Metaprogramming with Macros

Eugene Burmako
LAMP, I&C, EPFL

***Abstract*—Macros realize the notion of *textual abstraction*. Textual abstraction consists of recognizing pieces of text that match a specification and replacing them according to a procedure.**

In the focus of the study is semantics of syntactic macros in lexically scoped programming languages. We highlight the problems of *hygiene* and *referential transparency* and describe the solutions employed by Template Haskell [1], Nemerle [2] and Racket [3].

We discuss integration of hygienic macros into statically typed languages and propose to improve upon state of the art by providing a flexible type system for syntax templates and uncovering synergies with high-level language features such as path-dependent types and implicits [4].

***Index Terms*—metaprogramming, macros, quasiquotes, hygiene, referential transparency**

I. Introduction

PROCEDURAL abstraction is pervasive. Factoring out parameterized fragments of programs into procedures is a conventional best practice.

Modern programming languages integrate the notion of procedures into their semantics. Procedures are viewed as independent programs that can communicate with the main program. As of such they can be manipulated as units, and big procedures can be built from the smaller ones. This is a powerful way to manage complexity of software systems.

Proposal submitted to committee: September 3rd, 2012; Candidacy exam date: September 10th, 2012; Candidacy exam committee: Christoph Koch, Martin Odersky, Viktor Kuncak.

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(name and signature)

Thesis director: _____
(name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(R. Urbanke) (signature)

EDIC-ru/05.05.2009

However procedural abstraction is sometimes not expressive enough, because its manifestations are bound by language syntax and it operates within the semantics of the language.

For example, in most programming languages it is impossible to define short-circuiting logical operators as procedures, because procedures are usually not in control of evaluation order. Another example is a C-like `for` loop, which supports optional prologue that introduces variables visible in its body. Procedures typically cannot abstract over variable bindings, so they cannot readily express this language construct.

Textual abstraction consists of recognizing pieces of text that match a specification and replacing them according to a procedure. Matched fragments are called macro calls or macro applications, and procedures that transform them are dubbed macros or macro transformers [5]. In general sense these procedures can also be referred to as meta-programs [6]. The process of applying macros is called macro expansion.

Textual abstraction provides programmers with a multitude of techniques. Just to name a few of them:

- reification (having code as data),
- language virtualization (overloading/overriding language semantics to enable deep embedding of DSLs),
- domain-specific optimization (application of optimizations such as inlining or fusion based on knowledge about the program being optimized),
- static verification (using reified representation of the program and, possibly, its contracts defined alongside the program, to verify invariants at compile time),
- algorithmic program construction (generation of code that is tedious to write with supported abstractions).

One possibility to implement a macro system is having it as a standalone tool operating on character streams. This gives rise to lexical macros. Such a design warrants simplicity of the implementation, but undermines robustness, because macros operating on lexical level have no mechanism that prevents generation of syntactically invalid programs.

Alternative approach is to integrate a macro expander into the compiler and have macros work with syntax trees, introducing *syntactic macros*. In this model macro application is a node in the program tree, and macro expansion produces a new node that replaces the macro application without distorting the structure of the program.

A significant body of recent research in textual abstraction has been done on syntax extensibility ([**?**,**?**,**?**] to name only a few). This paper however focuses on semantics of macro-enabled languages, namely on ways to prevent syntactic macros from expanding into nonsense.

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

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

a) Examples of a macro application and a macro expansion of the `let` macro

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

b) Low-level creation of syntax objects (the code manipulates S-expressions with standard symbol and list processing functions)

```
(defmacro let (decls body)
 `(
    (lambda
      ,(map car decls)    ;; (x y)
      ,body               ;; (print (+ x y))
     ,@(map cadr decls))) ;; (40 2)
```

c) Quasiquotation [8] (backquote introduces a static code template, commas (also known as unquotes) insert dynamic values into the template)

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

d) Macro-by-example notation [9] (uses a tree matching macro that can extract and reassemble fragments of syntax objects; ellipses capture recurrent parts of the input)

```
(define-syntax (let stx)
  (syntax-parse stx
    ((let ((name:identifier expr:expr) ...) body:expr ...)
     #:fail-when (check-duplicate #'(name ...))
                 "duplicate variable name"
     #'((lambda (name ...) body ...) expr ...))))
```

e) Macro-by-example notation augmented with a syntax specification [10] (colons denote syntax classes, which are first-class and can be built from the ground up).

Figure 1. Assorted implementations of the `let` macro in Lisp dialects

II. Background

A. Notation

`let` is a language construct typical to functional programming. It introduces a scope for a computation and brings temporary variables with provided values into that scope. To implement `let` the compiler might wrap the computation in a lambda abstraction and apply it right away (Figure 1a).

This notion cannot be abstracted procedurally, because the body of the computation typically contains free variables. To the contrast, textual abstraction can help, because macros manipulate the program on a level where bindings don't exist and therefore don't impose restrictions.

The most straightforward implementation of `let` is a low-level macro transformer (Figure 1b). It takes an S-expression that represents a macro application, destructures it using standard list manipulation functions, such as `car` and `cdr`, and creates a new S-expression with `cons`. Even in this simple example the notation is very noisy. It's quite difficult to figure out the expected shapes of input and output expressions from the imperative algorithm.

Quasiquotes [8] make it possible to reduce obscurity of the macro by providing a domain-specific language for syntax templates (Figure 1c). The quasiquote operator (`` ` ``) demarcates a static template. Quasiquoted code is inserted verbatim into the output (that's why there's no longer need in explicit `cons`'ing). Unquote operators (`,` and `,@`) temporatily interrupt a quasiquote, creating "holes" filled in with dynamically calculated data. For example, for `let` we statically know the shape of code to produce (an application of a lambda abstraction) - this makes up the static part of the quasiquote. On the other hand, body and parameters of the lambda as well as the arguments of the application may vary from expansion to expansion - this is the dynamic part.

```
(defmacro or (x y)
  '(let ((temp ,x))
     (if temp temp ,y)))

(or 42 "the result is 42")
```

a) A simple yet erroneous implementation of the `or` macro

```
(let ((temp "451 Fahrenheit"))
  (or null temp))

(let ((temp "451 Fahrenheit"))
  (let ((temp null))
    (if temp temp temp)))
```

b) Violation of hygiene [5]: binding established during expansion affects call site

```
(let ((if hijacked))
  (or true false))

(let ((if hijacked))
  (let ((temp true))
    (if temp temp false)))
```

c) Violation of referential transparency [7]: binding established during expansion is affected by call site

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

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

d) Intentional variable capture: infinite looping construct `forever` provides a predefined identifier to break from the loop. `abort` is introduced during macro expansion, but it should be visible to the body of the loop, which comes from the macro call site.

Figure 2. Intentional and unintentional variable capture

Another simplification of the macro can be achieved with MBE, the macro-by-example notation [9]. In their seminal work Kohlbecker and Wand came up with a specification of a pattern matcher that extracts singular and repetitive parts of S-expressions. Identifiers which appear in the input pattern are treated as pattern variables. Ellipses (`...`) used as a last element of a list that contains pattern variables denote repetition. Ellipses can also be nested capturing pattern variables as lists of arbitrary depth. The revised version of the `let` macro clearly shows the shapes of the input and the output and the relation between them (Figure 1d).

A recent development of the MBE syntax has been proposed by Culpepper and Felleisen [10]. Their refinement addresses the need for principled input validation and error reporting. Indeed, MBE covers the success path, but doesn't help with detecting errors. For example, duplicate identifier names as in `(let ((x 40) (x 2)) (print (+ x y))` will go unnoticed until the compiler gets to the resulting lambda form, which will produce confusing error messages. Authors enhance MBE with both declarative and procedural means of validation (Figure 1e). In the example colons next to the names of pattern variables denote syntax classes, which put restrictions on the shape of the variables and the `#:fail-when` clause contains imperative validation code and error messages. These validation facilities can be packed into custom syntax classes, which can be built from the ground up.

B. Bindings

Syntactic macros operate on ASTs, so they cannot produce syntactically invalid code, but nothing prevents such macros from making semantic errors, i.e. expanding into syntactically valid nonsense. The most obvious mistakes will result in type errors, however there's an entire class of oversights that might silently change the behavior of the program.

When writing procedures in lexically scoped languages, programmers typically don't need to think about name clashes between variables declared inside procedures and at their call sites. Except for recursion, scopes of procedure bodies and procedure call sites are different, so neither variables defined inside the procedures can change the semantics at the call site, nor vice versa.

In a macro-enabled language, especially in presense of quasiquotes which make macros look like normal procedures, this intuition becomes compromised, because after macro expansion the scopes of macro definition and macro call site are mechanically merged by the expander.

Conider the `or` macro shown on Figure 2a. This macro picks one of its two arguments based on the truthiness of the first argument. To prevent double evaluation, the macro introduces a temporary variable `temp` that holds the result of evaluation of the first argument. The temporary variable is then tested with `if` , which selects the resulting value.

As simple as this code can be, it is also incorrect, having two potential bugs.

The first problem happens when the call site defines its own variable named `temp` and relies on it in the second argument of a call to `or` (Figure 2b). After the expansion, two `temp`s clash, which produces incorrect results if the first argument of the call is falsy. This problem is dubbed *hygiene violation*, and the macro is said to introduce the temporary variable *unhygienically*.

In his thesis [5] Kohlbecker defines the hygiene condition and presents a macro system that automatically prevents such naming collisions:

> Hygiene Condition for Macro Expansion.
> Generated identifiers that become binding instances in the completely expanded program must bind only identifiers that are generated during the same transcription step.

The second problem happens when the call site redefines one of the procedures or macros used in the expansion. For example, on Figure 2c the call site hijacks the meaning of `if`, which destroys the original intent of the macro author. This is a *referential opaqueness* problem.

Dybvig et al. [7] build on Kohlbecker's work and devise a macro expander that automatically avoids this class of errors:

> Macros defined in [our] 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.

This notion is also called *cross-stage persistence*, because bindings in place at the compilation stage are persisted into the runtime stage. When viewed in this light, it becomes clear why referential transparency is usually supported only for top-level definitions. Top-level values are uniquely identified by their compile-time signatures (e.g. a static method in Java can be addressed by a full name of a class, name of the method and types of its parameters). To the contrast local values are represented by transient state (contents of processor registers, stack, heap etc), which in general case cannot be persisted.

Despite the conveniency of automatic segregation of the scopes of macro definitions and macro call sites, at times it is necessary to have them melded. A looping macro `forever` from Figure 2d demonstrates the need for this. Expansion of this macro is an infinite loop that provides a magic identifier `abort` to exit the loop. In this case automatic hygiene facility will do harm, making the body of the loop unable to see the loop control lever.

Because of similar situations some programmers argue in favor of manual control over the scoping discipline, proposing manual (or macro-powered!) renaming of identifiers whose capture would be undesired [11,12].

Another popular line of thought favors further empowerment of macro systems to minimize the necessity for low-level tinkering. In fact, one of the macro systems reviewed below provides a design pattern [3] that solves the `abort` challenge in a fully hygienic way (i.e. without introducing identifiers with manually provided names).

C. Typechecking

An important feature of macros is that macro expansion happens before the code is executed (below in the paper this phase is referred to as *compile-time*, despite that, strictly speaking, interpreted languages can also support macros). Therefore macro expansions can be checked for absense of semantic errors (to the extent supported by the language) in an automatic fashion, without requiring effort from the programmer.

What's even better is that there is another line of defense. Macros operate on syntax objects, which are snippets of the code to be, thus it might be a good idea to typecheck the code represented by these snippets in advance. This amounts to extending typechecking to the before-compile-time phase (as in compile-time of macros themselves).

One approach is to treat quasiquotes similarly to functions, requiring all free variables to be bound in the enclosing lexical scope and assigning quasiquotes a definitive type (Figure 3a). This strategy is used in MetaML [13], a statically typechecked language with special support for program generation.

All quasiquotes in MetaML are typechecked individually and receive one of the `<a>` types (which means "code that evaluates to a value of type `a`"). For example, on Figure 3a the quasiquote in the body of `pow` represents a function from `int` to `int`, so it has the `<int -> int>` type.

Programs in MetaML can generate and run programs at runtime, these programs can do the same, and so on. Quasiquotes here represent templates of the programs to be generated and delineate execution stages. Due to its specific nature, MetaML requires strong guarantees from the type system. At runtime, if program generation fails because of an error in the meta-program, it's too late to report typechecking errors.

Unfortunately this means that MetaML has to give up the freedom of arbitrary manipulations of syntax objects, the most important of which are destructuring and introduction of new bindings. (e.g. in MetaML its impossible to express `let` as a macro).

MacroML [14,15], a macro system built atop of MetaML, provides a clever way to partially alleviate this restriction, letting the programmer introduce new binding constructs in a very controlled setting. This is achieved by having the type system track not only the regular environments containing declared macros and variables, but also a so called body parameter environment, which remembers which parts of quasiquotes see which variables introduced by those quasiquotes.

This still doesn't allow arbitrary manipulations, e.g. similarly to MetaML there is no way to pull apart a syntax object and reassemble it using a quasiquote as a template (as done in the `let` macro on Figure 1).

Therefore later research on macros focused on relaxing the typechecking discipline of MetaML.

```
-| fun pow n = <fn x =>
    ~(if n = 0 then <1> else <x * (~(pow (n - 1)) x)>)>;
val pow = fn : int -> <int -> int>

-| val cube = (pow 3);
val cube = <(fn a => x %* x %* x %* 1)> : <int -> int>

-| (run cube) 5;
val it = 125 : int
```

a) Strict typechecking [13]: each quasiquote is typechecked individually and is assigned a "code of something" type (`<a>` stands for code that evaluates to a value of type `a`).

```
[| 'a' + True |] -- rejected

printf :: String -> Expr -- allowed
$(printf "Error: %s on line %d") "urk" 341

f :: Q Type -> Q [Dec] -- rejected
f t = [d| data T = MkT $t; g (MkT x) = g+1 |]
```

b) Lenient typechecking [1]: quasiquotes are sanity checked to prevent obvious errors and are required to have their bindings established before macro expansions kick in. No additional checks are done, and all quasiquotes get the same type-agnostic `Expr` type.

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

c) No typechecking [2]: quasiquotes are not typechecked at all until a macro that uses them expands, after which the result is typechecked as a whole. This provides maximum freedom, permitting even unquotes in binding positions (like in `def $name`).

Figure 3. Typechecking strategies for quasiquotes

Template Haskell [1] employs a lenient typechecking strategy for meta-programs (but not for expanded programs, which are typechecked with the usual rigor of Haskell).

Unlike in MetaML, the type of syntax objects is non-polymorphic - `Expr` , not `Expr a` - and Template Haskell doesn't try to provide up-front guarantees that meta-programs produce well-typed code. For example, well-typedness of the `printf` macro application (Figure 3b) is checked on the spot, immediately after the expansion, not ahead of time.

However Template Haskell does require the bindings of variables used in quasiquotes to be resolved statically. This is done to enforce the "static scope is absolute" design principle (which is a rehash of hygiene and referential transparency conditions outlined above) as stated in [16]:

> If an occurrence of a variable `x` looks as if it is lexically bound to an enclosing binding for `x`, then it is so bound, and no Template-Haskell transformation can threaten that binding.

Unfortunately this lightweight typecheck has proven to impose unnecessary limitations. For example, the meta-program `f` from Figure 3b that uses another meta-program `t` to insert a type declaration in a quasiquote is rejected. This happens because the compiler cannot prove that the usage of this type declaration (`MkT x`) is correct for all possible `t`s.

It is due to this reason that Nemerle [2], a macro-enabled language inspired by Template Haskell, has ceased typechecking quasiquotes altogether. As we will see below this doesn't prevent the macro expander in Nemerle from being hygienic, but with respect to typechecking this brings the evolution to square one.

III. Case study

This section discusses Template Haskell [1], Nemerle [2] and Racket [3] in the light of hygiene and referential transparency.

Syntax objects in all three presented macro systems can be manipulated both at low level (using raw construction of corresponding data structure) and at high level (with the help of quasiquotation facilities).

However in this survey we will only be studying and evaluating high-level notations and APIs. Low-level facilities are undoubtedly useful for getting things done, but our intent here is to get to the bottom of abstractions introduced by the scrutinees.

We will also take for granted the ability of meta-programs to ask the expander for the information about the parts of the program that lie outside of a macro application current to a given expansion (the `reify` family of functions in Template Haskell, `Macros.ImplicitCTX` in Nemerle and the `syntax-local` family of functions in Racket) or for metadata of syntax objects (e.g. their types, scopes, etc).

Another aspect that we will omit from consideration is the way the macro expander acquires and loads binary code of macros. For the record, all three reviewed languages are compiled, but they differ in their treatment of the situation. Template Haskell and Nemerle require separate compilation, whereas Racket allows joint compilation with extra precautions against cross-stage pollution of top-level scope [17].

Finally, in Nemerle and Racket macros can change the syntax of the language. As mentioned in the introduction, syntax extensibility is a lively research topic, but it is outside the scope of this study.

A. Template Haskell

Template Haskell is a macro-enabled extension to a statically typed functional programming language. Macro transformers in Template Haskell are represented by regular functions that produce data of type `Expr`. Macro expansion is triggered explicitly by unquoting outside quasiquotes - either with `$` which unquotes expressions, or with `splice` which unquotes declarations.

```
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
```

The snippet above illustrates the introduced concepts. `printf` and `gen` are functions that operate on syntax objects. In particular, `printf` generates a lambda abstraction from a C-like string format specification using helpers `parse` (whose declaration is not present in a snippet) and `gen`. Quasiquotes are encoded with `[|...|]` brackets and can contain arbitrary Haskell code (to be precise, the notation for quoted patterns, declarations and types is slightly different, but it doesn't make much difference in the context of this discussion):

Quasiquotes in Template Haskell are hygienic and referentially transparent. The former is achieved by renaming bindings introduced in quasiquotes, while the latter is warranted by keeping track of original names that unambiguously refer to top-level variables, even if such variables are not visible at macro use site. A curious fact is that Template Haskell also provides cross-stage persistence for local variables that conform to the `Lift` type class.

A very interesting aspect of Template Haskell is its implementation of hygiene through renaming, an inherently stateful operation, in the pure Haskell environment.

`Expr`, the type of quasiquotes, is in fact a synonym to `Q Exp`, a type of low-level syntactic data structures lifted into the quotation monad `Q`that provides usual monadic operations along with `gensym :: String -> Q String`, a fresh name generator. In this setting quasiquotes are mechanically translated into calls to low-level constructors wrapped in `Q` with binders renamed using `gensym` and bindees either following the renamed binders or persisted across stages as outlined above:

```
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)]))
}
```

This approach is ingenious, but unfortunately it has limitations.

First of all, in general case roles of identifiers in quasiquotes (binder, bound within quasiquote, bound outside) cannot be determined before macro expansion. For example, in `[| f $p = x + y |]`, variables `x` and `y` are on the fence until the quasiquote is evaluated. To the contrast, the translation algorithm requires an up front decision, which makes splices into binding positions illegal.

Secondly, there is no way to opt out of hygiene without dropping to low level. Even worse, since the translation algorithm effectively forbids unbound free variables in quasiquotes, every call site of a macro that introduces variables visible outside would have to drop to low level. For instance, users of the `forever` macro (Figure 2d) won't be able to write `$(forever [| abort |])`, but will be forced to express this as `$(forever [| $(var "abort") |])`.

B. Nemerle

Nemerle is a general purpose object-oriented/functional programming language. Macros in Nemerle are introduced

with the keyword `macro` and can use quasiquotes (`<[...]>`) to pattern match and compose syntax objects. Expansions are triggered automatically when the compiler encounters a function application, and the callee is deemed to be a macro. Nemerle also supports macro attributes that expand when put on declarations. Such macros can modify the inheritance chain of classes, generate new members, etc.

The macro `<->` shown below swaps values of two expressions like in `x <-> arr[2]`. In the snippet we can see how quasiquotes can be used to pattern match against syntax objects, extract their parts and reassemble producing new syntax objects. The `[Hygienic]` attribute is discussed later.

```
[Hygienic]
def cache(e: Expr): Expr * Expr
{
  | <[ $obj.$mem ]> =>
    (<[ def tmp = $obj ]>, <[ tmp.$mem ]>)
  | <[ $tab[$idx] ]> =>
    (<[ def (tmp1, tmp2) = ($tab, $idx) ]>,
     <[ tmp1[tmp2] ]>)
  | _ => (<[ () ]>, e)
}

macro @<->(e1, e2) {
  def (cached1, safe1) = cache(e1);
  def (cached2, safe2) = cache(e2);
  <[
    $cached1;
    $cached2;
    def tmp = $safe1;
    $safe1 = $safe2;
    $safe2 = tmp;
  ]>
}
```

From the example above it is apparent that quasiquotes in Nemerle can have unbound free variables and even allow splicing into binding positions. Nevertheless Nemerle is hygienic and referentially transparent (except for cross-stage references to local variables, which are prohibited). This is achieved with an algorithm of impressive simplicity and power.

To separate lexical scopes of macro definitions and macro call sites, Nemerle colors the identifiers in the program. By default each macro expansion gets a unique color, different from the color of normal code, that gets assigned to the symbols it introduces. After the program is fully expanded Nemerle resolves bindings, i.e. for every bindee the compiler looks for a nearest preceding binder with the same color, performing renaming as necessary. If there's no such binder, the name is looked up in a global environment that corresponds to the definition site of the bindee (such environments are enclosed in every symbol).

An important characteristic of this algorithm is that colors are fully customizable. Using an API exposed by the expander it is possible to recolor individual symbols and entire bunches into arbitrary colors. It is also possible to share colors between macros to ensure that selected parts of corresponding expansions can see each other. Finally Nemerle supports polychromatic symbols that bind to the nearest definitions with the same name regardless of the color.

Nemerle also provides most frequently used patterns of tweaks to bindings as the part of the core language. `$(name: usesite)` introduces a symbol colored identically to the macro call site. `[Hygienic]` attribute on a function (usually used on helper functions that can be called multiple times from the same macro) colors binders that come from that function is a color that is generated afresh every invocation.

C. Racket

Racket is a dynamically typed descendant of Scheme. It uses Scheme's `syntax-case` [7], a hygienic and referentially transparent macro system, augmented with syntax classes [10].

To prevent inadvertent captures `syntax-case` uses Dybvig's algorithm, which is similar to Nemerle's color algorithm outlined above (in fact, the causal relation is inversed - the algorithm in Nemerle is inspired by Dybvig's work). Racket's algorithm generates fresh marks for every macro expansion, assigns these symbols introduced by expansions and then uses the marks to resolve bindings and perform renamings if necessary.

Much like in Nemerle, affiliation of symbols can be customized. The `datum->syntax` function can be used to put an arbitrary S-expression into a lexical context of an arbitrary syntax object, including that of macro use site and macro definition site.

The snippet below creates a non-hygienic symbol `abort` in the lexical context of the macro use site (`#'forever`) and then introduces it in macro expansion as a binder, making it possible for a macro argument to see an identifier in the generated code:

```
(define-syntax (forever stx)
  (syntax-case stx ()
    ((forever body ...)
     (with-syntax
      ((abort (datum->syntax #'forever 'abort)))
       #'(call/cc (lambda (abort)
            (let loop () body ... (loop))))))))

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

In [3] Barzilay et al. investigate the consequences of breaking hygiene to introduce special identifiers, because the need for that arises quite often (e.g. to implement the ubiquitous `this` in a class system, to internally communicate information between subsystems of a complex macro, etc). Authors find that the technique outlined above doesn't scale to derived macros:

```
(define-syntax while
  (syntax-rules ()
    ((while test body ...)
     (forever (unless test (abort)) body ...))))

(while #t (abort))
```

In the running example, `abort` will not be automatically propagated to use sites of the macros that build upon `forever`. As outlined below `abort` is visible inside the implementation of `while`, but using it in the body of `while` will produce a "reference to undefined identifier" error.

Barzilay et al. study a workaround that involves systematic introduction of `abort` into all derived macros (eventually

abstracting out the recurring pattern into an auxiliary macro) and another one that explicitly passes special identifiers between call sites. Both approaches however involve repeated boilerplate.

It becomes apparent that to address the `abort` problem elegantly, the macro system needs to have a mechanism of variables visible in multiple scopes.

Racket lacks the notion of Nemerle's polychromatic symbols that bind to whatever is in any scope nearby, but all Lisps have a tradition of dynamically scoped variables. By adapting this notion to compile-time (without changes to the compiler, just by introducing a couple of macros: `define-syntax-parameter` and `syntax-parameterize`) authors of [3] arrive at a design pattern, which doesn't impose boilerplate tax on both the developers and the users of derived macros:

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

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

As an additional bonus, syntax parameters are actually more robust than polychromatic symbols, because the dynamic variable visible though scopes is explicitly introduced as such and is easily discoverable, since it comes together with the originating macro.

IV. Conclusion

Macro expansions can go wrong in several ways. The simple one is introduction of type errors, the more sophisticated one is inadvertent collision of lexical scopes.

Evolution of Lisp identified the problems that lead to distortion of scopes and formulated the principles of hygiene and referential transparency that prevent such errors. This gave rise to manual [11,12] and automatic [5,7] techniques that increase robustness of macros. Among languages studied in this paper, Template Haskell derives monadic representations for quasiquotes and automatically renames introduced binders [1]. It is however incapable of breaking hygiene per programmer's request without dropping to a low-level notation. Nemerle [2] and Racket [3] use approaches derived from Dybvig's algorithm [7], which provides scope isolation by default as well as mechanisms for manual control.

Typechecking algorithms that catch errors in meta-programs are pioneered by MetaML [13–15]. Its static typechecking discipline for quasiquotes has proven to be overly restrictive when applied to macros. Template Haskell features a lenient typechecking algorithm, which unfortunately remains too restrictive. Therefore Nemerle doesn't typecheck quasiquotes at all, relying on a mandatory typecheck after macro expansion.

V. Research proposal

We implemented a macro system for the Scala programming language [18] and integrated it into a production version of the Scala compiler.

The main contribution of our work is bootstrapping of a minimalistic non-hygienic macro system without quasiquotes into a hygienic extension with high-level means of building syntax objects.

The `reify` macro that implements the bootstrapping is based on a notion similar to MacroML's [14], which forces the programmers to resort to low-level syntax manipulations when untypeable templates or destructuring are involved. Our experience and user reports show a clear need for a full-fledged hygienic quasiquoting facility. Therefore we plan to resolve the issues with `reify` while retaining as much type safety as possible.

Another direction of research is integration with Scala's rich type system. For example, our early adopters have suggested that the combination of macros and path-dependent types can be used to contain and propagate effects. One more conjecture is that synergy between macros and implicits might grant Scala theorem-proving powers.

References

[1] T. Sheard and S. Peyton Jones, Template meta-programming for Haskell. ACM SIGPLAN Notices, 2002.

[2] K. Skalski, M. Moskal and P. Olszta, Meta-programming in Nemerle. Generative Programming and Component Engineering, 2004.

[3] E. Barzilay, R. Culpepper and M. Flatt, Keeping it Clean with Syntax Parameters. Scheme and Functional Programming Workshop, 2011.

[4] M. Odersky, L. Spoon and B. Venners, Programming in Scala 2nd Edition. Artima, 2010.

[5] E. Kohlbecker, Syntactic Extensions in the Programming Language Lisp. PhD thesis, Indiana University, 1986.

[6] T. Sheard, Accomplishments and Research Challenges in Meta-Programming. Semantics, Applications, and Implementation of Program Generation, 2001.

[7] R. Dybvig, R. Hieb and C. Bruggeman, Syntactic Abstraction in Scheme. Lisp and Symbolic Computations, 1992.

[8] A. Bawden, Quasiquotation in Lisp. Partial Evaluation and Program Manipulation, 1999.

[9] E. Kohlbecker, M. Wand, Macro-by-Example: Deriving Syntactic Transformations from their Specifications. Principles of Programming Languages, 1987.

[10] R. Culpepper, M. Felleisen, Fortifying Macros. International Conference on Functional Programming, 2010.

[11] D. Hoyte, Let Over Lambda. Lulu, 2008.

[12] W. Clinger, Hygienic macros through explicit renaming. ACM SIGPLAN Lisp Pointers, 1991.

[13] W. Taha, Multi-Stage Programming: Its Theory and Applications. PhD Thesis, Oregon Graduate Institute, 1999.

[14] S. Ganz, A. Sabry, W. Taha, Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. International Conference on Functional Programming, 2001.

[15] W. Taha and P. Johann, Staged Notational Definitions. Generative Programming and Component Engineering, 2003.

[16] T. Sheard and S. Peyton Jones, Notes on Template Haskell Version 2. Glasgow Haskell Compiler, 2003.

[17] M. Flatt, Composable and Compilable Macros: You Want it When? International Conference on Functional Programming, 2002.

[18] E. Burmako and M. Odersky, Scala Macros, a Technical Report. Valentin Turchin Workshop on Metacomputation, 2012.