Evolving a DSL implementation

Laurence Tratt

Bournemouth University, Poole, Dorset, BH12 5BB, United Kingdom.
laurie@tratt.net, http://tratt.net/laurie/

Abstract. Domain Specific Languages (DSLs) are small languages designed for use in a specific domain. DSLs typically evolve quite radically throughout their lifetime, but current DSL implementation approaches are often clumsy in the face of such evolution. In this paper I present a case study of an DSL evolving in its syntax, semantics, and robustness, implemented in the Converge language. This shows how real-world DSL implementations can evolve along with changing requirements.

1 Introduction

Developing complex software in a General Purpose Language (GPL) often leads to situations where problems are not naturally expressible within the chosen GPL. This forces users to find a workaround, and encode their solution in as practical a fashion as they are able. Whilst such workarounds and encodings are often trivial, they can be exceedingly complex. DSLs aim to tackle the lack of expressivity in GPLS by allowing users to use mini-languages defined for specific problem areas. [1] define DSLs as 'languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with GPLs in their domain of application'. [2] describes the typical costs of a DSL, noting that a small extra initial investment in a DSL implementation typically leads to long term savings, in comparison to alternative routes. Exactly what identifies a particular language as being a 'DSL' is subjective, but intuitively I define it as a language with its own syntax and semantics, and which is smaller and less generic than a typical GPL such as Java.

Traditionally DSLs – for example the UNIX `make` program or the `yacc` parsing system – have been implemented as stand alone systems. The resulting high implementation costs, primarily due to the difficulties of practical reuse, have hindered the development of DSLs. Implementing DSLs as stand alone systems also leads to problems when DSLs evolve. DSLs tend to start out as small, declarative languages [3], but most tend to acquire new features as they are used in practise; such features tend to be directly borrowed from GPLs [2]. 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.

Recently, dedicated DSL implementation approaches such as Stratego [4], the commercial XMF [5], Converge [6], and others (e.g. [7–9]) have substantially reduced implementation costs through the embedding of DSLs in host languages.

As noted in [3, 2], DSLs tend to start small but grow rapidly when users find them useful, and desire more power. Specifically, such evolution often takes the form of functionality influenced by that found in GPLs. Continual evolution of DSL implementations is often difficult because such evolution is generally both unplanned and unanticipated, and therefore leads to the implementation becoming increasingly difficult to maintain [2]. In this paper I present a case study of a DSL for state machines implemented within Converge. I then show how this example can be easily evolved to a substantially more powerful version without compromising the quality of the implementation, and indeed improving the user experience. The evolution in this paper is intended to show typical unplanned evolution, where an implementation is gradually edited to reflect new and changing requirements.

This paper is structured as follows. First I present a brief overview of Converge, and its DSL related features (section 2). I then outline the case study and present an initial implementation (section 3) before extending its functionality (section 4) and increasing its robustness (section 5).

2 Converge

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 [10], it should allow readers familiar with a few other programming languages the opportunity to quickly come to grips with the most important areas of Converge, and to determine the areas where it differs from other languages.

2.1 Fundamental features

Converge's most obvious ancestor is Python [11] resulting in an indentation based syntax, a similar range and style of datatypes, and general sense of aesthetics. The most obvious initial 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 the first argument 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.new()
  Sys::println(fib_cache.fib(6))
```

2.2 Compile-time meta-programming

For the purposes of this paper, compile-time meta-programming can be largely thought of as being equivalent to macros; more precisely, 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 [7] or apply application specific optimizations [9]. Converge's compile-time meta-programming facilities were inspired by those found in Template Haskell (TH) [12], and are detailed in depth in [6]. In essence Converge provides a mechanism to allow its concrete syntax to naturally create Abstract Syntax Trees (ASTs), which can then be spliced into a source file.

The following program is a simple example of compile-time meta-programming, trivially adopted from its TH cousin in [8]. `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
```

2.3 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, 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'. DSL blocks make use of Converge's indentation based syntax; when the level of indentation falls, the DSL block is finished.

An example DSL block for a railway timetable DSL is as follows:

```
func timetable(dsl_block, src_infos):
  ...

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

As shall be seen later, DSL blocks have several useful features, particularly relating to high quality error reporting. 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.

3 Initial case study

The example used in this paper is that of a generic state machine. Although state machines are often represented graphically, they are easily represented textually. I start with a particularly simple variant of state machines which represents the basics: states and transitions with events. For example, Figure 1 shows a state machine which we wish to represent textually so that we can have a running state machine we can fire events at and examine its behaviour. In the rest of this section, I show the complete definition of a simple textual state machine DSL.

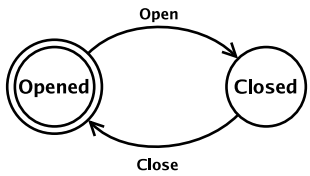

Fig. 1. Simple state machine of a door.

3.1 DSL grammar

Since we use DSL blocks within Converge, much of the potential difficulty with embedding a DSL is automatically taken care of. The first action of a DSL author is therefore to define a grammar his DSL must conform to. Converge allows DSL authors to parse the text of a DSL block in any way they choose. However most DSLs can be defined in a way which allows them to make use of Converge's flexible tokenizer (sometimes called a lexer), and its built-in Earley parser. This allows the implementation work for a DSL to be minimised as Earley parsing can deal with any context free grammar, without the restrictions common to most parsing approaches [13]. Expressing a suitable grammar for simple state machines is thus simple:

```
parse := $<<DSL::mk_parser("system", ["state", "transition", "and", \
  "or", "from", "to"], [])>>:
    system     ::= element ( "NEWLINE" element )*
    element    ::= state
                 | transition
    state      ::= "STATE" "ID"
    transition ::= "TRANSITION" "FROM" "ID" "TO" "ID" event
    event      ::= ":" "ID"
                 |
```

As this code fragment suggests, grammars are themselves a DSL in Converge. The above example creates a parser which uses Converges default tokenizer, adds new keywords (`state`, `transition` etc.) and using a specified top-level rule `system`.

3.2 Creating a parse tree

The DSL implementation function is passed a DSL block string which it should parse against the DSL's grammar. Since DSL implementation functions tend to follow the same form, Converge provides a convenience function `CEI::dsl_parse` which performs parsing in one single step. The state machine DSL implementation function and an example DSL block look as follows:

```
func sm(dsl_block, src_infos):
  parse_tree := CEI::dsl_parse(dsl_block, src_infos, ["state", \
   "transition", "and", "or", "from", "to"], [], GRAMMAR, "system")
  return SM_Translator.new().generate(parse_tree)

Door := $<<sm>>:
  state Opened
  state Closed

  transition from Opened to Closed: close
  transition from Closed to Opened: open
```

The `CEI` (Compiler External Interface) `dsl_parse` convenience function takes a DSL block, a list of src infos, a list of extra keywords above and beyond Converge's standard keywords, a list of extra symbols, a grammar, and the name of the grammar's start rule. It returns a parse tree (that is, a tree still containing tokens). Parse trees are Converge lists, with the first element in the list representing the production name, and the resulting elements being either tokens or lists representing the production. Tokens have two slots of particular interest: `type` contains the tokens type (e.g. `ID`); `value` contains the particular value of the token (e.g. `foo`). A subset of the parse tree for the above DSL block is as follows:

```
["system", ["element", ["state", <STATE state>, <ID Opened>]],
<NEWLINE>, ["element", ["state", <STATE state>, <ID Closed>]], ... ]
```

3.3 Translating the parse tree to an AST

The second, final, and most complex action a DSL author must take is to translate the parse tree into a Converge AST, using quasi-quotes and so on. Converge provides a simple framework for this translation, where a translation class (`SM_Translator` in the above DSL implementation function) contains a function `_t_`*production name* for each production in the grammar. For the simple state machine DSL, we wish to translate the parse tree into an anonymous class which can be instantiated to produce a running state machine, which can then receive and act upon events. The starting state is taken to be the first state in the DSL block. Transitions may have an event attached to them or not; if they have no event, they are unconditionally, and non-deterministically, taken. A slightly elided version of the translation is as follows:

```
1  class SM_Translator(Traverser::Strict_Traverser):
2    func _t_system(self, node):
3      sts := [all translated states]
```

```
4       tns := [all translated transitions]
5
6       return [|
7         class:
8           states := ${CEI::ilist(sts)}
9           transitions := ${CEI::ilist(tns)}
10
11          func init(self):
12            self.state := ${sts[0]}
13            while self.transition("")
14
15          func event(self, e):
16            if not self.transition(e):
17              raise Exceptions::User_Exception.new(Strings::format( \
18                "Error: No valid transition from state."))
19            while self.transition("")
20
21          func transition(self, e):
22            for tn := self.transitions.iter():
23              if tn.from == self.state & tn.event == e:
24                Sys::println("Event ", e, \
25                  " causes transition to state ", tn.to)
26                self.state := tn.to
27                break
28            exhausted:
29              return fail
30      |]
31
32    func _t_element(self, node):
33      return self._preorder(node[1])
34
35    func _t_state(self, node):
36      // state ::= "STATE" "ID"
37      return CEI::istring(node[2].value)
38
39    func _t_transition(self, node):
40      // transition ::= "TRANSITION" "FROM" "ID" "TO" "ID" event
41      return [| Transition.new(${CEI::istring(node[3].value)}, \
42        ${CEI::istring(node[5].value)}, ${self._preorder(node[-1])}) |]
43
44    func _t_event(self, node):
45      // event ::= ":" "ID"
46      //         |
47      if node.len() == 1:
48        return [| "" |]
49      else:
50        return CEI::istring(node[2].value)
51
52  class Transition:
53    func init(self, from, to, event):
54      self.from := from
55      self.to := to
56      self.event := event
```

At a high level, this translation is relatively simple: states are transformed into strings; transitions are transformed into instantiations of the `Transition` class. The resulting anonymous class thus knows the valid states and transitions of the state machine, and given an event can transition to the correct state, or

report errors. Certain low-level details require more explanation. The calls to `self._preorder` reference a method which, given a node in a parse tree, calls the appropriate `_t_` function. The CEI module defines functions for every Converge AST type allowing them to be created manually when quasi-quotes do not suffice. For example a call such as `CEI::istring("foo")` (e.g. lines 37) returns an AST string whose content is 'foo'. The reference to the `Transition` class in line 41 is possible since quasi-quotes can refer to top-level module definitions, as these inherently cross compile-time staging boundaries.

3.4 Using the DSL

Given the complete, if simplistic, definition of state machine DSL we now have, it is possible to instantiate a state machine and fire test events at it:

```
door := Door.new()
door.event("close")
door.event("open")
```

which results in the following output:

```
Event close causes transition to state Closed
Event open causes transition to state Opened
```

As this section has shown, we have been able to create a functioning DSL with its own syntax, whose complete definition is less than 75 lines of code. I assert that a corresponding implementation of this DSL as a stand-alone application would be considerably larger than this, having to deal with external parsing systems, IO, error messages and other boiler-plate aspects which are largely invisible in the Converge DSL implementation approach.

4 Extending the case study

As noted in [3, 2], DSLs tend to start small but grow rapidly when users find them useful, and desire more power. In a traditional stand alone implementation, such changes might be difficult to integrate. In this section I show how we can easily extend the Converge DSL implementation.

4.1 An extended state machine

As an example of a more complex type of state machine, we define a state machine of a vending machine which dispenses drinks and sweets:

```
drinks := 10
sweets := 20

state Waiting
state Vend_Drink
state Vend_Sweet
state Empty
```

```
transition from Waiting to Vend_Drink: Vend_Drink \
 [ drinks > 0 ] / drinks := drinks - 1
transition from Vend_Drink to Waiting: Vended [drinks > 0 or sweets > 0]

transition from Waiting to Vend_Sweet: Vend_Sweet \
 [ sweets > 0 ] / sweets := sweets - 1
transition from Vend_Sweet to Waiting: Vended [sweets > 0 or drinks > 0]

transition from Vend_Sweet to Empty: Vended [drinks == 0 and sweets == 0]
transition from Vend_Drink to Empty: Vended [drinks == 0 and sweets == 0]
```

This state machine makes use of variables, guards, and actions. Variables can hold integers or strings, and must be assigned an initial value. Guards such as `[drinks > 0]` are additional constraints to events; they must hold in order for a transition to be taken. Actions such as `sweets := sweets - 1` are executed once a transitions constraints have been evaluated and the transition has been taken.

4.2 Extending the grammar

As before, the DSL author's first action is to define – or in this case, to extend – the grammar of his DSL. An elided extension to the previous grammar is as follows:

```
element    ::= state
             | transition
             | var_def
transition ::= "TRANSITION" "FROM" "ID" "TO" "ID" event guard action
var_def    ::= "ID" ":=" const
guard      ::= "[" B "]"
             |
action     ::= "/" C
            ::=
B          ::= B "AND" B %precedence 5
             | B "OR" B  %precedence 5
             | B "==" B  %precedence 10
             | B ">" B   %precedence 10
             | E
C          ::= A ( ";" A )*
A          ::= "ID" ":=" E
             | E
E          ::= E "+" E
             | E "-" E
             | var_lookup
             | const
```

The `%precedence` markings signify to the Earley parser which of several alternatives is to be preferred in the event of an ambiguous parse, with higher precedence values having greater priority. Essentially the extended grammar implements a syntax in a form familiar to many state machine users. Guards are conditions or expressions; actions are sequences of assignments or expressions; and expressions include standard operators.

4.3 Extending the translation

The first thing to note is that the vast majority of the translation of section 3.3 can be used unchanged in our evolved DSL. The anonymous state machine class gains a `vars` slot which records all variable names and their current values, and `get_var` / `set_var` functions to read and update `vars`. Transitions gain `guard` and `action` slots which are functions. `transition` is then updated to call these functions, passing the state machine to them, so that it can read and write variables. The updated `transition` function is as follows:

```
func transition(self, e):
  for tn := self.transitions.iter():
    if tn.from == self.state & tn.event == e & tn.guard(self):
      Sys::println("Event ", e, " causes transition to state ", tn.to)
      self.state := tn.to
      tn.action(self)
      break
  exhausted:
    return fail
```

The remaining updates to the translation are purely to translate the new productions in the grammar. The full translation is less than 200 lines of code, but in the interests of brevity I show a representative subset; the translation of guards, and the translation of variable lookups and constants.

```
1  func _t_guard(self, node):
2    // guard ::= "[" B "]"
3    //         |
4    if node.len() == 1:
5      guard := [| 1 |]
6    else:
7      guard := self._preorder(node[2])
8    return [|
9      func (&sm):
10       return ${guard}
11   |]
12
13 func _t_B(self, node):
14   // B ::= B "AND" B
15   //     | B "OR" B
16   //     | B "==" B
17   //     | B ">" B
18   //     | E
19   if node.len() == 4:
20     lhs := self._preorder(node[1])
21     rhs := self._preorder(node[3])
22     ndif node[2].type == "AND":
23       return [| ${lhs} & ${rhs} |]
24     elif node[2].type == "OR":
25       return [| ${lhs} | ${rhs} |]
26     elif node[2].type == "==":
27       return [| ${lhs} == ${rhs} |]
28     elif node[2].type == ">":
29       return [| ${lhs} > ${rhs} |]
30   else:
31     return self._preorder(node[1])
```

```
32
33  func _t_const(self, node):
34    // const ::= "INT"
35    //         | "STRING"
36    ndif node[1].type == "INT":
37      return CEI::iint(Builtins::Int.new(node[1].value))
38    elif node[1].type == "STRING":
39      return CEI::istring(node[1].value)
40
41  func _t_var_lookup(self, node):
42    // var_lookup ::= "ID"
43    return [| &sm.get_var(${CEI::istring(node[1].value)}) |]
```

The majority of this translation is simple, and largely mechanical. Guards are turned into functions (lines 8–11) which take a single argument (a state machine) and return true or false. An empty guard always evaluates to 1 (line 5), which can be read as 'true'. The translation of guards (lines 20–29) is interesting, as it shows that syntactically distinct DSLs often have a very simple translation into a Converge AST, as Converge's expression language is unusually rich in expressive power by imperative programming language standards (including features such as backtracking which we do not use in this paper). State machine constants (strings and integers) are directly transformed into their Converge equivalents (lines 36–39).

4.4 Communication between AST fragments

One subtle aspect of the translation deserves special explanation, which are the two `&sm` variables (lines 43 and 9). These relate to the fact that the state machine which is passed by the `transition` function to the generated guard function (lines 8-11) needs to be used by the variable lookup translation (line 43). The effect we wish to achieve is that the translated guard function looks approximately as follows:

```
func (sm):
  return sm.get_var("x") < 1
```

By default, Converge's quasi-quote scheme generates *hygienic* ASTs. The concept of hygiene is defined in [14], 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`. This is 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' [14].

Converge's quasi-quote scheme therefore preemptively α-renames variables to a *fresh name* – guaranteed by the compiler to be unique – thus ensuring that unintended variable capture can not happen. While this is generally the required behaviour, it can cause practical problems when one is building up an AST in fragments, as we are doing in our state machine translation. In normal programming, one of the most common way for local chunks of code to interact is via variables; however, hygiene effectively means that variables are invisible between different code chunks. Thus, by default, there is no easy way for the variable passed to the generated guard function (lines 8-11) to be used by the variable lookup translation (line 43).

The traditional meta-programming solution to this problem is to manually generate a fresh name, which must then be manually passed to all translation functions which need it. A sketch of a solution for Converge would be as follows:

```
func _t_guard(self, node):
  // guard ::= "[" B "]"
  //         |
  sm_var_name := CEI::fresh_name()
  if node.len() == 1:
    guard := [| 1 |]
  else:
    guard := self._preorder(node[2], sm_var_name)
  return [|
    func (${CEI::iparam(CEI::ivar(sm_var_name))}):
      return ${guard}
  |]

func _t_var_lookup(self, node, sm_var_name):
  // var_lookup ::= "ID"
  return [| ${CEI::ivar(sm_var_name)}.get_var( \
   ${CEI::istring(node[1].value)}) |]
```

This idiom, while common in other approaches, is intricate and verbose. Indeed, the quantity and spread of the required boilerplate code can often overwhelm the fundamentals of the translation.

Converge therefore provides a way to switch off hygiene in quasi-quotes; variables which are prefixed by `&` are not α-renamed. Thus the `sm` variable in line 43 dynamically captures the `sm` variable defined in line 9, neatly obtaining the effect we desire. This is a very common translation idiom in Converge, and is entirely safe in this translation[1].

5 Evolving a robust DSL

In the previous section, I showed how a Converge DSL can easily evolve in expressive power. The DSL defined previously suffers in practice from one fun-

[1] Although I do not show it in this paper, translations which integrate arbitrary user code can cause this idiom to become unsafe; Converge provides a relatively simple work around for such cases.

damental flaw. When used correctly, it works well; when used incorrectly, it is difficult to understand what went wrong. This is a common theme in DSLs: initial versions with limited functionality are used only by knowledgeable users; as the DSLs grow in power, they are used by increasingly less knowledgeable users. This has a dual impact: the more powerful the DSL it is, the more difficult it is to interpret errors; and the less knowledgeable the user, the less their ability to understand whether they caused the error, if so, how to fix it.

In this section, I show how Converge DSLs can easily add debugging support which makes the use of complex DSLs practical.

5.1 DSL errors

Returning to the vending machine example of section 4.1, let us change the guard on the first transition from `drinks > 0` to `drinks > "foo"`. The vending machine state machine is contained in a file `ex.cv` and the state machine DSL definition in `SM.cv`. As we might expect, this change causes a run-time exception, as Converge does not define a size comparison between integers and strings. The inevitable run-time exception and traceback look as follows:

```
Traceback (most recent call at bottom):
  1: File "ex2.cv", line 29, column 22
  2: File "SM.cv", line 118, column 27
  3: File "SM.cv", line 124, column 57
  4: File "SM.cv", line 235, column 21
  5: (internal), in Int.>
Type_Exception: Expected arg 2 to be conformant to Number but got
instance of String.
```

This traceback gives very little clue as to where in the DSL the error occurred. Looking at line 235 of `SM.cv` merely pinpoints the `_t_B` translation function. Since we know in this case that comparing the size of a number and a string is invalid, we can rule out the translation itself being incorrect. The fundamental problem then becomes that errors are not reported in terms of the users input. Knowing that the error is related to an AST generated from the `_t_B` translation function is of limited use as several of the vending machines guards also involve the greater than comparison.

5.2 Src infos

DSL implementation functions take two arguments: a string representing the DSL block and a list of src infos. A src info is a (src path, char offset) pair which records a relationship with a character offset in a source file. The Converge tokenizer associates a src info with every token; the parse tree to AST conversion carries over the relevant src infos; and the bytecode compiler associates every bytecode instruction with the appropriate src infos. As this suggests, the src info concept is used uniformly throughout the Converge parser, compiler, and VM.

From this papers perspective, an important aspect of src infos is that tokens, AST elements, and bytecode instructions can be associated with more than one

src info. Converge provides a simple mechanism for augmenting the src infos that quasi-quoted code is associated with. Quasi-quotes have an extended form `[<e>| ... |]` where *e* is an expression which must evaluate to a list of src infos. As we shall see, a standard idiom is to read src infos straight from tokens (via its `src_infos` slot) into the extended quasi-quotes form.

5.3 Augmenting quasi-quoted code

Using the extended form of quasi-quotes, we can easily augment the translation of section 4.3 to the following:

```
func _t_B(self, node):
  // B ::= B "<" B
  ...
  elif node[2].type == ">":
    return [<node[2].src_infos>| ${lhs} > ${rhs} |]
  ...
```

In other words, we augment the quasi-quoted code with src infos directly relating it to the specific location in the user's DSL input where the size comparison was made. When we re-compile and re-run the altered vending machine DSL, we get the following backtrace:

```
Traceback (most recent call at bottom):
  1: File "ex2.cv", line 29, column 22
  2: File "SM.cv", line 118, column 27
  3: File "SM.cv", line 124, column 57
  4: File "SM.cv", line 235, column 21
     File "ex2.cv", line 15, column 74
  5: (internal), in Int.>
Type_Exception: Expected arg 2 to be conformant to Number but got
instance of String.
```

The way to read this is that the fourth entry in the backtrace is related to two source locations: one is the quasi-quoted code itself (in `SM.cv`) and the other is a location within the vending machine DSL (in `ex2.cv`). This allows the user to pinpoint precisely where within their DSL input the error occurred, which will then allow them – one hopes – to easily rectify it. This is a vital practical aid, making DSL debugging feasible where it was previously extremely challenging. Because of the extended form of quasi-quotes, augmenting a DSL translation with code to record such information is generally a simple mechanical exercise.

5.4 Statically detected errors

Src infos are not only useful for aiding run-time errors. Converge also uses the same concept to allow DSL implementations to report errors during the translation of the DSL. For example, as our state machine DSL requires the up-front declaration of all variables to be used within it, we can easily detect references to undefined variables. The `CEI::error` function takes an arbitrary error message, and a list of src infos and reports an error to the user. Given that the translation class defines a slot `var_names` which is a set of known variables, we can then alter the `_t_var_lookup` function as follows:

```
func _t_var_lookup(self, node):
  // var_lookup ::= "ID"
  if not self.var_names.find(node[1].value):
    CEI::error(Strings::format("Unknown state-machine variable '%s'.", \
     node[1].value), node[1].src_infos)
  return [<node[1].src_infos>| &sm.get_var( \
   ${CEI::istring(node[1].value)}) |]
```

When an unknown variable is encountered, an error message such as the following is printed, and compilation halts:

```
Error: Line 53, column 66: Unknown state-machine variable 'amuont'.
```

6 Related work

Several approaches have been suggested for DSL implementation. Hudak presented the notion of Domain Specific Embedded Languages (DSELs) [2] where DSLs are implemented using a languages normal features. 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. DSLs implemented in Converge have considerably more syntactic flexibility.

TXL [15] and ASF+SDF are similar, generic source to source transformation languages [16]. Both are are mature and efficient; TXL has been used to process billions of lines of code [15]. Furthermore such approaches are inherently flexible as they can be used with arbitrary source and target languages; unlike Converge, they can embed DSLs into any host language. However this flexibility means that they have little knowledge of the host language's semantics beyond the simple structure recorded in parse trees. This makes safe embeddings hard to create, whereas Converge based systems can use the compilers inherent knowledge of the host language to avoid such issues.

MetaBorg uses a combination of tools to allow language grammars to be extended in an arbitrary fashion using a rule rewriting system [4]. Although MetaBorg by default operates on parse trees in the same way as TXL, it 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. Although MetaBorg is in theory capable of defining any embedding, its authors deliberately narrow their vision for MetaBorg to a 'method for promoting APIs to the language level.' This is a sensible restriction since DSLs that result from promoting a particular API to the language level will tend to shadow that API; therefore instances of the DSL will generally translate fairly directly into API calls which limits the potential for safety violations. In contrast, Converge provides coarser-grained support for implementing larger DSLs.

Macro systems have long been used to implement DSLs. Lisp was the first language with a macro system, and although its syntax is inherently flexible, it is

not possible to change it in a completely arbitrary fashion as Converge allows – Lisp DSLs 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 [17]. Therefore despite Lisp's success in this area, for many years more modern systems struggled to successfully integrate similar features [6]. More recently languages such as Template Haskell [12] (which is effectively a refinement of the ideas in MetaML [18]; see [6] for a more detailed comparison of these languages with Converge) have shown how sophisticated compile-time meta-programming systems can be implemented in a modern language. However such languages still share Lisp's inability to extend the languages syntax.

Nemerle uses its macro system to augment the compilers grammar as compilation is in progress [19]. However only relatively simple, local additions to the syntax are possible and the grammar extensions must be pre-defined; it is not intended, or suitable, for implementing complex DSLs. In comparison to Nemerle, MetaLua allows more flexible additions to the grammar being compiled but has no support for e.g. hygiene as in Converge, which makes implementing large DSLs problematic [20].

7 Conclusions

In this paper, I showed a case study of a state machine DSL evolving in terms of functionality and robustness. There are many other areas in which the DSL could evolve, including showing how DSL code can naturally interact with 'normal' Converge code. However I hope this papers' case study gives a clear indication as to how a dedicated DSL implementation approach can make DSL evolution – in whatever form it takes – practical.

I am grateful to the anonymous referees whose comments have helped to improve this paper. Any remaining mistakes are my own.

This research was partly funded by Tata Consultancy Services.

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

References

1. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. Technical report, Centrum voor Wiskundeen Informatica (December 2003)
2. Hudak, P.: Modular domain specific languages and tools. In: Proceedings of Fifth International Conference on Software Reuse. (June 1998) 134–142
3. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. Volume 35 of SIGPLAN Notices. (June 2000) 26–36

4. Bravenboer, M., Visser, E.: Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Schmidt, D.C., ed.: Proc. OOPSLA'04, Vancouver, Canada, ACM SIGPLAN (October 2004)
5. Clark, T., Evans, A., Sammut, P., Willans, J.: An executable metamodelling facility for domain specific language design. In: Proc. 4th OOPSLA Workshop on Domain-Specific Modeling. (October 2004)
6. Tratt, L.: Compile-time meta-programming in a dynamically typed OO language. In: Proceedings Dynamic Languages Symposium. (October 2005) 49–64
7. Sheard, T., el Abidine Benaissa, Z., Pasalic, E.: DSL implementation using staging and monads. In: Proc. 2nd conference on Domain Specific Languages. Volume 35 of SIGPLAN., ACM (October 1999) 81–94
8. Czarnecki, K., O'Donnell, J., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. **3016** (2004) 50–71
9. Seefried, S., Chakravarty, M., Keller, G.: Optimising Embedded DSLs using Template Haskell. In: Third International Conference on Generative Programming and Component Engineering, Vancouver, Canada, Springer-Verlag (October 2004) 186–205
10. Tratt, L.: Converge Reference Manual. (July 2007) `http://www.convergepl.org/documentation/` Accessed Aug 16 2007.
11. van Rossum, G.: Python 2.3 reference manual (2003) `http://www.python.org/doc/2.3/ref/ref.html` Accessed Aug 31 2005.
12. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Proceedings of the Haskell workshop 2002, ACM (2002)
13. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM **13**(2) (February 1970)
14. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: Symposium on Lisp and Functional Programming, ACM (1986) 151–161
15. Cordy, J.R.: TXL - a language for programming language tools and applications. In: Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications. (April 2004)
16. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the asf+sdf compiler. Volume 24., New York, NY, USA, ACM Press (2002) 334–368
17. Bachrach, J., Playford, K.: D-expressions: Lisp power, Dylan style (1999) `http://www.ai.mit.edu/people/jrb/Projects/dexprs.pdf` Accessed Nov 22 2006.
18. Sheard, T.: Using MetaML: A staged programming language. (September 1998) 207–239
19. Skalski, K., Moskal, M., Olszta, P.: Meta-programming in Nemerle (2004) `http://nemerle.org/metaprogramming.pdf` Accessed Nov 5 2007.
20. Fleutot, F., Tratt, L.: Contrasting compile-time meta-programming in metalua and converge. In: Workshop on Dynamic Languages and Applications. (July 2007)