Domain Specific Language Implementation via Compile-Time Meta-Programming

LAURENCE TRATT
Bournemouth University, Poole, Dorset, BH12 5BB, United Kingdom.

Domain specific languages (DSLs) are mini-languages which are increasingly seen as being a valuable tool for software developers and non-developers alike. DSLs must currently be created in an ad-hoc fashion, often leading to high development costs and implementations of variable quality. In this paper I show how expressive DSLs can be hygienically embedded in the Converge programming language using its compile-time meta-programming facility, the concept of DSL blocks, and specialised error reporting techniques. By making use of pre-existing facilities, and following a simple methodology, DSL implementation costs can be significantly reduced whilst leading to higher quality DSL implementations.

Categories and Subject Descriptors: D.3.4 [**Software Engineering**]: Processors—*Translator writing systems and compiler generators*

General Terms: Languages

Additional Key Words and Phrases: Syntax extension, compile-time meta-programming, domain specific languages

1. INTRODUCTION

When developing complex software systems in a General Purpose Language (GPL), it is often the case that one comes to a problem which is not naturally expressible in the chosen GPL. In such cases the user has little choice but to find a suitable workaround, and encode their solution in as practical a fashion as they are able. Whilst such workarounds and encodings are often trivial, they can on occasion be exceedingly complex. In such cases the system can become far less comprehensible than the user may have wished. Although Steele argues that 'a main goal in designing a language should be to plan for growth' [Steele 1999], most modern GPLs only allow growth through the addition of libraries. The ability of a user to extend, or augment, their chosen programming language is thus severely restricted.

Domain Specific Languages (DSLs) are an attempt to work around the lack of expressivity in a GPL by presenting the user with a mini-language targeted to the particular domain they are working in. [Mernik et al. 2003] define DSLs as 'languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general purpose programming languages in their domain of application'. [Hudak 1998] describes the typical costs

of a DSL, noting that a small extra initial investment in a DSL implementation typically leads to long term savings compared with alternative routes. Exactly what identifies a particular language as being a 'DSL' is inherently subjective: for the purposes of this paper, it can be intuitively defined as a language smaller, and less generic, than a typical programming language such as Java, C++, or Python.

Traditionally DSLs – for example the UNIX `make` program or the `yacc` parsing system – have been implemented as entirely stand alone systems. Although such systems have many similarities in their implementations, each one tends to be engineered from scratch; this leads to increased work for the DSL language implementer, which inevitably results in implementations of variable quality. It is therefore a fundamental tenet of this paper that implementing DSLs as stand-alone systems is undesirable. There is another reason to shy away from implementing DSLs as stand alone systems. DSLs tend to start out as small, declarative languages [van Deursen et al. 2000], but most tend to acquire new features as they are used in practise; such features tend to be directly borrowed from GPLs [Hudak 1998]. So while DSL implementations tend over time to resemble programming language implementations, they frequently lack the quality one might expect in such a system due to the unplanned nature of this evolution.

In contrast to the traditional technique of implementing DSLs as stand alone systems, DSLs can be implemented by embedding the DSL into a *host language*. The advantage of this approach is that the DSL can inherit many of the features and benefits of the host language (including, one hopes, a robust implementation) with relatively little effort. The capabilities of the host language and the particular embedding mechanism dictate the class of DSLs that can be expressed in a particular combination; embedded DSLs range from those designed to express GUIs to constraint solving systems. In this paper I define and distinguish between *homogeneous* and *heterogeneous* embedding. Informally, heterogeneous embedding is when a system external to that used to compile[1] the host language is used to define the embedding, whereas homogeneous embedding is when the system used to compile the host language is also used to define the embedding. As shown in section 2, heterogeneous embedding systems are able to define a wider variety of DSLs than homogeneous embedding systems; conversely, by restricting the DSLs they can express, homogeneous systems can often define embeddings in a more concise and safe fashion.

This paper presents a practical, self-contained approach to DSL implementation in an homogeneous embedding environment. I show how an extension of the Converge language presents a coherent approach to DSL embedding that is aimed at facilitating rapid development and prototyping of DSLs. The novelty of this approach is in both its combination of features found in other languages, and the new features specific to Converge. Building on its compile-time meta-programming features, the main feature to allow DSL embeddings is the *DSL block*, which allows arbitrary syntaxes to be embedded in the language. I show how how DSLs can reuse the expression language from the main Converge programming language, how both run-time and compile-time error reports can be expressed in terms of

[1]This could of course just as easily be an interpreter; however in the interests of brevity I use the term 'compiler' throughout this paper.

the user's DSL input using the *src info* concept, and how DSL embeddings can be made hygienic (analogous to LISP macros [Kohlbecker et al. 1986]). The approach presented in this paper shows that homogeneous embedding approaches can express significantly more powerful DSLs than has previously been the case, and that these embeddings can be guaranteed to be safe.

This paper is structured as follows. First I detail existing approaches to DSL implementation via embedding, categorizing approaches as being either heterogeneous or homogeneous. I then outline the basics of the Converge programming language, including its compile-time meta-programming facility. I then introduce the features and techniques relating to DSL embedding in Converge, exploring them in relation to an evolving example of a model transformation language (the full version of the example can be found in [Tratt 2005c]). I conclude by separating out the parts of Converge fundamental to homogeneous DSL embedding, and explain how such features could be integrated into similar systems.

2. DSL IMPLEMENTATION VIA EMBEDDING

In the closely related area of meta-programming, [Sheard 2003] distinguishes between homogeneous and heterogeneous meta-programming systems. Similarly I choose to distinguish between homogeneous and heterogeneous embedding systems. [Sheard 2003] defines 'homogeneous systems [as those] where the meta-language and the object language are the same, and heterogeneous systems [as those] where the meta-language is different from the object-language.' In the context of DSL embedding it is important to weaken the final clause of this definition. Basing the definition on the languages involved is an objective choice, but not ideal as it focuses on implementation details rather than the way in which users perceive those systems in use.

I therefore update and alter the definition in [Sheard 2003] so that heterogeneous embedding is when the *system* used to compile the host language, and the *system* used to implement the embedding are different[2]. Note that this does not imply that the host language must be different than the language used to implement the embedding: it is possible to identify a heterogeneous embedding approach as one whose two separate systems just so happen to be written in the same language. Put differently, a homogeneous system is one where all the components are specifically designed to work with each other, whereas in heterogeneous systems at least one of the components is largely, or completely, ignorant of the existence of the other parts of the system.

An important reason for differentiating carefully between homogeneous and heterogeneous embedding approaches is that it allows readers, particularly those less familiar with the subject area, to quickly understand whether a given approach aims to be general or limited in its approach. There is no notion of one style being 'better' or 'worse' than the other; the normal trade-offs of generality and complexity versus restrictions and simplicity apply, and different approaches may be appropriate in different circumstances. The approaches detailed in this section cover a large spread of the possible spectrum.

[2]I also believe that an equivalent clarification might be usefully applied to meta-programming.

In the rest of the paper I use the terms *parse tree* and *abstract syntax tree* (AST) to differentiate representations of a program that record structure but carry little semantic information (parse trees) from those that record structure and a significant amount of semantic information (abstract syntax trees). Typically a parse tree is the automatic output from a parser, whereas an AST is manually created after one or more passes over a parse tree. For a given language every AST has an equivalent representation as a parse tree, but incorrect parse trees will have no equivalent representation as an AST.

2.1 Heterogeneous embedding approaches

In this section I describe three different types of heterogeneous embedding technologies: first generic hard-coded implementations, and then two specific technologies TXL and MetaBorg.

2.1.1 *Hard-coded implementations.* The majority of heterogeneous embedding systems in current existence are hard-coded to translate a specific DSL within a specific host language. For example, the standard implementation of the Icon programming language [Griswold and Griswold 1996] defines an embedded DSL called RTL[3], which allows those implementing Icon VM code in C to encode its goal-directed evaluation semantics in a natural fashion [Walker 1994]. A translator called RTT takes in files containing C and the embedded RTL DSL and converts them into pure C files. The system within the Icon implementation is indicative of the majority of heterogeneous embedding systems in that a bespoke translator is required to translate the embedded DSL. Such systems are only of marginal interest in the context of this paper, since my aim is to define a mechanism for defining arbitrary DSLs.

2.1.2 *TXL.* TXL [Cordy 2004] is a generic source to source transformation language. Although originally intended for transforming instances of the programming language Turing, it has evolved into a language capable of transforming instances of arbitrary language grammars. In so doing, TXL has morphed into a hybrid rule-based / functional programming language. ASF+SDF is a similar approach, although its implementation (which can see transformation systems compiled into machine code via C) is more like a traditional programming language [van den Brand et al. 2002].

TXL's general execution mechanism is simple. A TXL transformation has a grammar for the language to be transformed, and a number of rewrite rules. A source file is parsed according to the grammar and the rewrite rules execute on the parse tree. Although TXL was originally designed for source-to-source transformations on the same language, it has well developed mechanisms for overriding and extending grammars. By creating a *union grammar* of two separate languages, TXL can transform between languages with different grammars. Through this mechanism one can easily realise external embedding of DSLs, with few limits on the DSL or host language involved.

TXL has many advantages, including its maturity and efficiency, and has proved

[3]Although the Icon documentation does not explicitly identify RTL as such, it is clearly an embedded DSL.

itself capable of succinctly expressing many useful transformations. Its pragmatic rule-based approach is amongst the most refined available; [Cordy 2004] records a case where several billion lines of code were transformed with TXL. As TXL operates only at the parse tree level, it is left entirely to the DSL embedding author to be aware of the semantics of the host language beyond the simple structure recorded in the parse tree. To take a concrete issue, for some host languages the problem of unintended variable capture (well known in the analogous area of macro expansion [Kohlbecker et al. 1986]) can only be solved by understanding and accurately implementing the host languages' scoping rule. At its most extreme, this could mean each translator resembling a substantial subset of the host languages compiler which would in general be a prohibitive cost.

TXL is highly interesting because it shows the fundamental trade-off of heterogeneous systems. TXL can read and transform any language with a context free grammar, and its transformation language is powerful and easy to use. However because it is, by default, almost entirely ignorant of what it is transforming – TXL understands trees and tokens, but not what the trees or tokens represent – it is difficult to create transformations which require deep reasoning about the program being transformed, or to create transformations which guarantee basic safety properties such as variable capture.

2.1.3 *MetaBorg.* MetaBorg is a method which uses a combination of tools to provide a heterogeneous embedding approach, allowing language grammars to be extended in an arbitrary fashion using a rule rewriting system. [Bravenboer and Visser 2004] provides a compelling example of MetaBorg's power by using it to embed a DSL called SWing User-interface Language (SWUL) which allows Swing GUIs to be expressed natively within Java code.

As an example of the sort of DSL that MetaBorg is intended to create, the following is an example of some embedded SWUL code:

```
int cols = ...;

JPanel panel = panel of border layout {
  hgap = 12 vgap = 12
  north = label "Please enter your message"
  center = scrollpane of textarea {
    rows = 20
    columns = cols
  }
};
```

After translation by MetaBorg the resulting pure Java code is as follows:

```
JPanel panel = new JPanel(new BorderLayout(12, 12));
panel.add(BorderLayout.NORTH, new JLabel("Please enter your message"));
panel.add(BorderLayout.CENTER, new JScrollPane(text, 20, cols));
```

As this example shows, MetaBorg DSLs can interact with their surrounding environment. Although MetaBorg by default operates on parse trees in the same way as TXL, it does comes with standard support for representing some of the semantics of languages such as Java. This allows transformation authors to write more sophisticated transformations, and make some extra guarantees about the safety

of their transformations[4]. Unlike TXL, MetaBorg is also able to maintain a strict separation between the syntax of the hybrid DSL / host language input, and the host language output. MetaBorg has a sound mechanism for safely mixing different grammars, and can compose different embeddings in various ways, which further differentiates it from TXL.

Although MetaBorg is in theory capable of defining any embedding, it appears that the authors of the MetaBorg were aware that while it offers great power, that power is difficult to wield effectively. The MetaBorg authors deliberately narrow their vision for MetaBorg to a 'method for promoting APIs to the language level.' This is a sensible restriction since DSL that results from promoting a particular to the language level will tend to shadow that API; therefore instances of the DSL will generally translate fairly directly into API calls. It is unclear from MetaBorg examples whether it would be an appropriate platform in which to implement larger DSLs. Nevertheless MetaBorg is without doubt the most sophisticated heterogeneous approach yet created.

2.1.4 *Macro systems.* Macro systems which realise embedded DSLs through preprocessing are a valid form of heterogeneous embedding. The fact that such systems are often implemented in the same language as the host language does not effect the fact that the users of such systems are very aware of the difference between the two systems. Note that this shows the difference in the definition of homogeneous and heterogeneous embedding as defined in this paper relative to the original definition in [Sheard 2003], where systems such as the JSE [Bachrach and Playford 2001] would have been classified as homogeneous. In the context of this paper, systems such as the JSE are simply less capable (and therefore less interesting) cousins of homogeneous macro systems which are described in the following section.

2.1.5 *Summary of heterogeneous embedding systems.* Heterogeneous embedding systems can in theory realise any possible DSL embedding because they are not limited to any particular host language. However in practise such systems can be difficult to use precisely because of their inevitably limited knowledge of the host language. The more sophisticated a DSL is, the more complex its embedding becomes, which implies that a greater knowledge of the host language is required. Because of this, in practise such systems have rarely been used to embed sophisticated DSLs. Furthermore, the majority of heterogeneous embedding systems are hard-coded for a given host language and DSL, with only a few approaches designed to facilitate more generalised solutions.

2.2 Homogeneous embedding

2.2.1 *Macro systems.* The canonical, and arguably the original, example of homogeneous DSL embedding is via LISP macros. LISP's flexible syntax and powerful macro mechanism have been used to express countless DSLs. DSL embedding in LISP requires the presence of only one component: the LISP compiler. This component contains significant knowledge of the host language being embedded into.

[4]Although MetaBorg contains a `gensym` function, it is unclear whether this uses the semantic knowledge of the host language to generate guaranteed unique variable names in a given scope.

As a simple but compelling example, this means that a DSL embedding in LISP will be free of variable capture since this is a feature of all respectable LISP macro implementations.

In the case of DSL embedding in LISP, although its syntax is inherently flexible, it is not possible to change it in a completely arbitrary fashion – DSLs expressed via this mechanism are limited to what can be naturally expressed in LISP's syntax. Furthermore whilst this mechanism has been used to express many DSLs, its tight coupling to LISP's syntactic minimalism has largely prevented similar approaches being applied to other, more modern programming languages [Bachrach and Playford 1999]. Therefore despite LISP's success in this area, for many years more modern systems struggled to successfully integrate similar features [Tratt 2005a]. Dylan is one of the few such systems [Bachrach and Playford 1999], implementing a rewrite rule based macro system which is broadly equivalent to LISP's in power. However Dylan's syntax is not significantly more flexible than LISP's, and its macro related syntax is heavyweight, as it is a separate language from Dylan itself.

More recently languages such as Template Haskell [Sheard and Jones 2002] (which is effectively a refinement of the ideas in MetaML [Sheard 1998]; see [Tratt 2005a] for a more detailed comparison of these languages with Converge) have shown how sophisticated homogeneous meta-programming systems can be implemented in a modern language. However such languages suffer the same limitations as LISP in respect to DSL embedding, which is the inability to extend the syntax. I have more to say on Template Haskell, in particular, in section 4.

2.2.2 *Grammar extension through macros.* It is possible to use macro systems to create an homogeneous embedding approach similar in outlook to MetaBorg. In essence, macro calls inform the compiler of new grammar productions and provide a mechanism to compile these extensions into base code. Nemerle was the first example of such a language [Skalski et al. 2004], and is a statically typed OO language in the Java / C# vein. Normal Nemerle macros are very much in the LISP vein in that they are identified as such at compile-time and are not first-class objects i.e. they can not be called from other code. Nemerle macros can be specified with an optional `syntax` clause which inserts a new production into the Nemerle grammar. [Skalski et al. 2004] gives the following example of a macro which adds a C-style `for` loop to Nemerle:

```
macro for (init, cond, change, body)
syntax ("for", "(", init, ";", cond, ";", change, ")", body)
{ ... }
```

The Nemerle compiler defers to the `for` macro whenever a `for` loop is parsed in a source file. Nemerle's extensions are strictly limited to those which can be expressed as a single production extension to its grammar. Furthermore since Nemerle macros are not first-class citizens (similarly to LISP, but unlike Template Haskell), syntax extensions are effectively scoped over an entire file in a coarse-grained fashion.

Two subsequent approaches take a broadly similar tack to Nemerle, but allow richer extensions to be expressed. Metalua is a TH-derived extension to the Lua language, that allows new productions to be inserted into the Lua grammar [Fleutot and Tratt 2007]. xTc allows similar extensions for C [Grimm 2005]. xTc and Metalua are broadly similar in expressive power, although xTc has the burden

of coping with a larger, less regular base language. Both approaches allow much richer extensions than Nemerle, although neither has a correspondingly mature implementation. Metalua explicitly states that few guarantees can be made about the composition of extensions; it is unclear if xTc is able to make stronger guarantees than Metalua about composition.

2.2.3 *Function composition.* Homogeneous embedding should not be seen as being simply synonymous with macro systems. After describing the high costs of traditional DSL implementation, Hudak develops the notion of Domain Specific Embedded Languages (DSELs) [Hudak 1998]. Hudak specifically limits his vision of DSELs in two ways: he restricts the languages he considers suitable for embedding to strongly typed functional languages, particularly Haskell; he explicitly rules out any form of syntax extension. Hudak's DSELs therefore rely on the unusual feature set present in languages such as Haskell, such as the ability to compose functions in powerful ways, monads, and lazy evaluation. The advantage of this approach is that it allows an otherwise entirely ignorant language to be used to embed DSLs, and also allows DSLs to be relatively easily combined together. The disadvantage is that the embedding is indirect, and limited to what can be easily expressed using these pre-existing components.

2.3 Comparison of heterogeneous and homogeneous embedding systems

In theory there is a significant difference between heterogeneous and homogeneous embedding systems. Heterogeneous systems impose no limits on the DSLs they can express nor on the host languages they can embed into, whereas homogeneous embedding systems are inherently limited to expressing a certain class of DSLs within a specific host language. However it is notable that heterogeneous systems such as TXL and MetaBorg have only been used to implement relatively small DSLs. I believe that a major reason for this is the practical difficulty of defining a safe embedding of a DSL when the embedding system is largely, or completely, unaware of the semantics of the host language it is embedding into. Creating a system with suitable knowledge of the host languages' semantics is likely to involve encoding a significant subset of that languages compiler. In practice this is unrealistic, which is why heterogeneous embedding systems are generally limited to small DSLs (which even then may not have entirely safe embeddings). Homogeneous embedding systems on the other hand are limited to one compilation system and host language and can therefore take advantage of the knowledge of the host languages semantics embedded in the compiler. However existing homogeneous embedding systems are very limited and can embed only simple DSLs.

2.4 A lightweight approach to DSL embedding

Although the relative power of the various DSL embedding systems detailed thus far is hopefully clear, one important question remains unanswered: why have these systems seen relatively little real-world use? It is my belief that there are two complementary reasons. Firstly while heterogeneous embedding systems allow syntax extension, practical implementation concerns tend to limit them to small hard-coded DSLs for specific target languages; little of any such system can be reused to create another. Secondly, homogeneous embedding system's general inability to

permit syntax extension means that only a small subset of DSLs can be naturally embedded within them.

It is my assertion that generic heterogeneous embedding systems are fundamentally difficult systems to create. However, as argued in [Wilson 2005], syntax extension is the only way to realise powerful DSLs.

This paper thus presents a new approach to homogeneous embedding which allows DSLs of arbitrary syntaxes to be embedded. I do this by presenting a small extension to the Converge programming language [Tratt 2005a] which uses its compile-time meta-programming facility to be used to embed DSLs. Although this paper is largely expressed in terms of Converge, in section 10 I explain how these concepts could be adapted to other languages.

3. CONVERGE BASICS

This section gives a brief overview of basic Converge features that are relevant to the main subject of this paper. Whilst this is not a replacement for the language manual [Tratt 2007], it should allow readers familiar with a few other programming languages the opportunity to quickly come to grips with the most important areas of Converge, and to determine the areas where it differs from other languages.

Converge's most obvious ancestor is Python [van Rossum 2003] resulting in an indentation based syntax, a similar range and style of datatypes, and general sense of aesthetics. The most significant difference is that Converge is a slightly more static language: all namespaces (e.g. a modules' classes and functions, and all variable references) are determined statically at compile-time. Converge's scoping rules are different from 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. 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. Variable references search in order from the innermost block outwards, ultimately resulting in a compile-time error if a suitable reference is not found. 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). The overall justification for these rules is to ensure that, unlike similar languages such as Python, Converge's namespaces are entirely statically calculable.

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

```
import Sys

class Fib_Cache:
  func init():
```

```
    self.cache := [0, 1]

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

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

Another important, if less obvious, influence is Icon [Griswold and Griswold 1996]. As Icon, Converge is an expression-based language. Icon has a powerful notion of expression *success* and *failure*; for the purposes of this paper, these features are mostly irrelevant, and are explained only as needed.

Converge's OO features are reminiscent of Smalltalk's [Goldberg and Robson 1989] everything-is-an-object philosophy, but with a prototyping influence that was inspired by Abadi and Cardelli's theoretical work [Abadi and Cardelli 1996]. The internal object model is derived from ObjVLisp [**?**]. An object is said to be comprised of *slots*, which are name / value pairs typically corresponding to the functions and fields defined by the class which created the object.

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.

4. COMPILE-TIME META-PROGRAMMING IN CONVERGE

4.1 A first example

For the purposes of this paper, compile-time meta-programming can be largely thought of as being equivalent to macros; more formally, it allows the user of a programming language a mechanism to interact with the compiler to allow the construction of arbitrary program fragments by user code. Compile-time meta-programming allows users to e.g. add new features to a language [Sheard et al. 1999] or apply application specific optimizations [Seefried et al. 2004]. Converge's compile-time meta-programming facilities were inspired by those found in Template Haskell [Sheard and Jones 2002], and are detailed in depth in [Tratt 2005a]. In essence Converge provides a mechanism to allow its concrete syntax to describe Abstract Syntax Trees (ASTs) which can then be then spliced into a source file.

The following program is a simple example of compile-time meta-programming, trivially adopted from its TH cousin in [Czarnecki et al. 2004]. `expand_power` recursively creates an expression that multiplies `x` `n` 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 directly from TH. *Quasi-quoted* expressions `[| ... |]` build ASTs that represent the program code contained within them whilst ensuring that variable references respect Converge's lexical scoping rules. Splice annotations `$<...>` evaluate the expression within at compile-time (and before VM instruction generation), replacing the splice annotation itself with the AST resulting from its evaluation. This is achieved by creating a temporary module containing the splice expression in a function, compiling the temporary module into bytecode, injecting it into the running VM, and then evaluating the function therein. Insertions `${...}` are used within quasi-quotes; they evaluate the expression within and copy the resulting AST into the AST being generated by the quasi-quote.

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

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

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. As this example highlights, a substantial difference from traditional LISP derived macro schemes is that Converge functions are not explicitly identified as being macros — they are normal functions that happen to be called at compile-time.

This terse explanation hides much of the necessary detail which can allow readers who are unfamiliar with similar systems to make sense of the actual synthesis. In the context of this paper, the intent of Converge's compile-time meta-programming is more important than the precise details; any details vital for the understanding of this paper are described as they are needed.

5. SYNTAX EXTENSION IN CONVERGE

In this section I explain how arbitrary syntaxes can be embedded into Converge via the DSL block construct, and how that can be used to embed DSLs. Before that, I outline the paper's running example of a model transformation DSL, and briefly outline the parsing facilities available within the Converge system.

5.1 MT

For much of the rest of this paper I use a simplified version of the MT model transformation DSL. MT transforms one UML-like model into another and is described fully in [Tratt 2005c]. It does this by specifying a 'regular expressions for models' language which matches against source models, and a syntactically similar language for creating target model elements. An MT transformation has a name and consists of one or more rules, the ordering of which is significant. Rules are effectively functions which define a fixed number of parameters and which either succeed or fail depending on whether the rule matches against given arguments. The following fragment shows a transformation which converts a class model – effectively a simplified UML model – into a Relational DataBase Model Scheme (RDBMS):

```
transformation Classes_To_Tables

rule Class_To_Table:
  srcp:
    (Class)[name == <n>]

  tgtp:
    (Table)[name := n]
```

The `Class_To_Table` rule matches a `Class` model element binding whatever its name is to the variable `n`; if the match is successful it then produces a `Table` model element whose name is the same as the matched class. This example of converting class models to RDBMS is standard in the model transformations community and is useful because most readers have an intuitive idea of what the transformation should involve (converting classes to tables, attributes to columns etc.), and it is easy to progressively add complexity to the example.

The purpose of this paper is not to explain MT; however, using the class to RDBMS transformation as an example, as this paper progresses we will gradually add further complexity to the model transformation language under consideration to show Converge's DSL features.

5.2 Parsing in Converge

Converge provides a parser toolkit (the Converge Parser Kit or CPK) which contains a parsing algorithm based on that presented by Earley [Earley 1970]. 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. By allowing grammars to be expressed without regard for many of the traditional parsing headaches such as shift-reduce conflicts, one barrier to rapid DSL development is removed. 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 from [Aycock and Horspool 2002] 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 more than one thousand lines per second on an average machine, which is sufficient for the purposes of

this paper. The performance of more sophisticated Earley parsers such as Accent [Schröer 2005] suggest that the CPK's performance could be raised significantly with relatively little effort.

Parsing in Converge is preceded by a tokenization (also known as lexing) phase. Token objects record the value and type of the text they represent, as well as recording the source file and character offset within that file of the text. The CPK provides no special support for tokenization, since the built-in regular expression library makes the creation of custom tokenizers trivial. However the standard Converge tokenizer was designed to have a certain degree of flexibility built in, and can accept a list of symbols to identify as extra keywords. As we will see later it can often be used to tokenize DSLs; however it is perfectly acceptable to use other tokenizers for DSLs if more appropriate.

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:

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

Terminals in the grammar are those surrounded by quote marks. Groups of tokens can be surrounded by brackets and suffixed by `?` (must appear zero or one times), `*` (may appear zero or more times), or `+` (must appear one or more times). The `precedence` annotations allow an unambiguous parse tree to be created from an ambiguous parse forest. The parse tree resulting from parsing the expression `5 + 2 * 3` with this grammar is:

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

The Converge compiler itself uses the CPK to compile Converge files. A custom tokenizer is provided to deal with Converge's unusual indentation-based syntax. The parse tree produced by the CPK is traversed, and converted into an AST; the AST is then converted into bytecode. If one ignores the extra parts of the Converge compiler dedicated to compile-time meta-programming, its implementation is similar to standard compilers.

5.3 DSL blocks

A DSL can be embedded into a Converge source file via a *DSL block*. Such a block is introduced by a variant on the splice syntax `$<<`*`expr`*`>>` where *`expr`* should evaluate to a function (the *DSL implementation function*). The DSL implementation function is called at compile-time with a string representing the DSL block (section 8 explains why DSL blocks provide benefits above simply using raw strings), 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. Colloquially one uses the DSL implementation function to talk about the DSL block as being 'an *expr* block'.

Although in this paper I only discuss DSL blocks, Converge also supports *DSL phrases* which are essentially intra-line DSL inputs, suitable for smaller DSLs such as SQL queries.

5.4 A first example

We can now see the first example of our running example, the MT DSL, with the following DSL block:

```
$<<MT::mt>>:
  transformation Classes_To_Tables

  rule Class_To_Table:
    srcp:
      (Class)[name == <n>]

    tgtp:
      (Table)[name := n]
```

The `MT::mt` referred to between angled brackets is the DSL implementation function, and refers to a function defined in the `MT` module.

5.4.1 *Grammar.* In the interests of brevity, the MT DSL uses the standard Converge tokenizer. Therefore we start our example with a heavily simplified version of the MT grammar:

```
mt_rules ::= "TRANSFORMATION" "ID" ( "NEWLINE" mt_rule )+
mt_rule ::= "RULE" "ID" ":" "INDENT" mt_src "NEWLINE" mt_tgt "DEDENT"

mt_src ::= "SRCP" ":" "INDENT" pt_spattern "DEDENT"
mt_tgt ::= "TGTP" ":" "INDENT" mt_tgt_expr "DEDENT"

pt_spattern ::= pt_smodel_pattern
pt_smodel_pattern ::= "(" "ID" ")" "[" "ID" "==" pt_spattern_expr "]"
                    | "(" "ID" ")" "[" "]"
pt_smodel_pattern_self ::= "ID"
pt_spattern_expr ::= "<" "ID" ">"

mt_tgt_expr ::= "(" "ID" ")" "[" "ID" ":=" "<" "ID" ">" "]"
```

This specifies that a valid transformation consists of one or more rules. Each rule optionally contains matching (a *source clause*) and producing (a *target clause*) clauses. Source clauses contain a pattern matching expression. In this simplified grammar only simple *model element patterns*, of the form `(M)[x == p]` are expressible; this matches against a model element of type `M` which has an `x` slot which matches the pattern `p`, where `p` can be a *variable binding* of the form `<v>` which automatically matches the slots contents and binds it to `v`. Target clauses contain a *model element expression* of the form `(M)[x == v]` which creates a new `M` model element whose `x` slot is set to the value of the variable `v`. While this simplified MT variant is not very powerful, it does give a flavour of the full DSL.

5.4.2 *DSL implementation function.* DSL implementation functions tend to follow a set form. Since MT uses Converge tokenizer, it is able to use a single function call to obtain a parse tree; all it then has to do is pass it to the AST translator and return the resulting AST. The DSL implementation function thus looks as follows:

```
func mt(dsl_block, src_infos):
  parse_tree := CEI::dsl_parse(dsl_block, src_infos, ["transformation", "srcp", \
    "tgtp", "src_when", "tgt_where"], [".."], GRAMMAR, "mt_rules")

  return MT_Translater.new().generate(parse_tree)
```

The first argument to the function is a string representing the DSL block; the `src_infos` argument is covered later in section 8. The Compiler External Interface (CEI) module is the interface between a Converge program and the Converge compiler. The `dsl_parse` function is a convenience function which takes a DSL block, a list of src infos, a list of extra keywords for the tokenizer, a list of extra symbols for the tokenizer, a grammar (defined elsewhere, and bound to the variable `GRAMMAR`), and the name of the start rule in the grammar; it returns a parse tree. As this shows, the cut-down MT DSL defines five new keywords (`transformation`, `srcp` etc.) and a single new symbol '..'. Any errors that occur during parsing are automatically reported to the user.

5.4.3 *Translation to AST.* The next step is the translation of an MT DSL block into an AST representing a transformation. The CPK provides a generic parse tree `Traverser` class (influenced by that found in the SPARK parsing kit [Aycock and Horspool 2002]); for each relevant node ***n*** in the parse tree a corresponding *traversal function* `_t_`***n*** should be created. The `preorder` function takes a node in the parse tree and passes it to the appropriate traversal function. For this simplified MT variant, the translator class contains four traversal functions (note that `Traverser::Traverser` is a superclass):

```
1  class MT_Translator(Traverser::Traverser):
2    func _t_mt_rules(node):
3      // mt_rules ::= "TRANSFORMATION" "ID" ( "NEWLINE" mt_rule )+
4      rules := []
5      i := 4
6      while i < node.len():
7        rules.append(self.preorder(node[i]))
8        i += 2
9      return [|
10       class ${node[2].value}:
11         func init(*root_set):
12           for obj := root_set.iterate():
13             self.transform(obj)
14         func transform(obj):
15           Try each rule in order on obj.
16         ${rules}
17     |]
18
19   func _t_mt_rule(node):
20     // mt_rule ::= "RULE" "ID" ":" "INDENT" mt_src "NEWLINE" mt_tgt "DEDENT"
21     return [|
22       func ${node[2].value)}(obj):
```

```
23          // Try to match the input elements with the source pattern.
24          if bindings := ${self.preorder(node[5])}(obj):
25            // If we matched successfully, create the target object.
26            return ${self.preorder(node[7])}(bindings)
27          else:
28            return fail
29      |]
30
31    func _t_mt_src(node):
32      // mt_src ::= "SRCP" ":" "INDENT" pt_spattern "DEDENT"
33      // pt_smodel_pattern ::= "(" "ID" ")" "[" "ID" "==" pt_spattern_expr "]"
34      // pt_spattern_expr ::= "<" "ID" ">"
35      return [|
36        func (obj):
37          // Check the model types match.
38          if obj.instance_of.name != ${node[4][2].value}:
39            return fail
40          // Check the model element has the required slot name.
41          slot_name := ${CEI::lift(node[4][5].value)}
42          // Bindings always match.
43          bindings := Dict{${CEI::lift(node[4][7][2].value)} : obj.get_slot(slot_name)}
44          return bindings
45      |]
46
47    func _t_mt_tgt(node):
48      // Create target elements.
49      // ...
```

This translation is deliberately very simplistic. Essentially every transformation is translated to a class, which is initialised with a 'var args' list of objects. Each transformation rule is translated to a function in this class. Translated rules make use of Converge's Icon-esque semantics where a function `f` which returns the special `fail` object causes the expression `not f(...)` to evaluate to true. This means that if a translated rule fails to match against an input object, it will signal that failure to its caller (line 40); if it succeeds it returns a dictionary of variable bindings.

Two features used in the translation require particular explanation. First is the `${rules}` expression of line 13. In this case, `rules` is a list of ASTs representing functions; this list is inserted into the class being created, and each function in that list becomes a single function in the class. Second are the calls to `CEI::lift` (lines 42 and 44). The CEI module exposes functions to the user to build up ASTs that cannot be expressed via the quasi-quote mechanism (see section 4.1). The `lift` function[5] takes standard Converge objects (strings, lists etc) and recursively converts them into their AST equivalent (e.g. `[| 3 |] == CEI::lift(3)`). In the above example, the two calls to `lift` return AST strings.

For the original example, the resulting elided pretty-printed AST is the following:

```
class Classes_To_Tables:
  func init(*$$0$$root_set$$):
    ...

  func transform($$1$$obj$$):
```

[5]Note that the `lift` function is equivalent to its namesake in Template Haskell.

```
  ...

func Class_To_Table($$2$$obj$$):
  if $$3$$bindings$$ := func ($$4$$obj$$) {
    if $$4$$obj$$.instance_of.name != Class:
      return fail
    $$5$$slot_name$$ := "name"
    $$6$$bindings$$ := Dict{"n" : $$4$$obj$$.get_slot($$5$$slot_name$$)}
    return $$6$$bindings$$
  }($$2$$obj$$):
    return func ($$7$$obj$$) {
      ...
    }($$3$$bindings$$)
  else:
    return fail
```

The only surprise in this pretty printed AST is that variable names have been surrounded by `$$`; this is part of Converge's hygiene system and is explored in more detail in section 9.

Although this cut-down MT variant is too simplistic to be of real use, it outlines the basics both of DSL creation in Converge and of MT itself. While one can easily imagine implementing this particular DSL in a stand-alone fashion (in similar fashion to e.g. Yacc), this example does show that the embedding implementation in Converge is relatively small and simple. However this example does not use any of Converge's more advanced DSL features. In the following sections I show how this example can be extended to more advanced and realistic examples.

6. ADDING AN EXPRESSION LANGUAGE TO A DSL

The initial cut-down MT DSL allowed only very simplistic rules to be expressed; real model transformations require significantly more expressive power. For example the grammar of section 5.4.1 was particularly limited in its capacity to create new target model elements, only allowing references to variable bindings from the source model elements. In reality, model transformations often require arbitrary calculations when creating target model elements. In the class to RDBMS model transformation, a simple example presents itself. In most programming languages, class names conventionally begin with a capital letter; many RDBMS systems however require table names to be entirely in lower case.

There are two ways to extend the cut-down MT DSL to allow it to express more complex transformations. The traditional approach would be to add entirely new syntactic constructs for the desired new functionality. If we knew that occasionally making a string lower-case was the only extension we would need, this might be a good route to take. However repeatedly taking this approach as new requirements arise, as is common, tends to lead to a DSL that resembles 'a complex general purpose language... where one has to look hard to find the pure domain-specific abstractions that were its foundation' [Hudak 1998]. Since we can reasonably assume that our model transformation DSL will need many such small extensions, continually adding new syntax is unlikely to be a practical long-term situation. Alternatively one could add an expression language to the DSL, minimising the need for new syntactic constructs. However creating a new expression language designed

specifically for the DSL is often expensive in implementation terms; it also has an unwanted cost for users as it forces them to learn 'yet another' expression language which is similar, but not identical, to those they already know.

In an homogeneous embedding environment a different approach to adding an expression language is possible. Rather than creating a new expression language, it is possible to reuse the one already provided by the host language. This is particularly easy in the sort of embedding provided by LISP. However it is not clear how to do this in the presence of syntax extension of the sort provided by Converge — when writing code within a DSL block, there is no obvious way to escape back to standard Converge expressions.

Fortunately the Converge compiler allows grammars and DSL translations to be created as extensions to it. This means that DSLs can integrate the standard Converge expression language within them. Since CPK grammars are currently expressed as strings, unioning two grammars is simple, although it does not provide strong guarantees about the resulting grammar; hopefully future versions of Converge will provide modularisable grammars. In the interim, prefixing rules with `mt_` or `pt_` guarantees no clashes between the union grammar. Thus to make use of Converge's expression language we need only edit the `mt_tgt_expr` rule to the following:

```
mt_tgt_expr ::= "(" "ID" ")" "[" "ID" ":=" expr "]"
```

In so doing, target model expressions can now make use of the standard Converge expression language which is expressed in the `expr` grammar rule (Converge's full grammar can be found in [Tratt 2005b]). The translation class then needs to be updated to the following:

```
1  class MT_Translator(IModule_Generator::IModule_Generator):
2    ...
3    func _t_mt_tgt_expr(node):
4      // mt_tgt_expr ::= "(" "ID" ")" "[" "ID" ":=" expr "]"
5      return [|
6        func (bindings):
7          // Lookup the element type.
8          new_elem_type := CLASSES_REPOSITORY[${CEI::lift(node[2].value)}]
9          // Create a new element whose slots default value is null.
10         new_elem := new_elem_type.new()
11         expr := ${self.preorder(node[7])}
12         expr_b := Lookup variable references in bindings dict.
13         // Populate the specified slot.
14         new_elem.${node[5].value} := expr_b
15         return new_elem
16     |]
```

The crucial line in this line 11 - the `self.preorder` call is all that is needed to integrate Converge's expression language with the DSL. Line 12 calls a function, elided in the interests of brevity, which takes in an AST and replaces variable references with lookups in the `bindings` dictionary. One can use this feature to express model transformations such as the following:

```
rule Class_To_Table:
  srcp:
    (Class)[name == <n>]
```

```
tgtp:
  (Table)[name := n.to_lower_case()]
```

This example is interesting because it shows not only that it is possible to integrate the standard Converge expression language into a DSL, but that such integration requires a minimum of effort on the DSL authors behalf. Indeed this ease of use allows the full MT DSL to make use of the integrated expression language in several different contexts. Furthermore, it should be noted that by embedding the expression language, DSLs can be embedded within themselves, and within other DSLs, to an arbitrary depth.

Although it is not made explicit in this example, this integration is fully hygienic in the sense that no unintended variables can be captured. This important property is explored in more detail in section 9.

7. CONTRASTING DSL EVOLUTION THROUGH THE ADDITION OF SYNTACTIC CONSTRUCTS, AND THROUGH USING AN INTEGRATED EXPRESSION LANGUAGE

One of the problems noted in [Hudak 1998] is that DSLs tend to be subject to continual, unplanned evolution that can eventually lead to the DSL losing much of the domain-specific focus that made it desirable in the first place. In the context of Converge DSLs new requirements can be addressed either by adding new syntax to the DSL or by integrating an expression language. In this section I show how two different features of MT are best added to the cut-down MT DSL, one by adding a new syntactic construct, and one by using the integrated expression language.

7.1 Adding new syntactic constructs

In model transformations, it is often the case that one wants to match against more than one model element. For example, when transforming class models, one often needs to find all associations which have a given class as their source. One can certainly imagine using an integrated expression language to solve this problem, but the resulting solution would be somewhat verbose and clumsy since backtracking is required. However there is a related domain which provides a much neater syntactic solution to this problem. Most developers have at least a passing familiarity with textual regular expressions, where expressions such as `ab*` mean that the every string starting with an 'a' character followed by zero or more 'b's will match the regular expression. Complex regular expressions can encode sophisticated behaviour in terse text.

MT therefore co-opts the familiar syntax and semantics of regular expression multiplicities such as `*`. Every pattern in a source clause can optionally be suffixed by a multiplicity which specifies how many times the pattern should match, and a variable binding to which the resulting list of matched model elements will be assigned[6]. A simplified version of the grammar for multiplicities is as follows:

```
pt_spattern ::= pt_smodel_pattern ( pt_multiplicity "<" "ID" ">" )?
pt_multiplicity ::= "*"
```

[6]Note that in the full version of MT a list of dictionaries is assigned, but this complexity is elided in the interests of brevity in this paper.

```
                | "*" "!"
                | "*" "?"
```

In order, the three multiplicities in this fragment are 'greedily match zero or more times', 'must match all', 'non-greedily match zero or more times'. Assuming that the cut-down MT DSL is also enhanced so that model element patterns and model element expressions can now accept more than one slot, rules such as the following can be expressed:

```
rule Class_To_X:
  srcp:
    (Class, <c>)[name == <n>]
    (Association)[src = c] : * <assocs>

  tgtp:
    ...
```

In the interests of brevity, I do not include the full translation of this feature, since matching in the face of multiplicities involves backtracking, which requires careful, if tedious, book-keeping. However, the simple syntax addition of multiplicities hides a significant layer of complexity from the MT user.

7.2 Using the expression language for complex calculations

Although adding an expression language to model element expressions in section 6 allows more powerful model transformations to be expressed, there are occasions when more complex calculations can not be captured in a single expression. As in section 6, adding new syntax is not a scalable solution; but unlike section 6 we really need to embed more than just a simple expression language. What we need to embed is a sequence of expressions, and for that sequence of expressions to provide full programming language like abilities.

MT thus provides a 'where' sub-clause `tgt_where` for the target part of a rule (and also a similar, but not identical, 'when' sub-clause for the matching part of rule), which allows an arbitrary number of Converge expressions to be evaluated before the main `tgtp` clause is executed. In the `tgt_where` sub-clause, variable bindings can be accessed from the matching part of the rule, and new variables defined in the sub-clause can be referenced in the `tgtp` clause. The affected grammar rules are the following:

```
mt_tgt ::= "TGTP" ":" "INDENT" mt_tgt_expr ( mt_tgtw )? "DEDENT"
mt_tgtw ::= "NEWLINE" "TGT_WHERE" ":" "INDENT" expr_block "DEDENT"
```

`expr_block` references a rule in the Converge grammar in identical fashion to the previous reference to the `expr` rule. Similarly, the translation of an expression block in a DSL requires only a single line of code, but allows calculations such as the following to be expressed:

```
rule Class_To_Table:
  srcp:
    (Class)[name == <n>, attributes == <attrs>]
    (Association)[src = c] : * <assocs>

  tgtp:
    (Table)[name := n.to_lower_case(), cols := cols]
```

```
tgtp_where:
  cols := []
  for a := (attrs + assocs).iterate():
    cols.append(self.transform(a))
```

As we shall see in section 9, it is also possible for code in the embedded expression language to call normal Converge code outside of the DSL block.

7.3 Comparison

So far this section has skirted around an inherently subjective field, that of language design. There can be no absolute rules as to when it is best to add new syntax to a DSL, or when instead to embed an expression language. However since every DSL designer is by definition a language designer, guidance of some sort is useful. As these examples suggest, there are some simple questions that DSL designers can ask themselves that may aid in this decision. If the desired feature is fairly specific to the DSLs domain and if it expresses a concept that translates to a lot of back-end code, it may be a candidate for new syntax. If the feature seems as if it may be an instance of a more generic feature, consider embedding an expression language. If in doubt, it is often best to err on the side of embedding an expression language since – as many language designers will ruefully attest – removing syntax tends to be a politically difficult sell to users.

8. ERROR REPORTING

A significant usability problem associated with conventional DSL embedding, particularly when it involves syntax extension, concerns error reporting. This is both a problem at compile-time and particularly at run-time: when an error occurs, should the result be flagged in terms of the original user's input, or in terms of what it is translated into? The intuitive answer is the former; however none of the syntax extension systems described earlier in this paper are capable of reliably achieving this ([Tratt 2005a] evaluates a number of macro and related systems in this regard).

Converge takes a novel approach to this problem based on the concept of a *src info*. Not only can DSL authors can report errors to users at compile-time, but the error information displayed at run-time can be fully customised based on the user's input. In this section, I first explain Converge's src info concept, before showing how errors can be reported and customised.

8.1 Src infos

In section 5.4.2 we saw that DSL implementation functions take two arguments: a string representing the DSL block and the hitherto mysterious `src_infos` argument. This latter argument is a list of src infos. A src info is a (src path, src offset) pair which records a relationship back to a specific byte offset in a specific input file. Src infos start life in the Converge tokenizer, where every token records its src info; when the compiler converts parse trees to ASTs, the ASTs carry over the relevant src infos; and when the compiler compiles ASTs into bytecode, every single bytecode instruction records the src infos it relates to. Thus the src info concept is used uniformly throughout the Converge parser, compiler, and VM.

There are three important design decisions behind the src info concept. First, source paths are stored in a src info because compile-time meta-programming means that individual binary files may arbitrarily interleave bytecode generated from more than one source file. Second, storing the byte offset allow errors reporting to pinpoint errors within (and not just between) lines; when reporting errors in syntactically rich DSLs, knowing merely the line number of an error is often not enough information to debug problems. Third, most importantly and, I believe, entirely unique to Converge is that Converge always deals with lists of src infos, so that individual errors can be reported as belonging to more than one location. This latter concept is examined in section 8.4.

8.2 Using DSL blocks and src infos for accurate error reporting

Let us assume the user attempts to compile a file containing the following marginally incorrect DSL block (which contains a spurious colon character ':'):

```
$<<MT::mt>>:
  transformation Class_To_Table:

  ...
```

If we were to naïvely parse the `dsl_block` argument to a DSL implementation function, then the resulting parsing error that the user would see would be along the following lines:

```
Line 1, column 31: Parsing error at or near ':' token.
```

This is misleading since the DSL block must have been at least 2 lines into the file.

Up until this point, DSL blocks could have been seen to be somewhat superfluous as the same effect could have been achieved using normal strings[7] as in the following:

```
$<MT::mt("""
  transformation Class_To_Table:

  ...
""")>
```

The disadvantage of using strings to embed DSLs is that the Converge compiler has no way of knowing that it should pass information to the `MT::mt` function about the position of the DSL input within the overall source file. Thus a motivating factor in making DSL blocks a distinct syntactic construct in Converge is that it signifies that the DSL implementation function will be passed src infos, allowing it to accurately report errors to the user.

In fact, the Converge tokenizer is passed (generally via the `dsl_parse` function) the `src_infos` argument from the MT DSL implementation function, which it uses to add the appropriate byte offset to all the tokens it produces. Thus when the parser comes to report an error it reports the true line number of the error:

```
Line 37, column 31: Parsing error at or near ':' token.
```

[7] As in Python, triple quoted strings in Converge allow single quotes to be embedded within them.

8.3 Reporting DSL defined errors at compile-time

One of the advantages of using compile-time meta-programming to embed a DSL is that a significant amount of analysis can be done by a DSL at compile-time; Converge provides a simple framework for reporting resulting errors to the user.

In the case of the MT DSL, a simple error that can be detected is a variable being referenced despite not being bound anywhere, such as the following:

```
$<<MT::mt>>:
  transformation Class_To_Table:

  rule Class_To_Table:
    srcp:
      (Class)[name == <n>]

    tgtp:
      (Table)[name := name]
```

Such errors can be reported by passing a string and a list of src infos to the `error` (or `warning` if more appropriate) function in the compiler object, which is accessed via the `CEI::compiler()` method. The src infos associated with a given token can be accessed via the tokens `src_infos` slot. An elided example is as follows:

```
func _t_mt_tgt_expr(node):
  // mt_tgt_expr ::= "(" "ID" ")" "[" "ID" ":=" "<" "ID" ">" "]"
  if variable in node[8] not found:
    CEI::compiler().error(Strings::format("Variable '%s' not bound.",
      node[8].value), node[8].src_info)
```

Which results in an error such as the following being reported to the user:

```
Line 44, column 19: Variable 'name' not bound.
```

Using this facility, it is possible to create DSLs with e.g. sophisticated static type systems, and to report any errors in the input to the user.

8.4 The need for src infos as lists

Users generally assume that compilers are flawless programs. However bugs created by compilers in generated code can be exceedingly frustrating to track down, since it is generally unclear from the resulting run-time error whether the problem is due to a bug in the user's input program, or a bug in the code generator. This problem is identical in DSL development where run-time errors in particular may be the result of bugs in the DSL user's input, or bugs in the DSL implementation.

Taking one translation function from section 6, we introduce a random – but, for the sake of example, obvious – bug into the code:

```
1  func _t_mt_rule(node):
2    // mt_rule ::= "RULE" "ID" ":" "INDENT" mt_src "NEWLINE" mt_tgt "DEDENT"
3    return [|
4      func ${node[2].value)}(obj):
5        // Try to match the input elements with the source pattern.
6        if bindings := ${self.preorder(node[5])}(obj):
7          Sys::println("Matched ", obj.name, " class")
8          // If we matched successfully, create the target object.
9          return ${self.preorder(node[7])}(bindings)
```

```
10          else:
11            return fail
12      |]
```

The error here is in some ways subtle. When a source clause matches against a class (or any other object with a `name` slot), line 7 prints that name out. However if the user matches against a model element without a `name` slot, an exception will be raised. Let us assume we have the class to RDBMS transformation and a new (contrived) extra rule `X` where `Y` is a model element type without a `name` slot:

```
rule Class_To_Table:
  ...

rule X:
  srcp:
    (Y)[...]

  ...
```

Compiling a file `c2r.cv` with the above DSL block in it, with rule `X` starting at line 98, will proceed without error. When the user runs their transformation, class names will be printed out to screen until such point as the `X` rule matches against a `Y` element. At such point an exception along the following lines will be raised:

```
Traceback (most recent call last):
  1: File "c2r.cv", line 188, column 8, in main
  2: File "MT.cv", line 117, column 18, in _t_mt_tgt_expr
Exception: No such slot 'name' in instance of 'Y'.
```

If the user opens up the `MT.cv` file and looks at line 117, they will see the `Sys::println` function call. This is deeply confusing: the user has no idea why this line, which has clearly executed successfully on several occasions, suddenly fails. Similarly, line 88 in `c2r.cv` is simply the function call which runs a model transformation, providing no useful clues. The reason for this confusion is simple: ASTs with bugs in them have no direct relation to the user input that caused them to fail. Errors in the DSL user's input – a very different class of error – also lead to such confusing error reports.

The fundamental question thus becomes: what source location should be reported as being the source of the error? For many macro systems, unfortunately the answer is not to report any source location as being the source of the error. A few macro systems can pinpoint the splice-site of the AST as the source of the error; thus all errors appear to originate from a single source location. By reporting the precise location in the DSL translator, Converge is therefore already providing some useful information for debugging.

Recent versions of Converge have a powerful and novel feature, allowing AST elements to be associated with one or more src infos; similarly, bytecode instructions record their location to one or more src infos. This makes it possible to pinpoint an error as being related to multiple source locations. In our case it is therefore possible to relate an AST element both to its location in the AST generator and its location in the user's DSL input. This means that stack backtraces of the following form can be reported:

```
Traceback (most recent call last):
  1: File "c2r.cv", line 188, column 8, in main
  2: File "MT.cv", line 117, column 18, in _t_mt_tgt_expr
     File "c2r.cv", line 98, column 2, in X
Exception: No such slot 'name' in instance of 'Y'.
```

This should be read as 'location 2 in the stack backtrace is associated with both line 117 in `MT.cv` and line 98 `c2r.cv`'. Thus the user and the DSL developer can check both the relevant part of the input transformation and the DSL translator to determine the cause of an error. In general, AST elements can be associated with any number of src infos, and it is not unusual in complex DSL development for individual AST elements to be associated with three or more src infos.

8.4.1 *Adding extra src infos to an AST element.* As implied earlier, when an AST is generated via quasi-quotes, it automatically records its src info in the source file containing the quasi-quotes. Quasi-quotes provide a simple syntactic extension which allows extra src infos to be added to the standard src info. This extended quasi-quotes takes the form of `[<e>| ... |]` where `e` is an expression which must evaluate to a list of src infos. Since tokens record their src infos, the standard idiom for using this is as follows:

```
1  func _t_mt_rule(node):
2    // mt_rule ::= "RULE" "ID" ":" "INDENT" mt_src "NEWLINE" mt_tgt "DEDENT"
3    return [<node[1].src_infos>|
4      ...
5    |]
```

Each element of the AST created by the quasi-quotes thus contains the extra src infos from `node[1].src_infos`. By using different src infos to augment different quasi-quoted expressions, it is possible to provide fine-grained error reports to the user which pinpoint errors to individual tokens in a line.

9. ENSURING THAT EMBEDDINGS WITH AN INTEGRATED EXPRESSION LANGUAGE ARE HYGIENIC

All of the examples thus far in this paper have been hygienic, which is defined to mean (as in LISP-esque macro systems) that variables are not captured inadvertently, in this case between translated DSL code and integrated Converge expressions. Ensuring that this property is maintained in DSL embeddings is crucial to making reliable embeddings. In this section, I briefly define the concept of hygiene, explain how DSL embeddings may break hygiene and hence be unsafe, before presenting a complete solution to this problem for DSLs in Converge.

9.1 Defining hygiene

The concept of hygiene is defined in [Kohlbecker et al. 1986], and is most easily explained by example. Consider the Converge functions `f` and `g`:

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

func g():
  x := 10
```

```
$<f()>
Sys.println(x)
```

The question to ask oneself is simple: when `g` is executed, what is printed to screen? In older macro systems, the answer would be `4` since when, during compilation, the AST from `f` was spliced into `g`, the assignment of `x` in `f` would 'capture' the `x` in `g`. Such unintended variable capture can be incredibly difficult to debug. This is thus a serious issue since it makes embeddings and macros 'treacherous [, working] in all cases but one: when the user ... inadvertently picks the wrong identifier name' [Kohlbecker et al. 1986].

The solution to this problem, as outlined in [Kohlbecker et al. 1986], is to α-rename variables to avoid such capturing. In identical fashion to Template Haskell, quasi-quoted Converge expressions in fact have another highly desirable property besides allowing AST's to be created via concrete syntax. Variables which are bound in quasi-quotes are preemptively α-renamed to a *fresh name*, that is a name which the compiler guarantees will be unique (see [Tratt 2005a] for more details). It is an important property of Converge that the α-renaming of variables in this manner does not affect a programs semantics. α-renaming of variables in quasi-quoting is the default behaviour, but can be turned off by prefixing a variable with the '`&`' character to achieve dynamic scoping. Although an implementation detail, it can be instructive to note that the variable `x` in the quasi-quoted expression in `f` is α-renamed to a variable whose fresh name will be along the lines of `$$0$$x$$`, ensuring that when it is spliced into `g` there is no inadvertent variable capture.

9.2 Why hygiene could be violated in DSL embeddings

At first it may seem that the α-renaming performed by the quasi-quotes mechanism precludes the possibility of non-hygienic embeddings; indeed many simple embeddings – including those seen up until this point in this paper – can use this as a guarantor of hygiene. However the simplistic AST generation techniques we have used previously do not scale well in practise. For example, the full version of the class to RDBMS transformation in the full MT DSL is around 100 lines long; the translated AST however is an order of magnitude larger. Creating large ASTs from sub-ASTs that are entirely localised and do not communicate with each other is generally impractical. In practise, complex DSLs frequently use dynamic scoping to bind together AST fragments into larger AST chunks (the full MT DSL uses this feature frequently). However this use of dynamic scoping, while entirely necessary from a practical point of view, can violate hygiene.

An example of this problem can be seen by considering an optional clause that MT rules can have: the `tracing_override` clause. When model transformations execute, tracing information recording the link between source and target model elements is created. Since the default tracing information created is sometimes not what is desired by the MT user, it can be overridden by using a `tracing_override` clause. A (slightly contrived) example of this part of the translation is as follows:

```
1  func _t_mt_tgt(node):
2    // mt_tgt ::= "TGTP" ":" "INDENT" mt_tgt_expr mt_tgt_trc "DEDENT"
3    // mt_tgt_trc ::= "TRACING_OVERRIDE" expr
4    //              |
5    if node[5].len() == 1:
```

```
6       // mt_tgt_trc ::=
7       tracing_expr := [| &tracing := ... |]
8     else:
9       // mt_tgt_trc ::= "TRACING_OVERRIDE" expr
10      tracing_expr := [| &tracing := ${self.preorder(node[5][2])} |]
11
12    return [|
13      func (obj):
14        new_elem := ${self.preorder(node[4]}(obj)
15        ${tracing_expr}
16        self.tracing.add(&tracing)
17        return new_elem
18    |]
```

The difference between this traversal function and its predecessors in this paper is not that the AST is built in fragments: it is that the AST fragments are linked together by the dynamically scoped variable `tracing` (lines 7, 10, and 16). This unfortunately means that this variable is also scoped over the `tgtp` clause and could cause inadvertent variable capture on the expressions therein (note that the `new_elem` variable in lines 14 and 17 will be α-renamed, thus avoiding this problem).

An example of this problem in practise is the following:

```
tracing := ...

$<<MT::mt>>:
  transformation T
  rule X:
    ...
    tracing_override:
      tracing + ...
```

Here a variable `tracing` is defined outside the DSL block. The user expects that the `tracing` reference within the DSL block will refer to the `tracing` variable defined outside the DSL block; instead it will be bound to the `tracing` variable defined in the AST created in the traversal function. Allowing the integrated expression language to reference variables outside allows the user significant expressive power. However to make this feature consistent with Converge's quasi-quotes mechanism, it must fully respect lexical scoping (in a similar fashion to [Dybvig et al. 1992]) which would be easily achievable if not for the problem of variable capture.

The issue with variable capture in DSL embeddings is that it results from the *deliberate* use of dynamic scoping; when dynamic scoping is not used, hygiene can not be violated. This problem may appear to be entirely of Converge's own making, since LISP-like approaches do not suffer from this issue. I believe that this is because 'traditional' uses of compile-time meta-programming tend to generate small ASTs which can be built in one step. Converge DSLs, such as MT, on the other hand often generate ASTs that are the equivalent of thousands of lines of hand written code; practical experience has shown that building these in one step is impractical. It is therefore necessary to create large ASTs from smaller ASTs, and to provide a mechanism – dynamic scoping – for these ASTs to communicate naturally with each other.

A partial solution to this problem is presented in [Tratt 2005c] which α-renames free variables and assigns the value of the variable outside of the DSL block to

the renamed variable. In the example of outer variable capture given earlier the translated AST passed to the compiler looks along the lines of the following:

```
tracing := [...]

class T:
  func (...):
    $$11$$tracing$$ := tracing
    ...
      $$11$$tracing$$ + [...]
    ...
```

The problem with this approach is that the two variables `$$11$$tracing$$` and `tracing` shadow each other; assigning to either variable during the execution of the DSL block means that the contents of the two become desynchronized, with consequently unpredictable results. Although it is possible to reduce the window of opportunity for such desynchronizations, it can not be eradicated altogether.

9.3 The core ingredient of a solution

More recent versions of Converge provide a small and novel language feature that allows outer variable capture to be avoided whilst also sidestepping the synchronization problems associated with variable shadowing. The `rename x as y` declaration at the beginning of a function causes the variable `x` to be completely removed from the scope of the function, and a new variable `y` to be introduced. `x` and `y` do not shadow each other; rather `y` represents the same 'underlying' variable as `x`, modulo α-renaming. Renaming a variable also makes it 'nonlocal' to a block, meaning that assigning to the renamed variable does not create a local variable but assigns to the 'underlying' variable in the outer block[8]. Thus the following prints `7` when run:

```
x := 3
func f():
  rename x as y
  y := 7
f()
Sys::println(x)
```

There are two things to note about this example. First it would be illegal to add an expression such as `z := x` to `f` in isolation, because there is no variable `x` in scope in `f` to reference. Second that adding an expression such as `x := 5` to `f` would not affect the value printed to screen, since the `x` in `f` would be local to `f`.

It is important to realise that the `rename` declaration is not intended to be used for day-to-day programming: it was specifically added to Converge to facilitate the integration of expression languages in embedded DSLs. The `rename` declaration allows us to make fully hygienic, transparent embeddings even in the face of dynamic scoping and integrated expression languages. Furthermore it does this while ensuring that standard compile-time meta-programming in Converge is hygienic by default in the same fashion as languages such as Template Haskell and Scheme.

[8]Although not discussed in this paper, non-renamed variables can also be declared as being nonlocal with the `nonlocal` keyword.

9.4 Making an embedding hygienic

The key to hygienic DSLs is to use the `rename` declaration to α-rename free variables in any instance of the embedded expression language to fresh names. Thus free variables can never be captured by dynamically scoped variables in the outer scope of translated AST. Although it is possible to code this by hand, it is an entirely mechanical operation and as such Converge provides a convenience function `CEI::mk_hygienic_itree` which takes in an AST and returns an AST of a `rename` declaration (renaming all free variables in the input AST to fresh names) and a copy of the input AST with all free variables α-renamed to the appropriate fresh name. Although it is not used in this paper, `mk_hygienic_itree` takes a second input parameter which is a list of variable names which, even if they are free, should not be α-renamed; this is used to pass values in and out of instances of the embedded expression language. As an example, `CEI::mk_hygienic_itree([| x + 2 + y |], Set{y})` returns two ASTs `rename x as $$8$$x$$` and `$$8$$x$$ + 2 + y`.

In order to use the `mk_hygienic_itree` function, the translation function from section 9.2 needs a slight rewriting to the following:

```
1  func _t_mt_tgt(node):
2    if node[5].len() == 1:
3      tracing_expr := [| &tracing := ... |]
4      renames := []
5    else:
6      expr := self.preorder(node[5][2])
7      renamed_expr, renames := CEI::mk_hygienic_tree(expr, Set{})
8      tracing_expr := [| &tracing := ${renamed_expr} |]
9
10   return [|
11     func (obj):
12       ${renames}
13       new_elem := ${self.preorder(node[4]}(obj)
14       ${tracing_expr}
15       self.tracing.add(&tracing)
16       return new_elem
17   |]
```

The crucial parts of this translation function are lines 7 and 12. Line 7 gets an AST of a `rename` declaration, and a version of the `tracing_override` expression with free variables α-renamed. Line 12 inserts the `rename` declaration into the AST. We can thus now be sure that there is no unintended variable capture. In the example of variable capture given earlier, this means that what is passed to the compiler after the DSL blocks translation looks along the lines of the following:

```
tracing := [...]

class T:
  func (...):
    rename tracing as $$11$$tracing$$
    ...
      $$11$$tracing$$ + [...]
    ...
```

With fully hygienic embeddings, DSLs can interact fully with surrounding code,

calling Converge functions in arbitrary modules and so on.

Creating large ASTs using dynamic scoping and `CEI::mk_hygienic_itree` requires that sub-ASTs only dynamically scope agreed variables; all other variables should use Converge's default behaviour of α-renaming variables to fresh names. In other words, if one is gluing together various sub-ASTs which communicate around the dynamically scoped variable `tracing`, then every other variable in each sub-AST must have been α-renamed (using `CEI::mk_hygienic_itree` if necessary). This is not as onerous a requirement as may first seem: it is equivalent to stating that code should not arbitrarily update the contents of objects which are 'owned' by other sections of code.

10. SEPARATING OUT THE KEY DSL-RELATED ASPECTS

Up to this point, I have concentrated on showing how showing how embedded DSLs can be created in Converge, and how those embeddings can be made safe. Although I believe Converge to be the first technology which allows safe, powerful DSL embeddings in an homogeneous fashion, it should not be seen as being necessarily special. Rather it can be seen as a combination of language features and compiler APIs which collectively constitute a number of design lessons and implementation techniques that are equally applicable to other languages.

This section has two aims. First to separate out the features which are fundamental to homogeneous DSL embedding and to explain how these might be applied to other languages – both those already in existence, or those yet to be designed. Second to explain why it is this particular combination of features that is necessary to allow homogeneous DSL embedding.

10.1 A compile-time meta-programming facility

The distinguishing feature of a homogeneous embedding DSL approach is the existence of a compile-time meta-programming facility. Since DSL embeddings can potentially be extremely complex it is necessary that this facility is hygienic, which also implies that it works at the AST, rather than concrete syntax, level (meaning that while LISP-style macros are sufficient, C-esque pre-processing isn't; see [Brabrand and Schwartzbach 2000]). As suitable facilities now exist in a different flavours of programming languages from lazy functional languages (Template Haskell) to dynamically typed OO languages (Converge), it is reasonable to assume that adding such a facility should be possible for many programming languages.

Although in theory all that is needed in a raw compile-time meta-programming facility is an equivalent of the splice operator, I believe that in practical terms a mechanism similar to Converge's quasi-quotes facility is needed to allow AST's to be built up via concrete syntax. Without such a mechanism users are forced to build ASTs via an API, which tends towards the unusable [Weise and Crew 1993].

10.2 DSL blocks

As shown in section 8.2, DSL blocks are a vital part of the Converge approach to DSL. Although they initially appear to be almost identical to the normal splice annotation, DSL blocks signify to the Converge compiler that extra information needs to be passed to the DSL implementation function. This then allows both compile-time and run-time errors to be accurately reported to the user.

Any language with a compile-time meta-programming facility where macro definitions are normal first-class functions (such as Template Haskell) would need to have a similar syntactic construct to DSL blocks in order to signify that extra information should be passed to them. For LISP-esque macro systems, one could define a different category of macro (or some way of marking a normal macro as such) which signifies to the compiler that it requires extra information to be passed to it.

10.3 Accurate and complete error reporting

Traditional macro systems have paid little attention to the problem of accurately reporting errors to users. This is even more critical when embedding DSLs since errors could be due either to a faulty translation, or to a bug in the user's DSL input. Converge's src info concept, when threaded throughout a languages' compiler and execution environment, allows errors to be associated with multiple source locations leading to significantly easier debugging of errors.

10.4 Free variable calculation and α-renaming

If an expression in a DSL block can reference variables in scope outside of it, then various problems arise, including the violation of hygiene. It is my experience that even seemingly mundane embedded DSLs are likely to be non-hygienic if an integrated expression language is used. The principal part of a hygienic embedding solution is to be able to α-rename free variables in expressions in the DSL.

Any hygienic compile-time meta-programming facility is able to calculate an AST's free and bound variables, and α-rename variables as appropriate. Many such systems do not expose this functionality directly to the user, keeping it internal to the compiler; however users must be able to α-rename free variables in a given AST to fresh names, in order to make hygienic embeddings. The precise way in which this functionality is exposed is a matter of choice. Converge provides the `CEI::mk_hygienic_itree` convenience function which wraps the required functionality up into a single call. However it is also possible to obtain the same functionality at a much lower level e.g. by making use of the fact that each AST element has a `free_vars` slot which is a set detailing the elements' free variables.

10.5 Maintaining a link between α-renamed variables

Once issues concerning hygiene have been satisfied, a more subtle problem regarding the synchronization of the contents of an α-renamed variables to the contents of the non α-renamed variable arises (see section 9.2). It must be possible to 'link' an α-renamed variable to the unrenamed variable, so that changes to the contents of either variable are atomically reflected in the other. The use of the word atomic is deliberate: clever run-time tricks can reduce the problem, but can not fully resolve it (see [Tratt 2005c] for further explanation). Thus there needs to be some way of informing the compiler of the link between variables. This could be achieved in various ways; Converge's `rename` declaration is merely one approach, although it has the useful property that it doesn't effect 'normal' compile-time meta-programming notions of hygiene and so on.

10.6 Implications for a languages implementation

Assuming one has a compiler which can support compile-time meta-programming, adding support for DSL embedding of the type outlined in this paper is surprisingly trivial, since it can really be viewed as a variation on the semantics of the splice operator (this statement holds equally true for more traditional macro systems). In the case of the Converge compiler, the DSL embedding system detailed in this paper was only designed and added to the compiler after the complete base compiler, including the compile-time meta-programming facility, had been completed. I hope my experience with the Converge compiler is thus not unindicative of the effort needed for systems at a similar stage of development.

For the Converge compiler adding the concept of DSL blocks necessitated a rewrite of the tokenizer, but the addition of only around 10 lines of code to the rest of the compiler. Other changes to the compiler are mostly likely to involve small alterations to expose otherwise internal operations such as the ability to calculate an AST's free variables. In the case of Converge, only two substantial changes to the compilers internals were required: adding the `rename` declaration, which required touching a number of critical parts of the compiler, took around three man days of effort to design, implement, test and debug; adding the src info concept uniformly throughout the compiler took around one man day of effort since it touched nearly every part of the compiler. The only other component which may be missing from other systems is an equivalent parsing library to Converge's CPK. Since parser implementation is well understood, this is not of great concern.

10.7 Feature combination

A question which we are now able to consider is the following: is the complete set of features outlined earlier in this section necessary to achieve the same effect as Converge? In other words, can we drop features and still create the similar DSLs with similar levels of effort?

The first thing that can be said is that a compile-time meta-programming facility is the key enabling feature in a Converge-style approach. In order to make such a facility usable, one also needs a quasi-quotes mechanism to build ASTs. Although one can remove DSL blocks, this would prevent accurate intra-DSL error reporting; in practice this makes creating large, reliable DSLs very difficult.

Small DSLs can often be created in 'one shot'. However in order to implement large DSLs, one typically creates ASTs piece-meal, which means that dynamic scoping of variables of sub-ASTs is necessary to 'glue' sub-ASTs together. Allowing the host languages' expression language to be integrated into DSLs then raises the spectre of variable capture in the presence of such dynamic scoping; at that point, a facility such as the `rename` declaration (and possibly its associated `mk_hygienic_tree` function) is vital to retain the hygiene property.

Since a DSL implementation is as likely to contain bugs as any other program, the ability to track down errors quickly is important. Since errors reported to the user may result from bugs in their input or a bug in the translation, removing Converge's ability to easily associate multiple source locations with an AST fragment makes debugging significantly more difficult. Similarly, the fact that src infos can be associated with different files is vital, given that code is usually generated in one

file and spliced into another.

As this section suggests, it is possible to select a subset of the features presented in this section; however this also implies that only a subset of functionality is achieved. In certain cases, that may be all that is required. However full Converge-like functionality can only be achieved by combining the complete set of features outlined in this section. Note however that it might be possible to strip down some of the features, and still achieve similar functionality (e.g. the `mk_hygienic_tree` function is a convenience, rather than being a fundamental building block).

11. COMPARISON TO RELATED WORK

I detailed existing approaches to DSL embedding in detail in section 2; in this section I attempt to identify what place the approach outlined in this paper occupies within the spectrum of such systems. At a high level, a distinguishing feature of Converge is that it is geared towards facilitating the rapid development and prototyping of large DSLs; this particular class of problem has been left largely unexplored by previous approaches. However, there are many other areas for comparison with other approaches, which I now tackle.

Comparing homogeneous and heterogeneous embedding approaches has traditionally been very difficult. While both approaches have broadly similar aims, their wildly differing technical approaches mean that they were applied in very different contexts. Converge in some senses bridges these two extremes—it shows that homogeneous approaches can get much closer to heterogeneous approaches in expressive power than has previously been possible. Technologically speaking, heterogeneous embedding approaches are strictly more expressive than Converge, as they can be applied to any embedding problem, whereas Converge can only embed DSLs in Converge code. However practically speaking, heterogeneous approaches such as TXL are not well suited to embedding complex DSLs, as they have little knowledge of the language they are embedding into, placing this burden on the author of the embedding. However as Converge and TXL are specifically designed for very different tasks, there is little scope for a meaningful comparison.

More advanced approaches such as MetaBorg are capable of practically embedding vastly more complex DSLs than previous homogeneous embedding approaches. Some of the larger MetaBorg DSLs are similar in intent, and implementation, to some of the smaller Converge DSLs. Comparing Converge and MetaBorg is thus of practical interest. Clearly MetaBorg has the significant advantage that it is not tied to a specific target language. MetaBorg is also able to more neatly express small, fine-grained extensions to the target language, whereas Converge uses the slightly coarser-grained DSL block construct. However, for larger DSLs, Converge has several benefits. Whereas Converge allows full programming language expressivity to express complex DSLs, MetaBorg's term rewriting system is intentionally more limited. In a sense, MetaBorg can be thought of as a DSL for creating DSLs, whereas Converge is a programming language for creating DSLs. Perhaps more fundamentally, MetaBorg is also at the mercy of the target language in various aspects. For example, since run-time error reporting, in particular, depends on the target language, MetaBorg can not replicate Converge's advanced error reporting facilities. Similarly, MetaBorg can not practically allow large DSLs to be built out of sub-

ASTs which are glued together with dynamically scoped variables, as most target languages have no equivalent of Converge's `rename` declaration to restore hygiene. Thus MetaBorg's transformations can not be as easily decomposed or modularised as their Converge equivalents. In summary, a MetaBorg style approach is the only choice when an arbitrary target language is mandated, and will often be at least as practical as Converge for small DSLS; but for large DSLs, various Converge features, which do not (and in some cases, can not) have an equivalent in MetaBorg, can make a large practical difference.

The most obvious difference between the approach outlined in this paper and other homogeneous embedding approaches is with regard to syntax extension. Most homogeneous embedding approaches are not capable of syntax extension at all. The three languages capable of syntax extension – Nemerle, xTc, and Metalua – take a very different, more fine-grained, approach than Converge. Essentially these languages allow, at compile-time, their grammars to be extended. Therefore, and similarly to MetaBorg, for small extensions these languages allow a neater integration of new constructs than Converge can manage. Converge's DSL block construct is more coarse-grained, but has the advantage that DSL blocks need not be constrained by compatibility with the host language grammar. Furthermore Converge has a substantially more developed ability to report compile-time and run-time error messages in terms of the user's DSL input. Although the approach outlined in this paper is more involved than Hudak's DSEL concept for example, it is still relatively lightweight, and is directly comparable to these three languages in complexity.

12. FUTURE WORK

The main direction for future work is to continue implementing different DSLs for a wider variety of domains. In so doing, I would hope that useful techniques, idioms, and approaches for DSL implementation will be identified, codified, and documented. My experience of DSL implementation has already identified a number of common threads in DSL development, and I believe that further experience could help streamline the process substantially.

Currently, Converge consists of a compiler and associated libraries. Many development teams expect such systems to come with a plethora of associated tools, such as intelligent editors, debuggers, refactoring tools and so on. Compile-time meta-programming makes some of these more challenging—editors, and refactoring tools particularly. In order to make the technologies in Converge more acceptable to a wider audience, it will be necessary to uncover techniques for making compile-time meta-programming interact well with such tools.

13. CONCLUSIONS

In this paper I first identified the advantages of using DSLs to aid development. I then asserted that the traditional technique of implementing DSLs as stand-alone applications was slow, and resulted in implementations of variable quality. I then defined two different types of DSL embedding – homogeneous and heterogeneous. Crudely put, homogeneous embedding allowed simple DSLs to be implemented quickly and safely, whereas heterogeneous embedding allowed more complex DSLs

to be implemented but with fewer safety guarantees about e.g. hygiene; consequently most heterogeneous approaches have to restrict themselves to less complex embeddings.

This paper then presented a novel, practical, and complete approach to DSL implementation in an homogeneous embedding environment. By adding the simple concept of a *DSL block* to the Converge programming language, arbitrary syntaxes can be translated at compile-time using its compile-time meta-programming facility. I then showed how DSLs can integrate the expression language from the main Converge language, and how the src info concept allows accurate and detailed compile-time and run-time error reports. I then showed how DSL embeddings can be made fully hygienic in the presence of an integrated expression language. Finally I separated out the parts of Converge fundamental to homogeneous DSL embedding, and discussed how such features might be integrated into similar systems.

The running example in this paper was a cut-down version of the MT model transformation DSL; the full version of this DSL is documented in [Tratt 2005c]. MT is an example of a Converge DSL which implements sophisticated behaviour that is considered, by the model transformation community, to be inexpressible in a traditional homogeneous embedding approach. Despite the power of the resulting language, its implementation is relatively small at around 1000LoC, a fraction of the size of other model transformation implementations. MT is an example of a DSL which is particularly well suited to Converge since it is effectively a complete language in its own right; at its most extreme such DSLs effectively conflate the concept of Converge source files with DSL blocks. Conversely DSLs such as the TM (Typed Modelling language) DSL [Tratt 2005c] are not quite as easy a fit, since different DSL blocks, generally small in length, need to influence each other in some fashion; while this is possible, the resulting implementation is inevitably less neat and tidy.

Converge has also been used in industry to implement non-computing related DSLs such as DSLs for the telecoms and insurance domains. Although further experience will be required to make definitive statements in this regard, it already appears that Converge has a much wider applicability than was originally anticipated. Successful DSLs appear to share some common traits e.g.: careful thought has been put into the DSL design, most obviously the grammar; the DSL implementer is fluent and confident in the problem domain; the DSLs make use of as much 'normal' Converge code as possible to avoid duplicating effort. Although Converge's user's have reported DSL creation in Converge to be much swifter than traditional methods, the benefits become more pronounced for complex DSLs. In other words, the time needed to create a small DSL using 'traditional' techniques is relatively low, so there is not as much to improve upon as for larger DSLs, where the implementation effort may previously have been measured in weeks or months.

In summary I believe that the approach outlined in this paper details a useful new point in the DSL implementation spectrum — whilst more general than traditional homogeneous embedding approaches, the Converge system is often less complex in use than the more general heterogeneous embedding approaches. Although DSL design and implementation is a fundamentally challenging task, for many tasks Converge lowers the barrier to entry.

Free implementations of Converge (under a MIT / BSD-style license), which can execute all of the examples in this paper, can be found at `http://convergepl.org/`.

My thanks to Kelly Androutsopoulos for comments on an early draft, and to the anonymous referees whose detailed comments significantly improved latter versions. This research was partly funded by a grant from Tata Consultancy Services.

REFERENCES

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

Aycock, J. and Horspool, R. N. 2002. Practical Earley parsing. *The Computer Journal 45,* 6, 620–630.

Bachrach, J. and Playford, K. 1999. D-expressions: Lisp power, Dylan style. `http://www.ai.mit.edu/people/jrb/Projects/dexprs.pdf` Accessed Nov 22 2006.

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

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

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

Cordy, J. R. 2004. TXL - a language for programming language tools and applications. In *Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications*.

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

Dybvig, R. K., Hieb, R., and Bruggeman, C. 1992. Syntactic abstraction in scheme. In *Lisp and Symbolic Computation*. Vol. 5. 295–326.

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM 13,* 2 (Feb.).

Fleutot, F. and Tratt, L. 2007. Contrasting compile-time meta-programming in Metalua and Converge. In *Workshop on Dynamic Languages and Applications*.

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

Grimm, R. 2005. Systems need languages need systems! 2nd ECOOP Workshop on Programming Languages and Operating Systems.

Griswold, R. E. and Griswold, M. T. 1996. *The Icon Programming Language*, Third ed. Peer-to-Peer Communications.

Hudak, P. 1998. Modular domain specific languages and tools. In *Proc. Fifth International Conference on Software Reuse*. 134–142.

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

Mernik, M., Heering, J., and Sloane, A. M. 2003. When and how to develop domain-specific languages. Tech. rep., Centrum voor Wiskundeen Informatica. Dec.

Schröer, F. W. 2005. *The ACCENT Grammar Language*. `http://accent.compilertools.net/language.html` Accessed Jan 25 2005.

Seefried, S., Chakravarty, M., and Keller, G. 2004. Optimising Embedded DSLs using Template Haskell. In *Third International Conference on Generative Programming and Component Engineering*. Springer-Verlag, Vancouver, Canada, 186–205.

Sheard, T. 1998. Using MetaML: A staged programming language. *Advanced Functional Programming*, 207–239.

Sheard, T. 2003. Accomplishments and research challenges in meta-programming. *Proceedings of the Workshop on Semantics, Applications and Implementation of Program Generation (SAIG '01) 2196*, 2–44.

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

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

Skalski, K., Moskal, M., and Olszta, P. 2004. Meta-programming in Nemerle. `http://nemerle.org/metaprogramming.pdf` Accessed Nov 5 2007.

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

Tratt, L. 2005a. Compile-time meta-programming in a dynamically typed OO language. In *Proc. Dynamic Languages Symposium*. 49–64.

Tratt, L. 2005b. The Converge programming language. Tech. Rep. TR-05-01, Department of Computer Science, King's College London.

Tratt, L. 2005c. The MT model transformation language. Tech. Rep. TR-05-02, Department of Computer Science, King's College London. May.

Tratt, L. 2007. *Converge Reference Manual*. `http://www.convergepl.org/documentation/` Accessed June 3 2008.

van den Brand, M. G. J., Heering, J., Klint, P., and Olivier, P. A. 2002. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst. 24,* 4, 334–368.

van Deursen, A., Klint, P., and Visser, J. 2000. Domain-specific languages: An annotated bibliography. SIGPLAN Notices, vol. 35. 26–36.

van Rossum, G. 2003. Python 2.3 reference manual. `http://www.python.org/doc/2.3/ref/ref.html` Accessed June 3 2008.

Walker, K. 1994. The run-time implementation language for Icon. Tech. Rep. IPD261, University of Arizona.

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

Wilson, G. V. 2005. Extensible programming for the 21st century. *Queue 2,* 9 (Jan.), 48–57.