum->syntax #'aif 'it)))
       #'(let ((temp cond)
               (it temp)))
           (if temp then else))))))

(let* ((temp (calculate))
       (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Doesn't scale

```
(aunless (not (calculate))
  (print it)
  (error "does not compute"))

(define-syntax (aunless stx)
  (syntax-case stx ()
    ((aunless cond then else)
     #'(aif (not cond) then else))))

(let* ((temp (not (not (calculate))))
       (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Doesn't scale

```
(aunless (not (calculate))
  (print it)
  (error "does not compute"))

(define-syntax (aunless stx)
  (syntax-case stx ()
    ((aunless cond then else)
     #'(aif (not cond) then else))))

(let* ((temp (not (not (calculate))))
       (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Doesn't scale

```
(aunless (not (calculate))
  (print it)
  (error "does not compute"))

(define-syntax (aunless stx)
  (syntax-case stx ()
    ((aunless cond then else)
     #'(aif (not cond) then else))))

(let* ((temp (not (not (calculate))))
       (it temp))
  (if temp
    (print it)
    (error "does not compute")))
```

Solution: dynamic variables

```
(define-syntax-parameter it (syntax-rules ()))

(define-syntax (aif stx)
  (syntax-case stx ()
    ((aif cond then else)
     #'(let ((temp cond))
         (syntax-parameterize
         ((it (syntax-rules () ((_) temp))))
           (if temp then else))))))
```

Summary: overall

- There are algorithms that take care of hygiene and referential transparency
- These algorithms can work in automatic mode, but are flexible enough to give the programmer full control
- Like a silver bullet
- Nevertheless sometimes even better solutions come from integration with language features

Outline

scalamacros.org

- Since this semester Scala has macros
- Even better: macros are an official part of the language in the next production release 2.10.0
- Now it's time to put the pens down and think about the future

Implicits

```
def serialize[T](x: T): Pickle
```

Implicits

```
trait Serializer[T] {
  def write(pickle: Pickle, x: T): Unit
}

def serialize[T](x: T)(s: Serializer[T]): Pickle
```

Implicits

```
trait Serializer[T] {
  def write(pickle: Pickle, x: T): Unit
}

def serialize[T](x: T)(implicit s: Serializer[T]): Pickle

implicit object ByteSerializer extends Serializer[Byte] {
  def write(pickle: Pickle, x: Byte) = pickle.writeByte(x)
}
```

Implicits

```
trait Serializer[T] {
  def write(pickle: Pickle, x: T): Unit
}

def serialize[T](x: T)(implicit s: Serializer[T]): Pickle

implicit def generator: Serializer[T] = macro impl[T]
```

Research proposal

Marry macros and high-level language features:

- Macros + functions = programmable inlining, specialization, fusion
- Macros + annotations = code contracts
- Macros + path-dependent types = controlled effects
- Macros + implicits = static verification
- ...