Building Domain-Specific Languages over a Language Framework

(Extended Abstract)*

Roberto Ierusalimschy† Luiz Henrique de Figueiredo‡

Abstract

Lua is a language specifically designed to be tailored for specific domains. In this sense, Lua can be regarded as a *language framework*. This paper describes our experience using Lua for building different Domain-Specific Languages. These domains include bibliographic databases, user-interfaces, graphical metafiles, and finite element analysis.

1 Introduction

Many domain-specific languages can be classified as *description*, or *configuration*, languages. Under this label, we mean languages without strong run-time semantics, in which the output has a strong resemblance—sometimes an isomorphism—with the "program" tree itself. Well known examples of such languages are:

- the language used by BibTEX to specify bibliographic references [6];
- UIL (User Interface Language), used to describe graphical user interfaces in Motif [4];
- VRML (Virtual Reality Modeling Language), used to describe three-dimensional objects and views [1].

Although such languages are used mainly to describe structures and values, they usually provide some weak abstraction (i.e., programming) facilities. For instance, BibTEX allows the naming of strings, and UIL supports simple arithmetic calculations. However, if a language is at all successful, users will demand more power as soon as they learn the basics. For instance, a BibTEX user may want to create an abstraction to declare in-house technical reports, or to automate the creation of keys. To fulfill those needs, description languages should provide full programming facilities [8]. However, because they are so small, such languages are usually built from scratch, and their designers have little incentive to create a true programming language.

In this paper, we describe how a single, generic configuration language is being used to create many different domain-specific languages. By adopting this language framework, designers of domain-specific languages do not have to deal with lexical and syntactical issues, neither with common functionality, like control structures, function definitions, etc. Such issues are essential for implementing a language, but are time-consuming and really peripheral to the task. With a powerful language framework at hand, such designers can instead concentrate on the semantics of their particular domain.

*Submitted to the 1997 ACM SIGPLAN Workshop on Domain-Specific Languages. Last revision on October 11, 1996.

†Departamento de Informática, PUC-Rio, Rua Marquês de São Vicente 225, 22453-900 Rio de Janeiro, RJ, Brazil. (roberto@inf.puc-rio.br)

‡Laboratório Nacional de Computação Científica, Rua Lauro Müller 455, 22290-160 Rio de Janeiro, RJ, Brazil. (lhf@lncc.br)

2 The Configuration Language Lua as a Framework

Lua is a general purpose configuration language that arose from the need of TeCGraf[1] to use a single extension language to customize industrial applications [5]. By design, Lua integrates powerful data-description facilities and familiar imperative constructs. On the "traditional" side, Lua provides the usual control structures (*whiles*, *ifs*, etc.), function definitions with parameters and local variables, and the like. On the less traditional side, Lua provides functions as first order values, and dynamically created associative arrays (called *tables* in Lua) as a single, unifying data-structuring mechanism. As an example of Lua code, Figure 1 shows the implementation of the `map` function in Lua; this function receives a list $a = (a_1, a_2, \ldots, a_n)$ and a function f, and returns a new list $b = (f(a_1), f(a_2), \ldots, f(a_n))$.

```
function map (a, f)
  local b = {}  -- create a new table
  local i = 1
  while a[i] do
    b[i] = f(a[i])
    i = i+1
  end
  return b
end
```

Figure 1: An example of Lua code.

What makes Lua specially interesting for implementing description languages is its *constructor* mechanism. Syntactically, constructors are very similar to those passive BibTeX entries. For instance, the following is a typical piece of Lua code for representing a bibliographic database:

```
book{ author = "Charles Dickens",
      title = "David Copperfield",
      year = 1849
}
```

From a syntactic point of view, this looks indeed almost exactly like a BibTeX entry. Semantically, however, it is very different: In Lua, the code above is a *statement* (more specifically, it is an expression with side-effects), and its semantics are roughly equivalent to

```
temp = {}  -- new table
temp["author"] = "Charles Dickens"
temp["title"] = "David Copperfield"
temp["year"] = 1849
book(temp)  -- function call
```

In words, the basic constructor, denoted by `{...}`, creates a new table (like `new` in some languages). Inside the curly brackets, we specify how to initialize some fields of the table. Finally, the syntax

[1] TeCGraf is a research and development group at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro, Brazil, with industrial partners like the Brazilian Oil Company Research Center (CENPES/PETROBRAS).

`f{...}` is just sugar for `f({...})`. In the example above, the function `book` must be defined for that specific domain, and can output this entry, store it for future use, check fields for consistency, provide default values, etc.

Note that this is very unlike BibTEX, in which the descriptions are not "created", but only "recorded" in the file. Using Lua, different applications can declare different constructor functions (like `book` in the example above), tailoring the language for different domains.

Because constructors are expressions, they can be nested arbitrarily. For instance, in the IUP-Lua user interface system [7], the following code specifies a complete dialog box in a single step:[2]

```
w = iupdialog {
      iupbutton{label = "Ok", action = press_ok},
      iupbutton{label = "Cancel", action = press_cancel};
      title = "Overwrite existing file?",
      size = "100x50"
    }
```

Again we emphasize that, despite the declarative style, the above piece of code has a very clear imperative semantics.

In summary, the point being made here is simply that Lua provides a simple syntax for describing complex, nested (and *active*) data-structures, along with familiar constructs and operators, such as loops and string concatenation.

The main task in tailoring Lua for a new domain is to define new constructor functions, like `book` or `iupdialog` in the examples above. Such functions can be defined in Lua or in C. There is no need for a separate "named string" facility (as in BibTEX) or for providing arithmetic operators as a special case (as in UIL). Lua has variables, and expressions involving them can be used anywhere a value is required. For example, in a long bibliographic database, it is convenient to be able to use short cuts for full author names, as shown in Figure 2.

This is but one simple example of how a full programming language can be at the same time more useful and simpler than a custom-made language. Lua has been explicitly designed to cope with many different user domains, and thus is a good choice as a framework for implementing domain-specific languages.

3 Some Real Applications

Lua has been used in TeCGraf for over three years now, in many different applications. Most of these applications are commercial products, developed for industrial partners like PETROBRAS (the Brazilian Oil Company). These applications have tailored Lua for a range of specific domains, including:

- the storage and exchange of structured graphical metafiles [3] (see Figure 3);
- the description of user interfaces, as already shown above. This system, called IUP-Lua, is mainly an interface between Lua and the GUI toolkit IUP [7].
- the description of lithology profiles of oil wells, used in a visualization tool.
- generic attribute configuration for finite element meshes [2].

[NOTE: *In the final version, we intend to explain these examples better.*]

[2] The functions `iupdialog` and `iupbutton` are declared by IUP-Lua.

```
-- define shortcut for full names
name = {
Dickens = "Charles Dickens",
Doyle = "Arthur Conan Doyle",
Austen = "Jane Austen",
...
}
-- now create some entries
book{ author=name.Dickens, title="David Copperfield", year=1849 }
book{ author=name.Dickens, title="Bleak House", year=1853 }
...
book{ author=name.Doyle, title="The Hound of the Baskervilles", year=1902 }
book{ author=name.Doyle, title="The Sign of Four", year=1889 }
...
book{ author=name.Austen, title="Sense and Sensibility", year=1811 }
book{ author=name.Austen, title="Pride and Prejudice", year=1813 }
```

Figure 2: A bibliographic database in Lua.

4 Related Work

One traditional language framework for "little languages" is Lisp or one of its variants, such as Scheme [11]. Lisp-based languages are clearly very easy to parse and have built-in extensibility, but their simple syntax with its multitude of parentheses is not user-friendly. Nevertheless, many description languages have been based on Lisp, specially because it is simple to describe complex hierarchical structures in Lisp. This same strength is present in Lua, with its tables, but combined with a more familiar syntax.

Tcl [9, 10] is certainly a very successful configuration language nowadays. Although Tcl derives simplicity from being based on a single data type ("string") and a crude syntax, this also complicates user programming.

Certainly, the constructor mechanism is one of the main components for tailoring domain-specific languages based on Lua. What makes constructors specially attractive is its ability to mix constant ("quoted") and variable parts in a structure using a simple syntax. For instance, consider again a typical dialog description:

```
w = iupdialog {
      iupbutton{label = l1, action = a1},
      iupbutton{label = l2, action = a2};
      title = t1,
      size = x .. "x" .. y }
```

Notice that the whole constructor specifies a fixed structure, i.e., a dialog containing two buttons. However, for each dialog created with this structure, its button labels and actions, as well as its title and size, depend on the actual value of variables `l1`, `a1`, etc. Thus the same structure may mean different things depending on *when* the constructor is executed. This same effect can be obtained in Lisp, albeit in more obscure ways, using quasiquotes and unquotes, or with explicit calls to `cons`

```
pe = glbBox{
 matrix={1,0,0,0,1,0},
 framed=1,
 filled=1,
 frame={
         color={ 0, 0, 0 },
         style=0,
         width=1,
       },
 fill={
         color={ 0, 0, 255 },
         type=0,
         style=1,
      },
 xmin = 8.139535, xmax = 15.116279,
 ymin = 17.676768, ymax = 36.868687,
}
```

Figure 3: A piece of graphical metafile.

and `list` to build the structure.
[NOTE: *In the final version, we intend to expand this discussion.*]

5 Conclusions

Lua has been built to provide a framework for tailoring domain-specific languages. The language has been used in many different industrial applications, whose users and developers have benefited from the existence of a single, powerful and extensible extension language. Most of these applications ended up with versions of Lua that were highly specialized to their domains, ranging from user interface descriptions to graphical metafiles to attribute configuration for finite element meshes. Those applications provide strong evidence that Lua has fulfilled its design goal.

Lua has been implemented using traditional techniques: a hand-written lexer (for speed); a yacc-generated parser; a bytecode compiler and interpreter for a virtual stack machine; hash tables with linear probing and rehashing for associative arrays. The whole package has around 6000 lines of ANSI C, and includes libraries for pattern-matching, I/O, etc. The package is highly portable, and currently runs on all platforms we know, including Macs, MS-DOS, Windows, and Unix.

Lua is freely available in the internet at `http://www.inf.puc-rio.br/~roberto/lua.html`.

Acknowledgements

Waldemar Celes Filho is our partner in the design and implementation of Lua; we thank him for numerous discussions on the topics of this paper. We also thank the Brasilian Oil Company Research Center (CENPES/PETROBRAS) fot its support. The authors are partially supported by the Brazilian Council for Scientific and Technological Development (CNPq).

References

[1] Gavin Bell, Rikk Carey, and Chris Marrin. The Virtual Reality Modeling Language Specification—Version 2.0. `http://vag.vrml.org/VRML2.0/FINAL/`, August 1996. (ISO/IEC CD 14772).

[2] M. T. de Carvalho and L. F. Martha. Uma arquitetura para configuração de modeladores geométricos: aplicação a mecânica computacional. In *PANEL95 — XXI Conferência Latino Americana de Informática*, pages 123–134, 1995.

[3] W. Celes Filho, L. H. Figueiredo, and M. Gattass. Edg: Uma ferramenta para criação de interfaces gráficas interativas. In *SIBGRAPI 95*, pages 241–248, 1995.

[4] Open Software Foundation. OSF/Motif Programmer's Guide. Prentice-Hall, Inc., 1991. (ISBN 0-13-640673-4).

[5] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes Filho. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[6] L. Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, 1986.

[7] Carlos H. Levy, Luiz H. de Figueiredo, Marcelo Gattass, Carlos J. Lucena, and Don D. Cowan. IUP/LED: a portable user interface development tool. *Software: Practice and Experience*, 26(7):737–762, 1996.

[8] Bonnie A. Nardi. *A Small Matter of Programming: perspectives on end user computing.* The MIT Press, 1993.

[9] J. Ousterhout. Tcl: an embeddable command language. In *Proc. of the Winter 1990 USENIX Conference.* USENIX Association, 1990.

[10] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[11] A. Sah and J. Blow. A new architecture for the implementation of scripting languages. In *Proc. USENIX Symposium on Very High Level Languages*, 1994.