Contrasting compile-time meta-programming in Metalua and Converge

Fabien Fleutot[1] and Laurence Tratt[2]

[1] fleutot@gmail.com 11 rue Ferrus, 75014 Paris, France
[2] King's College London, Strand, London, WC2R 2LS, U.K.,
laurie@tratt.net, http://tratt.net/laurie/

Abstract. Powerful, safe macro systems allow programs to be programatically constructed by the user at compile-time. Such systems have traditionally been largely confined to LISP-like languages and their successors. In this paper we describe and compare two modern, dynamically typed languages Converge and Metalua, which both have macro-like systems. We show how, in different ways, they build upon traditional macro systems to explore new ways of constructing programs.

1 Introduction

Macros as found in the LISP family of languages, such as Scheme [1] allow program fragments to be built at compile-time, allowing users to extend the programming language. Such functionality allows users to e.g. add new features to a language [2] or apply application specific optimizations [3].

Macros come in two main flavours: those which operate at the syntactic level and those which operate at the lexing level [4]. LISP systems work at the syntactic level, operating on Abstract Syntax Trees (ASTs), which structure a program's representation in a way that facilitates making sophisticated decisions based on a node's context within the tree. Macro systems operating at the lexing level – commonly represented by the C preprocessor – are inherently less powerful, since they operate on a text string. Therefore they have little to no sense of context, and can cause bizarre programming headaches due to unexpected side effects of their use [5]). Unfortunately, despite the benefits of a syntactic macro system, the syntactic richness of most languages has presented a significant barrier to creating an equivalent of LISPs system which is often seen to rely on that languages particular syntactic minimalism [6].

Relatively recently languages such as the multi-staged MetaML [7] (specifically its MacroML variant [8]) and Template Haskell (TH) [9] have shown that statically typed functional languages can house powerful compile-time meta-programming (CTMP) systems. These systems are, in all but name, macro systems and showed that such functionality is compatible with the syntactic richness of modern languages. Converge was the first dynamically typed language based on this general style of macro system [10].

This paper describes and compares two modern dynamically typed CTMP languages: Metalua and Converge, which were designed separately by the respective authors of this paper. Each of these languages takes a TH-like macro system and extends it in different ways, leading to new functionality and, of course, new trade-offs. The aim of this paper is to describe the differing philosophies and implementations of these two systems, and thus to show how macro systems can be designed and implemented for non-traditional purposes.

2 Overview of Metalua and Converge

This section gives a very brief overview of Metalua and Converge; please see the respective documentation for more details [11, 12].

Converge Converge's most obvious ancestor is Python, resulting in an indentation based syntax, a similar range and style of datatypes, and general sense of aesthetics. The most significant difference is that Converge is a slightly more static language: all namespaces (e.g. a modules' classes and functions, and all variable references) are determined statically at compile-time. Converge is lexically scoped, and its scoping rules are simplified to ensure that macro hygiene is easily achievable. Converge programs are split into modules, which contain a series of *definitions* (imports, functions, classes and variable definitions). As in Python, Converge modules are executed from top to bottom when they are first imported.

Metalua Metalua is an extension to the existing dynamic language Lua [13]. Lua has several features that make it a good candidate for extending with CTMP features. It has an easy to parse syntax, and a simple yet powerful semantics favouring—in its authors' own words—"meta-mechanisms over a host of features": lexical scoping, closures, coroutines, function environments, and a single very versatile compound datatype—the table—which can be specialized as an array, structure, dictionary, object, class, module, function environment, etc. through a metatable defining its behaviour.

Lua being already equipped with excellent runtime meta-programming features, Metalua adds the mechanisms required for CTMP: an AST compiler, code quasi-quotation, compile-time code execution, splicing of generated and literal code, and a simple, dynamically extensible syntax parser.

3 Compile-time meta-programming

For the purposes of this paper, CTMP can be largely thought of as being equivalent to macros. More formally, it allows the user of a programming language a mechanism to interact with the compiler to allow the construction of arbitrary program fragments by user code. Although must such interaction is in terms of building and inserting abstract syntax trees into programs, it is not restricted to this.

3.1 A first example in Converge

The following program is a simple example of CTMP, trivially adopted from its TH cousin in [14]. `expand_power` recursively creates an expression that multiplies `n` `x` times; `mk_power` takes a parameter `n` and creates a function that takes a single argument `x` and calculates x^n; `power3` is a generated function which returns n^3:

```
func expand_power(n, x):
  if n == 0:
    return [| 1 |]
  else:
    return [| ${x} * ${expand_power(n - 1, x)} |]

func mk_power(n):
  return [|
    func (x):
      return ${expand_power(n, [| x |])}
  |]

power3 := $<mk_power(3)>
```

The user interface to compile-time meta-programming is inherited directly from TH. *Quasi-quoted* expressions `[| ... |]` build ASTs that represent the program code contained within them whilst ensuring that variable references respect Converge's lexical scoping rules. Splice annotations `$<...>` evaluate the expression within at compile-time (and before VM instruction generation), replacing the splice annotation itself with the AST resulting from its evaluation. This is achieved by creating a temporary module containing the splice expression in a function, compiling the temporary module into bytecode, injecting it into the running VM, and then evaluating the function therein. Insertions `${...}` are used within quasi-quotes; they evaluate the expression within and copy the resulting AST into the AST being generated by the quasi-quote.

When the above example has been compiled into VM instructions, `power3` essentially looks as follows:

```
power3 := func (x):
  return x * x * x * 1
```

As this example shows, Converge differs from traditional LISP macro schemes in that 'macros' are not explicitly identified—they are normal functions that happen to be called by a splice and thus are executed at compile-time.

3.2 Metalua example

We now show precisely the same example in Metalua:

```
1  -{block:
2    function expand_power(n, x)
3       if n==0 then return +{1}
4       else return +{ -{x} * -{expand_power (n-1, x)} } end
5    end
6
7    function mk_power(n)
```

```
8         return +{ |x| -{expand_power (n, +{x})} } }
9     end }
10
11  power3 = -{mk_power(3)}
```

Allowing for minor syntax differences, this example initially appears to be similar to the Converge version. However, Metalua's underlying philosophy is significantly different, with a much stricter separation of CTMP meta-levels combined with a more explicit concept of moving between layers of -meta-levels. Metalua 'moves' between meta-levels with the `+{...}` (equivalent to Converge's quasi-quotes) and `-{...}` (equivalent to Converge's `$<...>`, unless it is nested inside a `+{...}` when it is equivalent to Converge's `${...}`) operators. Code executed at compile time is referred to as 'level 0', and the result of compilation as 'level 1'. There can be other levels: if the CTMP code itself relies on generation, is will be produced by level -1 code; conversely, if the resulting code produces ASTs through quasi-quotes, these quotes are level 2. However, the majority of cases are handled entirely within levels 0 and 1. Although Converge can access the same rang of meta-levels as Metalua, it is much less common to use anything other than levels 0 and 1.

In Converge, all functions exist in the run-time meta-level, and some functions can also get called (and therefore exist) when a splice is invoked: the function call happens at a specific meta-level, but the function definition occurs at the same level as regular functions. In Metalua, functions called at compile-time metalevel must be defined at compile-time metalevel: by default, a definition belongs to a single level, and everything compile-time related disappears once compilation is finished. In the example above, there is no trace of `mk_power` nor `expand_power` left in the compiled file. If a function is to be used both at compile-time and runtime meta-levels, the preferred way is to put this code in a separate module and to require it twice, once in each meta-level using it: existing in several meta-levels simultaneously is considered as unusual, so the programmer's intent is explicit at the price of some verbosity.

3.3 Language philosophies

Metalua and Converge have different philosophies on CTMP, its integration into the host language, and the ways in which it is best used. Crudely put, Metalua aims to put greater power in the hands of the CTMP programmer, but relies on him to have a fairly intimate understanding of the language's internals in order to produce safe results. Conversely, Converge is more concerned with the reliability of CTMP programs, and thus attempts to structure features in such a way that they are hard to use incorrectly. These philosophies can be traced back to each languages ancestors: Lua's aim is to provide a small language which experts can easily extend [15], while Python (Converge's most obvious ancestor) generally aims to provide one obvious solution for most common programming problems.

We believe that both Converge's and Metalua's approaches have virtue, and that they represent interesting and important areas of the overall CTMP solution

space. Subsequent sections of this paper flesh out specific concrete issues which relate to these underlying philosophical differences.

4 Syntax extension

One of the most common uses of the macro system in LISP is to build efficient, powerful abstractions of concepts that are often found in modern programming languages, such as object systems. Since both Metalua and Converge either provide such features directly, or have a more idiomatic encoding, our experience is that 'traditional' macros are less useful than in LISP. We believe that to make CTMP practical, there needs to be a way of extending the syntax of the language in ways not anticipated by the language author, to allow natural integration. Both Metalua and Converge provide such functionality, although in very different manners.

Metalua Because of Metalua's approach to separating meta-levels, it is possible to dynamically extend its syntax. Dropping to meta-level 0, a user can plug-in to the Metalua parser which provides hooks for the most common cases for syntax extensions: prefix, infix and suffix expression modifiers, and new statements introduced by a dedicated keyword. This captures most of Lua's syntax effectively. Although there are limitations in what can be parsed—extensions need to carefully consider Lua's syntax, ensuring they do not undermine normal parsing—it has the advantage of not requiring the learning of a completely separate parsing DSL (as e.g.[6]). Once the parser is extended, all subsequent code in a file is parsed relative to that extension.

As an example, we introduce dedicated syntax for the power function generator presented earlier. "`^n`" represents the function raising its argument to the $\texttt{n}^{\text{th}}$ power, where `n` is a constant integer known at compile-time. Note that this does not conflict with the `^` infix operator, since this extension is only legal where an expression is expected, whereas infix operators appear *after* an expression.

```
-{ block:
   mlp.expr.prefix:add{ "^", builder = pow_builder }
   function pow_builder (op, expr)
      assert (expr.tag=="Number", "literal number expected by '^'")
      assert (expr[1]%1==0,       "constant for '^' must be an integer")
      return mk_power (expr[1])
   end }
```

It should be noted that by dropping to meta-level 0, one extends the parser for meta-level 1. This means that in order to extend the parser for meta-level 0, one needs to drop to meta-level -1. This helps make dynamic syntax extension more manageable: it's always clear when and where a syntax change takes effect, and syntax-changing code doesn't interfere with itself.

Backward compatibility with Lua's syntax raises two issues for Metalua. First, Lua's distinction between statements (non-value producing constructions) versus expressions (value producing constructions) makes some constructions

more complicated than necessary, as the programmer often wishes to use a statement where an expression is expected. To address this issue, Metalua introduces the `Stat{...}` AST node, which allows just this, without resorting to inefficient encoding (such as application of a local anonymous closure). Second, Lua's statement separators (semicolons) are optional, which can effect the design of new statements, whose end must be determinable without an explicit delimiter. This can then lead to subtle ambiguities in the grammar, especially between two independently designed macros. However, this problem is generally relatively easy to address, provided it is considered up-front.

Converge Relative to Metalua, Converge provides a coarser-grained approach to syntax extension. Built on top of the the standard splice operator is the concept of a *DSL block*. This allows arbitrary blocks of text to be embedded in a Converge file, and appropriate error information to be associated with that block. Whereas Metalua allows small extensions to be weaved into the languages parser, Converge aims to allow complete, localised DSLs to be embedded into the language.

When the Converge tokenizer encounters a DSL block, the text on the next level of indentation is left unparsed (for completeness we note that there is also an intra-line DSL phrase construct). This raw string is passed to a *DSL implementation function* which is called at compile-time. It is the DSL implementation function's job to parse the raw string and return an AST. Converge provides a number of convenience functions to capture standard idioms of DSL parsing, and DSL AST creation. The provided parser is an Earley parser [16] which means that arbitrary context free grammars (including ambiguous grammars) can be expressed. Converge also allows its expression language to be easily embedded into DSLs, and DSLs (and indeed normal Converge code) can be embedded arbitrarily deep within one another.

As Converge DSLs tend to be rather large, space constraints mean a full example is impractical. An indicative chunk of an example for a railways timetable DSL is as follows:

```
func timetable(dsl_block, src_infos):
  parse_tree := CEI::dsl_parse(dsl_block, src_infos, ["Premium", \
    "Cheap"], [], GRAMMAR, "start")
  return Translater.new().generate(parse_tree)

$<timetable>:
  8:25 "Exeter St. Davids" Premium
  10:20 "Salisbury" Premium, Cheap
  11:49 "London Waterloo"
```

In this example, `timetable` is the DSL implementation function, which is called at compile-time with a string `dsl_block` which is the raw, unparsed text from the DSL block. The `dsl_parse` function is a convenience function which takes a DSL block, the DSL blocks' src infos, extra keywords (over the base Converge language), extra symbols, a grammar, and a start rule in the grammar. With the parse tree in hand, the DSL implementation function then has to translate

this into an AST. Converge does provide a standard mechanism for doing this, but DSL authors are also free to tackle this problem as they wish.

Comparison The approach taken by each language has interesting trade-offs. Most obviously Metalua allows the base language to be naturally extended, whereas Converge's DSL blocks are distinct from normal code.

Metalua requires the CTMP programmer to have a very good understanding of the underlying parser and compiler. It can be difficult to extend some existing language features, and some seemingly plausible extensions are impractical. Composition of multiple syntax-extending features can be challenging.

Converge limits the ways in which new syntax can be embedded into the language, but allows any syntax to be embedded, without interfering with the main language. This allows a clean separation between, and composition of, languages even when DSLs are embedded within each other. However this also means that DSLs can have relatively limited interaction with each other.

Metalua's approach is both more pragmatic, and unforgiving: language extensions are intended to be directly incorporated in the standard language, rather than fenced off in separate blocks. This allows the language to easily assimilate idioms taken from other languages such as Python's list comprehension or ML's "`match...with`" construction. However this raises problems with composition of different language extensions, as there is no bulletproof protection against interference among extensions. If the macro designer uses only the extension places explicitly opened in the grammar (prefix/infix/suffix expression modifiers and statements introduced by specific keywords), his macros should cohabit correctly with others, but more advanced changes break compositionally of macros.

5 Error reporting

Good quality error reporting in the face of CTMP can be very challenging, and has hitherto received scant attention [14]. Many LISP implementations have a step-by-step macro expander which is one possible approach to this issue. In this section we discuss how Metalua and Converge tackle this issue in different ways. We break this into two problems: how to report run-time errors that result from an arbitrary AST created via CTMP; how to detect type-incorrect ASTs. In part because of the ability to easily change its VM and compiler in the early design phases, Converge is currently more mature than Metalua in both respects.

Converge has the concept of src infos; an individual src info is a (src path, src offset) pair which records a relationship to a specific byte offset in a specific input file. The Converge tokenizer associates every token with a src info; when the compiler converts parse trees to ASTs, the ASTs carry over the relevant src infos; and when the compiler compiles ASTs into bytecode, every single bytecode instruction records the src infos it relates to. Thus the src info concept is used uniformly throughout the Converge parser, compiler, and VM. Since src infos are lists, AST elements can be associated with more than one source location, and Converge provides an extended quasi-quote variant `[<e>| ... |]` which

appends the src infos in the expression `e` to the src info automatically created by the quasi-quotes. This leads to unusual backtraces such as the following:

```
Traceback (most recent call last):
  1: File "test.cv", line 162, column 8, in main
  2: File "Pf.cv", line 105, column 18, in _t_pf_tgt
     File "test.cv", line 78, column 2, in X
Exception: No such slot 'foo' in instance of 'Pf'.
```

The problem of creating type-incorrect ASTs can be debilitating in a dynamically typed languages. Type errors are not normally considered a major issue in such languages, because there is typically an obvious and direct fix when code accesses a slot `foo` in an object with such slot. CTMP often involves creating complex, nested ASTs, and type errors – which can include putting e.g. a string where an AST is expected, or putting an expression into an AST where a statement is expected – are relatively easy to make. However the resulting exceptions tend to be raised much later, and emanate from deep within the language's compiler, making debugging close to impossible. Converge thus has an AST type checker which rigorously checks it for type conformance. Whenever a new AST is created – by quasi-quotes or other means – it is type checked, to ensure that any type errors are reported as soon as possible, thus greatly increasing the chance that the offending code can be easily tracked down.

Metalua is currently not as advanced as Converge in this regard, although it is anticipated that it will develop similar error reporting features over time. Metalua is slightly hampered by the fact that some changes may require changes to the Lua VM, which somewhat goes against its philosophy.

6 Related work

The canonical example of a programming language with macros is LISP. Its minimal syntax allows powerful macros to be expressed, and much of the work in this paper ultimately traces its roots back to LISP. However LISP's minimalist nature means that many of the macros which add functionality to LISP are not needed in richer languages such as Converge and Metalua which contain such functionality as standard. Therefore the use cases for macros in Converge and Metalua are different than for LISP, and this explains features such as Converge's DSL blocks and Metalua's strict layering of meta-levels.

The tight coupling of LISP's macro system to its syntactic minimalism has largely prevented similar approaches being applied to other, more modern programming languages [6]. Therefore despite LISP's success in this area, for many years more modern systems struggled to successfully integrate similar features [10]. Dylan is one of the few such systems [6], implementing a rewrite rule based macro system which is broadly equivalent to LISP's in power. However Dylan's syntax is not significantly more flexible than LISP's, and its macro related syntax is heavyweight, as it is a separate language from Dylan itself.

More recently languages such as Template Haskell (TH) [9] (effectively a refinement of the ideas in MetaML [17]) have shown how sophisticated homogeneous meta-programming systems can be implemented in a modern language.

The design of both Converge's and Metalua's compile-time meta-programming functionality is heavily influenced by TH. One of TH's motivations for CTMP is to work around type system restrictions – a non-issue in Converge and Metalua – and thus its functionality is broadly equivalent to LISP. Both languages have used the TH design as a base upon which more powerful functionality is built.

Finally, it must be mentioned that Luiz Henrique de Figueiredo implemented a macro facility for Lua, called token filters [18]. However, this system acts at the lexeme stream level: this makes it substantially simpler to master than Metalua, but restricts its use to shallow transformations, which don't require a deep understanding of the language's grammar.

7 Future work

Both languages are still far from mature with many respects.

Large parts of Metalua's implementation are still rough prototypes, and some essential features are not adequately implemented: error diagnostic and debug informations are still largely unaddressed; lexer modification and extension is only possible through dirty hacks; the libraries which provide the features required for CTMP (e.g. code walking or support for macro hygiene) are also immature; and finally, it is highly desirable to find ways to improve macro compositionality. On some of these issues, Metalua is expected to benefit from Converge's experience.

Converge's CTMP features are in several respects fairly mature. However there are some aspects which require more work. For example, grammar merging (such as when embedding the Converge expression language into a DSL) is crude and dangerous; modularising grammars may provide a practical solution to this problem. The overall language implementation is much less mature; for example the VM is still relatively slow, and library support is lacking.

8 Conclusions

Whereas Metalua is built on top of an existing, mature language Converge has a new compiler and VM. Metalua therefore has the benefit of inheriting a mature language design, a capable and fast VM, and many libraries. However this has both advantages and disadvantages. For example, Lua's statement vs. expression based approach is less suited to CTMP than a purely expression based approach. Since Metalua is currently a pure Lua approach, patching the VM would be a severe task and may make synchronising with future versions of Lua difficult. On the other hand, while creating the Converge VM necessarily diverts energies away from language features for CTMP, features such as Converge's error reporting rely on the synchronisation of features between the compiler and VM.

The interest for runtime meta-programming in non-Lisp dynamic languages has dramatically increased, in large part due to interest in Ruby, and Ruby on Rails in particular. By providing additional features on top of raw CTMP, new possibilities in language extension arise, which we believe are of great interest to dynamic language programmers. In order to make this practical we believe that

CTMP needs to find a good compromise between power and abstraction, without compromising 'normal' programming. Metalua and Converge explore two different, if related, ways to balance these somewhat antagonist requirements: Metalua takes a fine-grained view and insists on clear separation of meta-programming from regular code, whereas Converge takes a more coarse-grained view, and allows fluid travelling across meta-levels. Both approaches present design challenges, and we believe that the experimentation that has forged both languages has been possible because of the flexibility of dynamically typed languages.

References

1. Kelsey, R., Clinger, W., Rees, J.: Revised(5) report on the algorithmic language Scheme. Higher-Order and Symbolic Computation **11**(1) (1998) 7–105
2. Sheard, T., el Abidine Benaissa, Z., Pasalic, E.: DSL implementation using staging and monads. In: Proc. 2nd conference on Domain Specific Languages. Volume 35 of SIGPLAN., ACM (October 1999) 81–94
3. Seefried, S., Chakravarty, M.M.T., Keller, G.: Optimising embedded DSLs using Template Haskell. In: Draft Proc. Implementation of Functional Languages. (2003)
4. Brabrand, C., Schwartzbach, M.: Growing languages with metamorphic syntax macros. In: Workshop on Partial Evaluation and Semantics-Based Program Manipulation. SIGPLAN, ACM (2000)
5. Ernst, M.D., Badros, G.J., Notkin, D.: An empirical analysis of C preprocessor use. IEEE Transactions on Software Engineering (2002)
6. Bachrach, J., Playford, K.: D-expressions: Lisp power, Dylan style (1999) `http://www.ai.mit.edu/people/jrb/Projects/dexprs.pdf` Accessed Nov 22 2006.
7. Taha, W.: Multi-Stage Programming: Its Theory and Applications. PhD thesis, Oregon Graduate Institute of Science and Technology (October 1999)
8. Ganz, S.E., Sabry, A., Taha, W.: Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In: Proc. International Conference on Functional Programming (ICFP). Volume 36 of SIGPLAN., ACM (September 2001)
9. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Proceedings of the Haskell workshop 2002, ACM (2002)
10. Tratt, L.: Compile-time meta-programming in a dynamically typed OO language. In: Proceedings Dynamic Languages Symposium. (October 2005) 49–64
11. Tratt, L.: Converge Manual. (May 2007) `http://www.convergepl.org/documentation/` Accessed May 14 2007.
12. Fleutot, F.: Man Metalua. (April 2007) `http://metalua.luaforge.net/metalua-manual.html`.
13. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: The evolution of lua. In: HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, New York, NY, USA, ACM Press (2007) 2–1–2–26
14. Czarnecki, K., O'Donnell, J., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. **3016** (2004) 50–71
15. Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: Lua — an extensible extension language. Software Practice and Experience **26**(6) (1996) 635–652
16. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM **13**(2) (February 1970)
17. Sheard, T.: Using MetaML: A staged programming language. (September 1998) 207–239
18. de Figueiredo, L.H.: Token filters `http://www.tecgraf.puc-rio.br/~lhf/ftp/lua/#tokenf`.