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 Template Haskell [1], Nemerle [2] and Racket [3].

We discuss integration of hygienic macros into statically typed languages and plan 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.

However procedural abstraction is sometimes not 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 operational semantics. Another example in this vein 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 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] (and in general sense these procedures can be referred to as meta-programs [6]). The process of applying macros is called macro expansion.

Having means of textual abstraction in their toolbox, programmers can use a multitude of techniques, some of which are:

- reification (providing code as data),
- language virtualization (overloading/overriding 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 ([7–9] 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.

II. 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 [14,15] and automatic [5,10] techniques that increase robustness of macros. Among languages studied in this paper, Template Haskell derives monadic representations for quasiquotes and automatically alpha-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 [10], 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 [16–18]. 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.

III. Research proposal

We implemented a macro system for the Scala programming language [20] 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 [17], 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 vanilla macros, type 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.