Compile-time meta-programming in a dynamically typed OO language

Laurence Tratt
Department of Computer Science, King's College London, Strand, London, WC2R 2LS, U.K.
laurie@tratt.net

ABSTRACT

Compile-time meta-programming allows programs to be constructed by the user at compile-time. Although LISP derived languages have long had such facilities, 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. In this paper I detail Converge's compile-time meta-programming facilities, much of which is adapted from Template Haskell, contain several features new to the paradigm. Finally I explain how such a facility might be integrated into similar languages.

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' [30] – 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 [27] or apply application specific optimizations [26].

The LISP family of languages, such as Scheme [20], 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 [14] 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 node's context within the tree. Macro systems operating at the lexing level are inherently less powerful, since they 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 C with its preprocessor (known as the CPP), a lexing system which is well-known for causing bizarre programming headaches due to unexpected side effects of its use (see e.g. [9, 5, 15]).

Despite the power of syntactic macro systems, and the widespread 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 the same 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 [2].

Relatively recently languages such as the multi-staged MetaML [31] (specifically its MacroML variant [16]) and Template Haskell (TH) [28] 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 a system, and LISP macro systems need the compiler to recognise that macro definitions are different from normal functions, languages such as TH distinguish only the macro call itself. In so doing, macros themselves can then be 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.

Dynamic Languages Symposium (DLS) '05, October 18, 2005, San Diego, CA, USA Copyright 2005 ACM ISBN 1-59593-283-6/05/10...$5.00

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 [34], I make frequent use of dynamically-typed Object Orientated (OO) languages such as Python [35]. Dynamically typed OO languages such as Python, Ruby [32] and Smalltalk [18] are increasingly recognised as having an important rôle to play in software development, particularly for the rapid development of software whose requirements evolve and change as the software itself develops [23]. Although they have traditionally been labelled somewhat dismissively as 'scripting languages', modern dynamic language implementations can often lead to programs which are close in run-time performance to their statically typed counterparts, whilst having a significantly lower development cost [24].

Since languages such as MetaML and TH are concerned with different aspects of program development (such as statically determinable type-safety), it is less than clear whether or not a similar compile-time system could naturally fit into a dynamically typed language. In this paper I present the Converge programming language, which can be seen in many ways as a Python derivative, both syntactically and semantically. However, Converge is a more experimental multi-paradigm language than Python and its ilk. It has been designed, in part, to explore if, and how, various language features can be integrated together. The purpose of this paper is not particularly to promote Converge as a new language. Rather, it is relatively speaking less effort to experiment with adding new features into a fresh language – that is a language with no historical baggage in either its specification or implementation. Note that although Converge is fundamentally an OO language, in the context of this paper this fact plays relatively little significance; the major focus is on Converge's dynamic nature.

In this paper I explore Converge's TH-derived compile-time meta-programming facilities, explaining the impact this has had on the language's design since it is important that the addition of such a feature does not unduly complicate other areas of the language. I also detail Converge features which add new power to the TH style of compile-time meta-programming. I then assess the relative utility of compile-time meta-programming in the context of a dynamically typed language, since this is one of the key novelties of this work. Finally I conclude the paper by drawing more general conclusions from the experience of integrating this feature into Converge, and suggest how compile-time meta-programming may be added to other dynamically typed languages such as Python and Ruby.

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 [33], 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 [35] 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 whereas even modern Python allows namespaces to be altered at run-time[1]. 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[2]. 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). Although both Converge and Python force fields to be referenced via the `self` variable, Converge's justification is subtly different. Since an objects slots are not always known at compile-time, without this feature namespaces would not be statically calculable.

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_Cache:
  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 [19]. As

[1]Prior to version 2.1, Python's namespaces were determined almost wholly dynamically; this often lead to subtle bugs, and hampered the utility of nested functions.

[2]Although not relevant to this paper, the `nonlocal` keyword allows assignment to variables scoped in an outer block.

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 Converge's OO features are reminiscent of Smalltalk's [18] everything-is-an-object philosophy, but with a prototyping influence that was inspired by Abadi and Cardelli's theoretical work [1]. The internal object model is derived from ObjVLisp [10]. An object is said to be comprised of *slots*, which are name / value pairs typically corresponding to the functions and fields defined by the class which created the object. Classes are provided as a useful, and common, convenience but are not fundamental to the object system. 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[3]: this provides a full metaclass ability whilst avoiding the class / metaclass dichotomy found in Smalltalk [8, 13]. Converge diverges from the Smalltalk school of OO since calls to functions within objects do not (unless the Meta-Object Protocol [21] 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: calling 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.

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 its TH cousin in [11]. `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)>>
```

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

The user interface to compile-time meta-programming is inherited directly from TH. Quasi-quote expressions `[| ... |]` build abstract syntax trees - ITree's in Converge's terminology - that represent the program code contained within them whilst respecting Converge's scoping rules. 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. This example also highlights a substantial difference from LISP derived macro schemes — Converge functions do not need to be explicitly identified as being macros in order that they are executed at compile-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 the necessary 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 necessary 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.

3.2.1 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 3.5.2).

Class fields. Splice annotations in place of class fields can return any normal ITree; by convention one would expect such ITree's to represent functions or variable definitions.

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' [37]. LISP's quasi-quote mechanism allows programmers to construct LISP S-expressions (which, for our purposes, are analogous to ITree's) by writing normal code prepended by the quote ` notation; the resulting S-expression can be easily manipulated by a LISP program. MetaML and, later TH, introduce a quasi-quotes mechanism suited to syntactically rich languages. Whilst the quasi-quotes mechanism is similar in operation to LISP's quote notation, the need to deal with the syntactic complexity of non-LISP languages raises several challenges which were largely left unanswered in previous meta-programming systems.

Converge inherits TH's Oxford quotes 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 if *x* does not contain a valid ITree).

3.3.1 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 do not cause a change of meta-level, variable references do not cause staging concerns.

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

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 has two important properties: it fully respects lexical scoping, and avoids inadvertent variable capture.

Consider first 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* 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. Although the implementations differ substantially, the concept of an original name is similar to its TH equivalent; in the case of this example, the original name can be considered to be `A.x`.

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 [22]. 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 name which is invalid in the normal concrete syntax. In so doing, Converge guarantees that the user can not inadvertently cause variable clashes. All references to the variable within the quasi-quotes 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 since this would almost certainly lead to run-time 'slot missing' exceptions being raised. Converge thus does not permit top-level definitions to be α-renamed.

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 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`. Converge's interface is similar to that found in systems such as Maya [4]. Although it may seem a mere implementation artifact, such interfaces to the compiler form a crucial and integral part of any compile-time meta-programming system.

3.5.1 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.

3.5.2 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. Although largely an implementation detail, all such such unique names are prefixed with `$$`. 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 ensures that *x* can not clash with unique names generated by Converge (by checking it is not prefixed by `$$`), 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). `fresh_name` takes an optional argument *x* which, if present, is incorporated into the generated name whilst still guaranteeing the uniqueness of the resulting name; this feature aids debugging by allowing the user to trace the origins of a fresh name. As a simple example, the expression `var := CEI.ivar(CEI.fresh_name())` creates a variable with a name guaranteed by Converge to be unique throughout the current compilation cycle, and which can not clash with any variable specified by the user.

Note that this interface 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, and is inherited from TH.

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. The `log` function is thus a safe means of performing what is often termed 'conditional compilation'. 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)>>`. Changing the segment in question 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 implicitly lifted `msg`. Although one might consider TH's implicit lifting somewhat bad design, TH's rules do have the virtue of being uniform. However in an imperative language such as Converge, variable assignment introduces a tricky corner case. To explain the 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 if the variable had been assigned to within the quasi-quotes or outside it. 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 TH's behaviour can be unnecessarily confusing. Converge's quasi-quote mechanism originally had the same behaviour in this respect as TH, but this resulted in fragile and hard to follow code. To avoid such problems, 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 two variables. In such a case, we *want* the result of the splice to capture the variables in the spliced environment. The following definition of `swap` expects to be passed two ITree's representing variables:

```
func swap(x, y):
  return [|
    temp := $<<x>>
    $<<x>> := $<<y>>
    $<<y>> := temp
  |]
```

It is initially tempting to try and use this function as follows:

```
a := 10
b := 20
$<<swap([| a |], [| b |])>>
```

However this causes a staging error since the `a` and `b` in the quasi-quotes do not refer to a bound variable either inside or outside the quasi-quotes when the splice expression is evaluated (the latter would be invalid anyway, due to Converge's lifting rules; see section 3.6). `swap` needs to be passed ITree's representing variable names, not references to variables. ITree's representing variable names can be constructed by the idiom `CEI.ivar(CEI.name(x))`. When such a variable name is spliced into a quasi-quotes it will not be renamed, thereby allowing dynamic scoping. A correct call to `swap` thus looks as follows:

```
$<<swap(CEI.ivar(CEI.name("a")), \
  CEI.ivar(CEI.name("b")))>>
```

In this case, the variable names constructed by the `CEI` interface are first spliced into the quasi-quotes in the `swap` function. The resulting ITree from the quasi-quotes is then spliced in place of the `swap` call, and the variable names dynamically capture the `a` and `b` variables.

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 function's 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. Notice that this notation prefixes a variable *name* — it has nothing to do with the value the variable contains. Using this syntax also allows `swap` to be called in the following less cumbersome fashion:

```
$<<swap([| &a |], [| &b |])>>
```

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. TH avoids this problem by allowing splice expressions to only refer to the import statements in its surrounding modules' contents. Converge's solution to the problem of forward references thus has no precedent in TH.

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. That same property is now used 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()
```

There are thus no unresolvable forward references in this example. Notice that Converge's approach to the forward reference problem is not a completely general solution since some forward references (particularly those to definitions beyond a splice site) are inherently unresolvable. Converge's approach is intended to significantly reduce the problem to the point that any unresolvable references are the result of programmer error.

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

3.9 Error reporting

Perhaps the most significant unresolved issue in compile-time meta-programming systems relates to error reporting [11]. Although Converge does not have complete solutions to all issues surrounding error reporting, it does contain some rudimentary features which may give insight into the form of more powerful error reporting features both in Converge and other compile-time meta-programming systems.

The first aspect of Converge's error reporting facilities relates to exceptions. When an exception is raised, detailed stack traces are printed out allowing the user to inspect the sequence of calls that led to the exception being raised. These stack traces differ from those found in e.g. Python in that each level in the stack trace displays the file name, line number and column number that led to the error. Displaying the column number allows users to make use of the fine-grained information to more quickly narrow down the precise source of an exception. Converge is able to display such detailed information because when it parses text, it stores the *source code information*: the file name, and character offset within the file of each token. Tokens are ordered into parse trees; parse trees are converted into ASTs; ASTs are eventually converted into VM instructions. At each point in this conversion, the source code information is retained. Thus every VM instruction in a binary Converge program has a corresponding debugging entry which records which file and character offset the VM instruction relates to. Whilst this does require more storage space than simpler forms of error information, the amount of extra space required is insignificant in the face of the vast storage resources now commonplace.

If Converge were a 'normal' programming language – i.e. *sans* compile-time meta-programming – then there would be no need to record the relation of each individual VM instruction, nor the source file that debugging entries are related to. However in order that error reporting in the presence of compile-time meta-programming is useful, such information must be recorded much differently in Converge. To see why this is the case, consider a file `A.cv`:

```
func f():
  return [| 2 + "3" |]
```

and a file B.cv:

```
import A

func main():
  $<<A.f()>>
```

When the quasi-quoted code in `A.f` is spliced in, and then executed an exception will be raised about the attempted addition of an integer and a string. The exception that results from running `B` is as follows:

```
Traceback (most recent call last):
  File "A.cv", line 2, column 13, in main
Type_Exception: Expected instance of Int, but
got instance of String.
```

The fact that the `A` module is pinpointed as the source of the exception may initially seem surprising, since the code raising the exception will have been spliced into the `B` module. This is however a deliberate design choice in Converge. Although the code from `A.f` has been spliced into `B.main`, when `B` is run the quasi-quoted code retains the information about its original source file, and not its splice location. To the best of my knowledge, this approach to customising error reporting in the face of compile-time meta-programming is unique.

3.9.1 Customizing source code information

Converge allows customization of the error-reporting information stored about a given ITree. Each ITree object has two slots `src_file` and `src_offset` which record the source file name and character offset that the ITree object relates to. I now assume the existence of a simple function `update_src_info` which takes three arguments: an ITree, a source file name, and a character offset. `update_src_info` should copy the input ITree, and as it does so alter the source info of the copied ITree to that provided by the user. By updating an ITree's source code information, the user can change the information printed in a stack traces.

However, whilst the simple definition of `update_src_info` is initially appealing, it has an unfortunate property that limits its use. Consider the following contrived example:

```
func f():
  return update_src_info([| 2 + "3" |], "X.cv", \
    39)

func g():
  return update_src_info([| 2 * $<<f()>> |], \
    "Y.cv", 78)
```

A user calling `g` would be forgiven for assuming that the ITree returned to them will contain some elements whose source code information relates to `X.cv`, and some which relate to `Y.cv`. However, the last call to `update_src_info` always 'wins', and thus the resulting ITree from `g` will have all its source code information relating to `Y.cv`. The reason for this is that the ITree created by `f` is spliced into the ITree created by `g` before the `update_src_info` function is called. In other words, the information that the ITree has been constructed in two parts is entirely lost to the `update_src_info` function. It should be noted that it would not be possible to work around this problem by modifying `update_src_info` to only alter the source code information of 'top level elements' since ITree's are nested to an arbitrary depth.

The behaviour of the `update_src_info` is thus limiting because its coarse-grained operation works particularly badly when ITree's are built up in stages, or by separate pieces of code, each of which wish to record different source code information.

3.9.2 Fine grained customization of source code information

In a TH-esque system only coarse-grained customization of source code information is possible. In Converge, fine grained customization of source code information is possible. This is achieved by adding a feature to Converge which is not present in TH: nested quasi-quotes. Essentially an outer quasi-quote returns the ITree of the code which would create the ITree of the nested quasi-quote. For example, using the `pp` function to pretty-print an ITree, the following nested code:

```
Sys.println([| [| 2 + "3" |] |].pp())
```

results in the following output:

```
CEI.ibinary_add(CEI.iint(2, "ct.cv", 484),
CEI.istring("3", "ct.cv", 488), "ct.cv", 486)
```

Nested quasi-quotes provide a facility which allows users to analyse the ITrees that plain quasi-quotes generate: one can see in the above that each ITree element contains a reference to the file it was contained within (`ct.cv` in this case) and to the offset within the file (`484` and so on). As this output shows, the ITree creating functions in the `CEI` module (see section 3.5) have a standard form. Each one takes zero or more initial arguments relative to the ITree item it creates, and ends with two mandatory arguments denoting the source file the ITree element is related to, and the character offset of the ITree element in that file.

The importance of nested quasi-quotes is they allow one to inspect and alter the code which creates an ITree i.e. a meta-level removed from the ITree itself. This means that one can manipulate the quasi-quoted code in isolation, without having to worry about other ITree's which might be spliced in when the quasi-quotes are actually executed. This is illustrated by changing the example code above to the following:

```
Sys.println([| [| 2 + $<<h()>> |] |].pp())
```

where `f` is an arbitrary function which returns a valid ITree. This code results in the following output:

```
CEI.ibinary_add(CEI.iint(2, "ct.cv", 129), \
  h(), "ct.cv", 131)
```

In a sense the nested quasi-quotes can be seen to defer the execution of h allowing the user to manipulate the ITree before the ITree from h is included. h will be when the result of the nested quasi-quotes is spliced into a program, thus 'cancelling out' one degree of nesting.

In order to facilitate working with nested quasi-quotes, the CEI module provides a function src_info_to_var which given an ITree representing quasi-quoted code essentially copies the ITree replacing the source code file and character offsets with variables src_file and src_offset. This new ITree is then embedded in a quasi-quoted function which takes two arguments src_file and src_offset. When the user splices in and then calls this function, they update the ITree's relation to source code files and offsets. Using this function in the following fashion:

```
Sys.println(CEI.src_info_to_var( \
  [| [| 2 + "3" |] |]).pp())
```

results in the following output:

```
unbound_func (src_file, src_offset){
  return CEI.ibinary_add(CEI.iint(2, \
    src_file, src_offset), CEI.istring("3", \
    src_file, src_offset), src_file, src_offset)
}
```

As an example of using this in practise, the original definitions of f and g would be changed to the following:

```
func f():
  return $<<CEI.src_info_to_var([| [| 2 + \
    "3" |] |])>>("X.cv", 39)

func g():
  return $<<CEI.src_info_to_var([| [| 2 * \
    $<<f()>> |] |])>>("Y.cv", 78)
```

Whilst this is perhaps slightly clumsy, it is interesting to note that by adding only the simple, uniform concept of nested quasi-quotes, complex manipulation of the meta-system is possible. In this case, this extra access to the meta-system has allowed fine-grained source code information to be controlled: the ITree generated by g will contain some elements whose source code information references X.cv and some who reference Y.cv. Note that quasi-quotes can be nested to arbitrary levels – as this may suggest, nested quasi-quotes expose that Converge's compile-time meta-programming is an analogue of ObjVLisp's 'golden braid' data model [10].

Converge's current approach is not without its limitations. Its chief problem is that it can only relate one source code location to any given VM instruction. There is thus an 'either / or' situation in that the user can choose to record either the definition point of the quasi-quoted code, or change it to elsewhere (e.g. to record the splice point). It would be of considerable benefit to the user if it is possible to record all locations which a given VM instruction relates to. Assuming the appropriate changes to the compiler and VM, then the only user-visible change would be that src_info_to_var would append src_file and src_offset information to elements in a given ITree, rather than overwriting the information it already possessed.

4. COMPILE-TIME META - PROGRAMMING IN USE

In this paper we have seen several uses of compile-time meta-programming. There are many potential uses for this feature, many of which are too involved to detail in the available space. For example, one of the most exciting uses of the feature has been in conjunction with Converge's extendable syntax feature (see section 7), allowing powerful DSL's to be expressed in an arbitrary concrete syntax. One can see similar work involving DSL's in e.g. [26, 11].

In this section I show two seemingly mundane uses of compile-time meta-programming: conditional compilation and compile-time optimization. Although mundane in some senses, both examples open up potential avenues not currently available to other dynamically typed OO languages.

4.1 Conditional compilation

Whereas languages such as Java attempt to insulate their users from the underlying platform an application is running on, languages such as Python and Ruby allow the user access to many of the lower-level features the platform provides. Many applications rely on such low-level features being available in some fashion. However for the developer who has to provide access to such features a significant problem arises: how does one sensibly provide access to such features when they are available, and to remove that access when they are unavailable?

The log function on page 54 was a small example of conditional compilation. Let us consider a simple but realistic example that is more interesting from an OO perspective. The POSIX fcntl (File CoNTrol) feature provides low-level control of file descriptors, for example allowing file reads and writes to be set to be non-blocking; it is generally only available on UNIX-like platforms. Assume that we wish to provide some access to the fcntl feature via a method within file objects; this method will need to call the raw function within the provided fcntl module iff that module is available on the current platform.

In Python for example, there are two chief ways of doing this. The first mechanism is for a File class to defer checking for the existence of the fcntl module until the fcntl method is called, raising an exception if the feature is not detected in the underlying platform. Callers who wish to avoid use of the fcntl method on platforms lacking this feature must use catch the appropriate exception. This rather heavy handed solution goes against the spirit of *duck typing* [32], a practise prevalent in languages such as Ruby and Python. In duck typing, one essentially checks for the presence of a method(s) which appear to satisfy a particular API without worrying about the type of the object in question. Whilst this is perhaps unappealing from a theoretical point of view, this approach is common in practise due to the low-cost flexibility it leads to. To ensure that duck typing is possible in our fcntl example, we are forced to use exception handling and the dynamic selection of an appropriate sub-class:

```
try:
  import fcntl
  _HAVE_FCNTL = True
except exceptions.ImportError:
  _HAVE_FCNTL = False
```

```
class Core_File:
  # ...

if _HAVE_FCNTL:
  class File(Core_File):
    def fcntl(op, arg):
      return fcntl.fcntl(self.fileno(), op, arg)
else:
  class File(Core_File):
    pass
```

Whilst this allows for duck typing, this idiom is far from elegant. The splitting of the `File` class into a core component and sub-classes to cope with the presence of the `fcntl` functionality is somewhat distasteful. This example is also far from scalable: if one wishes to use the same approach for more features in the same class then the resultant code is likely to be highly fragile and complex.

Although it appears that the above idiom can be encoded largely 'as is' in Converge, we immediately hit a problem due to the fact that module imports are statically determined. Thus a direct Converge analogue would compile correctly only on platforms with a `fcntl` module. However by using compile-time meta-programming one can create an equivalent which functions correctly on all platforms and which cuts out the ugly dynamic sub-class selection.

The core feature here is that class fields are permissible splice locations (see section 3.2.1). A splice which returns an ITree that is a function will have that function incorporated into the class; if the splice returns `pass` as an ITree then the class is unaffected. So at compile-time we first detect for the presence of a `fcntl` module (the `VM.loaded_module_names` function returns a list containing the names of all loaded modules); if it is detected, we splice in an appropriate `fcntl` method otherwise we splice in the no-op. This example make use of two hitherto unencountered features. Firstly, using an `if` construct as an expression requires a different syntax (to work around parsing limitations associated with indentation based grammars); the construct evaluates to the value of the final expression in whichever branch is taken, failing if no branch is taken[4]. Secondly the modified Oxford quotes `[d| ...|]` – *declaration quasi-quotes* – act like normal quasi-quotes except they do not α-rename variables in the top-level of the quasi-quotes; declaration quotes are typically most useful at the top-level of a module. The Converge example is as follows:

```
$<<if VM.loaded_module_names().contains("FCntl") {
  [d|
    import FCntl
    _HAVE_FCNTL := 1
  |]
}
else {
  [d| _HAVE_FCNTL := 0 |]
}>>

class File:
  $<<if _HAVE_FCNTL {
    [|
      func fcntl(op, arg):
        return FCntl.fcntl(self.fileno(), op, \
          arg)
    |]
  }
  else {
    [| pass |]
  }>>
```

[4]Note that failure in Converge is a concept inherited from Icon, and is not equivalent to raising an exception.

Although this example is simplistic in many ways, it shows that compile-time meta-programming can provide a conceptually neater solution than any purely run-time alternative since it allows related code fragments to be kept together. It also provides a potential solution to related problems. For example often portability related code in dynamically typed OO languages consists of many `if` statements which perform different actions depending on a condition which relates to querying the platform in use. Such code can become a performance bottleneck if called frequently within a program. The use of compile-time meta-programming can lead to a zero-cost run-time overhead. Perhaps significantly, the ability to tune a program at compile-time for portability purposes is the largest single use of the C preprocessor [15] – compile-time meta-programming of the sort found in Converge not only opens similar doors for dynamically typed OO languages, but allows the process to occur in a far safer, more consistent and more powerful environment than the C preprocessor.

4.2 Run-time efficiency

In this section I present the Converge equivalent of the TH compile-time `printf` function given in [28]. 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 the purposes of this paper, I deal only 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].conforms_to(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[5] – a potentially significant gain when a tight loop repeatedly calls printf.

[5]This large differential is in part due to the fact that the current Converge VM imposes a relatively high overhead on function application.

4.3 Compile-time meta-programming costs

Although compile-time meta-programming has a number of benefits, it would be naïve to assume that it has no costs associated with it. However although Converge's features have been used to build several small programs, and two systems of several thousand lines of code each, it will require a wider range of experience from multiple people working in different domains to make truly informed comments in this area.

One thing is clear from experience with LISP: compile-time meta-programming in its rawest form is not likely to be grasped by every potential developer [25]. To use it to its fullest potential requires a deeper understanding of the host language than many developers are traditionally used to; indeed, it is quite possible that it requires a greater degree of understanding than many developers are prepared to learn. Whilst features such as extendable syntax (see section 7) which are layered on top of compile-time meta-programming may smooth off many of the usability rough edges, fundamentally the power that compile-time meta-programming extends to the user comes at the cost of increased time to learn and master.

In Converge one issue that arises is that code which continually dips in and out of the meta-programming constructs can become rather messy and difficult to read on screen if overused in any one area of code. This is due in no small part to the syntactic considerations that necessitate a move away from the clean Python-esque syntax to something closer to the C family of languages. It is possible that the integration of similar features into other languages with a C-like syntax would lead to less obvious syntactic seams.

5. IMPLICATIONS FOR OTHER LANGUAGES AND THEIR IMPLEMENTATIONS

I believe that Converge shows that compile-time meta-programming facilities can be added in a seamless fashion to a dynamically-typed OO language. It seems reasonable to assume that this work could be equally applied to non-OO dynamically typed languages.

In this section I first pinpoint the relatively minimal requirements on language design necessary to allow the safe and practical integration of compile-time meta-programming facilities. Since the implementation of such a facility is quite different from a normal language compiler, I then outline the makeup of the Converge compiler to demonstrate how an implementation of such features may look in practice. Finally I discuss the requirements on the interface between user code and the languages compiler.

5.1 Language design implications

Although Converge's compile-time meta-programming facilities have benefited slightly from being incorporated in the early stages of the language design, there is surprisingly little coupling between the base language and the compile-time meta-programming constructs. The implications on the design of similar languages can thus be boiled down to the following two main requirements:

1. It must be possible to determine all namespaces stat-

ically, and also to resolve variable references between namespaces statically. This requirement is vital for ensuring that scoping rules in the presence of compile-time meta-programming are safe and practical (see section 3.4). Less importantly, this requirement also allows functions called at compile-time to be stored in the same module as splices which call them whilst avoiding the forward reference problem (see section 3.8).

2. Variables within namespaces other than the outermost module namespace must be α-renameable without affecting the programs semantics. This requirement is vital to avoid the problem of variable capture.

Note that there is an important, but non-obvious, corollary to the second point: when variables and slot names overlap then α-renaming can not take place. In section 3.4 we saw that, in Converge, top-level module definitions can not be renamed because the variable names are also the slot names of the module object. Since Converge forces all accesses of class fields via the `self` variable, Converge neatly sidesteps another potential place where this problem may arise. Fortunately, whilst many statically typed languages allow class fields to be treated as normal variables (i.e. making the `self.` prefix optional) many dynamically typed languages take a similar approach to Converge and should be equally immune to this issue in that context.

Only two constructs in Converge are dedicated to compile-time meta-programming. Practically speaking both constructs would need to be added to other languages:

1. A splicing mechanism. This is vital since it is the sole user mechanism for evaluating expressions at compile-time.
2. A quasi-quoting mechanism to build up AST's. Although such a facility is not strictly necessary, experience suggests that systems without such a facility tend towards the unusable [37].

5.2 Compiler structure

Typical language compilers follow a predictable structure: a parser creates a parse tree; the parse tree may be converted into an AST; the parse tree or AST is used to generate target code (be that VM bytecode, machine code or an intermediate language). Ignoring optional components such as optimizers, one can see that normal compilers need only two or three major components (depending on the inclusion or omission of an explicit AST generator). Importantly the process of compilation involves an entirely linear data flow from one component to the next. Compile-time meta-programming however necessitates a different compiler structure, with five major components and a non-linear data flow between its components. In this section I detail the structure of the Converge compiler, which hopefully serves as a practical example for compilers for other languages. Whether existing language compilers can be retro-fitted to conform to such a structure, or whether a new compiler would need to be written can only be determined on a case-by-case basis; however in either case this general structure serves as an example.

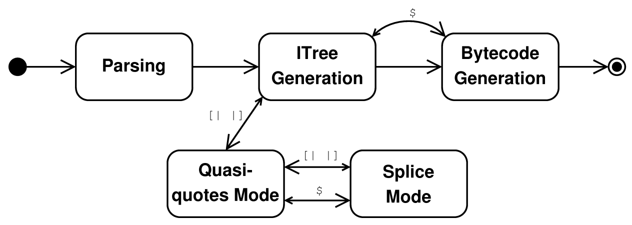

Figure 1: Converge compiler states.

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. 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. Since ITree's are exposed directly to the user, it is vital that the parse tree is converted into a format that the user can manipulate in a practical manner[6].
   (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. If, whilst processing a quasi-quoted expression, a nested quasi-quoted expression is encountered the compiler enters a new quasi-quotes mode.

[6]An early, and naïve, prototype of the Converge compiler exposed parse trees directly to the user. This quickly lead to spaghetti code.

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

5.3 Compiler interface

Converge provides the CEI module which user code can use to interact with the language compiler. Similar implementations will require a similar interface to allow two important activities:

1. The creation of fresh variable names (see section 3.5.2). This is vital to provide a mechanism for the user to generate unique names which will not clash with other names, and thus will prevent unintended variable capture. To ensure that all fresh names are unique, most practical implementations will probably choose to inspect and restrict the variable names that a user can use within ITree's via an analogue to Converge's `name` interface; this is purely to prevent the user inadvertently using a name which the compiler has guaranteed (or might in the future guarantee) to be unique.

2. The creation of arbitrary AST's. Since it is extremely difficult to make a quasi-quote mechanisms completely general without making it prohibitively complex to use, there are likely to be valid AST's which are not completely expressible via the quasi-quotes mechanism. Therefore the user will require a mechanism to allow them to create arbitrary AST fragments via a more-or-less traditional meta-programming interface [37].

5.3.1 Abstract syntax trees

One aspect of Converge's design that has proved to be more important than expected, is the issue of AST design. In typical languages, the particular AST used by the compiler is never exposed in any way to the user. Even in Converge, for many users the particulars of the AST's they generate via the quasi-quotes mechanism are largely irrelevant. However those users who find themselves needing to generate arbitrary AST's via the CEI interface, and especially those (admittedly few) who perform computations based on AST's, find themselves disproportionately affected by decisions surrounding the AST's representation.

At the highest level, there are two main choices surrounding AST's. Firstly should it be represented as an homogeneous, or heterogeneous tree? Secondly should the AST be mutable or immutable? The first question is relatively easy to answer: my experience suggests that homogeneous trees are not a practical representation of a rich AST. Whilst parse trees are naturally homogeneous, the conversion to an AST leads to a more structured and detailed tree that is naturally heterogeneous.

Let us then consider the issue of AST mutability. Initially Converge supported mutable AST's; whilst this feature has proved useful from time to time, it has also proved somewhat more dangerous than expected, in two ways. Firstly, one often naturally creates references to a given AST fragment from more than one node. Changing a node which is referenced by more than one other node can then result in unexpected changes, which all too frequently manifest themselves in hard to debug ways. Secondly, it prevents ITree's being partially type-checked as they are being created; thus invalid ITree's seep through to the bytecode generator, where the resulting error can be very difficult to relate to its origin. Future versions of Converge will force ITree's to be immutable, and I would recommend other languages consider this point carefully.

6. 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 [36, 11]. 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 [2] 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. 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' [25]. 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 the 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 such solutions are typically far less integrated into the language than Converge's systems. For example Bachrach and Playford's JSE system [3] system requires a heavy-weight pattern-matching to be added to Java, which adds a significant degree of complexity to the base language. A much different example is Weise and Crew's system for C [37] which is heavy-weight due to its lack of a quasi-quote equivalent.

Nemerle [29] 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 [16] aims to provide such a facility, but in order to guarantee type-correctness forbids inspection of code fragments which limits the features expressivity. MetaScheme [17] is a Scheme variant which is largely equivalent to MetaML, including possessing multi-level typing rules for statically determining some forms of correctness. Unlike most of the other systems in this section, MetaScheme is largely intended for generating program generators, and thus lacks features for practical use by programmers.

7. FUTURE WORK

Current versions of Converge sport an experimental extendable syntax system which allows new concrete syntaxes to be embedded within Converge code in a manner similar to [9]. At compile-time, user code is called to translate code in the new syntax into raw Converge code. This has already proved to be an immensely powerful approach, and is at least as good a justification for the utility of compile-time meta-programming as any other example in this paper. The current approach has one or two minor rough edges that require work before it is suitable for widespread use, but it has already proved sufficient to build three powerful and sizable (collectively approximately 10,000 lines of Converge code) model transformation system. Real-world implementations of a similar concept can be found in the Camlp4 pre-processor [12] which allows the normal OCaml grammar to be arbitrarily extended, and Nemerle [29]. The MetaBorg system [7] possibly has the closest resemblance to Converge's system from an external point of view, although Converge presents an homogeneous development environment to the developer whereas MetaBorg generally works in a heterogeneous environment.

Converge as presented in this paper is largely intended to be used as a two-stage meta-programming language in the TH vein; it is however possible to use it in a multi-stage fashion akin to MetaML. Making use of this facility currently involves calls to compiler internals. It should be possible to provide an analogue of MetaML's 'run' operator for synthesising and running code at run-time. Although I believe that Converge is best suited, and most useful, as a two-stage language, experimenting with multi-stage programming in Converge may open up interesting new avenues of research.

8. CONCLUSIONS

In this paper I have outlined the Converge language's compile-time meta-programming features, showing how they fit naturally into a dynamic OO programming language. Although much of Converge's compile-time meta-programming facilities are directly inherited from TH, I demonstrated how Converge elaborates on TH in several ways. Allowing splice expressions to refer to definitions in the surrounding module is an important usability gain; I then had to define the concept of forward references, and an appropriate solution to the problem they give rise to in this feature. I showed how Converge can customise error reporting error information, and how nested quasi-quotes, which generalize the meta-system, allow error reporting to be customised to a fine-grained level. I showed how Converge's features can be used to provide a number of desirable facilities such as conditional compilation which are either difficult or impossible to achieve with similar languages. Finally I used the experience gained from the Converge language and implementation to suggest how compile-time meta-programming might be added to similar languages.

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, and later the anonymous referees, for insightful comments this paper.

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

9. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer, 1996.

[2] J. Bachrach and K. Playford. D-expressions: Lisp power, Dylan style, 1999. `http://www.ai.mit.edu/people/jrb/Projects/dexprs.pdf` Accessed Sep 22 2004.

[3] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proc. OOPSLA*, pages 31–42, November 2001.

[4] J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proc. ACM SIGPLAN Conference on Programming language design and implementation*, pages 270 – 281, 2002.

[5] A. Bawden. Quasiquotation in LISP. Workshop on Partial Evaluation and Semantics-Based Program Manipulation, January 1999.

[6] C. Brabrand and M. Schwartzbach. Growing languages with metamorphic syntax macros. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN. ACM, 2000.

[7] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proc. OOPSLA'04*, Vancouver, Canada, October 2004. ACM SIGPLAN.

[8] J.-P. Briot and P. Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proc. OOPSLA '89*, October 1989.

[9] L. Cardelli, F. Matthes, and M. Abadi. Extensible grammars for language specialization. In *Proc. Fourth International Workshop on Database Programming Languages - Object Models and Languages*, pages 11–31, August 1993.

[10] P. Cointe. Metaclasses are first class: the ObjVLisp model. In *Object Oriented Programming Systems Languages and Applications*, pages 156–162, October 1987.

[11] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. 3016:50–71, 2004.

[12] D. de Rauglaudre. *Camlp4 - Reference Manual*, September 2003. `http://caml.inria.fr/camlp4/manual/` Accessed Sep 22 2004.

[13] F.-N. Demers and J. Malenfant. 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*, pages 29–38, August 1995.

[14] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. In *Lisp and Symbolic Computation*, volume 5, pages 295–326, December 1992.

[15] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 2002.

[16] S. E. Ganz, A. Sabry, and W. Taha. 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.

[17] R. Glück and J. Jrgensen. Multi-level specialization. volume 1706 of *LNCS*, pages 326 – 337, 1998.

[18] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, January 1989.

[19] R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.

[20] R. Kelsey, W. Clinger, and J. Rees. Revised(5) report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[21] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[22] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Symposium on Lisp and Functional Programming*, pages 151–161. ACM, 1986.

[23] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[24] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[25] C. Queinnec. Macroexpansion reflective tower. In *Proc. Reflection'96*, pages 93–104, April 1996.

[26] S. Seefried, M. M. T. Chakravarty, and G. Keller. Optimising embedded DSLs using Template Haskell. In *Draft Proc. Implementation of Functional Languages*, 2003.

[27] T. Sheard, Z. el Abidine Benaissa, and E. Pasalic. DSL implementation using staging and monads. In *Proc. 2nd conference on Domain Specific Languages*, volume 35 of *SIGPLAN*, pages 81–94. ACM, October 1999.

[28] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the Haskell workshop 2002*. ACM, 2002.

[29] K. Skalski, M. Moskal, and P. Olszta. Meta-programming in Nemerle, 2004. `http://nemerle.org/metaprogramming.pdf` Accessed Oct 1 2004.

[30] G. L. Steele, Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221 – 236, October 1999.

[31] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, October 1999.

[32] D. Thomas and A. Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.

[33] L. Tratt. *Converge Reference Manual*, September 2004. `http://www.convergepl.org/documentation/refmanual/` Accessed Sep 23 2004.

[34] L. Tratt. Model transformations and tool integration. *Journal of Software and Systems Modelling*, 4(2):112–122, May 2005.

[35] G. van Rossum. Python 2.3 reference manual, 2003. `http://www.python.org/doc/2.3/ref/ref.html` Accessed Aug 31 2005.

[36] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[37] D. Weise and R. Crew. Programmable syntax macros. In *Proc. SIGPLAN*, pages 156–165, 1993.