The Converge programming language

Technical report TR-05-01, Department of Computer Science, King's College London

Laurence Tratt
laurie@tratt.net

February 26, 2005

Contents

1 Introduction

This paper details the Converge programming language, a new dynamically typed imperative programming language capable of compile-time meta-programming, and with an extendable syntax. Although Converge has been designed with the aim of implementing different model transformation approaches as embedded DSL's in mind, it is also a General Purpose Language (GPL), albeit one with unusually powerful features.

The motivation for a new approach to implementing model transformation approaches is simple: existing languages, and their associated tool-chains, lead to long and costly implementation cycles for model transformation approaches. The justification for creating a new language, rather than altering an existing one, is far less obvious — it is reasonable to suggest that, given the vast number of programming languages already in existence, one of them should present itself as a likely candidate for modification.

There are two reasons why a new language is necessary to meet the aims of this paper. Firstly, in order to meet its aims, Converge contains a blend of features unique amongst programming languages; some fundamental design choices have been necessary to make these features coalesce, and imposing such choices retrospectively on an existing language would almost certainly lead to untidy results and backwards compatibility issues. Secondly, my personal experience strongly suggests that the complexity of modern languages implementations (when such implementations are available) can make adding new features a significant challenge. In short, I assert that it is easier in the context of model transformations to start with a fresh canvass than to alter an existing language.

This paper comes in three main parts. The first part documents the basics of the Converge language itself;. The second part details Converge's compile-time meta-programming and syntax extension facilities, including a section detailing suggestions for how some of Converge's novel features could be added to similar languages. The third part of this paper explains Converge's syntax extension facility, and documents a user extension which allows simple UML-esque modelling languages to be embedded within Converge. As well as being a practical demonstration of Converge's features, this facility is used extensively throughout the remainder of the paper.

2 Converge basics

This section gives a brief overview of the core Converge features that are relevant to the main subject of this paper. Since most of the basic features of Converge are similar to other similar programming language, this section is intentionally terse. However 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.

2.1 Syntax, scoping and modules

Converge's most obvious ancestor is Python [vR03] 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) 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's justification for this is subtly different than Python's, which has this feature to aid comprehension; although this is equally true in Converge, without this feature, namespaces would not be statically calculable since an objects slots are not always known at compile-time.

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))
```

[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.

Compiling and running this fragment looks as follows:

```
$ converge convergec -o fib.cvb fib.cv
$ converge convergel -o fib fib.cvb lib/libconverge.cvl
$ converge fib
8
```

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.

2.2 Functions

Converge uses the term function both in its traditional programming sense of a stand-alone function (or 'procedure'), and also for functions which reside in classes (often called methods). The reason for this is that 'normal' functions and 'methods' are not restricted in Converge to only their traditional rôles: 'normal' functions can reside in classes and 'methods' can reside outside of classes. When it is important to distinguish between the two, Converge has two distinct types: *unbound functions* ('normal' functions) and *bound functions* ('methods'). Bound functions expect to have an implicit first argument of the self object[2]; however they can not have arguments applied to it directly. Extracting a bound function from an object creates a *function binding* which wraps a bound function and a reference to the self object into an object which can then have arguments applied to it. Function bindings can be manually created by instantiating the `Func_Binding` class, which allows bound functions to be used with arbitrary self objects.

In normal use, Converge automatically assumes that the keyword `func` introduces an unbound function if it is used outside class, and a bound function if used inside a class. Using the `bound_func` or `unbound_func` keywords overrides this behaviour. Functions, bound or unbound, can have zero or more parameters; prefixing the final parameter in a function with a `*` denotes the 'var args' parameter.

An important feature of functions is their `apply` slot which applies a list of objects as parameters to the function. This allows argument lists of arbitrary size to be built and applied at run-time.

2.3 Goal-directed evaluation

An important, if less obvious, influence to Converge is Icon [GG96a]. Since Icon is likely to be unfamiliar to be most readers, a brief overview of Icon is instructive in understanding why it possesses an unusual, and interesting, feature set. Icon's chief designer was Ralph Griswold, and is a descendant of the SNOBOL series of programming languages – whose design team Griswold had been a part of – and SNOBOL's

[2] Note that unlike Python, Converge does not force the user to explicitly list `self` as a function parameter.

short-lived successor SL5. SNOBOL4 in particular was specifically designed for the task of string manipulation, but an unfortunate dichotomy between pattern matching and the rest of the language, and the more general problems encountered when trying to use it for more general programming issues ensured that, whilst successful, it never achieved mass acceptance; SL5 suffered from almost the opposite problem by having an over-generalized and unwieldy procedure mechanism. See Griswold and Griswold [GG93] for an insight into the process leading to Icon's conception. Since programs rarely manipulate strings in isolation, post-SL5 Griswold had as his aim to build a language which whilst being aimed at non-numeric manipulation also was usable as a general programming language. The eventual result was Icon [GG96a, GG96b], a language still in use and being developed to this day. In order to fulfil the goal of practical string manipulation, the premises on which Icon is founded are not only fundamentally different from those normally associated with GPLs, but are also tightly coupled with one another.

As Icon, Converge is an expression-based language, with similar notions of expression *success* and *failure*. In essence, expressions which succeed produce a value; expressions which fail do not produce a value and percolate the failure to their outer expression. For example the following fragment:

```
func main():
  x := 1 < 2
  y := 2 < 1
  Sys.println(x)
  Sys.println(y)
```

leads to the following output:

```
2
Traceback (most recent call last):
  File "expr.cv", line 5, column 13, in main
Unassigned_Var_Exception: Var has not yet been assigned to.
```

This is because when the expression `2 < 1` is evaluated, it fails (since 2 is not less than 1); the failure percolates outwards and prevents the assignment of a value to the variable `y`. Note that failure does not percolate outwards to arbitrary points: failure can not cross *bound expressions*. A bound expression thus denotes a 'stop point' for backtracking. The most obvious point at which bound expressions occur is when expressions are separated by newlines in an Converge program although bound expressions occur in various other points. For example, each branch of an `if` expression is bound, which prevents the failure of a branch causing the entire `if` expression to be re-evaluated. Converge directly inherits Icon's bound expression rules which largely preserve traditional imperative language evaluation strategies, even in the face of backtracking.

Success and failure are the building blocks for goal-directed evaluation, which is essentially a limited form of backtracking suitable for imperative programming languages. Functions which contain the `yield` statement are *generators* and can produce more than one return value. The `yield` statement is an alternative type of function return which effectively freezes the current functions closure and stack, and returns a value to the caller; if backtracking occurs, the function is resumed from its previous point of execution and may return another value. Generators complete by using the

`return` statement. Since the `return` statement returns the `null` object if no expression is specified, generators typically use `return fail` to ensure that the completion of the generator does not cause one final loop of the caller — `return`'ing the `fail` object causes a function to percolate failure to its caller immediately.

The most frequent use of generators is seemingly mundane, and occurs 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)
```

In simple terms, the `for` construct evaluates its condition expression and after each iteration of the loop backtracks in an attempt to pump the condition for more values. This idiom therefore subsumes the clumsy encoding of iterators found in most OO languages.

Generators can be used for much more sophisticated purposes. Consider first the following generator which generates all Fibonacci numbers from 1 to `high`:

```
func fib(high):
  a, b := [0, 1]
  while b < high:
    yield b
    a, b := [b, a + b]

  return fail
```

The `for` construct exhaustively evaluates its condition (via backtracking) until it can produce no more values. Therefore the following fragment prints all Fibonacci values from 1 to 100000:

```
for f := fib(100000):
  Sys.println(f)
```

The conjunction operator & conjoins two or more expressions; the failure of any part of the expression causes backtracking to occur. Backtracking resumes the most recent generator which is still capable of producing values, only resuming older generators when more recent ones are exhausted. Thus backtracking in Converge is entirely deterministic because the sequence in which alternatives are tried is explicitly specified by the programmer – this makes the evaluation strategy significantly different than that found in logic languages such as Prolog. If all expressions in a conjunction succeed, the value of the final expression is used as the value of the conjunction. If failure occurs, and there are no generators capable of producing more values to be resumed, then the conjunction itself fails.

Combining the `for` construct with the & operator can lead to terse, expressive examples such as the following which prints all Fibonacci numbers wholly divisible by 3 between 1 and 100000:

```
for Sys.println(f := fib(100000) & f % 3 == 0 & f)
```

A brief explanation of this can be instructive. Firstly `f := fib(100000)` pumps the `fib` generator and assigns each value it returns to the variable `f`. Since it is contained within the first expression of the & operator, when the `fib` generator completes,

its failure causes the `f := ...` assignment to fail, which causes the entire & operator to fail thus causing the `for` construct to fail and complete. Secondly `f % 3 == 0` checks whether `f` modulo 3 is equal to 0 or not; if it is not, failure occurs and backtracking occurs back to the `fib` generator. Since `f % 3 == 0`, if it succeeds, always evaluates to 0 (`==` evaluates to its right hand argument on success), the final expression of `f` produces the value of the variable which `Sys.println` then prints.

Neither Icon or Converge possess standard boolean logic since equivalent functionality is available through other means. The conjunction operator acts as an 'and' operator. Although the disjunction operator `|` is generally used as 'or', it is in fact a generator that successively evaluates all its expressions, producing values for those expressions which succeed. Thus in most circumstances the `|` operator neatly preserves the normal expectation of 'or' – that it evaluates expressions in order only until it finds one which succeeds – whilst also providing useful extra functionality.

This section has detailed the most important aspects of Converge's Icon-esque features, but for a more thorough treatment of these features I recommend Icon's manual [GG96a] — virtually all the material on goal-directed evaluation is trivially transferable from Icon to Converge. Gudeman [Gud92] presents a detailed explanation of goal-directed evaluation in general, with its main focus on Icon, and presents a denotational semantics for Icon's goal-directed evaluation scheme. Proebsting [Pro97] and Danvy et al. [DGR01] both take subsets of Icon chosen for their relevance to goal-directed evaluation, compiling the fragments into various programming languages (Danvy et. al also specify their Icon subset with a monadic semantics); both papers provide solid further reading on the topic.

2.3.1 While loops

Converge also contains a `while` construct. The difference between the `for` and `while` constructs is initially subtle, but is ultimately more pronounced than in most languages. In essence, each time a `for` loop completes, the construct backtracks to the condition expression and pumps it for a new value. In contrast, a `while` construct evaluates its expression anew after each iteration. This means that if the condition of a `while` construct is a generator it can only ever generate a maximum of one value before it is discarded. To emphasise this, the following code endlessly repeats, printing 1 on each iteration:

```
while f := fib(100000):
  Sys.println(f)
```

2.4 Data model

Converge's OO features are reminiscent of Smalltalk's [GR89] everything-is-an-object philosophy, but with a prototyping influence that was inspired by Abadi and Cardelli's theoretical work [AC96]. The internal data model is derived from ObjVLisp [Coi87]. Classes are provided as a useful, and common, convenience but are not fundamental to the object system in the way they are to most OO languages. The system is bootstrapped with two base classes `Object` and `Class`, with the latter being a subclass of

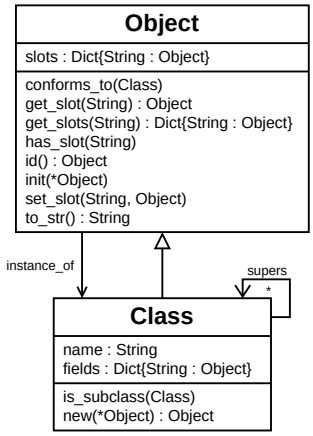

Figure 1: Core Converge data model.

the former and both being instances of `Class` itself: this provides a full metaclass ability whilst avoiding the class / metaclass dichotomy found in Smalltalk [BC89, DM95]. The core data model can be seen in figure 1. Note that the `slots` field in the `Object` class is conceptual and can only be accessed via the `get_slot` and `get_slots` functions.

In the Smalltalk school of OO, objects consist of a slot for each attribute in the class; calling a method on an object looks for a suitable method in the object's instantiating class (and possibly its superclasses). In contrast Converge, by default, creates objects with a slot for each field in a class, including methods. This therefore moves method overriding in classes to object creation time, rather than the more normal invocation time. This is possible since, as in Python, a functions name is the only factor to be taken into account when overriding. Object creation in Converge thus has a higher overhead than in most OO languages; this is offset by the fact that calling a function in an object is faster (since classes and super-classes do not need to be searched). The reason for this design decision is to ensure that all objects in a Converge system are 'free objects' in that they can be individually manipulated without directly affecting other objects, a feature which can prove useful when manipulating and transforming objects. This behaviour also mirrors the real world where, for example, changing a cars design on paper does not change actual cars on the road; it does not however reflect the behaviour of non-prototyping OO languages. For example, in Converge adding (or deleting) a method in a class does not automatically affect objects which are instances of that class, whereas in Python all of the classes instances would appear to grow (or lose) a method. From a practical point of view it is important to note that in normal use most users will be unaware of the difference between Converge's object creation scheme and its more normal counterparts. Note that this entire area of behaviour can be overridden by using meta-classes and the meta-object protocol (section 2.7).

In similar fashion to ObjVLisp, metaclasses are otherwise normal objects which possess a `new` slot. `Class` is the default metaclass; individual classes can instantiate

a different class via the `metaclass` keyword. Metaclasses typically subclass `Class` although this is not a requirement. A simple example of a useful metaclass is the following singleton metaclass [GHJV94] which allows classes to enforce that at most one instance of the class can exist in the system. Noting that `exbi` (EXtract and BInd) can be viewed as being broadly equivalent to other languages `super` keyword, the `Singleton` class is defined and used as follows:

```
class Singleton(Class):
  func new():
    if not self.has_slot("instance"):
      self.instance := exbi Class.new()
    return self.instance

class M metaclass Singleton:
  pass
```

Note that the `new` function in `Class` automatically calls the `init` function on the newly created object, passing it all the arguments that were passed to `new`.

2.4.1 Built-in data types

Converge provides a similar set of built-in data types to Python: strings, integers, dictionaries (key / value pairs) and sets. Dictionary keys and set elements should be immutable (though this is not enforced, violating this expectation can lead to unpredictable results), and must define `==` and `hash` functions, the latter of which should return an integer representing the object. All built-in types are subclasses of `Object`, and can be sub-classed by user classes (although the current implementation restricts user classes to sub-classing a maximum of one built-in type).

2.5 Comparisons and comparison overloading

Converge defines a largely standard set of binary operators. The lack of standard boolean logic in Converge means that the `not` operator is slightly unusual and is not classed as a comparison operator. Rather than `not` taking in a boolean value and returning its negated value, the `not` operator evaluates its expression and, if it fails, `not` succeeds and produces the `null` object. If the expression succeeds, the value produced is discarded and the `not` operator fails.

Objects can control their involvement in comparisons by defining, or overriding, the functions which are called by the various comparison operators. Functions are passed an object for comparison, and should fail if the comparison does not hold, or return the object passed to them if it does. Comparison operators are essentially syntactic sugar for calling a function of the same name in the left hand side object (e.g. the `==` operator looks up the `==` slot in an object).

Note that although the Converge grammar (section A) bundles the `is` operator into the `comparison_op` production, it is unlike the other comparison operators in that it tests two objects for equality of their identities, and can not be overridden by user objects.

2.6 Exceptions

Converge provides exception handling that is largely similar to Python. The `raise` expression raises an exception, printing a detailed stack-trace, the type of the exception and a message from the exception object itself. All exceptions must be instances of the `Exception` class in the `Exceptions` module. The `try ... catch` construct is used to test and capture exceptions.

2.7 Meta-object protocol

Converge implements a simple but powerful Meta-Object Protocol (MOP) [KdRB91], which allows objects to control all behaviour relating to slots. The default MOP is contained within the `Object` class and comprises the `get_slot`, `get_slots`, `has_slot` and `set_slot` functions. These can be arbitrarily overridden to control which slots the object claims it has (or has not), and what values such slots contain. Note that *all* accesses go through these functions; if they are overridden in a subclass, the user must exercise caution to call the 'master' MOP functions in the `Object` class to prevent infinite loops. The following example shows a MOP which returns a default value of `null` for unknown slot names:

```
class M:
  func get_slot(n):
    if not self.has_slot(n):
      return null
    return exbi Object.get_slot(n)
```

2.8 Differences from Python

Converge deliberately presents a feature set which can be used in a highly similar fashion to Python. Programmers used to Python can easily use Converge in a Python-esque fashion although they will miss out on some of Converge's more advanced features. The chief differences from Python are that Converge is a more static language, able to make stronger guarantees about namespaces, and that Converge is an expression based language rather than Python's statement based approach. Converge has a more uniform object system, and less reliance on a battery of globally available builtin functions than Python.

One small change from Python to Converge is a generalization of the somewhat confusingly named `finally` branch which can be attached to Python's `for` and `while` loops. The `finally` branch is executed if the loop construct terminates naturally (i.e. `break` is not called). Converge renames the `finally` branch to `exhausted` and also allows a `broken` branch to be added which will be called if a `break` is encountered. A slightly contrived example of this feature is as follows:

```
high := 10000
for x := fib(high):
  if x % 9 == 0:
    break
exhausted:
  Sys.println("No Fibonacci numbers wholly divisible by 9 upto ", high)
broken:
  Sys.println("Fibonacci number ", x, " wholly divisible by 9")
```

2.9 Differences from Icon

Converge's expression system is highly similar to Icon. Provided they can adjust to the Python-esque veneer, Icon programmers will have little difficulty exploiting Converge's expression system and implementation of goal-directed evaluation. There are however two significant differences in Converge's functions and generators.

Firstly, whereas Icon functions which do not have a `return` expression at the end of a function have an implicit `return fail` added, Converge functions instead default to `return null` (as do Python functions). Icon takes its approach so that generators do not accidentally return an extra object when they should instead fail, and Converge originally took the same approach as Icon. However in practise it is quite common, when developing code, to write incomplete functions — often one part of the code not initially filled in is the functions final `return` expression. Such functions then cause seemingly bizarre errors since they do not return a value, causing assignments in calling functions to fail and so on (indeed, this happened surprisingly frequently in the early stages of Converge development). Since the proportion of generators to normal functions is small, it seems more sensible to optimise the safety of normal functions at the expense of the safety of generators. As can be seen from section 2.3, generators in Converge generally have `return fail` as their final action in order to emulate Icon's behaviour.

Secondly, Converge does not propagate generation across a `return` expression. In Icon, if `f` is a generator then `return f()` turns the function containing the `return` expression into a generator itself which produces all the values that `f` produces. Converge does not emulate this behaviour, which somewhat arbitrarily turns `return` into a sort of `for` construct in certain situations that can only be determined by knowing whether the expression contains a generator. The same behaviour can be obtained in Converge via the following idiom:

```
for yield f()
return fail
```

2.10 Implementation

The current Converge implementation consists of a Virtual Machine (VM) written in C, and a compiler written in Converge itself (the current compiler was bootstrapped several generations ago from a much simpler Python version). The VM has a simplistic semi-conservative garbage collector which frees the user from memory management concerns. The VM uses a continuation passing technique at the C level to make the implementation of goal-directed evaluation reasonably simple and transparent from the point of view of extension modules. Its instruction set is largely based on Icon's, although the VM implementation itself shares more in common with modern VM's such as Python's.

This paper is not overly concerned with the implementation of the VM and compiler. Interested readers are encouraged to visit `http://convergepl.org/` where the VM and compiler can be downloaded and inspected.

2.11 Parsing

An aspect of Converge and its implementation that is particularly important throughout this paper is its ability to easily parse text. Converge implements a parser toolkit (the Converge Parser Kit or CPK) which contains a parsing algorithm based on that presented by Earley [Ear70]. Earley's parsing algorithm is interesting since it accepts and parses any Context Free Grammar (CFG) — this means that grammars do not need to be written in a restricted form to suit the parsing algorithm, as is the case with traditional parsing algorithms such as LALR. Practical implementations of Earley parsers have traditionally been scarce, since the flexibility of the algorithm results in slower parsing times than traditional parsing algorithms. The CPK utilises some (though not all) of the additional techniques developed by Aycock and Horspool [AH02] to improve its parsing time, particularly those relating to the ε production. Even though the CPK contains an inefficient implementation of the algorithm, on a modern machine, and even with a complex grammar, it is capable of parsing in the low hundreds of lines per second which is sufficient for the purposes of this paper. The performance of more sophisticated Earley parsers such as Accent [Sch] suggest that the CPK's performance could be raised by approximately an order of magnitude with relatively little effort.

Parsing in Converge is preceded by a tokenization (also known as lexing) phase. The CPK provides no special support for tokenization, since the built-in regular expression library makes the creation of custom tokenizers trivial. Tokenizers are expected to return a list of objects, each of which has slots `type`, `value`, `src_file` and `src_offset`. The first two slots represent the type (i.e. `ID`) and value (i.e. `height`) of a token and must be strings; the latter two slots record both the file and character offset within the file that a particular token originated in. The tokenizer for Converge itself is somewhat unusual in that it needs to understand about indentation in order that the grammar can be expressed satisfactorily. Essentially each increase in the level of indentation results in a `INDENT` token being generated; each decrease results in a `DEDENT` followed by a `NEWLINE` token. Each newline on the same level of indentation results in a `NEWLINE` token.

The CPK implements an EBNF style system – essentially a BNF system with the addition of the Kleene star. CPK production rules consist of a rule name, and one or more alternatives. Each alternative consists of tokens, references to other rules and groupings. Currently the only form of grouping accepted is the Kleene star. The CPK grammar itself is as follows:

⟨*grammar*⟩ ::= ⟨*rule*⟩*

⟨*rule*⟩ ::= `'ID'` ⟨*rule_alternative*⟩*

⟨*rule_alternative*⟩ ::= `':='` ⟨*rule_elem*⟩*
| `':='` ⟨*rule_elem*⟩* `'%PRECEDENCE'` `'INT'`

⟨*rule_elem*⟩ ::= ⟨*atom*⟩
| ⟨*grouping*⟩

⟨*grouping*⟩ ::= ‘`{`’ ⟨*atom*⟩ ‘`}*`’

⟨*atom*⟩ ::= ‘`"`’ ‘`TOKEN`’ ‘`"`’
| ‘`ID`’

Since Earley grammars can express any CFG, grammars can be ambiguous — that is, given inputs can satisfy the grammar in more than one way. In order to disambiguate between alternatives when building the parse tree, the CPK allows grammar rules to have a precedence attached to them; if more than one rule has been used to parse a given group of tokens, the rule with the highest precedence is used[3].

In order to use the CPK, the user must provide it with a grammar, the name of a start rule within the grammar, and a sequence of tokens. The result of a CPK parse is an automatically constructed parse tree, which is represented as a nested Converge list of the form [*production name, token or list*$_1$*, ..., token or list*$_n$]. The following program fragment shows a CPK grammar for a simple calculator:

```
GRAMMAR := """
S ::= E
E ::= E "+" E  %precedence 10
  ::= E "*" E  %precedence 30
  ::= "(" E ")"
  ::= N "INT"  %precedence 10
N ::= "-"
  ::=
"""
```

Assuming the existence of a suitable `tokenize` function, an example program which uses this grammar to parse input is as follows:

```
import CPK.Grammar, CPK.Parser

func calc_parse(input):
  grammar := Grammar.Grammar(GRAMMAR, "S")
  tokens := tokenize(input)
  parser := Parser.Parser(grammar)
  tree := parser.parse(tokens)
  Sys.println(tree)
```

The parse tree is printed out as:

```
["S", ["E", ["E", ["N"], <INT 5>], <+>, ["E", ["E", ["N"], <INT 2>],
<*>, ["E", ["N"], <INT 3>]]]]
```

This is somewhat easier to visualize when using the `parse_tree` function in a `Parser` instance to format the list as a tree:

```
S->
  E ->
    E ->
      N ->
      INT <5>
    +
    E ->
      E ->
```

[3]Note that there is another, much rarer, type of ambiguity involving alternatives which contain different number of tokens. These are currently always resolved in favour of the alternative containing the least number of tokens, no matter its precedence. This generally gives the expected behaviour, but can cause problems in some rare cases. This limitation is purely due to a naïve implementation.

```
    N ->
      -
    INT <2>
  *
  E ->
    N ->
    INT <3>
```

The full Converge grammar can be seen in appendix A.

2.12 Related work

This section has made several comparisons between Converge, and Icon and Python in particular. These are not repeated in this subsection.

The Unicon project [JMPP03] is in the reasonably advanced stages of extending Icon with object orientated features. It differs significantly from Converge in maintaining virtually 100% compatibility with Icon. Unicon's extensions to Icon, effectively being a bolt-on to the original, mean the resulting language contains more visible seams than does Converge. Godiva Godiva [Jef02], which aims to be a 'very high level dialect of Java' incorporating goal-directed evaluation. In reality, Godiva's claim to be a dialect of Java is slightly tenuous: whilst it shares some syntax, the semantics are substantially different. Neither Unicon nor Godiva are meta-circular, and both are less dynamic languages than Converge.

3 Compile-time meta-programming

3.1 Background

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' [Ste99] – 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 [SeABP99] or apply application specific optimizations [SCK03].

The LISP family of languages, such as Scheme [KCR98], 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 [DHB92] 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 [BS00]: 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. [CMA93, Baw99, EBN02]).

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 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 with the host language [BP99].

Relatively recently languages such as the multi-staged MetaML [Tah99] and Template Haskell (TH) [SJ02] 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. In this section I detail an extension to the core Converge language which adds compile-time meta-programming facilities similar to TH. Since this is the first time that facilities of this nature have been added to a dynamically-typed OO language such as Converge, section 4 details the implications of adding such a feature to similar languages.

3.2 A first example

The following program is a simple example of compile-time meta-programming, trivially adopted from it's TH cousin in [CJOT04]. `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.

3.3 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 3.5). 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 3.9.

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.

3.3.1 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 3.6.2).

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.

3.4 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' [WC93]. 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).

3.4.1 Splicing within quasi-quotes

In the 'powers' program, we saw the splice annotation being used within quasi-quotes. The explanation of splicing in section 3.3 would seem to suggest that 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).

3.5 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*[4] 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 [KFFD86]. 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

[4] This terminology is borrowed from TH, but with a much different implementation.

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.

3.6 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`.

3.6.1 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.

3.6.2 Names

Section 3.3 showed 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"))`. `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. Note that the `name` interface opens the door for dynamic scoping (see section 3.8).

3.7 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` 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)>>`. 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 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 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.

3.8 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 succinct 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.

3.9 Forward references and splicing

In section 3.3 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 ex-

cluding, 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 3.5 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 2.4): 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.

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.

3.10 Compile-time meta-programming in use

In this paper thus far 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 5), allowing powerful DSL's to be expressed in an arbitrary concrete syntax. One can see similar work involving DSL's in e.g. [SCK03, CJOT04].

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.

3.10.1 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 23 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* [TH00], 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 3.3.1). 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. Secondly the modified Oxford quotes `[d| ...|]` – *declaration quasi-quotes* – act like normal quasi-quotes except they do not α-rename variables; 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 |]
  }>>
```

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 portability related code in dynamically typed OO languages often 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 [EBN02] – 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.

3.11 Run-time efficiency

In this section I present the Converge equivalent of the TH compile-time `printf` function given in [SJ02]. 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].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.

3.12 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 [Que96]. 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 5) 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 over-used 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.

3.13 Error reporting

Perhaps the most significant unresolved issue in compile-time meta-programming systems relates to error reporting [CJOT04]. 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 file name, line number and column number 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, information about the source code elements is retained. Thus every VM instruction in a binary Converge program has a corresponding debugging entry which records which file, line number and column number the VM instruction relates

to. Whilst this does require more storage space than simpler forms of error information, the amount of space required is insignificant when the vast storage resources of modern hardware are considered.

Whilst the base language needs to record the related source offset of each VM instruction, the source file a VM instruction relates to is required only due to compile-time meta-programming. 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 error reporting in the face of compile-time meta-programming is unique. As points of comparison, TH is not able to produce any detailed information during a stack-trace and SCM Scheme [Jaf03] pinpoints the source file and line number of run-time errors as that of the macro call site. In SCM Scheme if the code that a macro produces contains an error, all the user can work out is which macro would have led to the problem — the user has no way of knowing which part of the macro may be at fault.

Converge allows customization of the error-reporting information stored about a given ITree. Firstly Converge adds a feature 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 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). The `CEI` module provides a function `src_info_to_var`

which given an ITree representing quasi-quoted code essentially copies the ITree[6] replacing the source code file and 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)
}
```

In practice when one wishes to customise the claimed location of quasi-quoted code, the nested quasi-quotes need to be cancelled out by a splice. For example, to change source information to be offset 77 in the file `nt.cv` we would use the following code:

```
return $<<CEI.src_info_to_var([| [| 2 + "3" |] |], "nt.cv", 77>>
```

Whilst this appears somewhat clumsy, it is worth noting that by adding only the simple concept of nested quasi-quotes, complex manipulation of the meta-system is possible.

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 within a given ITree, rather than overwriting the information it already possessed.

3.14 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 [Vel95, CJOT04]. 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 [BP99] 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 identify macros as distinct from normal functions (although Bawden has suggested a way

[6] In the current implementation, the `src_info_to_var` actually mutates ITrees, but for reasons explained in section 4.3.1 this will not be possible in the future.

to make macros first-class citizens [Baw00]). 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' [Que96]. 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 [BP01]), the lack of integration with their target language thwarts practical take-up.

Nemerle [SMO04] 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 section – 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 [GST01] 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.

4 Implications of Converge's compile-time meta-programming 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 and that such facilities provide useful functionality not available previously in such 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.

4.1 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 statically, 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 3.5). Slightly 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 3.9).

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 3.5 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) most 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 [WC93].

4.2 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

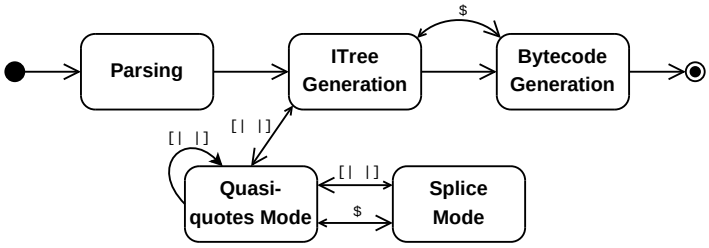

Figure 2: Converge compiler states.

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 2 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[7].

   (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

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

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 3.5.

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.

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

4.3 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 3.6.2). 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 [WC93].

4.3.1 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 ITree's they generate via the quasi-quotes mechanism are largely irrelevant. However those users who find themselves needing to generate arbitrary ITree's via the `CEI` interface, and especially those (admittedly few) who perform computations based on ITree's, find themselves disproportionately affected by decisions surrounding the ITree'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: 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. This is because 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. Since it is not possible to check for this problem in the general case, the user is ultimately responsible for ensuring it does not occur; in practise this has proved to be unrealistic, and gradually all ITree-mutating code has been banished from Converge code. Future versions of Converge will force ITree's to be immutable, and I would recommend other languages consider this point carefully.

5 Syntax extension for DSL's

Converge has a simple but powerful facility allowing users to embed arbitrary sequences of tokens within Converge source files. At compile-time these tokens are passed to a designated user function, which is expected to return an AST. This allows the user to extend the languages syntax in an arbitrary fashion, meaning that DSLs can be embedded within normal Converge code. Although this feature is somewhat less developed than the other aspects of Converge, it is useful even its current state.

Essentially a DSL fragment is an indented block containing an arbitrary sequence of tokens. The DSL block is introduced by a variant on the splice syntax `$< expr >` where *`expr`* should evaluate to a function (the *DSL implementation function*). The DSL function will be called at compile-time with a list of tokens, and is expected to return an AST which will replace the DSL block in the same way as a normal splice. Compile-time meta-programming is thus the mechanism which facilitates embedding DSLs.

An example DSL fragment is as follows. Colloquially this block is referred to as 'a `TM.model_class`':

```
import TM.TM

$<TM.model_class>:
```

```
abstract class ML1_Element {
  name : String;

  inv nonempty_name:
    name != null and name.len() > 0
}
```

Note that the DSL fragment is written in an entirely different syntax than Converge itself.

Currently DSL blocks are automatically tokenized by the Converge compiler using its default tokenization rules — this is not a fundamental requirement of the technique, but a peculiarity of the current implementation. More sophisticated implementations might choose to defer tokenization to the DSL implementation function. However using the Converge tokenizer has the advantage that normal Converge code can be embedded inside the DSL itself assuming an appropriate link from the DSL's grammar to the Converge grammar.

5.1 DSL implementation functions

DSL implementation functions follow a largely similar sequence of steps in order to translate the input tokens into an ITree:

1. Alter the input tokens as necessary. Since DSL's often use keywords that are not part of the main Converge grammar, such alterations mostly take the form of replacing `ID` tokens with specific keyword tokens.
2. Parse the input tokens according to the DSL's grammar.
3. Traverse the parse tree, translating it into an ITree.

Section 6 explores these steps in greater detail via a concrete example.

5.2 Related work

Real-world implementations of a similar concept are surprisingly rare. The Camlp4 pre-processor [dR03] allows the normal OCaml grammar to be arbitrarily extended, and is an example of a heterogeneous syntax extension system in that the system doing the extension is distinct from the system being extended. The MetaBorg system [BV04] is a heterogeneous system that can be applied to any language; more sophisticated than the Camlp4 pre-processor, from an external point of view it more closely resembles Converge's functionality, although the implementations and underlying philosophies are still very different.

I am currently aware of only two homogeneous syntax extension systems apart from Converge. Nemerle [SMO04] allows limited syntax extension via its macro scheme. The commercial XMF tool [CESW04] presents only a small core grammar, with many normal language concepts being grammar extensions on top of the core grammar. Grammar extensions are compiled down into XMF's AST. XMF is thus much closer in spirit to Converge, although the example grammar extensions available suggest that XMF's compile-time facilities may be less powerful than Converge's,

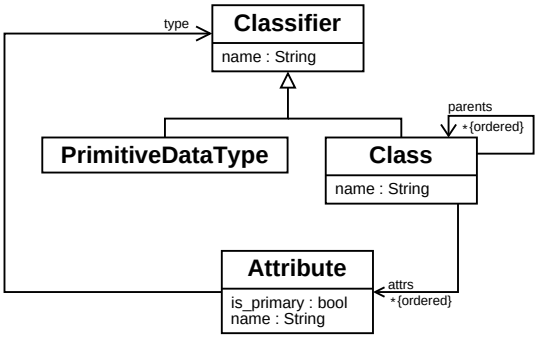

Figure 3: ‘Simple UML’ model.

seemingly being based on a simplified version of TH’s features. If true, this may limit the complexity of the grammar extensions.

6 Modelling language DSL

This section presents an example of a Converge DSL for expressing typed modelling languages; modelling languages can be instantiated create models. In its current simplistic form, the modelling DSL operates with a fixed number of meta-levels in that it defines modelling languages that can create models, but those models are terminal instances (in ObjVLisp’s terminology) — that is, they can not be used to create new objects.

This section serves as an example of Converge’s syntax extension system, and fleshes out the method of section 5.1.

6.1 Example of use

The typed modelling language DSL is housed within the package `TM`; the DSL implementation function `model_class` is contained within the `TM` module within the package. The following fragment uses the DSL to express a model of a simplified UML-esque modelling language as shown in figure 3:

```
import TM.TM

$<TM.model_class>:
  abstract class Classifier {
    name : String;
  }

  class PrimitiveDataType extends Classifier { }

  class Class extends Classifier {
    parents : Seq(Class);
    attrs : Seq(Attribute);

    inv unique_names:
      attrs->forAll(a1 a2 |
        a1 != a2 implies a1.name != a2.name)
  }
```

```
class Attribute {
  name : String;
  type : Classifier;
  is_primary : bool;
}
```

Note that although this particular examples shows a model of a modelling language, the DSL is capable of expressing any type of model.

The `TM.model_class` DSL implementation function translates each class in the model into a function in Converge which creates model objects. As a useful convenience, each constructor function takes arguments which correspond to the order attributes are specified in the model class. If a model class has parents, their attributes come first, and so on recursively. Model objects can have their slots accessed by name. Note that since the modelling language is typed, setting attributes either via the constructor function or through assigning to a slot forces the value to be of the correct type. Types can be of any model class (the DSL allows forward references), `int`, `String`, `bool` (where true and false are represented by 1 and 0 respectively), or sequences or sets of the preceding types. Note that sequences and sets can be nested arbitrarily. Model classes can contain invariants which are written in OCL; invariants are checked after an object has been initialized with values, and on every subsequent slot update. Currently only a subset of OCL 1.x is implemented, but the subset covers several different areas of OCL; implementing full OCL 1.x would be a relatively simple extension.

Assuming the above is held in a file `Simple_UML.cv`, one can then use the use the Simple UML modelling language to create models. The following example creates model classes `Dog` and `Person`, with `Dog` having an attribute `owner` of type `Person`:

```
person := Simple_UML.Class("Person")
dog := Simple_UML.Class("Dog")
dog.attrs.append(Simple_UML.Attribute("owner", person, 0))
```

One can arbitrarily manipulate models in a natural fashion:

```
dog.name := "Doggy"
```

Attempting to update a model in a way that would conflict with its type information results in an exception being raised. For example, attempting to assign an integer to the `Dog` model class' name raises the following exception:

```
Traceback (most recent call last):
  File "Ex1.cv", line 42, column 4, in main
  File "TM/TM.cv", line 162, column 5, in set_slot
Exception: Instance of 'Class' expected object of type 'String' for
slot 'name'.
```

In similar fashion, if one violates the `unique_names` constraint by adding two attributes called `owner` to the `Dog` model class, the following exception is raised:

```
Traceback (most recent call last):
  File "Ex1.cv", line 45, column 17, in main
  File "TM/TM.cv", line 327, column 31, in append
  File "TM/TM.cv", line 407, column 3, in _class_class_check_invs
Exception: Invariant 'unique_names' violated.
```

As can be seen, the result of using the `TM.model_class` DSL is a natural embedding of an arbitrary modelling language within Converge. Furthermore by recording, and enforcing, type information using the modelling language DSL provides guarantees about models that would not have been the case if they had been implemented as normal Converge class's. The typed modelling language DSL neatly sidesteps the data representation problems found in typical GPL's.

In the following sections I outline how this DSL is implemented.

6.2 Data model

The `TM` package provides its own ObjVLisp style data model which is similar to, but distinct from, the Converge data model of section 2.4. The `TM` package needs to provide a new data model since the default Converge data model is inherently untyped; whilst figure 1 showed the core data model with types, such type information is purely for the benefit of the reader. In contrast, the `TM` data model is inherently typed, and the type information is used to enforce the correctness of models. The only exception to this is that functions are currently untyped; it would be relatively simple to extend the implementation to record and enforce functions' type information.

Figure 4 shows the `TM` data model. As in Converge, a bootstrapping phase is needed to set up the meta-circular data model. `MObject` and `MClass` are so named to avoid clashing with the builtin classes `Object` and `Class`. Similarly, method and attribute names which might conflict, or be confused with, those found in normal Converge classes are named differently. For example `init` becomes `initialize`, `to_str` becomes `to_string` and `instance_of` becomes `of`. For brevity, and for easy interaction with external code, the `TM` package does not directly replicate all builtin Converge types such as strings; builtin Converge types are treated internally as instances of `MObject`.

Once cosmetic differences between the two are ignored, some important differences in the `TM` and Converge data models become apparent. Most importantly the `TM` data model has the standard statically typed OO languages notions of separate methods and attributes. `TM` mclasses are also different in that they can be abstract (i.e. can not be directly instantiated) and have at most one super class. `MObject` classes possess a `mod_id` slot which is a unique identifier, and which is typed as `String` to allow flexibility over the format of identifiers. The `mod_id` slot plays an important rôle in change propagating transformations.

As all this might suggest, the `TM` data model is intended to closely match the data model found within statically typed OO languages such as Java than the. Since methods and attributes are housed separately within classes, model instances require only a slot per attribute; invoking a method on an object searches the objects `of` class (and its superclasses) for an appropriate method. This is achieved by making use of the Converge MOP (see section 2.7). Although the actual implementation is relatively complex, a simplified version demonstrates the salient points. All model objects are instances of the Converge class `_Raw_Object` which is initialized with a blank slot per attribute

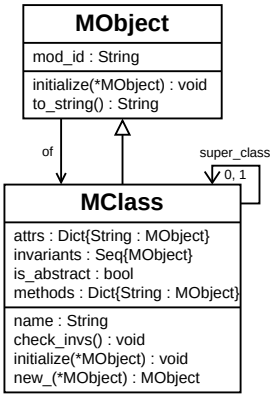

Figure 4: `TM` data model.

of a model class. The Converge MOP is overridden via a custom `get_slot` function in the `_Raw_Object` class. If a slot name matches an attribute slot, that value is returned. Otherwise the model objects `of` class and, if necessary its superclasses, are searched for a method of the appropriate name. Finally, if a method is not found then if the slot name matches that of a normal Converge slot in the `_Raw_Object` instance, the value is returned; otherwise an exception is raised. The following, much simplified, version of the code shows the skeleton of the `_Raw_Object` class and part of its MOP:

```
1  class _Raw_Object:
2    func init(attr_names):
3      self._attr_slots := Dict{}
4      for attr_name := attr_names.iterate():
5        self._attr_slots[attr_name] := null
6
7    func get_slot(name):
8      if self._attr_slots.contains(name):
9        return self._attr_slots[name]
10     else:
11       class_ := self._attr_slots["of"]
12       while 1:
13         if class_.methods.contains(name):
14           return Func_Binding(self, class_.methods[name])
15         if (class_ = class_.super_class) == null:
16           break
17       if exbi Object.has_slot(name):
18         return exbi Object.get_slot(name)
19       else:
20         raise Exceptions.Slot_Exception(Strings.format( \
21           "No such model / Converge slot '%s'", name))
```

A few notes are in order. Firstly the `class_` variable on line 11 is so named since `class` is a reserved keyword in Converge; by convention variable names are suffixed by '`_`' if they would otherwise clash with a reserved word. Note that definitions prefixed by '`_`' are conventionally considered to be private to the module or class they are contained within. On line 14, the `Func_Binding` class creates a binding which, when invoked, will call the Converge function `class_.methods[name]` with its `self` variable bound to `self` (i.e. the `_Raw_Object` instance; see section 2.2 for more details about functions and function bindings). The ability to create function

bindings in this fashion is an important feature of Converge, allowing a large deal of control over the behaviour of objects.

The `TM` packages data model can be considered to be a suitable template for suggesting how more powerful typed modelling languages – perhaps including packages and package inheritance [ACE+02], or allowing classes to inherit from more than one superclass) – might be naturally represented in a Converge DSL.

6.3 Pre-parsing and grammar

Before the `TM.model_class` DSL can parse its input, it first iterates through the input tokens searching for tokens which have type `ID` and value any of `abstract`, `and`, `at`, `collect`, `extends`, `forAll`, `implies`, `inv`, `Seq`, `Set`. Such tokens are replaced by a keyword token, whose type is the `ID`'s value. Furthermore since the `TM.model_class` DSL is intended to emulate typed languages such as C and Java, it implements a white space insensitive grammar; thus all `INDENT`, `DEDENT`, and `NEWLINE` tokens are removed from the input. The modified token list is then parsed according to the following grammar:

```
top_level      ::= { class }*

class          ::= class_abstract "CLASS" "ID" class_super "{"
                   { class_field }* "}"
class_abstract ::= "ABSTRACT"
               ::=
class_super    ::= "EXTENDS" "ID"
               ::=

class_field    ::= field_type
               ::= invariant

field_type     ::= "ID" ":" type ";"

type           ::= "ID"
               ::= "SEQ" "(" type ")"
               ::= "SET" "(" type ")"

invariant      ::= "INV" "ID" ":" expr

expr           ::= int
               ::= string
               ::= slot_lookup          %precedence 20
               ::= application          %precedence 15
               ::= binary               %precedence 10
               ::= seq
               ::= set
               ::= "ID"

int            ::= "INT"

string         ::= "STRING"

slot_lookup    ::= expr "." "ID"
               ::= expr "-" ">" forall
               ::= expr "-" ">" at
               ::= expr "-" ">" collect
forall         ::= "FORALL" "(" "ID" "|" expr ")"
               ::= "FORALL" "(" "ID" "ID" "|" expr ")"
at             ::= "AT" "(" expr ")"
collect        ::= "COLLECT" "(" "ID" "ID" "=" expr "|" expr ")"

application    ::= expr "(" expr { "," expr }* ")"
               ::= expr "(" ")"
```

```
binary          ::= expr "+" expr        %precedence 30
                ::= expr "-" expr        %precedence 30
                ::= expr ">" expr        %precedence 20
                ::= expr "<" expr        %precedence 20
                ::= expr "==" expr       %precedence 20
                ::= expr "!=" expr       %precedence 20
                ::= expr "IMPLIES" expr  %precedence 10
                ::= expr "AND" expr      %precedence 10

seq             ::= "SEQ" "{" expr ".." expr "}"
                ::= "SEQ" "{" expr { "," expr }* "}"
                ::= "SEQ" "{" "}"

set             ::= "SET{" expr { "," expr }* "}"
                ::= "SET{" "}"
```

Most of this grammar is straightforward, although it is worth noting a few peculiarities that result from the fact that tokenization is performed by the Converge tokenizer. For example, '`Set{`' is a single token (since `Set{...}` builds up a set in normal Converge). The equivalent notation for sets is represented by two tokens: '`Seq`' (a new keyword introduced by the DSL) followed by '{'. Fortunately in practise, such idiosyncrasies are largely hidden from, and irrelevant to, the DSL's users.

6.4 Traversing the parse tree

The main part of the DSL implementation function is concerned with traversing the parse tree, and translating it into an appropriate ITree. At a high-level, the translation is fairly simple: each model class is converted into an object capable of creating model instances. The CPK provides a simple traversal class (essentially a Converge equivalent of that found in the SPARK parser [AH02]) which provides the basis for most such translations. Users need only subclass the `Traverser` class and create a function prefixed by `_t_`*`name`* for each rule in the grammar. The `Traverser` class provides a `preorder` function will traverse an input parse tree in preorder fashion, calling the appropriate `_t_`*`name`* function for each node encountered in the tree. Note that each `_t_`*`name`* function can choose whether to invoke the `preorder` rule on sub-nodes, or whether it is capable of processing the sub-nodes itself.

For example, the `TM.model_class` function defines a traversal class `Model_Class_Creator` which translates the DSL's parse tree. An idealized version of the beginning of this class looks as follows:

```
import CPK.Traverser

class Model_Class_Creator(Traverser.Traverser):
  func translate():
    return self.preorder()

  func _t_top_level(node):
    // top_level ::= { class }*
    classes := []
    for class_node := node[1 : ].iterate()
      classes.extend(self.preorder(class_node))
    return classes

  func _t_class(node):
    // class ::= class_abstract "CLASS" "ID" class_super "{"  {
    //           class_field }* "}"
    ...
    return [|
```

```
      class $<<CEI.name(node[3].value)>>:
        ...
    |]
```

As this example shows, by automatically creating a parse tree and presenting a simple but powerful mechanism for traversing that parse tree, the CPK imposes a low burden on its users.

6.5 Translating

The actual translation of the parse tree to a ITree involves much repetition, and contains implementation details which are irrelevant to this paper. The first point to note about the translation is that the resulting ITree largely follows the structure of the parse tree. Having the translation follow the structure of the parse tree is desirable because it significantly lowers the conceptual burden involved in creating and comprehending the translation.

In this subsection I highlight some interesting aspects of the translation; interested readers can use this as a step to exploring the full translation in the `TM` package.

6.5.1 OCL expressions

Translating the OCL subset into Converge is a simple place to start in the translation because it is mostly simple and repetitive. For example, converting binary expressions from OCL into Converge is mostly a direct translation as the elided `_t_binary` traversal function shows:

```
func _t_binary(node):
  // binary ::= expr "+" expr
  //        ::= expr "<" expr
  //        ::= expr "==" expr
  lhs := self.preorder(node[1])
  rhs := self.preorder(node[3])
  if node[2].type == "+":
    return [| $<<lhs>> + $<<rhs>> |]
  elif node[2].type == ">":
    return [| $<<lhs>> > $<<rhs>> |]
  elif node[2].type == "==":
    return [|
      func ocl_equals() {
        lhs := $<<lhs>>
        if lhs.conforms_to(Int) | lhs.conforms_to(String):
          return lhs == $<<rhs>>
        else:
          return lhs is $<<rhs>>
      }()
    |]
```

Note that there is a slight complexity in translating the `==` operator, since OCL defines equality between objects to be based on their value if they are a primitive type, and on their identity if they are a model element. Whilst this is simple to encode as a sequence of expressions, it slightly complicates the `_t_class` traversal function, which is expected to a return only a single quasi-quoted expression. In order to work around this limitation we combine the necessary sequence of instructions into a function; the quasi-quotes returns the invocation of this function which is thus a single expression. This idiom occurs frequently in the translation.

6.5.2 Forward references

Forward references between model classes might appear to slightly muddy the structure of the translation. Consider the following example:

```
$<TM.model_class>:
  class Dog {
    owner : Person;
  }

  class Person {
    name : String;
  }
```

Assuming a simple translation, the result would be similar to the following converge code:

```
class Dog:
  attributes := Dict{"owner" : Person}
  name := "Dog"

class Person:
  attributes := Dict{"name" : String}
  name := "Person"
```

Such code would compile correctly, but lead to an exception being raised at run-time since the `Person` class will not have assigned a value to the `Person` variable when it is accessed in the `Dog` class. A standard approach to this problem would be to make the `attributes` field a function; by placing the reference to `Person` a function, the variable access would be deferred until after the `Person` variable contained a value.

The `TM.model_class` DSL takes an alternative approach which is simplistic and, whilst not generally applicable, effective. Essentially the `TM` module keeps a record of all model classes encountered. When a new model class is created (i.e. when importing a model containing a `TM.model_class` block), it registers itself with the `TM` module. Rather than directly referring to model classes, type references are strings of the target model class name – when a model class needs to be retrieved, its name is looked up in the `TM` registry, and the appropriate object returned. Thus forward references are a non-issue, since references are only resolved when necessary.

The reason the `TM.model_class` DSL takes this approach is that all model classes live in the same namespace; when we come to transforming model elements it aids brevity that model classes do not need to be prefixed by a package or module name.

6.5.3 Model class translation

The suggestion up to this point has largely been that model classes have been directly translated to normal Converge classes. In fact, model classes are instances of the `MClass` class. Whilst this would suggest that we can use the `metaclass` keyword shown in section 2.4, this is not possible since `MClass` requires more information than a normal Converge class. Normal Converge classes take only three parameters `name, supers, fields` whereas a model class requires information about whether it is

abstract, its invariants and so on. It is thus necessary to create the class manually. Fortunately this is relatively simple largely because the `bound_func` keyword allows bound functions to be expressed outside a class.

A much elided, and slightly simplified, version of the `_t_class` traversal function is as follows:

```
is_abstract := bool
class_name := String
super_ := String or null
attrs := Dict{name : type}
invariants := List of tuples [name, function]
operations := List of tuples [name, function]

init_func_var := CEI.ivar(CEI.fresh_name())

return [d|
  bound_func initialize(*args):
    super_attrs := _all_attrs($<<super_>>, 1)
    if args.len() > (super_attrs.len() + \
      $<<CEI.lift(attrs.len())>>):
      raise Exceptions.Parameters_Exception("Too many args")
    super_args_pos := Maths.min(super_attrs.len(), args.len())
    Func_Binding(self, $<<super_>>.methods["initialize"]). \
      apply(args[ : super_args_pos])
    $<<init_func_body>>


  $<<CEI.ivar(CEI.name(class_name))>> := MClass( \
    $<<is_abstract>>, $<<CEI.lift(class_name)>>, \
    $<<super_>>, $<<CEI.lift(attrs)>>, \
    $<<CEI.idict(operations + [[CEI.lift("initialize"), \
    init_func_var]])>>, $<<CEI.ilist(invariants)>>)
|]
```

Essentially the `_t_class` traversal function first evaluates and transforms the details of the model class, placing information such as whether the class is abstract into appropriately named variables. Finally it returns quasi-quoted code which contains two things: a function to initialize model class instances, and finally the instantiation of `MClass` itself. The arguments that `MClass` itself requires are hopefully obvious due to the names of the variables passed to it in this example.

6.5.4 Summary of translation

Whilst this section has tersely presented the translation of a `TM.model_class` block, I hope that it shows enough detail to suggest that the bulk of the translation is simple work, with only one or two areas requiring the use of more esoteric Converge features. Section B shows the pretty printed ITree resulting from the translation of the example in section 6.1.

6.6 Diagrammatic visualization

A useful additional feature of the `TM` package is its ability to visualize modelling languages and model languages as diagrams. Visualization makes use of the fact

that the `TM` data model is fully reflective, making the traversal and querying of objects trivial. The `Visualizer` module defines several visualization functions, all of which use the GraphViz package [GN00] to create diagrams. For example, the `visualize_modelling_language` function takes a list of model classes, and visualizes them as a standard class diagram. Figure 5 shows the automatic visualization of the Simple UML modelling language originally shown in figure 3. Note that the modelling language visualization function explicitly shows that all model classes are subclasses of `MObject`.

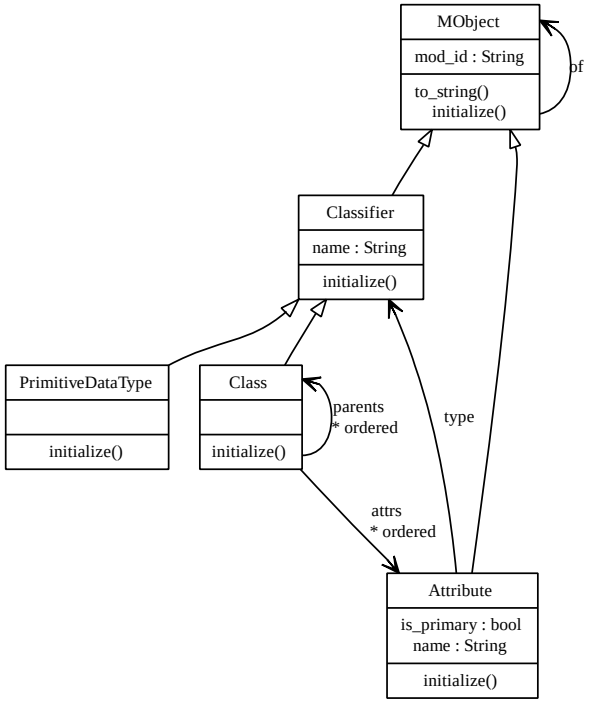

Figure 5: 'Simple UML' modelling language visualized.

The Visualizer is also able to visualize models as UML object diagrams. Figure 6 shows the visualization of the following module:

```
person := Simple_UML.Class("Person")
dog := Simple_UML.Class("Dog")
dog.attrs.append(Simple_UML.Attribute("owner", person, 0))
```

7 Future work

Because the core of Converge is a mix of established languages, the core language is largely stable. The implementation of the compile-time meta-programming facilities is currently less than satisfactory in one or two areas, but is eminently usable. One feature in particular that appears to be confuse new users to the language relates to the very different effects of the splice annotation. The syntax, inherited from Template

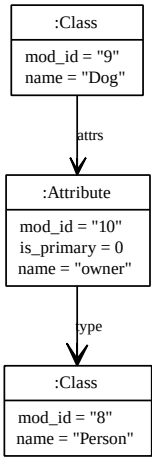

Figure 6: A ‘Simple UML’ model language visualized.

Haskell, means that $ behaves very differently when inside (a simple replacement of the splice annotation) and outside (cause compile-time evaluation) quasi-quotes. A simple change of syntax may suffice to solve this problem, or it may be considered to be an inevitable part of the learning curve that compile-time meta-programming presents.

Although Converge’s error reporting facilities are at least as good as any comparable language, there is still room for considerable improvement from the users point of view. Although section 3.13 used nested quasi-quotes to customise error reports, it may be necessary to find a lighter weight technique if DSL authors are to be encouraged to provide high quality error reporting.

The syntax extension feature presents the greatest opportunity for future work. The most obvious improvement would be to allow the user to provide their own tokenization facility. This may result in simply passing a single string to the DSL implementation function, or it may involve a more sophisticated interaction between the Converge tokenizer and parser and the DSL tokenizer and parser. For example parsing algorithms such as Pack Rat parsing [For02] allow the conflation of tokenizing and parsing in a way that might lend themselves to syntax extension.

8 Acknowledgments

This work has been funded by a grant from Tata Consultancy Services. My thanks to Kelly Androutsopoulos for insightful comments on an early draft of part of this paper.

A Converge grammar

This section lists the CPK grammar for Converge. This is extracted directly from the Converge compiler file `Compiler/CV_Parser.cv`:

```
top_level ::= definition { "NEWLINE" definition }*
          ::=

definition  ::= class_def
            ::= func_def
            ::= import
            ::= var { "," var }* ":=" expr
            ::= splice

import      ::= "IMPORT" dotted_name import_as { "," dotted_name
                import_as }*
dotted_name ::= "ID" { "." "ID" }*
import_as   ::= "AS" "ID"
            ::=

class_def       ::= "CLASS" class_name class_supers class_metaclass ":"
                    "INDENT" class_fields "DEDENT"
class_name      ::= "ID"
                ::= splice
class_supers    ::= "(" expr { "," expr }* ")"
                ::=
class_metaclass ::= "METACLASS" expr
                ::=
class_fields    ::= class_field { "NEWLINE" class_field }*
class_field     ::= class_def
                ::= func_def
                ::= var ":=" expr
                ::= splice
                ::= "PASS"

func_def           ::= func_type func_name "(" func_params ")" ":"
                       "INDENT" func_nonlocals expr_body "DEDENT"
                   ::= func_type func_name "(" func_params ")" "{"
                       "INDENT" func_nonlocals expr_body "DEDENT"
                       "NEWLINE" "}"
func_type          ::= "FUNC"
                   ::= "BOUND_FUNC"
                   ::= "UNBOUND_FUNC"
func_name          ::= "ID"
                   ::= "+"
                   ::= "-"
                   ::= "/"
                   ::= "*"
                   ::= "<"
                   ::= ">"
                   ::= "=="
                   ::= "!="
                   ::= ">="
                   ::= "<="
                   ::= splice
                   ::=
func_params        ::= func_params_elems "," func_varargs
                   ::= func_params_elems
                   ::= func_varargs
                   ::=
func_params_elems  ::= var func_param_default { "," var
                       func_param_default }*
                   ::= splice
func_param_default ::= ":=" expr
                   ::=
func_varargs       ::= "*" var
                   ::= splice
func_nonlocals     ::= "NONLOCAL" "ID" { "," "ID" }* "NEWLINE"
                   ::=

expr_body ::= expr { "NEWLINE" expr }*
```

```
expr ::= class_def
     ::= func_def
     ::= while
     ::= if
     ::= for
     ::= try
     ::= number
     ::= var
     ::= dict
     ::= set
     ::= list
     ::= dict
     ::= string
     ::= slot_lookup     %precedence 50
     ::= list
     ::= application     %precedence 40
     ::= lookup          %precedence 40
     ::= slice           %precedence 40
     ::= exbi
     ::= return
     ::= yield
     ::= raise
     ::= assert
     ::= break
     ::= continue
     ::= conjunction     %precedence 10
     ::= alternation     %precedence 10
     ::= assignment      %precedence 15
     ::= not             %precedence 17
     ::= neg             %precedence 35
     ::= binary          %precedence 30
     ::= comparison      %precedence 20
     ::= pass
     ::= import
     ::= splice          %precedence 100
     ::= quasi_quotes
     ::= brackets

if      ::= "IF" expr ":" "INDENT" expr_body "DEDENT" { if_elif }*
            if_else
        ::= "IF" expr "{" "INDENT" expr_body "DEDENT" "NEWLINE" "}"
            { if_elif }* if_else
if_elif ::= "NEWLINE" "ELIF" expr ":" "INDENT" expr_body "DEDENT"
        ::= "NEWLINE" "ELIF" expr "{" "INDENT" expr_body "DEDENT"
            "NEWLINE" "}"
if_else ::= "NEWLINE" "ELSE" ":" "INDENT" expr_body "DEDENT"
        ::= "NEWLINE" "ELSE" "{" "INDENT" expr_body "DEDENT" "NEWLINE"
            "}"
        ::=

while ::= "WHILE" expr ":" "INDENT" expr_body "DEDENT" exhausted broken
      ::= "WHILE" expr

for ::= "FOR" expr ":" "INDENT" expr_body "DEDENT" exhausted broken
    ::= "FOR" expr

try           ::= "TRY" ":" "INDENT" expr_body "DEDENT" { try_catch }*
                  try_else
try_catch     ::= "NEWLINE" "CATCH" expr try_catch_var ":" "INDENT"
                   expr_body "DEDENT"
try_catch_var ::= "INTO" var
              ::=
try_else      ::= "NEWLINE" "ELSE" ":" "INDENT" expr_body "DEDENT"
              ::=

exhausted ::= "NEWLINE" "EXHAUSTED" ":" "INDENT" expr_body "DEDENT"
          ::=

broken ::= "NEWLINE" "BROKEN" ":" "INDENT" expr_body "DEDENT"
       ::=

number ::= "INT"

var ::= "ID"
```

```
    ::= "&" "ID"
    ::= splice

string ::= "STRING"

slot_lookup ::= expr "." "ID"
            ::= expr "." splice

list ::= "[" expr { "," expr }* "]"
     ::= "[" "]"

dict ::= "DICT{" expr ":" expr { "," expr ":" expr }* "}"
     ::= "DICT{" "}"

set ::= "SET{" expr { "," expr }* "}"
    ::= "SET{" "}"

application ::= expr "(" expr { "," expr }* ")"
            ::= expr "(" ")"

lookup ::= expr "[" expr "]"

slice ::= expr "[" expr ":" expr "]"
      ::= expr "[" ":" expr "]"
      ::= expr "[" expr ":" "]"
      ::= expr "[" ":" "]"

exbi ::= "EXBI" expr "." "ID"

return ::= "RETURN" expr
       ::= "RETURN"

yield ::= "YIELD" expr

raise ::= "RAISE" expr

assert ::= "ASSERT" expr

break ::= "BREAK"

continue ::= "CONTINUE"

conjunction ::= expr "&" expr { "&" expr }*

alternation ::= expr "|" expr { "|" expr }*

assignment        ::= assignment_target { "," assignment_target }*
                      assignment_type expr
assignment_target ::= var
                  ::= slot_lookup
                  ::= lookup
                  ::= slice
assignment_type   ::= ":="
                  ::= "*="
                  ::= "/="
                  ::= "+="
                  ::= "-="

not ::= "NOT" expr

neg ::= "-" expr

binary    ::= expr binary_op expr
binary_op ::= "*"       %precedence 40
          ::= "/"       %precedence 30
          ::= "%"       %precedence 30
          ::= "+"       %precedence 20
          ::= "-"       %precedence 20

comparison    ::= expr comparison_op expr
comparison_op ::= "IS"
              ::= "=="
              ::= "!="
              ::= "<="
```

```
              ::= ">="
              ::= "<"
              ::= ">"

pass ::= "PASS"

splice       ::= expr_splice
             ::= block_splice
expr_splice  ::= "$" "<" "<" expr ">" ">"
block_splice ::= "$" "<" expr ">" ":" "INDENT" "JUMBO" "DEDENT"

quasi_quotes      ::= expr_quasi_quotes
                  ::= defn_quasi_quotes
expr_quasi_quotes ::= "[|" "INDENT" expr { "NEWLINE" expr }* "DEDENT"
                      "NEWLINE" "|]"
                  ::= "[|" expr { "NEWLINE" expr }* "|]"
defn_quasi_quotes ::= "[D|" definition { "NEWLINE" definition }* "|]"
                  ::= "[D|" "INDENT" definition { "NEWLINE" definition }*
                      "DEDENT" "NEWLINE" "|]"

brackets ::= "(" expr ")"
```

B The ‘Simple UML’ modelling language translation

The following is the pretty printed ITree that results from translating the Simple UML modelling language shown in section 6.1:

```
Simple_UML.cvb Simple_UML.cv
$$1$$ := bound_func initialize_Classifier(*args){
  super_attrs := TM._all_attrs(TM.MObject, 1)
  if args.len() > super_attrs.len() + 1:
    raise TM.Exceptions.Parameters_Exception("Too many args")
  super_args_pos := TM.Maths.min(super_attrs.len(), args.len())
  TM.Func_Binding(self, TM.MObject.methods["initialize"]).apply(args[0 : \
    super_args_pos])
  if 0 < args.len() - super_args_pos:
    self.name := args[super_args_pos + 0]
}

Classifier := TM.MClass(1, "Classifier", TM.MObject, Dict{"name" : [3]}, \
  ["name"], Dict{"initialize" : $$1$$}, [])

$$2$$ := bound_func initialize_PrimitiveDataType(*args){
  super_attrs := TM._all_attrs(TM._CLASSES_REPOSITORY["Classifier"], 1)
  if args.len() > super_attrs.len() + 0:
    raise TM.Exceptions.Parameters_Exception("Too many args")
  super_args_pos := TM.Maths.min(super_attrs.len(), args.len())
  TM.Func_Binding(self, TM._CLASSES_REPOSITORY["Classifier"]. \
    methods["initialize"]).apply(args[0 : super_args_pos])
}

PrimitiveDataType := TM.MClass(0, "PrimitiveDataType", \
TM._CLASSES_REPOSITORY["Classifier"], Dict{}, [], Dict{"initialize" : $$2$$}, [])

$$3$$ := bound_func initialize_Class(*args){
  super_attrs := TM._all_attrs(TM._CLASSES_REPOSITORY["Classifier"], 1)
  if args.len() > super_attrs.len() + 2:
    raise TM.Exceptions.Parameters_Exception("Too many args")
  super_args_pos := TM.Maths.min(super_attrs.len(), args.len())
  TM.Func_Binding(self, TM._CLASSES_REPOSITORY["Classifier"]. \
    methods["initialize"]).apply(args[0 : super_args_pos])
  if 0 < args.len() - super_args_pos:
    self.parents := args[super_args_pos + 0]
  if 0 >= args.len() - super_args_pos:
    self.parents := TM.TM_List(self)
  if 1 < args.len() - super_args_pos:
    self.attrs := args[super_args_pos + 1]
  if 1 >= args.len() - super_args_pos:
    self.attrs := TM.TM_List(self)
}
```

```
Class := TM.MClass(0, "Class", TM._CLASSES_REPOSITORY["Classifier"], \
Dict{"parents" : [0, "Class"], "attrs" : [0, "Attribute"]}, ["parents", \
"attrs"], Dict{"initialize" : $$3$$}, [["unique_names", \
  unbound_func unique_names(self){
  return unbound_func ocl_for(){
    expr := self.attrs
    for a1 := expr.iterate():
      for a2 := expr.iterate():
        if not unbound_func ocl_implies(){
          if not unbound_func ocl_not_equals(){
            lhs := a1
            if lhs.conforms_to(TM.Int) | lhs.conforms_to(TM.String):
              return lhs != a2
            else:
              return not lhs is a2
          }():
            return 1
          if unbound_func ocl_not_equals(){
            lhs := a1.name
            if lhs.conforms_to(TM.Int) | lhs.conforms_to(TM.String):
              return lhs != a2.name
            else:
              return not lhs is a2.name
          }():
            return 1
          return TM.fail
        }():
          return TM.fail
    return 1
  }()
}]])

$$4$$ := bound_func initialize_Attribute(*args){
  super_attrs := TM._all_attrs(TM.MObject, 1)
  if args.len() > super_attrs.len() + 3:
    raise TM.Exceptions.Parameters_Exception("Too many args")
  super_args_pos := TM.Maths.min(super_attrs.len(), args.len())
  TM.Func_Binding(self, TM.MObject.methods["initialize"]).apply(args[0 : \
    super_args_pos])
  if 0 < args.len() - super_args_pos:
    self.name := args[super_args_pos + 0]
  if 1 < args.len() - super_args_pos:
    self.type := args[super_args_pos + 1]
  if 2 < args.len() - super_args_pos:
    self.is_primary := args[super_args_pos + 2]
}

Attribute := TM.MClass(0, "Attribute", TM.MObject, Dict{"name" : [3], \
"type" : "Classifier", "is_primary" : [5]}, ["name", "type", "is_primary"], \
Dict{"initialize" : $$4$$}, [])
```

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[ACE+02] Biju Appukuttan, Tony Clark, Andy Evans, Stuart Kent, Girish Maskeri, Paul Sammut, Laurence Tratt, and James S. Willans. Unambiguous uml submission to uml 2 infrastructure rfp, September 2002. OMG document `ad/2002-06-14`.

[AH02] John Aycock and R. Nigel Horspool. Practical earley parsing. *The Computer Journal*, 45(6):620–630, 2002.

[Baw99] Alan Bawden. Quasiquotation in lisp. Workshop on Partial Evaluation and Semantics-Based Program Manipulation, January 1999.

[Baw00] Alan Bawden. First-class macros have types. In *Proc. 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 133–141, January 2000.

[BC89] Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proc. OOPSLA '89*, October 1989.

[BP99] Jonathan Bachrach and Keith Playford. D-expressions: Lisp power, dylan style, 1999. `http://www.ai.mit.edu/people/jrb/Projects/dexprs.pdf` Accessed Sep 22 2004.

[BP01] Jonathan Bachrach and Keith Playford. The java syntactic extender (jse). In *Proc. OOPSLA*, pages 31–42, November 2001.

[BS00] Claus Brabrand and Michael Schwartzbach. Growing languages with metamorphic syntax macros. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN. ACM, 2000.

[BV04] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proc. OOPSLA'04*, Vancouver, Canada, October 2004. ACM SIGPLAN.

[CESW04] Tony Clark, Andy Evans, Paul Sammut, and James Willans. Applied metamodelling: A foundation for language driven development, September 2004. Available from `http://www.xactium.com/` Accessed Sep 22 2004.

[CJOT04] Krzysztof Czarnecki, Jörg Striegnitz John O'Donnell, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. 3016:50–71, 2004.

[CMA93] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible grammars for language specialization. In *Proc. Fourth International Workshop on Database Programming Languages - Object Models and Languages*, pages 11–31, August 1993.

[Coi87] Pierre Cointe. Metaclasses are first class: the ObjVLisp model. In *Object Oriented Programming Systems Languages and Applications*, pages 156–162, October 1987.

[DGR01] Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20(1):53–73, Nov 2001.

[DHB92] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. In *Lisp and Symbolic Computation*, volume 5, pages 295–326, December 1992.

[DM95] François-Nicola Demers and Jacques 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.

[dR03] Daniel de Rauglaudre. *Camlp4 - Reference Manual*, September 2003. `http://caml.inria.fr/camlp4/manual/` Accessed Sep 22 2004.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2), February 1970.

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 2002.

[For02] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *International Conference on Functional Programming*, pages 36–47, October 2002.

[GG93] Ralph E. Griswold and Madge T. Griswold. History of the Icon programming language. *j-SIGPLAN*, 28(3):53–68, March 1993.

[GG96a] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.

[GG96b] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Peer-to-Peer Communications, third edition, 1996.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Orientated Software*. Addison Wesley, 1994.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, January 1989.

[GST01] Steven E. Ganz, Amr Sabry, and Walid 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.

[Gud92] David A. Gudeman. Denotational semantics of a goal-directed language. *ACM Transactions on Programming Languages and System*, 14(1):107–125, January 1992.

[Jaf03] Aubrey Jaffer. *SCM Scheme Implementation*, November 2003. `http://www.swiss.ai.mit.edu/~jaffer/scm_toc` Accessed Sep 16 2004.

[Jef02] Clinton L. Jeffery. *Godiva Language Reference Manual*, November 2002. `http://www.cs.nmsu.edu/~jeffery/godiva/godiva.html`.

[JMPP03] Clinton Jeffery, Shamim Mohamed, Ray Pereda, and Robert Parlett. *Programming with Unicon*, April 2003. `http://unicon.sourceforge.net/book/ub.pdf`.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised(5) report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Symposium on Lisp and Functional Programming*, pages 151–161. ACM, 1986.

[Pro97] Todd A. Proebsting. Simple translation of goal-directed evaluation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–6, 1997.

[Que96] Christian Queinnec. Macroexpansion reflective tower. In *Proc. Reflection'96*, pages 93–104, April 1996.

[Sch] Friedrich Wilhelm Schröer. *The ACCENT Grammar Language*. `http://accent.compilertools.net/language.html` Accessed Jan 25 2005.

[SCK03] Sean Seefried, Manuel M. T. Chakravarty, and Gabriele Keller. Optimising embedded DSLs using Template Haskell. In *Draft Proc. Implementation of Functional Languages*, 2003.

[SeABP99] Tim Sheard, Zine el Abidine Benaissa, and Emir 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.

[SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the Haskell workshop 2002*. ACM, 2002.

[SMO04] Kamil Skalski, Michal Moskal, and Pawel Olszta. Meta-programming in Nemerle, 2004. `http://nemerle.org/metaprogramming.pdf` Accessed Oct 1 2004.

[Ste99] Guy L. Steele, Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221 – 236, October 1999.

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, October 1999.

[TH00] David Thomas and Andrew Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.

[Vel95] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[vR03] Guido van Rossum. Python 2.3 reference manual, 2003. `http://www.python.org/doc/2.3/ref/ref.html` Accessed Sep 23 2004.

[WC93] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proc. SIGPLAN*, pages 156–165, 1993.