Compile-time meta-programming in Converge

Department of Computer Science, Technical Report TR-04-11

Laurence Tratt

King's College London, Strand, London, WC2R 2LS, U.K.,
`laurie@tratt.net`,
`http://tratt.net/laurie/`

Abstract. Compile-time meta-programming allows programs to be constructed by the user at compile-time. Few modern languages are capable of compile-time meta-programming, and of those that do, many of the most powerful are statically typed functional languages. In this paper I present the dynamically typed, object orientated language Converge which allows compile-time meta-programming in the spirit of Template Haskell. Converge demonstrates that integrating powerful, safe compile-time meta-programming features into a dynamic language requires few restrictions to the flexible development style facilitated by the paradigm.

1 Introduction

Compile-time meta-programming 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. As Steele argues, 'a main goal in designing a language should be to plan for growth' [1] – compile-time meta-programming is a powerful mechanism for allowing a language to be grown in ways limited only by a users imagination. Compile-time meta-programming allows users to e.g. add new features to a language [2] or apply application specific optimizations [3].

The LISP family of languages, such as Scheme [4], have long had powerful macro facilities allowing program fragments to be built up at compile-time. Such macro schemes suffered for many years from the problem of variable capture; fortunately modern implementations of hygienic macros [5] allow macros to be used safely. LISP and Scheme programs make frequent use of macros, which are an integral and vital feature of the language. Compile-time meta-programming is, at first glance, just a new name for an old concept – macros. However, LISP-esque macros are but one way of realizing compile-time meta-programming.

Brabrand and Schwartzbach differentiate between two main categories of macros [6]: those which operate at the syntactic level and those which operate at the lexing level. Scheme's macro system works at the syntactic level: it operates on Abstract Syntax Trees (AST's), which structure a programs representation in a way that facilitates making sophisticated decisions based on a nodes context within the tree. Macro systems operating at the lexing level are inherently less powerful, since they essentially operate on a text string, and have little to no

sense of context. Despite this, of the relatively few mainstream programming languages which have macro systems, by far the most widely used is the C preprocessor (CPP), a lexing system which is well-known for causing bizarre programming headaches due to unexpected side effects of its use (see e.g. [7–9]).

Despite the power of syntactic macro systems, and the wide-spread usage of the CPP, relatively few programming languages other than LISP and C explicitly incorporate such systems (of course, a lexing system such as the CPP can be used with other text files that share lexing rules). One of the reasons for the lack of macro systems in programming languages is that whilst lexing systems are recognised as being inadequate, modern languages do not share LISP's syntactic minimalism. This creates a significant barrier to creating a system which matches LISP's power and seamless integration with the host language [10].

Relatively recently languages such as the multi-staged MetaML [11] and Template Haskell (TH) [12] have shown that statically typed functional languages can house powerful compile-time meta-programming facilities where the run-time and compile-time languages are one and the same. Whereas lexing macro systems typically introduce an entirely new language to proceedings, and LISP macro systems need the compiler to recognise that macro definitions are different from normal functions, languages such as TH move the macro burden from the point of definition to the macro call point. In so doing, macros suddenly become as any other function within the host language, making this form of compile-time meta-programming in some way distinct from more traditional macro systems. Importantly these languages also provide powerful, but usable, ways of coping with the syntactic richness of modern languages.

Most of the languages which fall into this new category of compile-time meta-programming languages are statically typed functional languages. Whilst such languages have many uses, there are many situations where other language paradigms are useful. In my main body of research on transforming UML-esque models [13], I make frequent use of so-called dynamic object orientated (OO) languages such as Python [14] which are aimed at facilitating rapid development. Since languages such as MetaML and TH are concerned with different aspects of program development (such as statically determinable type-safety), it is not obvious that the compile-time systems devised for those languages would work within the context of a more flexible dynamic language.

In this paper I present a dynamic OO language similar to Python, which contains compile-time meta-programming features similar to TH – the result is Converge. The first part of this paper describes the essential features of the Converge language. The second part of this paper describes Converge's compile-time meta-programming features in more detail, looking at both user-visible features (such as scoping rules) and 'under the hood' features relevant to the compiler (such as dealing with forward references). This forms the main part of the paper, and demonstrates that compile-time meta-programming is not only compatible with dynamic languages (requiring few compromises to the dynamic nature of such languages) but provides features to the users of dynamic languages which have hitherto been largely unavailable. The final part of the paper presents

a high-level look at the Converge compiler, showing how the details of preceding sections dictate the structure of the compiler implementation.

2 Converge basics

This section gives a brief overview of basic Converge features that are relevant to the main subject of this paper. Whilst this is not a replacement for the language manual [15], it should allow readers familiar with a few other programming languages the opportunity to quickly come to grips with the most important areas of Converge, and to determine the areas where it differs from other languages.

Converge's most obvious ancestor is Python [14] 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: namespaces (e.g. a modules classes and functions, and all variable references) are determined statically at compile-time whereas Python's namespaces can be altered at run-time. Converge's scoping rules are also different from Python's and many other languages, and are intentionally very simple. Essentially Converge's functions are synonymous with both closures and blocks. Converge is lexically scoped, and there is only one type of scope (as opposed to Python's notion of local and global scopes). Variables do not need to be declared before their use: assigning to a variable anywhere in a block makes that variable local throughout the block (and accessible to inner blocks) unless the variable is declared via the `nonlocal` keyword to refer to a variable in an outer block. Variable references search in order from the innermost block outwards, ultimately resulting in a compile-time error if a suitable reference is not found. As in Python, fields within a class are not accessible via the default scoping mechanism: they must be referenced via the `self` variable which is automatically brought into scope in any *bound function* (functions declared within a class are automatically bound functions).

Converge programs are split into modules, which contain a series of *definitions* (imports, functions, classes and variable definitions). Unlike Python, each module is individually compiled into a bytecode file by the Converge compiler `convergec` and linked by `convergel` to produce a static bytecode executable which can be run by the Converge VM. If a module is the *main module* of a program (i.e. passed first to the linker), Converge calls its `main` function to start execution. The following module shows a caching Fibonacci generating class, and indirectly shows Converge's scoping rules (the `i` and `fib_cache` variables are local to the functions they are contained within), printing `8` when run:

```
import Sys

class Fib:
  func init():
    self.cache := [0, 1]

  func fib(x):
    i := self.cache.len()
```

```
    while i <= x:
      self.cache.append(self.cache[i - 2] + self.cache[i - 1])
      i += 1
    return self.cache[x]

func main():
  fib_cache := Fib_Cache()
  Sys.println(fib_cache.fib(6))
```

Another important, if less obvious, influence is Icon [16]. As Icon, Converge is an expression-based language. Icon has a powerful notion of expression *success* and *failure*; for the purposes of this paper, these features are largely irrelevant. The most important feature inherited from Icon is functions which generate more than one return value via the `yield` keyword: these are known as *generators*. In this paper we only encounter generators in the following idiom, which uses the `iterate` generator on a list to print each list element `l` on a newline:

```
l := [3, 9, 27]
for x := l.iterate():
  Sys.println(x)
```

Converge's OO features are reminiscent of Smalltalk's [17] everything-is-an-object philosophy, but with a prototyping influence that was inspired by Abadi and Cardelli's theoretical work [18]. The internal object model is derived from ObjVLisp [19]. The system is bootstrapped with two base classes `Object` and `Class`, with the latter being a subclass of the former and both being instances of `Class` itself[1]: this provides a full metaclass ability whilst avoiding the class / metaclass dichotomy found in Smalltalk [20, 21]. Converge diverges from the Smalltalk school of OO since calls to functions within objects do not (unless the Meta-Object Protocol [22] is overridden) lookup those functions within the objects class: objects are created with slots containing direct references to the relevant functions. This allows objects to be freely and arbitrarily manipulated. Object instantiation in Converge is similar to Python: performing an application on a class creates a new object. Objects are created by the meta-classes' `new` method; the `init` function in the new object is then called to allow it to initialize itself. Note that whilst namespaces are determined statically at compile-time, slot references within objects are resolved entirely at run-time.

As in Python, Converge modules are executed from top to bottom when they are first imported. This is because functions, classes and so on are normal objects within a Converge system that need to be instantiated from the appropriate builtin classes – therefore the order of their creation can be significant e.g. a class *must* be declared before its use by a subsequent class as a superclass. Note that this only effects references made at the modules top-level – references e.g. inside functions are not restricted thus.

[1] The class an object is an instance of can be determined via its `instance_of` slot.

3 Compile-time meta-programming

3.1 A first example

The following program is a simple example of compile-time meta-programming, trivially adopted from it's TH cousin in [23]. `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 specific power function which calculates 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 fairly directly from TH: quasi-quote expressions `[| ... |]` build abstract syntax trees - ITree's in Converge's terminology - that represent the program code contained within them, and the splice annotation `$<<...>>` evaluates its expression at compile-time (and before VM instruction generation), replacing the splice annotation itself with the ITree resulting from its evaluation. When the above example has been compiled into VM instructions, `power3` essentially looks as follows:

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

By using the quasi-quotes and splicing mechanisms, we have been able to synthesise at compile-time a function which can efficiently calculate powers without resorting to recursion, or even iteration. Note how apart from the quasi-quotes and splicing mechanisms no extra features have been added to the base language – unlike LISP style languages, all parts of a Converge program are first-class elements regardless of whether they are executed at compile-time or run-time.

This terse explanation hides much of the necessary detail which can allow readers who are unfamiliar with similar systems to make sense of this synthesis. In the following sections, I explore the interface to compile-time meta-programming in more detail, building up the picture step by step.

3.2 Splicing

The key part of the 'powers' program is the splice annotation in the line `power3 := $<<mk_power(3)>>`. The top-level splice tells the compiler to evaluate the expression between the chevrons at compile-time, and to include the result of that

evaluation in the module for ultimate bytecode generation. In order to perform this evaluation, the compiler creates a temporary or 'dummy' module which contains all definitions up to, but excluding, the definition the splice annotation is a part of; to this temporary module a new splice function (conventionally called `$$splice$$`) is added which contains a single expression `return` *`splice expr`*. This temporary module is compiled to bytecode and injected into the running VM, whereupon the splice function is called. Thus the splice function 'sees' all the definitions prior to it in the module, and can call them freely – there are no other limits on the splice expression. The splice function must return a valid ITree which the compiler uses in place of the splice annotation.

Evaluating a splice expression leads to a new 'stage' in the compiler being executed. Converge's rules about which references can cross the staging boundary are simple: only references to top-level module definitions can be carried across the staging boundary (see section 3.4). For example the following code is invalid since the variable `x` will only have a value at run-time, and hence is unavailable to the splice expression which is evaluated at compile-time:

```
func f(x): $<<g(x)>>
```

Although the implementation of splicing in Converge is more flexible than in TH – where splice expressions can only refer to definitions in imported modules – it raises a new issue regarding forward references. This is tackled in section 3.8.

Note that splice annotations within a file are executed strictly in order from top to bottom, and that splice annotations can not contain splice annotations.

Permissible splice locations Converge is more flexible than TH in where it allows splice annotations. A representative sample of permissible locations is:

Top-level definitions. Splice annotations in place of top-level definitions must return an ITree, or a list of ITree's, each of which must be an assignment.

Function names. Splice annotations in place of function names must return a `Name` (see section 3.5).

Expressions. Splice annotations as expressions can return any normal ITree. A simple example is `$<<x>> + 2`. We saw another example in the 'powers' program with `power3 := $<<mk_power(3)>>`.

Within a block body. Splice annotations in block bodies (e.g. a functions body) accept either a single ITree, or a list of ITree's. Lists of ITree's will be spliced in as if they were expressions separated by newlines.

A contrived example that shows the last three of these splice locations (in order) in one piece of code is as follows:

```
func $<<create_a_name()>>():
  x := $<<f()>> + g()
  $<<list_of_exprs()>>
```

At compile-time, this will result in a function named by the result of `create_a_name` and containing 1 or more expressions, depending on the number of expressions returned in the list by `list_of_exprs`.

Note that the splice expressions must return a valid ITree for the location of a splice annotation. For example, attempting to splice in a sequence of expressions into an expression splice such as `$<<x>> + 2` results in a compile-time error.

3.3 The quasi-quotes mechanism

In the previous section we saw that splice annotations are replaced by ITree's. In many systems the only way to create ITree's is to use a verbose and tedious interface of ITree creating functions which results in a 'style of code [which] plagues meta-programming systems' [24]. LISP's quasi-quote mechanism allows programmers to build up LISP S-expressions (which, for our purposes, are analogous to be ITree's) by writing normal code prepended by the backquote ` notation; the resulting S-expression can be easily manipulated by a LISP program. Unfortunately LISP's syntactic minimalism is unrepresentative of modern languages, whose rich syntaxes are not as easily represented and manipulated.

MetaML and, later TH, introduce a quasi-quotes mechanism suited to syntactically rich languages. Converge inherits TH's Oxford quotes notation `[| ...|]` notation to represent a quasi-quoted piece of code. Essentially a quasi-quoted expression evaluates to the ITree which represents the expression inside it. For example, whilst the raw Converge expression `4 + 2` prints `6` when evaluated, `[| 4 + 2 |]` evaluates to an ITree which prints out as `4 + 2`. Thus the quasi-quote mechanism constructs an ITree directly from the users input - the exact nature of the ITree is of immaterial to the casual ITree user, who need not know that the resulting ITree is structured along the lines of *add(int(4), int(2))*.

To match the fact that splice annotations in blocks can accept sequences of expressions to splice in, the quasi-quotes mechanism allows multiple expressions to be expressed within it, split over newlines. The result of evaluating such an expression is, unsurprisingly, a list of ITree's.

Note that as in TH, Converge's splicing and quasi-quote mechanisms cancel each other out: `$<<[| x |]>>` is equivalent to `x` (though not necessarily vice versa).

Splicing within quasi-quotes In the 'powers' program, we saw the splice annotation being used within quasi-quotes. The explanation of splicing in section 3.2 would suggest that e.g. the splice inside the quasi-quoted expression in the `expand_power` function should lead to a staging error since it refers to variables `n` and `x` which were defined outside of the splice annotation. In fact, splices within quasi-quotes work rather differently to splices outside quasi-quotes: most significantly the splice expression itself is *not* evaluated at compile-time. Instead the splice expression is essentially copied as-is into the code that the quasi-quotes transforms to. For example, the quasi-quoted expression `[| $<<x>> + 2 |]` leads to an ITree along the lines of *add(x, int(2))* – the variable `x` in this case would need to contain a valid ITree. As this example shows, since splice annotations within quasi-quotes are executed at run-time they can access variables without staging concerns.

This feature completes the cancelling out relationship between splicing and quasi-quoting: `[| $<<x>> |]` is equivalent to *x* (though not necessarily vice versa).

3.4 Basic scoping rules in the presence of quasi-quotes

The quasi-quote mechanism can be used to surround any Converge expression to allow the easy construction of ITree's. Quasi-quoting an expression also has another important feature: it fully respects lexical scoping. Take the following contrived example of module `A`:

```
func x(): return 4

func y(): return [| x() * 2 |]
```

and module `B`:

```
import A, Sys

func x(): return 2

func main(): Sys.println($<<A.y()>>)
```

The quasi-quotes mechanisms ensures that since the reference to `x` in the quasi-quoted expression in `A.y` refers lexically to `A.x`, that running module `B` prints out `8`. This example shows one of the reasons why Converge needs to be able to statically determine namespaces: since the reference of `x` in `A.y` is lexically resolved to the function `A.x`, the quasi-quotes mechanism can replace the simple reference with an *original name*[2] that always evaluates to the slot `x` within the specific module `A` wherever it is spliced into, even if `A` is not in scope (or a different `A` is in scope) in the splice location.

Some other aspects of scoping and quasi-quoting require a more subtle approach. Consider the following (again contrived) example:

```
func f(): return [| x := 4 |]

func g():
  x := 10
  $<<f()>>
  y := x
```

What might one expect the value of `y` in function `g` to be after the value of `x` is assigned to it? A naïve splicing of `f()` into `g` would mean that the `x` within `[| x := 4 |]` would be captured by the `x` already in `g` – `y` would end with the value `4`. If this was the case, using the quasi-quote mechanism could potentially cause all sorts of unexpected interactions and problems. This problem of variable capture is well known in the LISP community, and hampered LISP macro implementations for many years until the concept of hygienic macros was invented

[2] Whilst this terminology is borrowed from TH, the implementations are quite different.

[25]. A new subtlety is now uncovered: not only is Converge able to statically determine namespaces, but variable names can be α-renamed without affecting the programs semantics. This is a significant deviation from the Python heritage. The quasi-quotes mechanism determines all bound variables in a quasi-quoted expression, and preemptively α-renames each bound variable to a guaranteed unique name that the user can not specify; all references to the variable are updated similarly. Thus the `x` within `[| x := 4 |]` will not cause variable capture to occur, and the variable `y` in function `g` will be set to `10`.

There is one potential catch: top-level definitions (all of which are assignments to a variable, although syntactic sugar generally obscures this fact) can not be α-renamed without affecting the programs semantics. This is because Converge's dynamic typing means that referencing a slot within a module can not in all cases be statically checked at runtime. Thus renaming top-level definitions could lead to run-time 'slot missing' exceptions being raised. Although the current compiler does not catch this case, since the user is unlikely to have cause to quasi-quote top-level definitions, barring it should be of little practical consequence.

Whilst the above rules explain the most important of Converge's scoping rules in the presence of quasi-quotes, upcoming sections add extra detail to the basic scoping rules explained in this section.

3.5 The `CEI` interface

At various points when compile-time meta-programming, one needs to interact with the Converge compiler. The Converge compiler is entirely contained within a package called `Compiler` which is available to every Converge program. The `CEI` module within the `Compiler` package is the officially sanctioned interface to the Compiler, and can be imported with `import Compiler.CEI`.

ITree functions Although the quasi-quotes mechanism allows the easy, and safe, creation of many required ITree's, there are certain legal ITree's which it can not express. Most such cases come under the heading of 'create an arbitrary number of *X*' e.g. a function with an arbitrary number of parameters, or an `if` expression with an arbitrary number of `elif` clauses. In such cases the `CEI` interface presents a more traditional meta-programming interface to the user that allows ITree's that are not expressible via quasi-quotes to be built. The downside to this approach is that recourse to the manual is virtually guaranteed: the user needs to know the name of the ITree element(s) required (each element has a corresponding function with a lower case name and a prepended 'i' in the `CEI` interface e.g. `ivar`), what the functions requirements are etc. Fortunately this interface needs to be used relatively infrequently; all uses of it are explained explicitly in this paper.

Names We saw in section 3.2 that the Converge compiler sometimes uses names for variables that the user can not specify using concrete syntax. The same technique is used by the quasi-quote mechanism to α-rename variables to ensure

that variable capture does not occur. However one of the by-products of the arbitrary ITree creating interface provided by the `CEI` interface is that the user is not constrained by Converge's concrete syntax; potentially they could create variable names which would clash with the 'safe' names used by the compiler. To ensure this does not occur, the `CEI` interface contains several functions – similar to those in recent versions of TH – related to names which the user is forced to use; these functions guarantee that there can be no inadvertent clashes between names used by the compiler and by the user.

In order to do this, the `CEI` interface deals in terms of instances of the `CEI.Name` class. In order to create a variable, a slot reference etc, the user must pass an instance of this class to the relevant function in the `CEI` interface. New names can be created by one of two functions. The `name(`*x*`)` function validates *x*, raising an exception if it is invalid, and returning a `Name` otherwise. The `fresh_name` function guarantees to create a unique `Name` each time it is called (this is the interface used by the quasi-quotes mechanism). This allows e.g. variable names to be created safely with the idiom `var := CEI.ivar(CEI.name("var_name"))`. Note that this facility opens the door for dynamic scoping (see section 3.7).

3.6 Lifting values

When meta-programming, one often needs to take a normal Converge value (e.g. a string) and obtain its ITree equivalent: this is known as *lifting* a value.

Consider a debugging function `log` which prints out the debug string passed to it; this function is called at compile-time so that if the global `DEBUG_BUILD` variable is set to `fail` (essentially the Converge analogue of 'false') there is no run-time penalty for using its facility. Noting that `pass` is the Converge no-op, a first attempt at such a function is as follows:

```
func log(msg):
  if DEBUG_BUILD:
    return [| Sys.println(msg) |]
  else:
    return [| pass |]
```

This function fails to compile: the reference to the `msg` variable causes the Converge compiler to raise the error `Var 'msg' is not in scope when in quasi-quotes (consider using $<<CEI.lift(msg)>>`. Rewriting the offending piece of code to the following gives the correct solution:

```
return [| Sys.println($<<CEI.lift(x)>>) |]
```

What has happened here is that the string value of `x` is transformed by the `lift` function into its abstract syntax equivalent. Constants are automatically lifted by the quasi-quotes mechanism: the two expressions `[| $<<CEI.lift("str")>> |]` and `[| "str" |]` are therefore equivalent.

Converge's refusal to lift the raw reference to `msg` in the original definition of `log` is a significant difference from TH, whose scoping rules would have caused `msg` to be lifted without an explicit call to `CEI.lift`. To explain this difference, assume the `log` function is rewritten to include the following fragment:

```
return [|
  msg := "Debug: " + $<<CEI.lift(msg)>>
  Sys.println(msg)
|]
```

In a sense, the quasi-quotes mechanism can be considered to introduce its own block: the assignment to the `msg` variable forces it to be local to the quasi-quote block. This needs to be the case since the alternative behaviour is nonsensical: if the assignment referenced to the `msg` variable outside the quasi-quotes then what would the effect of splicing in the quasi-quoted expression to a different context be? The implication of this is that referencing a variable within quasi-quotes would have a significantly different meaning depending on whether that variable has been assigned to within the quasi-quotes or not. Whilst it is easy for the Converge compiler writer to determine that a given variable was defined outside the quasi-quotes and should be automatically lifted in (or vice versa), from a user perspective the behaviour can be unnecessarily confusing. In fact Converge's quasi-quote mechanism originally did automatically lift variable references when possible, but this feature proved confusing in practise. To avoid this, Converge forces variables defined outside of quasi-quotes to be explicitly lifted into it. This also maintains a simple symmetry with Converge's main scoping rules: assigning to a variable in a block makes it local to that block.

3.7 Dynamic scoping

Sometimes the quasi-quote mechanisms automatic α-renaming of variables is not what is needed. For example consider a function `swap(x, y)` which should swap the values of the two variables passed as strings in its parameters. In such a case, we *want* the result of the splice to capture the variables in the spliced environment. Because the quasi-quotes mechanism only renames variables which it can determine statically at compile time, any variables created via the idiom `CEI.ivar(CEI.name(x))` and spliced into the quasi-quotes will not be renamed. The following simple definition of `swap` takes advantage of this fact:

```
func swap(x, y):
  x_var := CEI.ivar(CEI.name(x))
  y_var := CEI.ivar(CEI.name(y))
  return [|
    temp := $<<x_var>>
    $<<x_var>> := $<<y_var>>
    $<<y_var>> := temp
  |]
```

Note that the variable `temp` within the quasi-quotes *will* be α-renamed and thus will be effectively invisible to the code that it is spliced into, but that the two variables referred to by `x` and `y` will be scoped by their splice location. This function can be used thus:

```
a := 10
b := 20
$<<swap("a", "b")>>
```

Dynamic scoping also tends to be useful when a quasi-quoted function is created piecemeal with many separate quasi-quote expressions. In such a case, variable references can only be resolved successfully when all the resulting ITree's are spliced together since references to the functions parameters and so on will not be determined until that point. Since it is highly tedious to continually write `CEI.ivar(CEI.name("foo"))`, Converge provides the special syntax `&foo` which is equivalent.

3.8 Forward references and splicing

In section 3.2 we saw that when a splice annotation outside quasi-quotes is encountered, a temporary module is created which contains all the definitions up to, but excluding, the definition holding the splice annotation. This is a very useful feature since compile-time functions used only in one module can be kept in that module. However this introduces a real problem involving forward references. A forward reference is defined to be a reference to a definition within a module, where the reference occurs at an earlier point in the source file than the definition. If a splice annotation is encountered and compiles a subset of the module, then some definitions involved in forward references may not be included: thus the temporary module will fail to compile, leading to the entire module not compiling. Worse still, the user is likely to be presented with a highly confusing error telling them that a particular reference is undefined when, as far as they are concerned, the definition is staring at them within their text editor!

Consider the following contrived example:

```
func f1(): return [| 7 |]

func f2(): x := f4()

func f3(): return $<<f1()>>

func f4(): pass
```

If `f2` is included in the temporary module created when evaluating the splice annotation in `f3`, then the forward reference to `f4` will be unresolvable.

The solution taken by Converge ensures that, by including only a minimal subset of definitions in the temporary module, most forward references do not raise a compile-time error. We saw in section 3.4 that the quasi-quotes mechanism uses Converge's statically determined namespaces to calculate bound variables. We now use the same property to determine an expressions free variables.

When a splice annotation is encountered, the Converge compiler does not immediately create a temporary module. First it calculates the splice expressions free variables; any previously encountered definition which has a name in the set of free variables is added to a set of definitions to include. These definitions themselves then have their free variables calculated, and again any previously encountered definition which has a name in the set of free variables is added to the set of definitions to include. This last step is repeated until an iteration adds no new definitions to the set. At this point, Converge then goes back in order

over all previously encountered definitions, and if the definition is in the list of definitions to include, it is added to the temporary module. Recall that the order of definitions in a Converge file can be significant (see section 2): this last stage ensures that definitions are not reordered in the temporary module. Note also that free variables which genuinely do not refer to any definitions (i.e. a mistake on the part of the programmer) will pass through this scheme unmolested and will raise an appropriate error when the temporary module is compiled.

Using this method, the temporary module that is created and evaluated for the example looks as follows:

```
func f1(): return [| 7 |]

func $$splice$$(): return f1()
```

Thus there are no unresolvable forward references.

There is another advantage to this method: since it reduces the number of definitions in temporary modules it can lead to a significant saving in compile time, especially in files containing multiple splice annotations.

3.9 Compile-time `printf`

In this section I present the Converge equivalent of the TH compile-time `printf` function given in [12]. Such a function takes a format string such as `"%s has %d %s"` and returns a quasi-quoted function which takes an argument per '%' specifier and intermingles that argument with the main text string. For our purposes, we deal with decimal numbers `%d` and strings `%s`.

The motivation for a TH `printf` is that such a function is not expressible in base Haskell. Although Converge functions can take a variable number of arguments (as Python, but unlike Haskell), having a compile-time version still has two benefits over its run-time version: any errors in the format string are caught at compile-time; an efficiency boost.

This example assumes the existence of a function `split_format` which given a string such as `"%s has %d %s"` returns a list of the form `[PRINTF_STRING, " has ", PRINTF_INT, " ", PRINTF_STRING]` where `PRINTF_STRING` and `PRINTF_INT` are constants.

First we define the main `printf` function which creates the appropriate number of parameters for the format string (of the form `p0`, `p1` etc.). Parameters must be created by the `CEI` interface. An `iparam` has two components: a variable, and a default value (the latter can be set to `null` to signify the parameter is mandatory and has no default value). `printf` then returns an anonymous quasi-quoted function which contains the parameters, and a spliced-in expression returned by `printf_expr`:

```
func printf(format):
  split := split_format(format)
  params := []
  i := 0
  for part := split.iterate():
    if part == PRINTF_INT | part == PRINTF_STRING:
```

```
        params.append(CEI.iparam(CEI.ivar(CEI.name("p" + i.to_str())), null))
        i += 1
    return [|
      func ($<<params>>):
        Sys.println($<<printf_expr(split, 0)>>)
    |]
```

printf_expr is a recursive function which takes two parameters: a list representing the parts of the format string yet to be processed; an integer which signifies which parameter of the quasi-quoted function has been reached.

```
func printf_expr(split, param_i):
  if split.len() == 0:
    return [| "" |]
  param := CEI.ivar(CEI.name("p" + param_i.to_str()))
  if split[0].is_instance(String):
    return [| $<<CEI.lift(split[0])>> + $<<printf_expr(split[1 : ], param_i)>> |]
  elif split[0] == PRINTF_INT:
    return [| $<<param>>.to_str() + $<<printf_expr(split[1 : ], param_i + 1)>> |]
  elif split[0] == PRINTF_STRING:
    return [| $<<param>> + $<<printf_expr(split[1 : ], param_i + 1)>> |]
```

Essentially, printf_expr recursively calls itself, each time removing the first element from the format string list, and incrementing the param_i variable iff a parameter has been processed. This latter condition is invoked when a string or integer '%' specifier is encountered; raw text in the input is included as is, and as it does not involve any of the functions parameters, does not increment param_i. When the format string list is empty, the recursion starts to unwind.

When the result of printf_expr is spliced into the quasi-quoted function, the dynamically scoped references to parameter names in printf_expr become bound to the quasi-quoted functions' parameters. As an example of calling this function, $<<printf("%s has %d %s")>> generates the following function:

```
func (p0, p1, p2):
  Sys.println(p0 + " has " + p1.to_str() + " " + p2 + "")
```

so that evaluating the following:

```
$<<printf("%s has %d %s")>>("England", 39, "traditional counties")
```

results in England has 39 traditional counties being printed to screen.

This definition of printf is simplistic and lacks error reporting, partly because it is intended to be written in a similar spirit to its TH equivalent. Converge comes with a more complete compile-time printf function as an example, which uses an iterative solution with more compile-time and run-time error-checking. Simple benchmarking of the latter function reveals that it runs nearly an order of magnitude faster than its run-time equivalent – a potentially significant gain when a tight loop repeatedly calls printf.

4 Compiler structure

We have now seen a description of the majority of Converge's most important rules regarding compile-time meta-programming, and have seen several examples

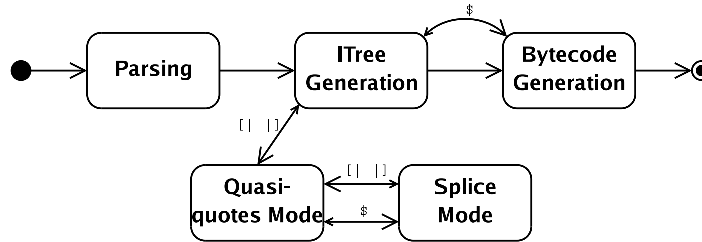

Fig. 1. Converge compiler states.

of useful functions which run at compile-time. However, most of this information has been given by concentrating on small, focused areas during each step – the 'big picture' has thus far been absent. Now that the reader hopefully has an appreciation of the components of Converge, we have sufficient information to take a look at the overall structure and operation of the compiler possible.

Figure 1 shows a (slightly non-standard) state-machine representing the most important states of the Converge compiler. Large arrows indicate a transition between compiler states; small arrows indicate a corresponding return transition from one state to another (in such cases, the compiler transitions to a state to perform a particular action and, when complete, returns to its previous state to carry on as before). Each of these states also corresponds to a distinct component within the compiler.

The stages of the Converge compiler can be described thus:

1. **Parsing.** The compiler parses an input file into a parse tree with an Earley parser [26]. Once complete, the compiler transitions to the next state.
2. **ITree Generation.** The compiler converts the parse tree into an ITree; this stage continues until the complete parse tree has been converted into an ITree.
   (a) **Splice mode / bytecode generation.** When it encounters a splice annotation in the parse tree, the compiler creates a temporary ITree representing a module. It then transitions temporarily to the bytecode generation state to compile. The compiled temporary module is injected into the running VM and executed; the result of the splice is used in place of the annotation itself when creating the ITree.
   (b) **Quasi-quotes mode / splice mode.** As the ITree generator encounters quasi-quotes in the parse tree, it transitions to the quasi-quote mode. Quasi-quote mode creates an ITree respecting the scoping rules and other features of section 3.4.
   If, whilst processing a quasi-quoted expression, a splice annotation is encountered, the compiler enters the splice mode state. In this state, the parse tree is converted to an ITree in a manner mostly similar to the normal ITree Generation state. If, whilst processing a splice annotation, a quasi-quoted expression is encountered, the compiler enters the quasi-quotes mode state again.

3. **Bytecode generation.** The complete ITree is converted into bytecode and written to disk.

5 Related work

Perhaps surprisingly, the template system in C++ has been found to be a fairly effective, if crude, mechanism for performing compile-time meta-programming [27, 23]. Essentially the template system can be seen as an ad-hoc functional language which is interpreted at compile-time. However this approach is inherently limited compared to the other approaches described in this section.

The dynamic OO language Dylan – perhaps one of the closest languages in spirit to Converge – has a similar macro system [10] to Scheme. In both languages there is a dichotomy between macro code and normal code; this is particularly pronounced in Dylan, where the macro language is quite different from the main Dylan language. As explained in the introduction, languages such as Scheme need to be able to explicitly identify macros over normal functions (although Bawden has suggested a way to make macros first-class citizens [28]). The advantage of explicitly identifying macros is that there is no added syntax for calling a macro: macro calls look like normal function calls. Of course, this could just as easily be considered a disadvantage: a macro call is in many senses rather different than a function call. In both schemes, macros are evaluated by a macro expander based on patterns – neither executes arbitrary code during macro expansion. This means that their facilities are limited in some respects – furthermore, overuse of Scheme's macros can lead to complex and confusing 'language towers' [29]. Since it can execute arbitrary code at compile-time Converge does not suffer from the same macro expansion limitations, but whether moving the syntax burden from the point of macro definition to call site will prevent the comprehension problems associated with Scheme is an open question.

Whilst there are several proposals to add macros of one sort or another to existing languages (e.g. Bachrach and Playford's Java macro system [30]), the lack of integration with their target language thwarts practical take-up.

Nemerle [31] is a statically typed OO language, in the Java / C# vein, which includes a macro system mixing elements of Scheme and TH's systems. Macros are not first-class citizens, but AST's are built in a manner reminiscent of TH. The disadvantage of this approach is that calculations often need to be arbitrarily pushed into normal functions if they need to be performed at compile-time.

Comparisons between Converge and TH have been made throughout this paper – I do not repeat them here. MetaML is TH's most obvious forebear and much of the terminology in Converge has come from MetaML via TH. MetaML differs from TH and Converge by being a multi-stage language. Using its 'run' operator, code can be constructed and run (via an interpreter) at run-time, whilst still benefiting from MetaML's type guarantees that all generated programs are type-correct. The downside of MetaML is that new definitions can not be introduced into programs. The MacroML proposal [32] aims to provide

such a facility, but in order to guarantee type-correctness forbids inspection of code fragments which limits the features expressivity.

Significantly, with the exception of Dylan, I know of no other dynamically typed OO language in the vein of Converge which supports any form of compile-time meta-programming natively.

6 Future work

Error reporting in compile-time systems is a largely unexplored topic [23]. Although Converge displays detailed stack traces when exceptions are raised and allows rudimentary customising of debugging data, compile-time meta-programming raises many issues related to debugging e.g. what is the source(s) of an error?

It would present a far more natural interface to the user if the Converge grammar itself can be extended to allow new syntaxes to be present in the source code in the manner outlined in [7]. Real-world implementations of a similar concept can be found in the Camlp4 pre-processor [33] which allows the normal OCaml grammar to be arbitrarily extended, and Nemerle [31]. A prototype Converge compiler showed promise with a simplified version of this feature.

7 Conclusions

In this paper I have outlined the Converge language, and in particular how its compile-time meta-programming features fit naturally into a dynamic OO programming language. By restricting namespaces to be statically calculated, Converge is able to provide a safe macro-esque facility that provides significant extra functionality over other dynamically typed OO languages, whilst maintaining the flexible development virtues of the dynamic language paradigm.

An implementation of Converge, which can execute all of the examples in this paper, is freely available under a MIT/BSD-style licence from `http://convergepl.org/`.

My thanks to Kelly Androutsopoulos for insightful comments on this paper.

This research was funded by a grant from Tata Consultancy Services.

References

1. Steele, Jr, G.L.: Growing a language. Higher-Order and Symbolic Computation **12** (1999) 221 – 236
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 (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. Kelsey, R., Clinger, W., Rees, J.: Revised(5) report on the algorithmic language Scheme. Higher-Order and Symbolic Computation **11** (1998) 7–105
5. Dybvig, R.K., Hieb, R., Bruggeman, C.: Syntactic abstraction in scheme. In: Lisp and Symbolic Computation. Volume 5. (1992) 295–326

6. Brabrand, C., Schwartzbach, M.: Growing languages with metamorphic syntax macros. In: Workshop on Partial Evaluation and Semantics-Based Program Manipulation. SIGPLAN, ACM (2000)
7. Cardelli, L., Matthes, F., Abadi, M.: Extensible grammars for language specialization. In: Proc. Fourth International Workshop on Database Programming Languages - Object Models and Languages. (1993) 11–31
8. Bawden, A.: Quasiquotation in lisp. Workshop on Partial Evaluation and Semantics-Based Program Manipulation (1999)
9. Ernst, M.D., Badros, G.J., Notkin, D.: An empirical analysis of C preprocessor use. IEEE Transactions on Software Engineering (2002)
10. Bachrach, J., Playford, K.: D-expressions: Lisp power, dylan style (1999) `http://www.ai.mit.edu/people/jrb/Projects/dexprs.pdf` Accessed Sep 22 2004.
11. Taha, W.: Multi-Stage Programming: Its Theory and Applications. PhD thesis, Oregon Graduate Institute of Science and Technology (1999)
12. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Proceedings of the Haskell workshop 2002, ACM (2002)
13. Tratt, L.: Model transformations and tool integration. Journal of Software and Systems Modelling (2004) Accepted for publication.
14. van Rossum, G.: Python 2.3 reference manual (2003) `http://www.python.org/doc/2.3/ref/ref.html` Accessed Sep 23 2004.
15. Tratt, L. (2004) `http://www.convergepl.org/documentation/refmanual/` Accessed Sep 23 2004.
16. Griswold, R.E., Griswold, M.T.: The Icon Programming Language. Third edn. Peer-to-Peer Communications (1996)
17. Goldberg, A., Robson, D.: Smalltalk-80: The Language. Addison-Wesley (1989)
18. Abadi, M., Cardelli, L.: A Theory of Objects. Springer (1996)
19. Cointe, P.: Metaclasses are first class: the ObjVLisp model. In: Object Oriented Programming Systems Languages and Applications. (1987) 156–162
20. Briot, J.P., Cointe, P.: Programming with explicit metaclasses in Smalltalk-80. In: Proc. OOPSLA '89. (1989)
21. Demers, F.N., Malenfant, J.: Reflection in logic, functional and object-oriented programming: a short comparative study. In: Proc. IJCAI'95 Workshop on Reflection and Metalevel Architectures and Their Applications in AI. (1995) 29–38
22. Kiczales, G., des Rivieres, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press (1991)
23. Czarnecki, K., John O'Donnell, J.S., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. **3016** (2004) 50–71
24. Weise, D., Crew, R.: Programmable syntax macros. In: Proc. SIGPLAN. (1993) 156–165
25. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: Symposium on Lisp and Functional Programming, ACM (1986) 151–161
26. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM **13** (1970)
27. Veldhuizen, T.: Using C++ template metaprograms. C++ Report **7** (1995) 36–43
28. Bawden, A.: First-class macros have types. In: Proc. 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. (2000) 133–141
29. Queinnec, C.: Macroexpansion reflective tower. In: Proc. Reflection'96. (1996) 93–104
30. Bachrach, J., Playford, K.: The java syntactic extender (jse). In: Proc. OOPSLA. (2001) 31–42

31. Skalski, K., Moskal, M., Olszta, P.: Meta-programming in Nemerle (2004) `http://nemerle.org/metaprogramming.pdf` Accessed Oct 1 2004.
32. 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 (2001)
33. de Rauglaudre, D.: Camlp4 - Reference Manual. (2003) `http://caml.inria.fr/camlp4/manual/` Accessed Sep 22 2004.