PROJECT KEPLER: COMPILE-TIME METAPROGRAMMING FOR SCALA

Eugene Burmako
LAMP, EPFL
2011-09-27
updated on 2011-10-02

Quick info

Goal: Implement macros, a type-safe compile-time metaprogramming facility for Scala.

Inspiration: Mostly Nemerle, though it was enlightening to learn about certain aspects of Scheme, MetaML and Template Haskell.

Resources: Currently only I am involved. It's a part-time effort, since I'm also studying in a doctoral program.

Status: Early design phase.

Outline

Today's talk is about current approaches to compile-time metaprogramming and about what can be built on this foundation.

Topics covered:

- Macros 101
- State of the art in macrology
- Project ideas

Most code examples will come from the articles about Nemerle. Thanks to Kamil Skalsky and Vlad Chistyakov!

Macros 101

An example

```
macro printf(format : string, params parms : array[expr])
{
  def (evals, refs) = make_exprs(parse(format), parms);
  def seq = evals + refs.Map(x => <[ Console.Write($x) ]>);
  <[ { ..$seq } ]>
}

printf("Value = %d", 123 + 877)

{
  def _N_1812 = (123 + 877 : int);
  Console.Write("Value = ");
  Console.Write(_N_1812)
}
```

Macros

That was a pretty involved code fragment. Let's take a closer look at the stuff that relates to macros:

- Printf is a macro. It hosts metacode, i.e. the code that gets executed during the compilation of the program.
- Arguments of printf are compile-time values, namely: a string literal and a bunch of expressions, i.e. AST nodes. Return value is an AST as well.
- When compiler sees a macro invocation, it will call the metacode (i.e. the body of the macro) and splice its result (i.e. the generated code) into the callsite.

Quasi-quotations

Such a fluent coding experience would be impossible without a prominent language construct:

- Fancy brackets <[...]> denote quasi-quotations, a facility to easily compose and decompose code snippets.
- Small snippets can be incorporated into bigger ones. This is called splicing. $x and ..$seq are splices.
- Quasi-quotations produce compile-time values. Note, the return value of our macro is a quasi-quotation, i.e. an AST.

Pattern matching

Quasi-quotations can also be used to match against code snippets. For example, in a hypothetical optimizer one could write:

```
match (e) {
    ...
    | <[ $x * 1 ]> => <[ $x ]>
}
```

Note how splices turn into named holes when used on the left-side of a pattern match. Neat trick, eh?

Flavors

We've seen and went through what I call an AST rewriting macro. However, macros come in different flavors.

Metamorphic macros provide the possibility to transparently change the syntax of the host language. For example, the x => <[Console.Write($x)]> expression is a macro as well.

Top-level macros don't return anything. Instead of churning out expressions, they perform codegen in the scope of the entire program.

These two examples do not represent a comprehensive classification, they are just to whet your appetite. More on that in the next section.

State of the art

Taxonomy

Compile-time metaprogramming can operate on different levels. Here's a bit contrived though useful classification that will guide us during our dive into macrology:

Flavors:

- Lexical substitution
- Metamorphic syntax
- AST rewriting
- Codegen in the large

I won't mention type-level computations, since they are out of the scope of this talk.

Lexical substitution

```
#define MIN(a,b) ((a)>(b)?(b):(a))
```

One of the approaches to compile-time metaprogramming is to rely solely on text substitution. This has proven to be an unreliable and unwieldy path.

But don’t dismiss macros just because their C/C++ cousins are so infamous. Later we’ll see how more sophisticated techniques completely overcome the problems inherent to preprocessor macros.

Metamorphic syntax

```
macro Cons is Expr
  syntax expr1 : 11 "::" expr2 : 10;
{
  <[ @::(expr1, expr2) ]>
}
```

Roughly speaking, metamorphic macros operate on parser level. They can introduce new syntaxes or refine already existing non-terminals of the grammar. Entire syntax of a language can be designed with such macros. This very example is based on a hybrid of PEG and TDOP.

Metamorphic syntax: discussion

Metamorphic macros usually operate separately from the compiler (CamlP4), which makes them noticeably limited, but that's not a hard requirement.

Authors of Nemerle 2 are planning to implement a DSL for expressing arbitrary syntaxes using metamorphic macros. In addition to syntax services, the DSL will also provide ways to declaratively specify type information for its macros. This will effectively make N2 a metacircular metacompiler.

AST rewriting

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

using ("x", Foo(), { x.Compute() })
```

We've already seen an example of AST rewriting in Macros 101, but this is another thought-provoking snippet.

AST rewriting: hygiene

In the example, one can notice a refinement of a plain splice expression, namely: $(name : usesite).

This tells the compiler to bind the variable used in body to a variable declared in a macro. Okay, that makes sense, but why do we need special ceremony for that?

That’s because Nemerle macros are hygienic, i.e. they alpha-rename the identifiers to prevent inadvertent name clashes. Typically that’s what one wants, but sometimes (say, in string splices) that’s undesired and can be disabled.

AST rewriting: typing

All macros we've seen so far use untyped expressions (i.e. ones of type PExpr instead of TExpr[T]). Why's that?

That's because in most cases macros are either too generic to be typed or snippets they produce are just fragments of code that make sense only together with other snippets. That's not a problem – after macro expansion the compiler is going to typecheck generated code anyways.

However, sometimes it makes sense to restrict certain expressions for better error diagnostic and IDE support. Say, "if" macro could read: "match ($cond : bool) { ... }".

Codegen in the large

```
[Usage (Phase.BeforeInheritance, Targets.Class)]
macro Serializable (t : TypeBuilder)
{
  t.AddImplementedInterface (<[ ISerializable ]>)
}

[Serializable]
class S {
  public this (v : int, m : S) { a = v; my = m; }
  my : S;
  a : int;
}
```

Codegen in the large: discussion

Top-level macros, which are applied by annotating program elements, are Nemerle's answer to boilerplate.

That's very-very neat, especially because most of the time one generates the code (types, methods, whatever) using quasi-quotations, which are expressive and composable.

The problem is that all top-level macros share global compiler environment, that's why they need to be carefully orchestrated to play well with each other. Another significant challenge is IDE support.

Summary

- Unlike infamous C macros, state of the art macros work with ASTs and are deeply integrated with compilers, which makes them safe, expressive and composable.

- With hygienic quasi-quotations, analysis and generation of object code literally becomes a walk in the park.

- Codegen in the large provides amazing boilerplate reduction facilities, though at times it might become rather complex because all top-level macros (and also an IDE) share the same environment.

Project ideas

This fall

Will be about AST rewriting macros.

I believe that it’d be impossible to implement all the aspects of the codegen in the large during this semester, that’s why I’ll go for a low-hanging fruit.

Project ideas

Speaking of AST rewriting macros, there are quite a few compelling use-cases for them, e.g. computational expressions, string interpolation, hassle-free regexen, code lifting for the convenience of DSL authors and much more.

One of the most prominent applications of AST rewriting macros is LINQ. With macros we get code lifting for free, and with a few lines of metacode we can express the AST accumulation logic of IQueryable without hardcoding anything into the compiler.

Next steps

Nothing concrete yet. Stay tuned to Scala meeting reports (subscribe to scala-internals mailing list!).

Wrapping up

Acknowledgements

I owe big thanks to Vlad Chistyakov, principal developer of Nemerle programming language, for immensely helpful discussions about compile-time metaprogramming.

Also, I am very inspired by Nemerle. It features a mature metaprogramming system for a mainstream programming platform, which is pretty remarkable in itself. Moreover, Nemerle is a testbed for an ongoing research in metaprogramming, and that makes it even more interesting.

Links

- Project Kepler, Compile-Time Metaprogramming for Scala
  https://github.com/xeno-by/kepler
- My blog that hosts fine-grained status updates
  http://blog-en.xeno.by
- A collection of enlightening papers about macros
  http://macros.xeno.by
- Nemerle Programming Language
  http://nemerle.org/

Questions and answers

eugene.burmako@epfl.ch

Bonus slides

Languages as libraries

Recently presented by Racket developers, this idea also applies to languages with syntax. In fact, this has already been implemented in Nemerle for more than five years.

There’s a story that Vlad told me yesterday. Some guy went to their forum and proclaimed that F# is superior to Nemerle, because it has computational expressions. Folks explained to him how to use macros and in a few weeks Nemerle also sported computational expressions =)

This is modular, composable and enables quick prototyping and deployment of language features. Pimp my language!

Engineering challenges

Metacompilers are klondikes for inventing and implementing advanced ways of software development.

Due to excessive complexity of the domain, imperative approaches to coding quickly become unwieldy – both for metacompiler developers and users.

Unfortunate restrictions of macro power made Nemerle developers completely rethink the compiler pipeline, problems with state shared between top-level macros beg for declarative solutions, quality IDE support requires reactive programming. It’s a major fun, isn’t it?