Lua , plugin Lua, AReVi et Lua

D. Herviou

28 juillet 2005

1 Evolution

1. 5 juillet 2005 : création de cette documentation
2. 28 juillet 2005 : mise à jour de la documentation
   - création des macros CHECK_RETRIEVE_ARREF et CHECK_LUA_ARG
   - modification de checkError pour avoir l'affichage du message d'erreur
   - modification de luaL_error du wrapper lua
   - création d'une classe LuaInterpreter

2 Lua

2.1 Introduction de Lua (short)

Pré-requis : aucun

Lua est un langage de script[1], portable, léger et aisement embarquable dans une application. Facile d'utilisation de par sa syntaxe,son typage implicite, son garbage collector, on y retrouve des traits de programmation impérative et fonctionnelle. De plus des mécanismes (appelé meta-mecanismes) sont offerts à travers le langage pour implémenter des classes et de l'héritage de classes. Grâce à son API-C, le langage peut-être étendu mais aussi peut permettre d'étendre une application, c'est pourquoi Lua est décrit comme *"Lua-an extensible extension language"*[2]. La documentation en ligne peut-être trouvée à http://www.lua.org/pil/ .

[1] http://www.lua.org
[2] http://www.lua.org/docs.html

> *"Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, interpreted from bytecodes, and has automatic memory management with garbage collection, making it ideal for configuration, scripting, and rapid prototyping."*

Bientôt peut-être des exemples tutoriaux pour Lua. En attendant RTFM.

2.2 La philosophie Lua

Pré-requis : fonctionnement d'une pile

Lua utilise une API-C qui permet de venir s'interfacer. Elle offre des fonctionnalités pour venir travailler sur la pile d'execution du moteur Lua. Ces fonctions permettent :

- d'ajouter des données de différents types (fonctions, données utilisateurs, nombres, chaînes de caractères)
- d'enlever des données
- de répliquer des données
- d'insérer des données
- de remplacer des données
- d'obtenir la taille de la pile
- ...

Lorsque l'on insére des données dans la pile, le typage de celles-ci est explicite, ce qui permet ultérieurement de vérifier le type de donnée stocké à un endroit particulier de la pile : c'est très utile notamment lorsque l'on veut vérifier que le type des arguments passés à une fonction correspondent bien à ceux requis pour l'execution de celle-ci.

En ce qui concerne l'execution, Lua utilise un état appelé `lua_State` qui contient l'ensemble des librairies ouvertes, l'ensemble des symboles globaux à l'execution ainsi que la pile et ses données. Il est possible de créer plusieurs de ces états au sein d'une même application, procurant ainsi un moyen simple de séparation des données entre les différentes executions. Cet état est le premier argument passer en paramêtre des fonctions en Lua, qu'il s'agisse des fonctions core de Lua ou des fonctions d'extensions. Ainsi, il est possible de récupérer n'importe qu'elle donnée de la pile et notamment les arguments d'une fonction appelée.

3 Plugin Lua pour ARéVi

Pré-requis : savoir ce qu'est un plugin ARéVi

Pour pouvoir utiliser Lua avec ARéVi, il convient de fournir un plugin Lua pour rester dans l'esprit des extensions pour ARéVi. Le plugin Lua fournit ici utilise la convention de nommage suivante pour les fonctions "wrappées" :

- Elles sont dans une classe `Lua`
- le nom des fonctions de wrap est le même que celui des fonctions Lua : pour la fonction Lua `lua_gettop(lua_State * L)` nous aurons `Lua::lua_gettop(void * L)`
- le nom des pointeurs de fonctions utilisés pour l'initialisation du plugin est le suivant : _arlua_*le_nom_de_la_fonction_de_wrap*

4 Lua et ARéVi

Afin de faciliter l'utilisation de Lua avec ARéVi nous avons créer plusieurs classes de services qui permettent de partager des données entre ARéVi et Lua et de débuger les états Lua.

4.1 La classe LuaBind

4.1.1 Fonctionnalités

Pré-requis : savoir ce qu'est un garbage collector

La classe `LuaBind` est déclaré dans le fichier `arlua.h` . Elle contient une structure :

```
lua_data_share{
   ArObject * data;
};
```

Cette structure représente la donnée partagé entre Lua et ARéVi : elle encapsule un pointeur sur un ArObject.

La fonction de partage d'une ArRef est :

```
int shareArRef (void * L,
                ArRef<ArObject> objShare);
```

Pour récupérer une donnée partagée en Lua il faut utiliser :

```
int                retrieveArRef       (void * L,
                                        int index);
```

- shareArRef permet de partager une ArRef avec Lua.
- retrieveArRef permet de récupérer une ArRef depuis un état Lua L et se trouvant dans la pile à l'index index

Des fonctionnalités supplémentaires sont ajoutées afin de partagerdes vecteurs d'ArRef, des vecteurs de doubles, des vecteurs d'entiers ou encore des vecteurs de StlString.

Lors d'un passage de garbage collection les objets partagés avec shareArRef seront détruit avec la fonction int destroy(void * L). Cette fonction récupère l'ArRef partagée et l'enlève de la map _shareAref qui ressence toutes les ArRef partagées. Ainsi, si AReVi reference encore cette ArRef, elle ne sera pas totalement détruite même si celle a été créé en Lua.

La méthode exportClass

```
void exportClass(void    * L,
                 const  char * clsName,
                 const  arluaL_reg  * methods  ,
                 const  arluaL_reg  * constructor);
```

est une méthode (sympatique) pour permettre d'exporter une de ses classes ARéVi vers Lua en permettant de garder la notion d'héritage faite en ARéVi, de plus elle autorise l'écriture "à la mode objet" en Lua pour les objets de cette classe.

Par exemple : Exportons la classe ArObjet vers Lua, puis ensuite la classe Base3D ; la méthode exportClass vérifie de quelle classe Base3D est issue, si la classe mère a été enregistré préalablement (ici ArObject) alors la classe Base3D héritera des méthodes d'ArObject. Une seule restriction à l'exportation des classes doient se faire dans un ordre particulier : des mères vers les filles pour que le mécanisme puisse récupérer toutes les méthodes pour les classes filles. Les paramêtres nécessaires à l'exportation d'une classe sont le nom de la classe (il faut par convention utiliser le nom donné par la méthode de classe AReVi getName()) et ensuite un tableau de fonctions C qui représentent les méthodes de la classe et un tableau de fonctions C qui représentent les constructeurs de la classe. Un exemple d'exportation de classe vers Lua sera donné plus tard dans le document.

Explication complémentaire : une table de lookUp (_lookUp) est maintenue dans la classe LuaBind qui permet de stocker les méthodes des objets que l'on partage. Ainsi en exportant une classe Base3D, on enregistre les méthodes exportées pour cette classe et lorsque l'on voudra exporter une classe Object3D, nous pourrons automatiquement récupérer les méthodes de la classe Base3D pour les ajouter à celle d'Object3D.

4.1.2 Du coté (clair) du partage de données

Pré-requis : savoir ce qu'est une table et une métatable en Lua, ainsi que les 2 façons de créer des données utilisateurs en Lua.

La classe `LuaBind` , en plus des fonctionnalités proposées pour le partage des données, enregistre les données partagées dans une table Lua. Cette table est utilisé dans le cas suivant : lors d'un partage d'une ArRef pour la première fois, la donnée partagée doit être alloué par Lua (pour permettre de fournir un type à la donnée stocké sur la pile Lua : ici `lua_newuserdata`).

Lorsque l'utilisateur désirera partagé une deuxième fois cette même ArRef, il n'est pas question de créer une nouvelle donnée partagée encapsulant ce même pointeur : en effet, cela poserait des problèmes d'égalités sur les données du coté de Lua[3].

La solution est donc de dupliquer la donnée précédemment partagée et de la mettre sur le haut de la pile. Ceci est rendu possible par le stockage dans une table des données partagées avec en clé le pointeur de l'ArRef (c_ptr()) et en valeur la donnée partagée créé par Lua.

Pour récupérer cette donnée partagée il faut mettre la metatable sur la pile, puis mettre la clé sur la pile (`c_ptr()`) et ensuite faire une recherche dans la table pour la clé donnée à l'aide d'un `lua_gettable` (c'est ce que fait `shareArRef`).

4.1.3 Du coté (obscur) du partage de données

Pré-requis : idem 4.1.2. Etre conscient des problèmes d'égalités de données utilisateurs en Lua.

Si vous n'êtes pas familier avec le fonctionnement du moteur Lua passer votre chemin, ceci ne vous apportera rien. En revanche si vous voulez connaître plus en détail (que le paragraphe précédent) le fonctionnement du partage de données entre ARéVi et Lua je vous conseille vivement de lire ce qui suit.

Pourquoi avons-nous opté pour le partage avec des données utilisateurs fortes (`lua_newuserdata`) et pas pour le partage avec des données faibles (`lua_lightuserdata`) ?

En effet, l'utilisation de données faibles permet de facilement effectuer les opérations d'égalités car il suffit de mettre dans la pile des pointeurs sur n'importe quelle donnée utilisateur pour que l'opération d'égalité soit vraie (arithmétique des pointeurs). Dans notre cas, il aurait suffit de mettre des pointeurs sur des `ArObject` .

[3] L'égalité des données utilisateurs en Lua n'est obtenu que lorsque les pointeurs sont égaux

Cependant, l'utilisation de cette technique est très limité, en fait le garbage collector ne gère pas ce type de donnée (c'est aussi pour cela qu'elles sont appelées faibles) de plus tous les mécanismes puissants de Lua (metatables pour fabriquer des objets) ne sont pas disponibles avec l'utilisation des données faibles.

C'est pourquoi il faut passer par la création de données forte (utilisation de lua_newuserdata) qui créer une donnée partagée avec une métatable et qui de plus sera soumis au garbage collector.

Mais ceci n'est pas sans poser de problème, l'opération d'égalité sur les données utilisateurs deviennent maintenant beaucoup plus difficile car lorsque l'on partage une ArRef on passe par lua_newuserdata qui génére une nouvelle zone mémoire donc deux pointeurs différents donc une inégalité systématique même lorsque la donnée encapsulée est identique.

Il faut donc un moyen pour permettre l'égalité sur ces données, c'est à dire de faire en sorte que le pointeur obtenu lors du premier appel à lua_newuserdata soit récupéré lorsque l'utilisateur cherche à partager à nouveau la même ArRef.

La solution choisie est la suivante : lors de l'ouverture des binds arevi (fichiers arevi_lua_bindings) nous créons une metatable (à clé faible pour ne pas faire de référencement sur ce qu'elle contient) avec un nom aléatoire, cette métatable sera utilisée comme tableau associatif de stockage pour mettre les données obtenues par lua_newuserdata (utilisé comme valeur) ainsi que les pointeurs de données encapsulées (pointeur d'ArObject utilisé comme clé). Il faut aussi remplir la map `_objectsTable` de `LuaBind` :: avec le pointeur sur l'état Lua et le nom de la métatable dans laquelle la donnée partagée est stocké.Ceci permettra lors d'un second partage de la donnée de récupérer la métatable de stockage à partir du pointeur de l'état Lua. Dans un second temps, nous utiliserons le pointeur d'`ArObject` comme clé de recherche dans cette métatable, la valeur associée à cette clé étant la donnée partagée lors du premier appel.

Pour générer cette metatable de stockage des données partagées il faut faire appel à la fonction `LuaBind : :createMarshallArea(lua_State) ;`

4.1.4 L'héritage et la notation objet

Pré-requis : Connaître la syntaxe d'écriture de code Lua.

Ce paragraphe est technique, si votre objectif est simplement d'utiliser le plugin passer à la section suivante.

Pouvoir écrire du code Lua avec la notation objet est en fait un sucre syntaxique proposé à l'utilisateur basé en plus sur l'utilisation des métamécanismes

de Lua. En effet prenons l'exemple d'une classe Counter qui aurait une méthode add deux solutions sont proposées pour écrire l'appel à cette méthode à partir d'un objet de "type" Counter appelé obj_count :

1. Counter.add(obj_count,dt)
2. obj_count :add(dt)

La seconde écriture est beaucoup moins lourde et souligne l'aspect objet de obj_count. Lorsque l'on veut écrire du code sous cette forme en Lua il suffit d'utiliser un petit artifice qui permet d'utiliser cette notation. Cet artifice est à utiliser dans le constructeur de l'objet counter. Cet artifice est détaillé à http://www.lua.org/pil/16.html.

Prenons un exemple : la classe Counter avec 1 méthode add et 1 constructeur new. Nous disons lors de la construction d'un objet Counter que :

```
Counter={counter=0}

function Counter:new(o)    --equivaut à Counter.new(self,o)
  self=o or {}
  setmetatable(o,self)
  self.__index=self
  counter=0
  return o
end

function Counter:add(dt)
  self.counter = self.counter + dt
end
```

Ce constructeur prend 0 ou 1 paramêtre (un autre counter), premièrement nous disons que self (ici Counter) est une table vide ou une table égale à l'objet passé en paramêtre : cette table contiendra toutes les méthodes de Counter dans laquelle Lua ira chercher en premier. Deuxièmement, nous disons que la métatable de la classe est self (Counter). Troisièmement, nous disons que le champs __index de self est self : soit Counter.__index = Counter. Cet artifice permet l'écriture à la mode objet.

En effet, lorsque l'on écrit obj_count :add(dt) nous faisons en fait appel à obj_count.add(obj_count,dt), Lua regarde dans la table obj_count pour une entrée add qu'il ne trouve pas, il vérifie donc dans la métatable stocké dans le champs __index, nous avons donc : getmetatable(obj_count).__index.add(obj_count,dt) avec getmetatable(obj_count) qui est Counter nous avons ainsi : Counter.__index.add(obj_count,dt) et par conséquent : Counter.add(obj_count,dt). Lua fait donc l'appel à la fonction add initiale de Counter avec comme paramêtre self obj_count.

Pour plus de précision sur l'écriture d'objet et l'héritage en Lua je vous conseille vivement d'aller voir les urls suivantes :

- http://www.lua.org/pil/16.html
- http://www.lua.org/pil/16.1.htm l
- http://www.lua.org/pil/16.2.htm l

Maintenant que nous avons vu les bases pour pouvoir écrire à la mode objet en Lua, nous allons voir comment il est possible d'utiliser la notation objet pour les données utilisateurs (données partagées). L'idée est de recréer pour les données utilisateurs le même mécanisme que celui évoqué ci-dessus.

Pour ce faire il faut procéder en 8 étapes :

1. créer une métatable (MT) pour l'objet
2. créer un champ __index dans cette MT
3. affecter ce champ comme égal à MT (MT.__index=MT)
4. créer un champ __gc pour permettre à l'objet d'être soumis au garbage
5. affecter ce champ comme égal à une fonction de destruction(FD) : MT.__gc=FD
6. remplir la métatable avec les méthodes de l'objet (les fonctions "bindées" : voir 5)
7. remplir la métatable avec les méthodes des classes parents (lookUp)
8. remplir avec les constructeurs de l'objet

Ces étapes sont faites dans la méthode de classe LuaBind::exportClass.

4.2 La classe LuaInterpreter

Pré-requis : connaître la syntaxe lua.
Cette classe permet d'instancier un interpreter de commande Lua. Il suffit pour cela de faire :

```
void * lua_State   L = Lua::lua_open();
ArRef<LuaInterpreter>      lua  = new_Interpreter();
StlString   code  = ''print(4+3,\''toto\'')''
if(lua->interpert(L,code)){        //L peut être un autre lua_State...
  cerr<<lua->getError()<<endl;
}
```

4.3 La classe LuaInfo

Pré-requis : connaître les types Lua et le fonctionnement de sa pile.

La classe LuaInfo permet d'afficher des informations sur la pile d'un état Lua à un instant donné. 3 fonctions sont fournis pour cela :

```
void checkLuaType(void * L, int index);
void dumpLuaStack(void * L);
void sizeofLuaStack(void * L);
```

- checkLuaType donne le type de la donnée dans la pile à l'index index
- dumpLuaStack affiche le contenu de la pile
- sizeofLuaStack donne la taille de la pile

5 Créer ses bindings ARéVi pour Lua

Pré-requis : savoir suivre des instructions aveugléments et savoir manipuler la pile Lua.

Pour créer ses bindings vers Lua nous détaillons ici la démarche à suivre (aveuglément). Cette méthode vous permettra d'éviter toute erreur d'incohérence dans vos binds mais également vous permettra de pouvoir utiliser la notation objet Lua pour vos classes "bindées".

Le fichier arevi_lua_bindings.cpp contient déja un ensemble de Binds de classes ARéVi. L'utilisation des binds est intéressante dans le cas ou il s'agit de classe utilisateur. Voici les étapes à suivre pour effectuer vos binds :

1. créer les fonctions de bind selon un prototype fixe
2. créer un tableau de structure, structure contenant le nom et la fonction wrappé
3. créer une fonction d'ouverture des binds

Nous prenons dans la suite l'exemple de la classe ARéVi Base3D dont nous voudrions binder certaines méthodes.

5.1 fonctions de bind

Les fonctions de binds doivent respecter le prototype suivant :

```
int name_of_bind_function(void * L)
```

L'entier de retour donne le nombre d'arguments de retour de la fonction de bind, L est l'état Lua qui sera passer en paramêtre par le moteur Lua lorsque cette fonction sera appelée.

Pour bindée la méthode globalToLocal de la classe Base3D ne ferons donc :

```
int arevi_lua_Base3D_globalToLocal(v        oid * L){
 //recuperation    du premier parametre   de la fonction   (self)
 ArRef<Base3D>    obj=ar_down_cast<Base3D>retrie       veAr Ref( L,1);
 //recuperation    des param êtres  de la fonction
 double  z = Lua::lua_tonumber(L,-1);
 double  y = Lua::lua_tonumber(L,-2);
 double  x = Lua::lua_tonumber(L,-3);

 obj->globalToLocal(x,y,z);

 //affectation    des résultats
 Lua::lua_pushnumber(L,x);
 Lua::lua_pushnumber(L,y);
 Lua::lua_pushnumber(L,z);

 //nombre   d'élément  mis  sur  la pile
 return  3;
}
```

Depuis le 28/08/2005 nous pouvons écrire

```
int arevi_lua_Base3D_globalToLocal(v        oid * L){
 //recuperation    du premier   parametre   de la fonction   (self)
 CHECK_RETRIEVE_ARREF(L,obj,Bas       e3D,1 )
 CHECK_LUA_ARG(L,number,-1)
 CHECK_LUA_ARG(L,number,-2)
 CHECK_LUA_ARG(L,number,-3)

 //recuperation    des param êtres  de la fonction
 double  z = Lua::lua_tonumber(L,-1);
 double  y = Lua::lua_tonumber(L,-2);
 double  x = Lua::lua_tonumber(L,-3);

 obj->globalToLocal(x,y,z);

 //affectation    des résultats
 Lua::lua_pushnumber(L,x);
 Lua::lua_pushnumber(L,y);
 Lua::lua_pushnumber(L,z);

 //nombre   d'élément  mis  sur  la pile
 return  3;
```

```
}
```

La macro CHECK_RETRIEVE_ARREF(lua_State, variable, Classe, index) permet de tester si l'argument (index) est une donnée utilisateur et de récupérer une ArRef (variable) du bon type(Classe). Si une erreur est détectée elle est indiqué à l'utilisateur par un message lors de l'execution. La macro CHECK_LUA_ARG(lua_State,type permet de tester si l'argument à l'index (index) est du type (type) qui peut être tout ceux proposé par lua_is*().

Pour le constructeur d'une Base3D nous ferons :

```
int arevi_lua_Base3D_new(void * L){
  //creation de l'objet
  ArRef<Base3D> base = new Base3D();

  //partage de l'objet : true declare l'objet
  //comme soumis au garbage lua
  int nb = LuaBind::shareArRef(L,base,tr ue);

  base->setTransient(false);
  return nb;
}
```

5.2 déclaration des tableaux de structures

Pour permettre à Lua d'utiliser ces méthodes nous devons déclarer un tableau particulier contenant le nom de la fonction et la fonction de bind comme suit :

```
const arluaL_reg arevi_lua_Base3D_method[]={
  {"localToGlobal",arevi_lua_Ba se3D_ loca lToGl obal },
  {NULL,NULL} //sentinelle
};

const arluaL_reg arevi_lua_Base3D_function[]= {
  {"new",arevi_lua_Base3D_new},
  {NULL,NULL} //sentinelle
};
```

5.3 fonction d'ouverture des binds

Pour enregistrer tout ceci dans Lua il faut créer une fonction d'ouverture. Pour cela il faut procéder comme suit :

```
int luaopen_arevi_lua_base3D_bindings(void* L){

 //creation de la table de stockage pour l'état L dans
 //laquelle nous stockerons les données partagées
 //si elle n'existe pas elle sera crée sinon rien.
 LuaBind::createMarshallArea(L);

 LuaBind::export_class(L,"Base3D",
                       arevi_lua_Base3D_method,
                       arevi_lua_Base3D_function);

 return 0;
}
```

Cette fonction enregistre en Lua les méthodes et les fonctions pour un objet Base3D. Lorsque l'on voudra utiliser un objet de ce type en Lua nous pourrons donc ecrire après avoir fait appel à cette fonction :

```
base = Base3D:new()
x_local,y_local,z_local=10,50,0.2
x_global,y_global,z_global=base:localToGlobal(x,y,z)
```

5.4 Un exemple complet (du moins les bases)

Pour permettre de faciliter encore plus l'utilisation de Lua dans ARéVi nous avons construit une classe LuaObject qui a pour objectif de fournir à l'utilisateur un moyen simple pour définir le comportement de cet objet en Lua. La classe LuaObject à pour cela une activité qui execute un code Lua contenu dans un fichier qui possède une fonction `run`. Par défaut, cette activité partage deux ArRef (l'objet d'execution et l'activité, puis elle transmet un pas de temps et effectue l'appel protégé de l'execution de `run`. Si l'execution ne se passe pas correctemenent un appel à `checkError` est fait et détermine l'erreur d'execution.

L'initialisation d'un LuaObject contient l'ouverture d'un état Lua et l'ouverture des différentes extension de base de Lua (math, io, string, table,base) mais aussi les extensions utilisateurs (les bindings arevi_lua). Il est possible d'ajouter d'autres extensions en faisant appel à la méthode `openLib` qui prend en argument une fonction d'ouverture de type celle que nous avons créees au paragraphe **??**

6 Documentation des fonctions