SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 0(0), 1–18 (? 19??)

Lua—an extensible extension language

ROBERTO IERUSALIMSCHY LUIZ HENRIQUE DE FIGUEIREDO WALDEMAR CELES FILHO
TeCGraf, Computer Science Department, PUC–Rio
{roberto,lhf,celes}@icad.puc-rio.br

SUMMARY

This paper describes Lua, a language for extending applications. Lua combines procedural features with powerful data description facilities, by using a simple, yet powerful, mechanism of *tables*. This mechanism implements the concepts of records, arrays, and recursive data types (pointers), and adds some object-oriented facilities, such as methods with dynamic dispatching.

Lua presents a mechanism of *fallbacks* that allows programmers to extend the semantics of the language in some unconventional ways. As a noteworthy example, fallbacks allow the user to add different kinds of inheritance to the language.

Currently, Lua is being extensively used in production for several tasks, including user configuration, general-purpose data-entry, description of user interfaces, storage of structured graphical metafiles, and generic attribute configuration for finite element meshes.

KEY WORDS Extension languages End-user programming Programming languages

INTRODUCTION

There is increasing demand for customizable applications. As applications became more complex, customization with simple parameters became impossible: users now want to make configuration decisions at execution time; users also want to write macros and scripts to increase productivity.[1-4] In response to these needs, there is an important trend nowadays to split complex systems in two parts: *kernel* and *configuration*. The kernel implements the basic classes and objects of the system, and is usually written in a compiled, statically typed language, like C or Modula-2. The configuration part, usually written in an interpreted, flexible language, connects these classes and objects to give the final shape to the application.[5]

Configuration languages come in several flavors, ranging from simple languages for selecting preferences, usually implemented as parameter lists in command lines or as variable-value pairs read from configuration files (e.g., MS-Windows' `.ini` files, X11 resource files), to *embedded languages*, for extending applications with user defined functions based on primitives provided by the applications. Embedded languages can be quite powerful, being sometimes simplified variants of mainstream programming languages such as Lisp and C. Such configuration languages are also called *extension languages*, since they allow the extension of the basic kernel semantics with new, user defined capabilities.

CCC 0038–0644/??/000001–18

Received ?
Revised 23 August 1995

What makes extension languages different from stand alone languages is that they only work *embedded* in a host client, called the *host* program. Moreover, the host program can usually provide domain-specific extensions to customize the embedded language for its own purposes, typically by providing higher level abstractions. For this, an embedded language has both a syntax for its own programs and an application program interface (API) for communicating with hosts. Unlike simpler configuration languages, which are used to supply parameter values and sequences of actions to hosts, there is a two-way communication between embedded languages and host programs.

It is important to note that the requirements on extension languages are different from those on general purpose programming languages. The main requirements for extension languages are:

- extension languages need good data description facilities, since they are frequently used as configuration languages;
- extension languages should have a clear and simple syntax, because their main users are not professional programmers;
- extension languages should be small, and have a small implementation. Otherwise, the cost of adding the library to an application may be too high;
- extension languages are not for writing large pieces of software, with hundreds of thousands lines. Therefore, mechanisms for supporting programming-in-the large, like static type checking, information hiding, and exception handling, are not essential;
- finally, extension languages should also be *extensible*. Unlike conventional languages, extension languages are used in a very high abstraction level, adequate for interfacing with users in quite diverse domains.

This paper describes Lua, an extensible procedural language with powerful data description facilities, designed to be used as a general purpose extension language. Lua arose as the fusion of two descriptive languages, designed for the configuration of two specific applications: one for scientific data entry,[6] the other for visualizing lithology profiles obtained from geological probes. When users began to demand increasingly more power in these languages, it became clear that real programming facilities were needed. Instead of upgrading and maintaining two different languages in parallel, the solution adopted was to design a single language that could be used not only for these two applications, but for any other application. Therefore, Lua incorporates facilities common to most procedural programming languages — control structures (*while*s, *if*s, etc.), assignments, subroutines, and infix operators — but abstracts out facilities specific to any particular domain. In this way, Lua can be used not only as a complete language but also as a *language framework*.

Lua satisfies the requirements listed above quite well. Its syntax and control structures are quite simple, Pascal-like. Lua is small; the whole library is around six thousand lines of ANSI C, of which almost two thousand are generated by yacc. Finally, Lua is extensible. In its design, the addition of many different features has been replaced by the creation of a few meta mechanisms that allow programmers to implement those features themselves. These meta mechanisms are: *dynamic associative arrays*, *reflexive facilities*, and *fallbacks*.

Dynamic associative arrays directly implement a multitude of data types, like ordinary arrays, records, sets, and bags. They also lever the data description power of the language, by means of *constructors*.

```
#include <stdio.h>
#include "lua.h"               /* lua header file */
#include "lualib.h"            /* extra libraries (optional) */

int main (int argc, char *argv[])
{
 char line[BUFSIZ];
 iolib_open();                /* opens I/O library (optional) */
 strlib_open();               /* opens string lib (optional) */
 mathlib_open();              /* opens math lib (optional) */
 while (gets(line) != 0)
   lua_dostring(line);
}
```

Figure 1. An interactive interpreter for Lua

Reflexive facilities allow the creation of highly polymorphic parts. Persistence and multiple name spaces are examples of features not directly present in Lua, but that can be easily implemented in Lua itself using reflexive facilities.

Finally, although Lua has a fixed syntax, fallbacks can extend the meaning of many syntactical constructions. For instance, fallbacks can be used to implement different kinds of *inheritance*, a feature not present in Lua.

AN OVERVIEW OF LUA

This section contains a brief description of the main concepts in Lua. Some examples of actual code are included, to give a flavor of the language. A complete definition of the language can be found in its reference manual.[7]

Lua is a general purpose embedded programming language designed to support procedural programming with data description facilities. Being an embedded language, Lua has no notion of a "main" program; it only works embedded in a host client. Lua is provided as a library of C functions to be linked to host applications. The host can invoke functions in the library to execute a piece of code in Lua, write and read Lua variables, and register C functions to be called by Lua code. Moreover, *fallbacks* can be specified to be called whenever Lua does not know how to proceed. In this way, Lua can be augmented to cope with rather different domains, thus creating customized programming languages sharing a single syntactical framework.[8] It is in this sense that Lua is a *language framework*. On the other hand, it is very easy to write an interactive, stand alone interpreter for Lua (Figure 1).

All statements in Lua are executed in a global environment, which keeps all global variables and functions. This environment is initialized at the beginning of the host program and persists until its end.

The unit of execution of Lua is called a *chunk*. A chunk may contain statements and function definitions. When a chunk is executed, first all its functions and statements are compiled, and the functions added to the global environment; then the statements

```
width = 420
height = width*3/2     -- ensures 3/2 aspect ratio
color = "blue"
```

Figure 2. A very simple configuration file

```
function Bound (w, h)
  if w < 20 then w = 20
  elseif w > 500 then w = 500
  end
  local minH = w*3/2             -- local variable
  if h < minH then h = minH end
  return w, h
end

width, height = Bound(420, 500)
if monochrome then color = "black" else color = "blue" end
```

Figure 3. Configuration file using functions

are executed in sequential order.

Figure 2 shows an example of how Lua can be used as a very simple configuration language. This code defines three global variables and assigns values to them. Lua is a dynamically typed language: variables do not have types; only values do. All values carry their own type. Therefore, there are no type definitions in Lua.

More powerful configurations can be written using flow control and function definitions. Lua uses a traditional Pascal-like syntax, with reserved words and explicitly terminated blocks; semicolons are optional. Such syntax is familiar, robust, and easily parsed. A small example is presented in Figure 3. Notice that functions can return multiple values, and multiple assignments can be used to collect these values. Thus, parameter passing by reference, always a source of small semantic difficulties, can be discarded from the language.

Functions in Lua are *first class* values. A function definition creates a value of type `function`, and assigns this value to a global variable (`Bound`, in Figure 3). Like any other value, function values can be stored in variables, passed as arguments to other functions and returned as results. This feature greatly simplifies the implementation of object-oriented facilities, as described later in this section.

Besides the basic types `number` (floats) and `string`, and the type `function`, Lua provides three other data types: `nil`, `userdata`, and `table`. Whenever explicit type checking is needed, the primitive function `type` may be used; it returns a string describing the type of its argument.

The type `nil` has a single value, also called `nil`, whose main property is to be different from any other value. Before the first assignment, the value of a variable is

```
list = {}                      -- creates an empty table
current = list
i = 0
while i < 10 do
  current.value = i
  current.next = {}
  current = current.next
  i = i+1
end
current.value = i
current.next = list
```

Figure 4. A circular linked list in Lua

nil. Therefore, uninitialized variables, a major source of programming errors, do not exist in Lua. Using `nil` in a context where an actual value is needed (for instance, in an arithmetic expression) results in an execution error, alerting the programmer that the variable was not properly initialized.

The type `userdata` is provided to allow arbitrary host data, represented as `void*` C pointers, to be stored in Lua variables. The only valid operations on values of this type are assignment and equality test.

Finally, the type `table` implements associative arrays, that is, arrays that can be indexed not only with integers, but with strings, reals, tables, and function values.

Associative arrays

Associative arrays are a powerful language construct; many algorithms are simplified to the point of triviality because the required data structures and algorithms for searching them are implicitly provided by the language.[9] Most typical data containers, like ordinary arrays, sets, bags, and symbol tables, can be directly implemented by tables. Tables can also simulate records by simply using field names as indices. Lua supports this representation by providing `a.name` as syntactic sugar for `a["name"]`.

Unlike other languages that implement associative arrays, such as AWK,[10] Tcl,[11] and Perl,[12] tables in Lua are not bound to a variable name; instead, they are dynamically created objects that can be manipulated much like pointers in conventional languages. The disadvantage of this choice is that a table must be explicitly created before used. The advantage is that tables can freely refer to other tables, and therefore have expressive power to model recursive data types, and to create generic graph structures, possibly with cycles. As an example, Figure 4 shows how to build circular linked lists in Lua.

Lua provides a number of interesting ways for creating a table. The simplest form is the expression `{}`, which returns a new empty table. A more descriptive way, which creates a table and initializes some fields, is shown below; the syntax is somewhat inspired in the BibTeX[13] database format:

```
window1 = {x = 200, y = 300, foreground = "blue"}
```

This command creates a table, initializes its fields x, y, and foreground, and assigns it to the variable window1. Note that tables need not be homogeneous; they can simultaneously store values of all types.

A similar syntax can be used to create lists:

```
colors = {"blue", "yellow", "red", "green", "black"}
```

This statement is equivalent to:

```
colors = {}
colors[1] = "blue";  colors[2] = "yellow"; colors[3] = "red"
colors[4] = "green"; colors[5] = "black"
```

Sometimes, more powerful construction facilities are needed. Instead of trying to provide everything, Lua provides a simple *constructor* mechanism. Constructors are written name{...}, which is just syntactic sugar for name({...}). Thus, with a constructor, a table is created, initialized, and passed as parameter to a function. This function can do whatever initialization is needed, such as (dynamic) type checking, initialization of absent fields, and auxiliary data structures update, even in the host program. Typically, the constructor function is pre-defined, in C or in Lua, and often configuration users are not aware that the constructor is a function; they simply write something like:

```
window1 = Window{ x = 200, y = 300, foreground = "blue" }
```

and think about "windows" and other high level abstractions. Thus, although Lua is dynamically typed, it provides user controlled *type constructors*.

Because constructors are expressions, they can be nested to describe more complex structures in a declarative style, as in the code below:

```
d = dialog{
           hbox{
                button{ label = "ok" },
                button{ label = "cancel" }
           }
    }
```

Reflexive facilities

Another powerful mechanism of Lua is its ability to traverse tables, using the built-in function next. This function takes two arguments: a table to be traversed and an index of this table. When the index is nil, the function returns a first index of the given table and the value associated to this index; when the index is not nil, the function returns a next index and its value. The indices are retrieved in an arbitrary order, and a nil index is returned to signal the end of the traversal. As an example of the use of Lua's traversal facilities, Figure 5 shows a routine for cloning objects. The local variable i runs over the indices of the object o, while v receives their values. These values, associated to their corresponding indices, are stored in a local table new_o.

The same way next traverses a table, a related function, nextvar, traverses the global variables of Lua. Figure 6 presents a function that saves the global environment of Lua in a table. As in function clone, a local variable n runs over the names of all

```
function clone (o)
  local new_o = {}              -- creates a new object
  local i, v = next(o,nil)      -- get first index of "o" and its value
  while i do
    new_o[i] = v                -- store them in new table
    i, v = next(o,i)            -- get next index and its value
  end
  return new_o
end
```

Figure 5. Function to clone a generic object

```
function save ()
  local env = {}                -- create a new table
  local n, v = nextvar(nil)     -- get first global var and its value
  while n do
    env[n] = v                  -- store global variable in table
    n, v = nextvar(n)           -- get next global var and its value
  end
  return env
end
```

Figure 6. Function to save Lua environment

global variables, while `v` receives their values, which are stored in a local table `env`. On exit, the function `save` returns this table, which can be later given to function `restore` to restore the environment (Figure 7). This function has two phases. First, the whole current environment is erased, including predefined functions. Then, local variables `n` and `v` run over the indices and values of the given table, storing these values in the corresponding global variables. A tricky point is that the functions called by `restore` must be kept in local variables, because *all* global names are erased.

Although it is an interesting example, the manipulation of the global environment in Lua is scarcely needed, since tables, used as objects, provide a better way to maintain multiple environments.

Support for object oriented programming

Because functions are first class values, table fields can refer to functions. This property allows the implementation of some interesting object-oriented facilities, which are made easier by syntactic sugar for defining and calling methods.

First, method definitions can be written as

```
function object:method (params)
```

```
function restore (env)
  -- save some built-in functions before erasing global environment
  local nextvar, next, setglobal = nextvar, next, setglobal
  -- erase all global variables
  local n, v = nextvar(nil)
  while n do
    setglobal(n, nil)
    n, v = nextvar(n)
  end
  -- restore old values
  n, v = next(env, nil)      -- get first index; v = env[n]
  while n do
   setglobal(n, v)           -- set global variable with name n
   n, v = next(env, n)
  end
end
```

Figure 7. Function to restore a Lua environment

```
      ...
    end
```

which is equivalent to

```
    function dummy_name (self, params)
      ...
    end
    object.method = dummy_name
```

That is, an anonymous function is created and stored in a table field; moreover, this function has a hidden parameter called `self`.

Second, a method call can be written as

```
    receiver:method(params)
```

which is translated to

```
    receiver.method(receiver,params)
```

In words, the receiver of the method is passed as its first argument, giving the expected meaning to the parameter `self`.

It is worthwhile to note some characteristics of the above construction. First, it does not provide information hiding. So, purists may (correctly) claim that an important part of object orientation is missing. Second, it does not provide classes; each object carries its operations. Nevertheless, this construction is extremely light (only syntactic sugar), and classes can be simulated using inheritance, as is common in other prototype based languages, like Self.[14] However, before discussing inheritance, it is necessary to discuss fallbacks.

FALLBACKS

Being an untyped language, Lua has a semantics with many run-time abnormal conditions. Examples are arithmetic operations applied to non numerical operands, trying to index a non table value, or trying to call a non function value. Because halting in these situations would be unsuitable for an embedded language, Lua allows programmers to set their own functions to handle error conditions; such functions are called *fallback functions*. Fallbacks are also used to provide hooks to handle other situations that are not strictly error conditions, such as accessing an absent field in a table and signaling garbage collection.

To set a fallback function, the programmer calls the function `setfallback`, with two arguments: a string identifying the fallback, and the new function to be called whenever the corresponding condition occurs. Function `setfallback` returns the old fallback function, so programs can chain fallbacks for different kinds of objects.

Lua supports the following fallbacks, identified by the given strings:

`"arith"`, `"order"`, `"concat"` – These fallbacks are called when an operation is applied to invalid operands. They receive three arguments: the two operands and a string describing the offended operator (`"add"`, `"sub"`, ...). Their return value is the final result of the operation. The default functions for these fallbacks issue an error.

`"index"` – This fallback is called when Lua tries to retrieve the value of an index not present in a table. It receives as arguments the table and the index. Its return value is the final result of the indexing operation. The default function returns `nil`.

`"gettable"`, `"settable"` – Called when Lua tries to read or write the value of an index in a non table value. The default functions issue an error.

`"function"` – Called when Lua tries to call a non function value. It receives as arguments the non function value and the arguments given in the original call. Its return values are the final results of the call operation. The default function issues an error.

`"gc"` – Called during the garbage collection. It receives as argument the table being collected, and `nil` to signal the end of garbage collection. The default function does nothing.

Before going on, it is important to notice that fallbacks are not usually set by ordinary Lua programmers. Fallbacks are used mainly by expert programmers when binding Lua to a specific application. After that, the facility is used as an integral part of the language. As a typical example, most real applications use fallbacks to implement inheritance, as described below, but most Lua programmers use inheritance without even knowing (or caring) how it is implemented.

Using fallbacks

Figure 8 shows an example that uses fallbacks to allow a more object oriented style of interpreting binary operators. When this fallback is set, expressions like `a+b`, where `a`

```
function dispatch (receiver, parameter, operator)
  if type(receiver) == "table" then
    return receiver[operator](receiver, parameter)
  else
    return oldFallback(receiver, parameter, operator)
  end
end

oldFallback = setfallback("arith", dispatch)
```

Figure 8. An example of fallbacks

is a table, are executed as `a:add(b)`. Notice the use of the global variable `oldFallback` to chain fallback functions.

Another unusual facility provided by fallbacks is the reuse of Lua's parser. Many applications would benefit from an arithmetic expression parser, but do not include one because not everyone has the required expertise or the inclination to write a parser from scratch or to use a parser generator such as yacc. Figure 9 shows the complete implementation of an expression parser using fallbacks. This program reads an arithmetic expression on the variables `a`, ..., `z`, and outputs the series of primitive operations needed to evaluate the expression, using variables `t1`, `t2`, ... as temporary variables. For example, the code generated for the expression

```
(a*a+b*b)*(a*a-b*b)/(a*a+b*b+c)+(a*(b*b)*c)
```

is

```
t1=mul(a,a)      t2=mul(b,b)      t3=add(t1,t2)
t4=sub(t1,t2)    t5=mul(t3,t4)    t6=add(t3,c)
t7=div(t5,t6)    t8=mul(a,t2)     t9=mul(t8,c)
t10=add(t7,t9)
```

The main part of this program is the function `arithfb`, which is set as a fallback for arithmetic operations. Function `create` is used to initialize the variables `a`, ..., `z` with tables, each with a field `name` containing the variable name. After this initialization, a loop reads lines containing arithmetic expressions, builds an assignment to the variable `E` and passes it to the Lua interpreter, calling `dostring`. Every time the interpreter tries to execute code like `a*a`, it calls the `"arith"` fallback, since the value of `a` is a table, not a number. The fallback creates a temporary variable to store a symbolic representation of the result of each primitive arithmetic operation.

Although small, this code actually performs global common sub-expression identification and generates optimized code. Notice in the example above how `a*a+b*b` and `a*a-b*b` are both evaluated based on a single evaluation of `a*a` and `b*b`. Notice also that `a*a+b*b` is evaluated once only. Code optimization is done simply by caching previously computed quantities in a table `T`, indexed by a textual representation of the primitive operations, whose values are the temporary variables containing the results. For example, the value of `T["mul(a,a)"]` is `t1`.

```
n=0                           -- counter of temporary variables
T={}                          -- table of temporary variables

function arithfb(a,b,op)
 local i=op .. "(" .. a.name .. "," .. b.name .. ")"
 if T[i]==nil then            -- expression not seen yet
   n=n+1
   T[i]=create("t"..n)        -- save result in cache
   print(T[i].name ..'='..i)
 end
 return T[i]
end

setfallback("arith",arithfb)  -- set arithmetic fallback

function create(v)            -- create symbolic variable
 local t={name=v}
 setglobal(v,t)
 return t
end

create("a") create("b") create("c") ... create("z")

while 1 do                    -- read expressions
 local s=read()
 if (s==nil) then exit() end
 dostring("E="..s)            -- execute fake assignment
 print(s.."="..E.name.."\n")
end
```

Figure 9. An optimizing arithmetic expression compiler in Lua

The code in Figure 9 can be easily modified to handle commutativity of addition and multiplication and anti-commutativity of subtraction and division. It is also easy to change it to output postfix representations or other formats.

In a real application, the variables `a`, ..., `z` would represent application objects, such as complex numbers, matrices, or even images, and the `"arith"` fallback would call application functions to perform the actual computation on these objects. Thus, the main use of Lua's parser is to allow programmers to use familiar arithmetic expressions to represent complex calculations on application objects.

Inheritance via fallbacks

Certainly, one of the most interesting uses of fallbacks is in implementing inheritance in Lua. Simple inheritance allows an object to look for the value of an absent field

```
function Inherit (object, field)
  if field == "parent" then     -- avoid loops
    return nil
  end
  local p = object.parent       -- access parent object
  if type(p) == "table" then    -- check if parent is a table
    return p[field]             -- (this may call Inherit again)
  else
    return nil
  end
end

setfallback("index", Inherit)
```

Figure 10. Implementing simple inheritance in Lua

in another object, called its *parent*; in particular, this field can be a method. This mechanism is a kind of object inheritance, in contrast to the more traditional class inheritance, adopted in Smalltalk and C++. One way to implement simple inheritance in Lua is to store the parent object in a distinguished field, called `parent` for instance, and set an index fallback function as shown in Figure 10. This code defines a function `Inherit` and sets it as the `"index"` fallback. Whenever Lua attempts to access a field that is absent in an object, the fallback mechanism calls the function `Inherit`. This function first checks whether the object has a field `parent` containing a table value. If so, it attempts to access the desired field in this parent object. If this field is not present in the parent, the fallback is automatically called again; this process is repeated "upwards" until a value for the field is found or the parent chain ends.

The above scheme allows endless variations. For instance, only methods could be inherited, or only fields starting with an underscore. Many forms of multiple inheritance can also be implemented. Among them, a frequently used form is *double inheritance.* In this model, whenever a field is not found in the parent hierarchy, the search continues through an alternative parent, usually called `"godparent"`. In most cases, one extra parent is enough. Moreover, double inheritance can model generic multiple inheritance. In the code below, for instance, `a` inherits from `a1`, `a2`, and `a3`, in this order:

```
a = {parent = a1, godparent = {parent = a2, godparent = a3}}
```

THE USE OF LUA IN REAL APPLICATIONS

TeCGraf is a research and development laboratory at the Pontifical Catholic University in Rio de Janeiro (PUC-Rio) with many industrial partners. Some forty programmers at TeCGraf have used Lua in the past two years to develop several substantial products. This section describes some of these uses.

```
Grid{
  name = "log",
  log = TRUE,
  h_step = 25,
  v_step = 25,
  v_tick = 5,
  step_line = Line {color = RED, width = SIMPLE},
  tick_line = Line {color = CORAL}
}
```

Figure 11. Description of a lithology profile object in Lua

Configurable report generator for lithology profiles

As mentioned in the introduction, Lua initially arose for supporting two different applications that had their own, but limited, extension languages. One of these applications is a tool for visualizing lithology profiles obtained from geological probes. Its main characteristic is to allow the user to configure profile layout, combining instances of objects and specifying the data to be shown. The program supports several kinds of objects, such as continuous curves, histograms, lithology representation, scales, etc.

To build a layout, users may write Lua code describing these objects (Figure 11). The application itself also has Lua code that allows the creation of such descriptions by means of a graphical user interface. This facility was built over the EDG framework, described below.

Storing structured graphical metafiles

Another important use of Lua is for the storage of structured graphical metafiles. The generic drawing editor TeCDraw, developed by TeCGraf, saves metafiles containing high level descriptions, in Lua, of the graphic objects that compose the drawing. Figure 12 illustrates these descriptions.

Such generic structured metafiles bring several benefits for development:

- As a direct consequence, the Lua interpreter can be used to load and parse the metafile; the editor only provides functions for holding Lua objects and converting them to the corresponding application objects.
- Applications can share graphical objects by using the same metafile format. Moreover, graphical objects generated in such applications can be edited with TeCDraw.
- The structured description with Lua syntax makes the metafile editable by humans: it is easy to identify and modify an object using conventional text editors.
- Since each object is easily identified, it can be individually manipulated. This feature is exploited in the EDG system for implementing support to active graphic objects.
- A graphical metafile in Lua allows the instantiation of procedural objects. For example, it is possible to describe curves using mathematical expressions.

```
line{
   x = { 0.0, 1.0 },
   y = { 5.0, 8.0 },
   color = RED
}
text{
   x = 0.8,
   y = 0.5,
   text = 'an example of text',
   color = BLUE
}
circle{
   x = 1.0,
   y = 1.0,
   r = 5.0
}
```

Figure 12. A excerpt from a structured graphical metafile

High level, generic graphical data entry

Lua features are also heavily exploited in the implementation of EDG, a system for supporting the development of data entry programs, with high abstraction level. The system provides manipulation of interface objects (such as buttons, menus, lists) and graphic objects (such as lines, circles, and groups of primitives). Hence, programmers can build sophisticated interface dialogs in a high abstraction programming level. Programmers can also associate callback actions to graphic objects, thus creating active objects that react procedurally to user input.

The EDG system uses the Lua fallback feature for implementing double inheritance, as explained above. Thus, new interface and graphic objects can be built, inheriting original object behavior. Another interesting use of inheritance present in EDG is *cross-language inheritance*. EDG is built upon the portable user interface toolkit IUP.[15] To avoid duplicating in Lua IUP data residing in the host, EDG uses fallbacks for `"gettable"` and `"settable"` to access fields in the toolkit directly from Lua. Thus, host data can be accessed directly, using an intuitive record syntax, without creating an access function for each exported data item in the host.

The EDG system has been used in the development of several data entry programs. In many engineering systems, the complete analysis is divided in three steps: data entry, called pre-processing; the analysis itself, called processing or simulation; and result report and verification, called post-processing. The data entry task can be made easier by drawing graphical representation of the data that must be specified as input to the analysis. For such applications, the EDG system is extremely helpful and provides a fast development tool for customized data entry. These graphical data entry tools have given new life to the legacy code of batch simulation programs.

```
ISO_MAT = ctrclass{ parent = MATERIAL,
                    name = "Isotropic",
                    vars = {"e", "nu"}
          }

function ISO_MAT:CrtDlg ()
  ...  -- creates a dialog to specify this material
end
```

Figure 13. Creating a new material in ESAM

Generic attribute configuration for finite element meshes

Another engineering area where Lua is being used is the generation of finite element meshes. A finite element mesh is composed by nodes and elements, which decompose the domain of analysis. To complete the model, physical properties (attributes) must be associated to nodes and elements, such as material type, support conditions and loading cases. The set of attributes that must be specified varies widely according to the analysis to be done. Thus, to implement versatile finite element mesh generators, it is recommended that the attributes remain configurable by the user, and not hard coded in the program.

ESAM[16] is a generic system that uses Lua to provide support for attribute configuration. Like EDG, ESAM adopts an object oriented approach: users create specific properties deriving from pre-defined core classes. Figure 13 shows an example of how to create a new kind of material, called "Isotropic".

RELATED WORK

This section discusses some other extension languages, and compares them with Lua. There is no intention of being comprehensive; instead, some representatives of current trends in extension languages have been selected: Scheme, Tcl, and Python. A comprehensive list of embedded languages is available in the Internet.[17] This section also compares the fallback mechanism with some other language mechanisms.

Lisp dialects, particularly Scheme, have always been a popular choice for extension languages, for their simple, easily parsed syntax and built-in extensibility.[8, 18, 19] For instance, a major part of the text editor Emacs is actually written in its own variant of Lisp; several other text editors have followed the same path. There are currently many implementations of Scheme in the form of libraries, especially designed to be used as an embedded language (for instance, libscheme,[18] OScheme,[20] and Elk[3]). However, Lisp cannot be called user-friendly when it comes to customization. Its syntax is rather crude for non-programmers. Moreover, few implementations of Lisp or Scheme are truly portable.

Another very popular extension language nowadays is Tcl.[11] Undoubtedly, one of the reasons for its success is the existence of Tk, a powerful Tcl toolkit for building graphical user interfaces. Tcl has a very primitive syntax, which greatly simplifies its interpreter, but also complicates writing even slightly complex constructions. For example, the Tcl code to double the value of a variable `A` is `set A [expr $A*2]`. Tcl supports a single primitive type, *string*. This fact, added to the absence of pre-compilation, makes Tcl rather inefficient, even for an extension language. Correcting these problems can improve the efficiency of Tcl by a factor of 5 to 10, as shown by TC.[21] Lua, with more adequate data types and pre-compilation, runs 10 to 20 times faster than Tcl. A simple test shows that a procedure call with no arguments, in Tcl 7.3 running in a Sparcstation 1, costs around 44 μs, while the increment of a global variable takes 76 μs. In Lua v. 2.1, the same operations cost 6 μs and 4 μs, respectively. On the other hand, Lua is approximately 20 times slower than C. This seems to be a typical value for interpreted languages.[22]

Tcl does not have built-in control structures, such as *while*s and *if*s. Instead, control structures are programmable via delayed evaluation, as in Smalltalk. Although powerful and elegant, programmable control structures can lead to very cryptic programs, and are seldom used in practice. Moreover, they often bring a high performance penalty.

Python[23] is an interesting new language that has also been proposed as an extension language. However, according to its own author, there is still a need for "improved support for embedding Python in other applications, e.g., by renaming most global symbols to have a 'Py' prefix".[24] Python is not a tiny language, and has many features not necessary in extension languages, like modules and exception handling. These features add extra cost to applications using the language.

Lua has been designed to combine the best of existing languages in order to fulfill its aim as an extensible extension language. Like Tcl, Lua is a small library, with a simple interface to C; this interface is a single header file with 100 lines. Unlike Tcl, however, Lua is pre-compiled to a standard bytecode intermediate form. Like Python, Lua has a clean but familiar syntax, and a built-in notion of objects. Like Lisp, Lua has a single data structure mechanism (tables), powerful enough to *efficiently* implement most data structures. Tables are implemented using hashing. Collisions are handled by linear probing, with automatic reallocation and rehashing when the table becames more than 70% full. Hash values are cached to improve access performance.

The fallback mechanism presented in Lua can be viewed as a kind of exception handling mechanism with resumption.[25] However, the dynamic nature of Lua allows its use in many cases where a statically typed language would issue an error at compile time; both examples presented above are of this kind. Three particular fallbacks, `"arith"`, `"order"` and `"concat"`, are mainly used to implement overloading. In particular, the example in Figure 9 could be readily translated to other languages with overloading, like Ada or C++. However, because of its dynamic nature, fallbacks are more flexible than exception handling or overloading mechanisms. On the other hand, some authors[26] argue that programs that use these mechanisms tend to be difficult to verify, understand, and debug; these difficulties are worsened when using fallbacks. Fallbacks should be written with care and moderation, and only by expert programmers.

CONCLUSION

The increasing demand for configuration applications is changing the structure of programs. Nowadays, many programs are written in two different languages: one for writing a powerful "virtual machine", and another for writing single programs for this machine. Lua is a language designed specifically for the latter task. It is small: as already noted, the whole library is around six thousand lines of ANSI C. It is portable: Lua is being used in platforms ranging from PC-DOS to CRAY. It has a simple syntax and a simple semantics. And it is flexible.

Such flexibility has been achieved through some unusual mechanisms that make the language highly extensible. Among these mechanisms, we emphasize the following:

Associative arrays are a strong unifying data constructor. Moreover, it allows more efficient algorithms than other unifying constructors like strings or lists. Unlike other languages that implement associative arrays,[10-12] tables in Lua are dynamically created objects with an identity. This greatly simplifies the use of tables as objects, and the addition of object-oriented facilities.

Fallbacks allow programmers to extend the meaning of most built-in operations. Particularly, with the fallbacks for indexing operations, different kinds of inheritance can be added to the language, while fallbacks for `"arith"` and other operators can implement dynamic overloading.

Reflexive facilities for data structure traversal help produce highly polymorphic code. Many operations that must be supplied as primitives in other systems, or coded individually for each new type, can be programmed in a single generic form in Lua. Examples are cloning objects and manipulating the global environment.

In addition to using Lua in several industrial applications, we are currently experimenting with Lua in a number of research projects, ranging from computing with distributed objects that send each other messages containing Lua code[27] (an idea previously proposed in Tcl[4]), to transparently extending WWW browsers with client-side Lua code. Because all functions that interface Lua with the operating system are provided in external libraries, it is easy to restrict the power of the interpreter in order to provide adequate security.

We also plan to improve the facilities for debugging Lua; currently, only a simple stack traceback is available. Following the philosophy of providing powerful meta mechanisms that allow programmers to build their own extensions, we plan to add simple hooks to the run time system to allow user programs to be informed when important events happen, such as entering or exiting a function, executing a line of user code, etc. Different debugging interfaces can be built on top of these basic hooks. Moreover, the hooks are also useful for building other tools, such as profilers for performance analysis.

The implementation of Lua described in this paper is available in the Internet at:

```
ftp://ftp.icad.puc-rio.br/pub/lua/lua-2.1.tar.gz
http://www.inf.puc-rio.br/~roberto/lua.html
```

ACKNOWLEDGEMENTS

We would like to thank the staff at ICAD and TeCGraf for using and testing Lua, and John Roll, for valuable suggestions by mail concerning fallbacks in a previous version of Lua. The industrial applications mentioned in the text are being developed in partnership with the research centers at PETROBRAS (The Brazilian Oil Company) and ELETROBRAS (The Brazilian Electricity Company). The authors are partially supported by research and development grants from the Brazilian government (CNPq and CAPES). Lua means *moon* in Portuguese.

REFERENCES

1. B. Ryan, 'Scripts unbounded', *Byte*, **15**(8), 235–240 (1990).
2. N. Franks, 'Adding an extension language to your software', *Dr. Dobb's Journal*, **16**(9), 34–43 (1991).
3. O. Laumann and C. Bormann. Elk: The extension language kit. `ftp://ftp.cs.indiana.edu:/pub/scheme-repository/imp/elk-2.2.tar.gz`, Technische Universität Berlin, Germany.
4. J. Ousterhout, 'Tcl: an embeddable command language', *Proc. of the Winter 1990 USENIX Conference*. USENIX Association, 1990.
5. D. Cowan, R. Ierusalimschy, and T. Stepien, 'Programming environments for end-users', *12th World Computer Congress*. IFIP, Sep 1992, pp. 54–60 Vol. A–14.
6. L. H. Figueiredo, C. S. Souza, M. Gattass, and L. C. Coelho, 'Geração de interfaces para captura de dados sobre desenhos', *V SIBGRAPI*, 1992, pp. 169–175.
7. R. Ierusalimschy, L. H. Figueiredo, and W. Celes, 'Reference manual of the programming language Lua version 2.1', *Monografias em Ciência da Computação 08/95*, PUC-Rio, Rio de Janeiro, Brazil, 1995. (available by ftp at `ftp.inf.puc-rio.br/pub/docs/techreports`).
8. B. Beckman, 'A scheme for little languages in interactive graphics', *Software, Practice & Experience*, **21**, 187–207 (1991).
9. J. Bentley, *More programming pearls*, Addison-Wesley, 1988.
10. A. V. Aho, B. W. Kerninghan, and P. J. Weinberger, *The AWK programming language*, Addison-Wesley, 1988.
11. J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
12. L. Wall and R. L. Schwartz, *Programming perl*, O'Reilly & Associates, Inc., 1991.
13. L. Lamport, LaTeX*: A Document Preparation System*, Addison-Wesley, 1986.
14. D. Ungar et al., 'Self: The power of simplicity', *Sigplan Notices*, **22**(12), 227–242 (1987). (OOPSLA'87).
15. C. H. Levy, L. H. de Figueiredo, C. J. Lucena, and D. D. Cowan. IUP/LED: a portable user interface development tool. submited to *Software: Practice & Experience*.
16. M. T. de Carvalho and L. F. Martha, 'Uma arquitetura para configuração de modeladores geométricos: aplicação a mecânica computacional', *PANEL95 — XXI Conferência Latino Americana de Informática*, 1995, pp. 123–134.
17. C. Nahaboo. A catalog of embedded languages. `ftp://koala.inria.fr:/pub/EmbeddedInterpretersCatalog.txt`.
18. B. W. Benson Jr., 'libscheme: Scheme as a C Library', *Proceedings of the 1994 USENIX Symposium on Very High Level Languages*. USENIX, October 1994, pp. 7–19.
19. A. Sah and J. Blow, 'A new architecture for the implementation of scripting languages', *Proc. USENIX Symposium on Very High Level Languages*, 1994.
20. A. Baird-Smith. OScheme manual. `http://www.inria.fr/koala/abaird/oscheme/manual.html`, 1995.
21. A. Sah, 'TC: An efficient implementation of the Tcl language', *Master's Thesis*, University of California at Berkeley, Dept. of Computer Science, Berkeley, CA, 1994.
22. Sun Microsystems, *Java, The Language*, 1995. http://java.sun.com/people/avh/talk.ps.
23. G. van Rossum, 'An introduction to Python for UNIX/C programmers', *Proc. of the NLUUG najaarsconferentie*. Dutch UNIX users group, 1993. (`ftp://ftp.cwi.nl/pub/python/nluug-paper.ps`).
24. G. van Rossum. Python frequently asked questions. `ftp://ftp.cwi.nl/pub/python/python-FAQ`, March 1995. version 1.20++.
25. S. Yemini and D. Berry, 'A modular verifiable exception handling mechanism', *ACM Transactions on Programming Languages and Systems*, **7**(2) (1985).
26. A. Black, 'Exception handling: the case against', *Ph.D. Thesis*, University of Oxford, 1982.
27. R. Cerqueira, N. Rodriguez, and R. Ierusalimschy, 'Uma experiência em programação distribuída dirigida por eventos', *PANEL95 — XXI Conferência Latino Americana de Informática*, 1995, pp. 225–236.