Are there Domain Specific Languages?

Conference Paper · March 2016

DOI: 10.1145/2889420.2892271

Greg Michaelson
Heriot-Watt University
185 PUBLICATIONS **1,144** CITATIONS
SEE PROFILE

Are there Domain Specific Languages?

Greg Michaelson
School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton, Scotland
G.Michaelson@hw.ac.uk

ABSTRACT

Turing complete languages can express unbounded computations over unbounded structures, either directly or by a suitable encoding. In contrast, Domain Specific Languages (DSLs) are intended to simplify the expression of computations over structures in restricted contexts. However, such simplification often proves irksome, especially for constructing more elaborate programs where the domain, though central, is one of many considerations. Thus, it is often tempting to extend a DSL with more general abstractions, typically to encompass common programming tropes, typically from favourite languages. The question then arises: once a DSL becomes Turing complete, then in what sense is it still domain specific?

Keywords

Domain Specific Languages; expressiveness

1. INTRODUCTION

Domain Specific Languages (DSLs) are notations oriented to specific problem domain with specialised types and control structures.

Constructing DSLs is motivated naturally by programming practice. It can become tiresome writing lots of small programs for the same problem area; typically, one ends up using the same set type and control abstractions, configured by standard programming tropes. Thus, it is not uncommon to construct command based framework, with scripts to invoke and configure individual program components. Here, the allowable structure and behaviour of scripts constitutes a first DSL.

It is tempting to add ad-hoc extensions to a command style DSL, for example to parameterise commands with flags or argument, but this can quickly become an unwieldy assemblage of special cases. Instead, it may be more fruitful to systematically elaborate some consistent DSL notation, oriented to the problem area, with built in constructs capturing specialised abstractions.

RWDSL '16, March 12 2016, Barcelona, Spain
© 2016 ACM. ISBN 978-1-4503-4051-9/16/03...$15.00
DOI: http://dx.doi.org/10.1145/2889420.2892271

However, the more we use such a restricted DSL, the more we tend to want familiar general purpose programming language abstractions as well. Thus, we may be seek to add, say, arithmetic and logic, sequences, selections, iteration, sub programs and data structures. Thus, as the DSL grows, it tends to become less and less domain specific, and also more and more like some favourite language.

Historically, high level languages were developed as abstractions from machine codes. However, their designers were often motivated in their choice of abstractions by the need to solve problems in some specific application domain.

This is strongly reflected in Sammet's pioneering survey of programming languages from 1969[6]. Here, Sammet distinguishes *problem oriented* languages, that is "any language which is easier for writing solutions to a particular problem than assembler", from *application-oriented* languages, which "have facilities and/or notations which are useful primarily for a single application area". To illustrates this, Sammet contrasts APT for controlling machine tools and COGO for civil engineering, with the less application-oriented FORTRAN for numerical mathematics and COBOL for for business data processing. Thus, she notes that "the term [application-oriented] is somewhat relative" and "The wider the application area, the more general the language must be."

More recently, the taxonomy in Baron's 1986 popularisation[1] includes *function*, distinguishing, amongst others, the languages:

- ALGOL - general numerical analysis in scientific computing (p106);
- BASIC - general purpose though especially popular in lower education; (p144)
- C - systems programming (p156);
- COBOL - business (p170);
- FORTRAN - science and engineering (p203);
- LISP - artificial intelligence (p223);

Despite these languages originating in the desire to solve domain specific problems, in fact they are all *general purpose languages*. Formally, they are all *Turing complete (TC)*.

A language is TC if it can be shown to be equivalent in expressive power to some formal model of computability, originally Turing machines or λ calculus[5]. Thus, a TC language is not, on the face of it, domain specific, in the sense of being *restricted* to some domain.

2. WHAT IS DOMAIN SPECIFICITY?

To explore domain specificity further, let's consider the very simple Light Bulb language (LBL) to control a single bulb. The switch can be in the ON or OFF states. The command `SWITCH` changes the ON state to OFF and vice versa:

program -> *switch*
switch -> `SWITCH` | ϵ
m_1 [`SWITCH`] ON = OFF
m_1 [`SWITCH`] OFF = ON
m_1 [ϵ] ON = ON
m_1 [ϵ] OFF = OFF

Let us now consider the Linear Light Bulb Language (LLBL) which can control a row of light bulbs:

program -> *row* | *row program*
row -> *switch* | *switch row*
$s_i \in row$
$b_i \in \{$`ON`,`OFF`$\}$
$m_2\ [s_1,...,s_n]\ \{b_1,...,b_N\} =$
$\{m_1\ s_1\ b_1,...,m_1\ s_N\ b_N\}$

Finally, let us consider the Grid Light Bulbs Language (GLBL) which can control a grid of light bulbs:

program -> *grid*
grid -> *row* | *row grid*
$r_i \in row$
$br_i \in b*$
$m_3\ [r_1,...,r_N]\ \{br_1 1,...,br_N\} =$
$\{m_2\ r_1\ br_1,...,m_2\ r_1\ br_N\}$

Now is GLBL a language of turning on and off grids of light bulbs, or negating black and white images, or manipulating anything representable as an array of booleans? That is, how domain specific is GLBL? Indeed, how domain specific is any language which can represent operations on values in multiple domains.

2.1 Embedded DSLs

A common approach is to implementing a DSL is to *embed* it in some extant host language. We will illustrate this by considering the SUCC language, for constructing integers by incrementation from zero:

e -> `z` | `s` e
m `z` = 0
m `s` e = 1+$m\ e$

We will implement SUCC by embedding it in Haskell.

A first stage is to add a *library* called through an API to the host language, so arbitrary host language constructs may be used to configure API calls:

```
ssucc n = n+1;

... ssucc(ssucc (ssucc 0)))...
```

This does not constitute a distinct DSL.

A second stage is to add *abstract syntax* and then *interpret abstract syntax trees (ASTs)* through calls to *interpreter* components from arbitrary host language constructs:

```
data Succ = Z | S Succ

eval Z = 0
eval (S s) = ssucc (eval s)

... eval (S(S(S Z)))...
```

A third stage is to design a *concrete syntax*:

program -> `Z` | `S` *program*

and build a *compiler* from concrete syntax to ASTs. The interpreter is then called with parsed strings:

```
data Lex = LZ | LS

llex [] = []
llex (' ':t) = llex t
llex ('Z':t) = LZ:llex t
llex ('S':t) = LS:llex t

parse [LZ] = (Z,[])
parse (LS:t) =
 let (e,r) = parse t
 in (S e,r)
parse l = error (showLs l)

comp s =
 let (e,r) = parse(llex s)
 in e

... eval (comp "S S S Z")...
```

A fourth stage is to extend the host language syntax, with a pre-processor to generate API calls.

Finally, the host language semantics may be extended: a true extension.

If the parser/interpreter for the DSL are exposed only as stand alone language processors, DSL programs can only be constructed using the domain specific syntax. Then we have *implemented* the DSL in the host language.

3. EXPRESSIVENESS

It is often alleged that DSLs have greater *expressiveness* than general purpose languages, that is they can express:

- the same things as other languages, but more succinctly, or
- things other languages cannot express.

Thus, for DSLs, expressiveness is a comparative quality.

Felleisen's account of expressiveness[3] depends on the notion of *extending* a language. Suppose language X has constructors which are not in language Y. Then, language X is a *conservative extension* of language Y if instances of X can be translated into instances of Y without changing the semantics of Y. Felleisen terms this *weak expressibility*.

For example, Glasgow Parallel Haskell (GpH)[4] extends Haskell with the `seq` and `par` operators, indicating a desire for sequential or parallel activity respectively. This is a conservative extension as simple elimination of the operators preserves the meanings of Haskell programs.

In contrast, if constructors in X cannot be eliminated in translation to Y then X has semantic properties Y lacks, and X is said to *extend* Y. For example, Scheme[7] extends λ calculus with the `set!` assignment operator to enable stateful programming. `set!` cannot be expressed in pure λ calculus.

All TC languages capture a common notion of algorithm, that is *effectively calculable* in Church's terminology or *computable* in Turings. Thus, in Fellesien's terms, TC languages with different semantics are mutually extending, so all TC languages have the same expressiveness.

Designers of DSLs may think they have enabled something that other languages cannot do so well. But every new TC language is equivalent to some old TC language plus some syntax and a library. So is every programming language really an embedded DSL with a TC host?

Felleisen suggests that comparisons between languages might be made in a common language universe. We might then establish that some TC languages can express some algorithms more succinctly than other TC languages. But questions then arise as to how to choose a language universe and whether or not language universes have language biases.

4. HOST LANGUAGE AND EDSL

Now, suppose that, rather than following the discipline of only programming in the EDSL (other than calling the interpreter and compiler, and, say I/O), EDSL programs may include arbitrary use of host language constructs, for example:

```
makeS 0 = "Z"
makeS n = "S"++(makeS (n-1))

...eval (comp(makeS (123*456))...
```

Now, it is not clear for practical purposes where the host language stops and the EDSL stops.

Furthermore, the EDSL now constitutes a conservative extension of the host language, with the same properties as the host language. In particular, if the host language is TC then so is the extended host language.

5. LANGUAGE AND PROGRAM

A programming language is formally defined by the association of semantics with syntactic constructs. Typically, the semantics may be characterised as a mapping from some initial state to some final state guided by the structure of a program instance:

semantics: *program* * *state* -> *state*

Now, a program may be characterised as a mapping from some intial state and an input, to some final state and an output:

program: *input* * *state* -> *output* * *state*

If we treat the output as part of the final state:

program : *input* * *state* -> *state*

That is, the program changes initial state to final state depending on input structure.

We could define the input structure with a syntax and then view the program as the semantics of its inputs.

So, does every program define a DSL?

6. CONCLUSIONS: DSLS ARE ABOUT PRAGMATICS

Felleisen asks:

> "...what advantages there are to programming in the more expressive language when equivalent programs in the simpler language already exist."

He observes that:

> "...programs in less expressive languages exhibit repeated occurrences of programming patterns and this pattern oriented style is detrimental to the programming process."

and elaborates his *Conciseness conjecture* that:

> "Programs in more expressive languages that use the additional facilities in a sensible manner contain fewer programming patterns than equivalent programs in less expressive languages."

That is, more expressive languages result in more *succinct* programs[2].

We may conclude that DSL abstractions and constructs are chosen pragmatically, to make it easier to express particular things. Thus, what may be complex in an arbitrary TC language may become simpler in a DSL. Nonetheless, while the domain may frame the choice of DSL abstractions and constructs, those from one domain may well be appropriate for other domains.

7. REFERENCES

[1] N. S. Baron. *Computer Languages: a guide for the perplexed.* Penguin, 1986.

[2] J. Davidson. *An Information Theoretic Approach to the Expressiveness of Programming Languages.* PhD thesis, University of Glasgow, 2016.

[3] M. Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming*, pages 134–151, 1990.

[4] Kevin Hammond. Glasgow Parallel Haskell (GpH). In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 768–779. Springer US, Boston, MA, 2011.

[5] L. MacKenzie P. Cockshott and G. Michaelson. *Computation and its Limits.* OUP, 2012.

[6] J. Sammet. *Programming Languages: Historyand Fundamentasl.* Prentice-Hall, 1969.

[7] Gerald Jay Sussman and Guy L Steele Jr. Scheme: An interpreter for extended lambda calculus. In *MEMO 349, MIT AI LAB*, 1975.

8. ACKNOWLEDGMENTS

This work was supported by EPSRC EP/K009931/1 'Programmable Embedded Platforms for Remote and Compute Intensive Image Process'.