Game Developers Need Lua AiR
Static Analysis of Lua Using Interface Models

Paul Klint[1], Loren Roosendaal[2], and Riemer van Rozen[3]

[1] Centrum Wiskunde & Informatica*
[2] IC3D Media
[3] Amsterdam University of Applied Sciences*

Abstract. Game development businesses often choose Lua for separating scripted game logic from reusable engine code. Lua can easily be embedded, has simple interfaces, and offers a powerful and extensible scripting language. Using Lua, developers can create prototypes and scripts at early development stages. However, when larger quantities of engine code and script are available, developers encounter maintainability and quality problems. First, the available automated solutions for interoperability do not take domain-specific optimizations into account. Maintaining a coupling by hand between the Lua interpreter and the engine code, usually in `C++`, is labour intensive and error-prone. Second, assessing the quality of Lua scripts is hard due to a lack of tools that support static analysis. Lua scripts for dynamic analysis only report warnings and errors at run-time and are limited to code coverage. A common solution to the first problem is developing an Interface Definition Language (IDL) from which *"glue code"*, interoperability code between interfaces, is generated automatically. We address quality problems by proposing a method to complement techniques for Lua analysis. We introduce Lua AiR (Lua Analysis in Rascal), a framework for static analysis of Lua script in its embedded context, using IDL models and Rascal.

1 Introduction

Game developers use script languages to develop and maintain game logic separately from game engine libraries. Lua is a script language [1] in the form of an ANSI `C` library[1] developed by Ierusalimschy, de Figuieredo and Celes. During its evolution [2] Lua has gradually matured and has remained light-weight. Moreover, it is an embeddable, minimalistic yet extensible, general purpose, dynamically typed script language. Its language design trade-offs, displayed in Table 1, have shaped the co-development of the embedding API and script features [3]. Lua is popular in game development. Using Lua, developers can quickly create prototypes. However, using Lua also comes at a cost. In later development stages, when larger quantities of engine library code and game script are available, developers encounter maintainability and quality problems.

* This work is part of the EQuA project. `http://www.equaproject.nl/`
[1] `http://www.lua.org/`

Table 1. Lua Design Trade-offs

Lua feature	**Trade-off**	**Mitigating argument**
Light-weight	Large responsibility to its users	Well-defined responsibilities
Dynamic typing	Lack of static type checking	Flexibility of use
Maintainable `C`	Lack of pure speed	Embedded in efficient `C`
General-purpose	Lack of domain-specific features	Extensible script language
Simple	High use of few APIs	Low learning curve
Embeddable	Need for interoperability code	This code can be generated

First, using a standard generator for the coupling between Lua and the game libraries sacrifices speed, and maintaining a hand-written coupling with domain-specific optimizations is labour intensive and error-prone. Second, assessing the quality of Lua scripts is hard due to a lack of tools that support source level *static analysis*, which entails computing information about scripts before run-time. The commonly used Lua scripts for dynamic analysis only report warnings and errors at run-time, and are limited to code coverage. Static analyses do exist, but are mainly applied to intermediate representations in Single Static Assignment (SSA) forms for run-time optimization, e.g. LuaJIT[2] and the run-time specializations of Williams *et al.* [4].

Both problems increase with scale and special measures are necessary for ensuring maintainability and code quality. A common solution to the first problem is to develop an Interface Definition Language (IDL) from which optimized *"glue code"*, interoperability code between interfaces, is generated automatically. However, code quality problems remain to be addressed. Quality problems are not unique to Lua. Blow [5] expresses increased complexity and lack of development tools and White *et al.* [6] describe the need for better script notation. Ramsey and Assis [7] express the need of the Lua community for machine-checkable APIs including types, whole-program and modular static analysis and static type inference. Providing practical methods for static analysis of Lua is hard because static analysis algorithms are subject to a trade-off between speed and precision and developers require exact and immediate feedback during development.

We address script quality problems by proposing a method to complement analysis techniques of Lua. We describe an approach in collaboration with IC3D Media that uses the Rascal Meta-Programming Language[3] [8]. IC3D Media is a Dutch SME located in Breda, active in games for entertainment and training. We introduce Lua Analysis in Rascal (Lua AiR), a framework for static analysis of Lua script in its embedded context, using IDL models.

2 Static Analysis of Lua

We can think of scripts as having many run-time states reachable via possibly many execution paths. Running every program path is infeasible, but abstracting

[2] `http://luajit.org/`

[3] `http://www.rascal-mpl.org/`

Table 2. Static Analysis: Features and Tools

		Tools				
		LDT	**Lua Inspect**	**Lua Checker**	**Lua for IDEA**	**Lua AiR**
	tool characterization	Eclipse IDE	SciTE/VIM plug-in, HTML output	command-line simplifier & checker tools	IntelliJIDEA plug-in	Meta-Framework, Eclipse IDE
	based on	Metalua	Metalua	lex/yacc, Lua	Kahlua	Rascal MPL
	affiliation	Eclipse, Sierra Wireless	David Manura	Google	Jon Akhtar	HvA, CWI, EQuA project
Features	1 globals & locals (definition and use)	yes	yes	strict declare before use	yes	yes
	2 var use must be undefined	no	yes	yes (locals)	yes (locals)	yes
	3 var use may be undefined	no	no	no, disallowed	no	not yet
	4 link a definition to its uses	highlight all occurrences	yes	no	yes	yes (no color)
	5 link a use to its definitions		yes	no	yes	yes (no color)
	4 definition must be unused	no	yes	no	yes	not yet
	5 definition may be unused	no	no	no	no	not yet
	6 duplicate local declaration	no	yes, mask(ed)	yes	no	yes
	7 assignment discards expression	no	no	no	yes "unbalanced no. exps"	yes
	8 assignment implicitly deletes var	no	yes, value nil	no	"unbalanced"	yes
	9 operator applies coercion to operand	no	no	no	no	limited
	10 constant folding	no	"infer value"	no	no	limited
	11 dead code detection	no	not working	no	no	no
	12 type inference	no	"infer value"	no (todo)	"infer nullity"	no
	13 function signature inference	no	not working	no	no, call sites show definition	static IDL

from specific execution states enables us to reason about software properties and to compute them timely. *Static analysis* refers to the extraction of information about states and behavior from a software application without executing it.

Table 2 enumerates static analysis features applicable to Lua and compares existing analysis tools. Features include distinguishing between global and local variables (1) and relating declarations to uses (4), possibly using syntax highlighting. Many of these features (1-6) can be approximated by data flow analysis techniques [9] such as *reaching definitions*, which computes for each program point which assignments can reach it. Others features require no flow analysis (7, 8) or require more advanced inference techniques (9-13). Figure 1 illustrates features described in Table 2 using comments and references (fn), where (a) illustrates features 1, 2, 4, 7, 8 and 9 and (b) shows features 1-3.

```
function f(c)       --(f1) assign function to f
  a       = 1       --(f1) creates global a
  local b = true    --(f1) creates local b
  a, b    = b, a    --(f4) swap a and b
  a, b    = 1,2,3   --(f7) discards 3
  a, b    = c       --(f8) implicitly deletes b
  print(b)          --(f2) nil, undeclared b
end                 --     close scope
f("4")              --(f4) call f,bind c to "4"
print(a)            --(f4) 4, read global a
d = 2 .. a          --(f9) coerces 2 to string
d = d / "12"        --(f9) coerces 12 to number
print(c, d)         --(f2) nil 2, undeclared c
```

(a) Sequential Assignments

```
function hit(self, damage)
  if self.health < damage then
    isDown = true --(f1) assign global
  else
    print(haelth) --(f2) must be undefined
  end
  self.health = self.health - damage
  print(isDown)   --(f3) may be nil or true
  return self
end
unit = {health=10}  --create a test unit
unit = hit(unit, 4) --call hit, bind unit, 4
unit = hit(unit, 8) --call hit, bind unit, 8
```

(b) Event Handler

Fig. 1. Lua Script demonstrating potential errors that static analysis can find

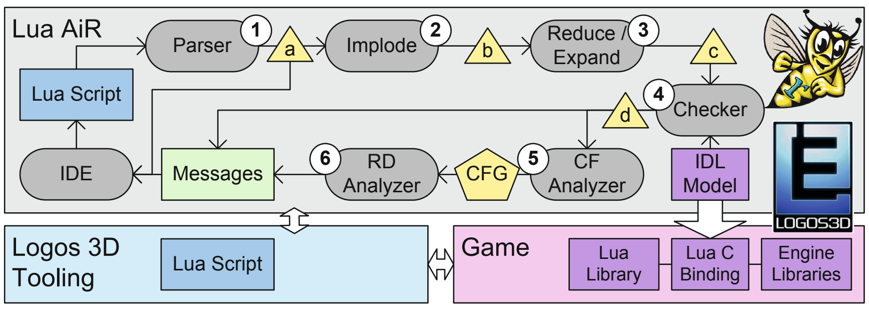

Fig. 2. Lua AiR Model Transformations

Koneki Lua Development Tools (LDT)[4] is an Eclipse plug-in that provides remote debugging and rudimentary static analysis for highlighting and refactoring. Lua Inspect[5] is an experimental static analysis tool that infers values, but incorrectly for conditional assignments. Lua Checker[6] is a basic command-line tool that checks local variable declarations against their uses. Lua for IDEA[7] enriches call sites with API structure (e.g. for World of Warcraft). LDT and Lua Inspect are based on Metalua [10], a Lua extension for static meta-programming.

2.1 Lua AiR Framework

This section explains the approach of the Lua AiR framework. IC3D Media has developed two languages for interoperability between Lua and their Logos3D game engine called Interface Definition Language (IDL) and Interface Generator Language (IGL). IDL defines function signatures and data types. IGL defines the generator format of the mapping between engine functionality defined in IDL models and Lua. Unlike other approaches shown in Table 2, we utilize information from the embedded context in our analysis. Functions and data structures exposed to Lua, and managed by the Logos3D engine, are statically defined and strongly typed. Sharing function signatures and data types modeled in IDL between the embedded environment and Lua enables checking function call site arguments against formal parameter types. Furthermore, it reduces the need for type inference and saves computation time in inter-procedural analysis. Additionally, code documentation can be shared between script proxies and the embedded context.

Lua AiR is a Rascal meta-program that implements the analysis as a pipeline, as illustrated by Figure 2. Rascal generates a specialized Eclipse IDE for Lua editing, highlighting, and static analysis. The analysis consists of the following model transformations. 1) The Lua script under analysis is fed into the parser generated by Rascal from our *Lua Grammar* (130 LOC). This produces a *parse*

[4] `http://www.eclipse.org/koneki/ldt/`
[5] `https://github.com/davidm/lua-inspect`
[6] `http://code.google.com/p/lua-checker/`
[7] `https://bitbucket.org/sylvanaar2/lua-for-idea/wiki/Home`

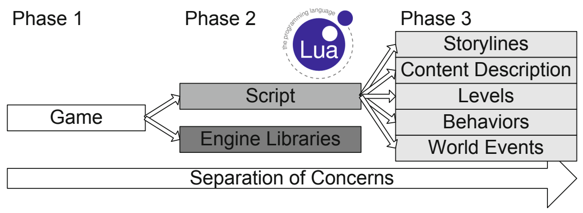

Fig. 3. Increased Complexity requires Separating Concerns

tree (a), depicted as a triangle. 2) The *implode* function matches the nodes of this tree to an Algebraic Data Type (ADT) that represents our Abstract Syntax Tree (AST) (b). This model transformation relies on compatible names and types between the Lua grammar and the ADT. 3) *Reduce* and *Expand* rewrite the AST to simplify the analysis (c). 4) The *Checker* provides static type checking and annotates the AST with scope information (d). 5) Given this AST, the *Control Flow (CF) Analyzer* generates a Control Flow Graph (CFG). 6) The *Reaching Definitions (RD) Analyzer* uses the CFG and performs fixed-point computation over *generate* and *kill* sets to generate the reaching definitions. Finally, the tool displays a log and the view in the IDE is updated by annotating the parse tree with results. The meta-program currently comprises approximately 3 KLOC.

3 Discussion and Future Work

This section discusses problems and describes opportunities for future work.

Empirical Validation. We believe that providing developers with better tools will improve code quality, but we have no proof yet this assumption is correct. Our approach can be validated by verifying if programmers can improve code quality by using our framework.

Improved Precision. Our analysis is *context insensitive* with respect to individual program states and execution paths and lacks type inference. Our analyis can be improved by using context sensitive techniques based generating and evaluating logical constraints [8,9].

Tool Integration. Our framework cannot be used yet by existing tools. We plan to create a *Query API* to interface with other tools.

DLSs. Lua lacks the domain-specific notation which non-programmer game developers need to model software artefacts. A separation of concerns, as shown in Figure 3, is necessary to tackle challenges resulting from increased complexity in game development. Games can be modeled using sets of complementary lightweight little languages, one for each concern, as demonstrated by Palmer [11] and advocated by Furtado [12]. We observe that Lua AiR can be extended to support DLSs for higher level game concerns such as world events, character behavior and mission design, using IDL bindings to check if models conform to the interfaces of their library foundations.

4 Conclusion

In this paper we related Lua script quality problems to a lack of tools that support its static analysis. We evaluated the features of available tools and proposed a method to complement techniques for analysing Lua. We introduced Lua AiR, a framework for static analysis of Lua script in its embedded context, using IDL models and Rascal. Its main goal is to provide the immediate script analysis developers need to improve code quality. Preliminary results show that Lua AiR can provide additional information about Lua scripts. In future case studies we plan to use Lua AiR to analyse existing game code on a larger scale.

Acknowledgements. We thank the reviewers for their constructive comments.

References

1. Ierusalimschy, R.: Programming in Lua, 2nd edn (2006), `Lua.org`
2. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: The Evolution of Lua. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, HOPL III, pp. 2–1–2–26. ACM, New York (2007)
3. Ierusalimschy, R., De Figueiredo, L.H., Celes, W.: Passing a Language through the Eye of a Needle. Commun. ACM 54(7), 38–43 (2011)
4. Williams, K., McCandless, J., Gregg, D.: Dynamic Interpretation for Dynamic Scripting Languages. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2010, pp. 278–287. ACM, New York (2010)
5. Blow, J.: Game Development: Harder Than You Think. ACM Queue 1, 28–37 (2004)
6. White, W., Koch, C., Gehrke, J., Demers, A.: Better Scripts, Better Games. Commun. ACM 52, 42–47 (2009)
7. Ramsey, N., Assis, F.: Almost Good Enough to Scale: A Lua Mail Handler and Spam Filter (presentation slides). In: Lua Workshop (2008)
8. Klint, P., van der Storm, T., Vinju, J.: RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, pp. 168–177. IEEE Computer Society, Washington, DC (2009)
9. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus (1999)
10. Fleutot, F., Tratt, L.: Contrasting Compile-Time Meta-Programming in Metalua and Converge. In: Workshop on Dynamic Languages and Applications (July 2007)
11. Palmer, J.D.: Ficticious: MicroLanguages for Interactive Fiction. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH 2010, pp. 61–68. ACM, New York (2010)
12. Furtado, A.W.B., Santos, A.L.M., Ramalho, G.L.: SharpLudus Revisited: from ad hoc and Monolithic Digital Game DSLs to Effectively Customized DSM Approaches. In: DSM 2011. SPLASH 2011 Workshops, pp. 57–62. ACM, New York (2011)