Scouting Reports

COMP 250RTS

September 29, 2017

Attached you will find the scouting reports, listed in the order in which we will discuss them on Monday. The V8 system lacks an advocate, so it will not get any air time.

Rating	*System*	*Scout*
Strong Pro	Chez Scheme	Brian LaChance
Strong Pro	Erlang	Matt Ahrens
Strong Pro	Lua	Nate Bragg
Strong Pro	Self	Moses Huang
Weak Pro	Squeak	Diogenes Nunez
Weak Con	Dart	Noah Cooper
Con	V8	Ethan Pailes
Strong Con	V8	Karl Cronburg

Chez Scheme Scouting Report

Brian Lachance

What papers would help us understand the system?

The last two papers may not help a ton with technical details of the runtime, but they may help us get our bearings:

1. Hieb, Dybvig, Bruggeman. *Representing Control in the Presence of First-Class Continuations*. PLDI 1990
2. Dybvig, Eby, Bruggeman. *Don't Stop the BIBOP: Flexible and Efficient Storage Management for Dynamically Typed Languages*. TR-400 Indiana Computer Science Department.
3. Dybvig, Bruggeman, Eby. *Guardians in a Generation-Based Garbage Collector*. PLDI 1993
4. Hilsdale, Ashley, Dybvig, Friedman. *Compiler Construction Using Scheme*. FPLE 1995
5. Dybvig. *The Development of Chez Scheme*. ICFP 2006

Each one of these can be found online at `https://www.cs.indiana.edu/chezscheme/pubs/`.

What are the components of the system? How big is each one?

Below, I focus on the major parts of Chez's runtime that were implemented in C. The other parts of the runtime, which are implemented in Chez Scheme, I think all refer to the C parts for support. In total, the C code I found is around 15,000 lines of code.

Storage management 3,299 lines of code

Continuations, signals 701 lines of code

Numeric primitives 1,531 lines of code

Other primitives 2,851 lines of code

Threading, synchronization 367 lines of code

What components appear especially interesting or well done?

Storage management The implementation looks small enough for us to study. It's different enough from Appel's generational collector to be interesting without being so different that we would burn a month getting our feet wet.

Continuations, signals The segmented stacks are a new idea to me and they appear to be why `call/cc` isn't horrendously slow (which benefits the features that Chez builds atop `call/cc`). Segmented stacks apparently also simplified multi-threaded support.

Numeric primitives Scheme provides a richer set of numbers than in most languages and it would be interesting to see what has been done (in either the compiler or the runtime) to make them efficient.

What should our studies focus on?

I think we should focus on three things:

1. How the storage management system can stay simple and performant despite its differences from the designs we've seen so far
2. How the compiler can use the stack (as opposed to the heap) to handle all call frames
3. How the compiler and runtime cooperate to support foreign calls

How do you rate the system? Why?

I give Chez Scheme a **strong pro**; I am willing to argue for why we should study this system.

My rating is in part because the authors chose designs that payed off in the end. For example, supporting multi-threaded code was apparently easier because of decisions in stack and heap representation that were motivated by unrelated features.

Further, if we do have to dig into the compiler, I am hopeful that the nanopass architecture will help us.

Although the system doesn't appear to have a formal developer's guide to motivate design choices, I've enjoyed reading the next the research papers as a stand-in for such a guide. As for orienting oneself with the code-base, I think Chez could do better here; thankfully, the C code base is at least easy to `grep`.

Scouting Erlang: BEAM VM

Matthew Ahrens

September 28, 2017

1 BEAM Overview and Outline

BEAM is the virtual machine for languages that support lightweight processes like Erlang, Elixir, and Lisp Flavored Erlang (LFE). BEAM describes a specific instance of the general specification for an Erlang Runtime System (ERTS). Studying BEAM would allow us to better understand GC, Parallelism and process scheduling, and interprocess communication via message passing among other topics.

2 Papers and Documentation

After reaching out to the Erlang and Elixir community, I found two great documentation resources on BEAM.

- Spwaned Shelter's collections of Erlang papers
- The BEAM Book, an open source book maintained by Erik Stenmen

From this pool of material, I wish to highlight the specific papers and sections of the book and the topics they will facilitate.

2.1 Garbage collection

2.1.1 Why Garbage Collection Matters in Erlang

In the blog post, Erlang Garbage Collection Details and Why It Matters, Hamidreza motivates why Erlang has two GC algorithms in play, one local to each process and one for the shared heap. This post is a small, easy to digest way of understanding how language design decisions, like independent processes, can lead to optimizations in other parts of the RTS, like GC.

2.1.2 How Garbage Collection works in the latest Erlang (v19

In the blog post, Erlang 19.0 Garbage Collector, Lukas describes the generational copying garbage collector employed by BEAM and clarifies the boxed/unboxed and other value representations that we were confused about when postulating how Appel's system would apply to Erlang. In the final parts of the blog post, Lukas describes the specific considerations of GC on inter-process messages since messages are allocated. This seems like an interesting connection between message-passing concurrency and GC.

2.2 Concurrency and Parallelism

2.2.1 Process and Thread Scheduling

In How Erlang does scheduling, Jesper gives the motivation and high level

view of how BEAM compares to other languages when typically using an OS thread per core to schedule Erlang processes (user threads).

In The BEAM Book chapter on scheduling we see a more general view of the scheduler than Jesper's description, and also some insight onto the work stealing algorithm used.

2.2.2 Message Passing

In the Processes and the BEAM instructions sections of the BEAM Book, we see the method by which the high level message passing and process management expressions (spawn, send, receive) are handled by BEAM.

3 Components of the BEAM

BEAM has the following Major components:

- Tag Recognition
- Instruction Decoder (Dispatcher)
- Process Scheduler(s)
- Garbage Collector(s)

4 Metrics on Components and Source

Estimates from the source: `https://github.com/erlang/otp/tree/master/erts/emulator/beam`

Interesting Component	LoC
Scheduler(s)	>21118
Garbage Collector(s)	>3831
BEAM overall	219700

5 Interesting Components

I believe we should focus on *process scheduling* and *message passing*. If the GC minded folks in our group would like to see a two-algorithm GC, then it may be of use to them to look at BEAM's GC algorithms after understanding how processes are scheduled, run, and communicate.

6 Rating and Conclusion

I rate BEAM and the general idea of the ERTS **strongly pro**. My reasoning is that parallelism and concurrency are very interesting to our group and because BEAM has work stealing, message passing, and some considerations for distributed computation which Joe Armstrong puts in direct contrast to RPCs, it is a good system to study. My only reservation is that BEAM is very large, and so it will take some preprocessing to look at code that is specifically relevant to one component.

Scouting Run-Time Systems: Lua
COMP 250RTS

Nate Bragg

September 26, 2017

Introduction

Lua presents a tempting target for study. It is relatively feature-rich, while remaining small and straightforward. Lua is a scripting language, with many features common to that class of language. As in other scripting languages, it is dynamically typed, and the main data type is an associative array. It is embeddable, with a clean FFI. It straddles the line between procedural, object oriented, and functional. It supports dynamic compilation, and features a rich debugging and profiling interface, a garbage collector, modules, coroutines, and limited supported for exceptions.

Lua has eight basic types: nil, boolean, number, string, function, userdata, thread, and table. Numbers can be represented either by floats or integers. Strings are immutable and interned. Thread, a misnomer, is a backing structure for coroutines. Tables (associative arrays) are the only data-structuring mechanism.

Papers that would help us understand Lua

There is relatively extensive literature on Lua, both for the user and on the language itself.

The authoritative book, *Programming in Lua* [Ierusalimschy, 2016] by Lua's chief architect is highly regarded. The manual [Ierusalimschy et al., 2017] is extensive and informative; I referred to it continuously while preparing this scouting report. I also found a light-weight architecture overview [Glasberg and Bresler, 2006].

The main academic paper serves as an introduction [Ierusalimschy et al., 1996], and the paper covering Lua 5.0 serves as a reintroduction [Ierusalimschy et al., 2005]. The authors also discuss Lua's evolution over time [Ierusalimschy et al., 2007].

Some special topic papers may also be of interest, covering first-class functions [Ierusalimschy, 2017], coroutines [de Moura et al., 2004] and message passing [Skyrme et al., 2008].

Some other fun-looking papers I encountered that may or may not be helpful for our purposes include embedding Lua into functional languages [Ramsey, 2003] and operating systems [Vieira Neto et al., 2014], and adding static types [Maidl et al., 2014, 2015].

Lua's components

Below, I break down the components of Lua that are most relevant to the run-time system. I separate these into four groups: data types, run-time core, memory, and key libraries. Each module is listed along with the number of lines in the file as reported by `wc -l`, tabulated per group, as well as a brief description of what is interesting about that module. Everything else is lumped together; this includes the lexer, parser, code generator, front end driver for the interpreter and compiler, as well as the remaining standard libraries, and various odds-and-ends.

Data Types	1678	
`lobject.h`	549	Data Definitions
`ltable.c`	669	Tables
`lstring.c`	248	Strings
`lfunc.{h,c}`	212	Closures
Run-Time Core	4301	
`lstate.{h,c}`	582	Run-Time State, Threads
`lopcodes.h`	297	Instructions
`lvm.c`	1322	Virtual Machine
`ldo.c`	802	Exceptions, Stacks, Calls, Coroutines
`lapi.c`	1298	C Interface
Memory	1347	
`lmem.{h,c}`	169	Memory Management
`lgc.c`	1178	Garbage Collector
Key Libraries	2627	
`lbaselib.c`	498	Basic Library
`lcorolib.c`	168	Coroutine Library
`lua.h`	17	Debug Library
`ldblib.c`	456	Debug Library
`ldebug.c`	698	Debug Library
`loadlib.c`	790	Package Library
Total	9953	
Everything Else	14350	

Especially interesting and well done components

All four groups listed are interesting, and should be covered in approximately the order given. Of these, the most interesting ones are undoubtedly `lvm.c`, `ldo.c`, `lgc.c`, `lstate.c`, and `lapi.c`. The others I include for context. More than anything else in a program, I care about data definitions because they are the fulcrum on which everything else hinges. Likewise, how the language core is exposed to a user's program through the standard library is informative about intent and structure.

In each group, there are certain key types and functions. For data definitions, these include `Value`, `TString`, `Closure`, `Table`, etc. Run-time state is built around `global_State` and `lua_State`. The virtual machine is best understood by considering `OpCode` and `luaV_execute`. The garbage collector's main entry point is `luaC_fullgc`.

The virtual machine is a register machine, redesigned from an earlier stack machine. It has with 47 opcodes, including arithmetic and logical operations, comparisons, control flow, and various table and closure primitives.

The garbage collector is mark-and-sweep, but in the past was optionally generational. It exposes its settings and controls to the user's program, including setting the collector's aggressiveness, as well as starting, stopping, and stepping the collector, and first-class control over whether a reference is strong, weak, or what Lua calls "ephemeron," a special type of reference for tables with weak keys.

As for the libraries, I include the basic library both as a window into the core built-in functions, and that it is a consummate example of Lua's FFI. The coroutine, debug, and package libraries integrate tightly with the rest of the run time. I mention `lua.h` for the sake of `lua_Debug`.

What we should focus on if we study Lua

The *Scouting Run-Time Systems* handout states that for Lua, the focus may be "[w]hat we can steal to put into more ambitious systems." While this is a perfectly viable focus, I want to additionally propose that Lua is a great case study in run-time evolution and experimentation under the pressure staying true to a set of core guiding constraints.

Not only have there been the above-mentioned relatively-recent notable VM and GC developments, but over its life has added types (including very recently, integers), interfaces like debug, and features like coroutines and modules. It's authors have had to strike a balance: when to extend the language, and when to cut back; when to add syntax, and when to add a library. At the same time, some parts of the design, like the FFI, have been stable since inception.

Rating

While another candidate language may better fit our purposes, Lua holds up well to the proposed ideals. Let's review.

Lua is clear. My audit of the source code only took about 4 hours, and I came away with a healthy respect for its design and the beginnings of insight into its function. Its modules are low-coupling, and high-cohesion.

Lua is simple. This is not a language rife with corner cases. The execution model and abstractions are easy to hold in your head, and this is reflected in the code, whose modules are to the point, with very few fancy tricks.

Lua is performant. Its performance is even better with the register-based VM, the improvement of which is covered thoroughly by the authors.

Lua is well documented. Both in books and on the web, there are ample resources for learning Lua. The code shares a similar spirit of thorough documentation, and comments are found at key points throughout.

Lua provides a lot of useful abstractions. In addition to the ones mentioned previously, Lua includes regular expressions (extended with brace-matching), syntactic sugar for object-oriented code, first-class functions, iterators, powerful reflection facilities, and much more.

Lua is popular. Okay, this one might be up for debate, since by some metrics Lua is only somewhat popular. On TIOBE [TIOBE, 2017], Lua ranks at a lukewarm #35, after such masterpieces as SAS and Visual FoxPro. On PYPL [Carbonnelle, 2017] and Redmonk [O'Grady, 2017] however, Lua ranks at #20, and at IEEE [Diakopoulos and Cass, 2017], #21. The authors write [Ierusalimschy et al., 2007]:

> Lua has been used successfully in many large companies, such as Adobe, Bombardier, Disney, Electronic Arts, Intel, LucasArts, Microsoft, Nasa, Olivetti, and Philips. [...]
>
> Lua has been especially successful in games. It was said recently that "Lua is rapidly becoming the de facto standard for game scripting." Two informal polls con- ducted by gamedev.net [...] showed Lua as the most popular scripting language for game development.

If I have to admit a criticism, it is perhaps that no single piece of Lua is flashy or exciting. While a good example of design and implementation, Lua is not a crucible of innovation, driving the state of the art in run-time system concepts. One of its only innovations was the addition of a register machine; it otherwise conservatively concerns itself with solid implementations of ideas invented elsewhere. However, this is only a criticism through a certain lens, and may also be considered a point in Lua's favor. I digress.

As an additional note, studying Lua will mean largely foregoing studying parallelism; while it provides concurrency through coroutines, Lua itself does not include multi-threading.

Altogether, Lua presents a tight case for its consideration as a candidate for study. As such, I rate Lua as a strong pro.

References

Pierre Carbonnelle. Pypl popularity of programming language. http://pypl.github.io/PYPL.html, 2017. Retrieved: 2017-09-24.

Ana Lcia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in lua. *Journal of Universal Computer Science*, 10(7):910–925, jul 2004. http://www.jucs.org/jucs_10_7/coroutines_in_lua.

Nick Diakopoulos and Stephen Cass. Interactive: The top programming languages 2017. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017, 2017. Retrieved: 2017-09-24.

Mark Stroetzel Glasberg and Jim Bresler. The lua architecture. *Advanced Topics in Software Engineering*, 2006.

Roberto Ierusalimschy. *Programming in Lua, Fourth Edition*. Lua.Org, 2016. ISBN 8590379868, 9788590379867.

Roberto Ierusalimschy. First-class functions in an imperative world. *Journal of Universal Computer Science*, 23(1):112–126, jan 2017. http://www.jucs.org/jucs_23_1/first_class_functions_in.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua-an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996. doi: 10.1002/(SICI)1097-024X(199606)26:6⟨635::AID-SPE26⟩3.0.CO;2-P. URL https://doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, jul 2005. http://www.jucs.org/jucs_11_7/the_implementation_of_lua.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 2–1–2–26, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. doi: 10.1145/1238844.1238846. URL http://doi.acm.org/10.1145/1238844.1238846.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Lua 5.3 reference manual. https://www.lua.org/manual/5.3/manual.html, 2017. Retrieved: 2017-09-24.

André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed lua: An optional type system for lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 3:1–3:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2916-3. doi: 10.1145/2617548.2617553. URL http://doi.acm.org/10.1145/2617548.2617553.

André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. A formalization of typed lua. *SIGPLAN Not.*, 51(2):13–25, October 2015. ISSN 0362-1340. doi: 10.1145/2936313.2816709. URL http://doi.acm.org/10.1145/2936313.2816709.

Stephen O'Grady. The redmonk programming language rankings: June 2017. http://redmonk.com/sogrady/2017/06/08/language-rankings-6-17/, 2017. Retrieved: 2017-09-24.

Norman Ramsey. Embedding an interpreted language using higher-order functions and types. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03, pages 6–14, New York, NY, USA, 2003. ACM. ISBN 1-58113-655-2. doi: 10.1145/858570.858571. URL http://doi.acm.org/10.1145/858570.858571.

Alexandre Skyrme, Noemi Rodriguez, and Roberto Ierusalimschy. Exploring lua for concurrent programming. *Journal of Universal Computer Science*, 14 (21):3556–3572, dec 2008. http://www.jucs.org/jucs_14_21/exploring_lua_for_concurrent.

TIOBE. Tiobe index for september 2017. https://www.tiobe.com/tiobe-index/, 2017. Retrieved: 2017-09-24.

Lourival Vieira Neto, Roberto Ierusalimschy, Ana Lúcia de Moura, and Marc Balmer. Scriptable operating systems with lua. *SIGPLAN Not.*, 50(2):2–10, October 2014. ISSN 0362-1340. doi: 10.1145/2775052.2661096. URL http://doi.acm.org/10.1145/2775052.2661096.

Scouting Report : SELF

Moses Huang

September 28, 2017

Resources to consult

- "Self" Handbook - `http://handbook.selflanguage.org/2017.1/index.html`
- Source code on Github - `https://github.com/russellallen/self`
- Introductory video on the use of the language and the UI - `https://youtu.be/Ox5P7QyL774`
- 1987 Video where David Ungar talks about the design decisions in the language implementation. Watch from 36:00 - 49:30 - `https://youtu.be/3ka4KY7TMTU`

What papers would help us understand the system

- Introduction to the Self Language [2]
- 20-year Retrospective [1]
- Craig's thesis detailing most of the original ideas: `https://www.cs.ucsb.edu/~urs/oocsb/self/papers/craig-thesis.html`
- Urs's thesis detailing the techniques that make Self performant (focus on chapters 3,4,5,6) - `http://hoelzle.org/publications/urs-thesis.pdf`

What are the components of the system and how big is each component?

- Garbage Collector - Generational Scavenging.
- Non-inlining compiler (fast but produces non-optimized code) [2.6k C++ code]
- Profiler
- Optimizing compiler (slow but produces optimized code) [11k C++ code]

What components appear to be especially interesting or well done?

- Garbage Collector - Generational Scavenging
- Non-inlining compiler (NIC) - fast but produces non-optimized code.
- Dynamic deoptimization to support source level debugging

If we study this system, what should we focus on?

- Interaction between profiler and NIC and the recompilation process.
- Dynamic deoptimization

Using the four-point scale described below, how do you rate the system as a candidate for study? What are the reasons for your rating?

Strong Pro - The ideas in Self of dynamic optimization heavily influenced Java in the Hotspot VM and is currently the basis of many dynamic object oriented languages that we have today. Studying Self will give us a good baseline to evaluate future design approaches for object oriented languages.

References

[1] Craig Chambers and David Ungar. A retrospective on: "customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language". *SIGPLAN Not.*, 39(4):295–312, April 2004.

[2] David Ungar and Randall B. Smith. Self: The power of simplicity. *LISP and Symbolic Computation*, 4(3):187–205, Jul 1991.

Scouting Squeak: A Smalltalk VM

Diogenes Nunez

Recommendation

Squeak is mostly written in Smalltalk, a high-level language. The benefits and drawbacks of writing a VM in a language like Smalltalk are the most interesting part of the system. This paper by Andrew Greenberg describes an interesting benefit: extending the VM in a high-level language.

- Extending the Squeak VM: `http://stephane.ducasse.free.fr/FreeBooks/CollectiveNBlueBook/greenberg.pdf`

I would give this system a *weak pro*. Since the VM is written in mostly Smalltalk, Squeak was easier to read and reason about in my short foray than other VMs. However, the VM's interesting or familiar components call primitives, which I could only find as translated C code instead of Smalltalk code, the selling point of Squeak. The related reading below describes the specification of the system, but do not work well as documentation alongside a Squeak implementation. The reading describes a general Smalltalk runtime system or an older version of the Squeak VM. Ultimately, this lack of direct documentation is why I gave Squeak a *weak* rating instead of a *Strong* rating.

Components

The components of Squeak are listed below. Interesting components are *emphasized*. Line numbers are estimated from browsing the VM code in the latest Squeak 6.0 alpha and the C code in an old Squeak 4.0 source code repository.

- Interpreter: Split between Smalltalk and C (where the primitives are implemented). I could not discern the Smalltalk implementation of the interpreter. The interpreter is about 30k lines of C translated from Smalltalk.
- *Object Memory*: Squeak's object layout as well as the *become* functionality. From what I can discern, about 100 lines of Smalltalk from the ImageSegmentLoader are part of the become system. Allocation, which is part of the storage management below, should cover how Squeak uses tags for objects.

- *Storage Management*: Squeak's memory management subsystem. Smalltalk with primitive calls. The primitive calls have been translated into C. About 200 lines of Smalltalk and about 550 lines of translated C in the interpreter (primitives for collection). Looking through the Smalltalk code, the Smalltalk code is split between allocation and collection. Allocation is contained in the ImageSegmentLoader in the System-Object Storage module. Collection code is contained in the SmalltalkImage class as part of the System-Support module.
- Sound System: Squeak's audio system, complete with synthesizers.
- BitBlt and WarpBlt: Squeak's graphic system.

Related Reading

- Back to the Future: `http://worrydream.com/refs/Ingalls%20-%20Back%20to%20the%20Future.pdf`
- Back to the Future Once Again: Look at whether Smalltalk meta-circularity helped or hindered
- Blue Book: Relevant part available here: `http://www.mirandabanda.org/bluebook/`. Ingalls et al. refers to this book. Part 4 (the chapters linked on the page) describes the formal specification for a Smalltalk system, which Ingalls et al. implemented in the above papers.

One book worth noting, albeit not about Squeak, is *Bits of History*. This book, also called the "Green Book" describes different implementations of the Smalltalk 80 runtime system and choices that must be made to implement said system.

Scouting Report - Dart Language Runtime

Noah Cooper
COMP 250RTS
September 28, 2017

1 Introduction

Dart is a cross-platform, general-purpose language first released by Google in 2011. Google offers a JavaScript transpiler for Dart, a fully ahead-of-time machine code compiler, and a Dart VM[1]. The Dart VM is the subject of this report.

Dart is garbage collected and optionally typed. It has exceptions. It is unicode-aware. It features a message-passing concurrency model in which *isolates* are the basic unit of execution. Isolates communicate by passing *object snapshots*.

2 Compiler: Types and Snapshots

The current stable version of Dart is optionally typed. Static type annotations are available, which prompt the VM to generate runtime assertions when run in *checked mode*[2]. An experimental sound static type system is available in *strong mode*[3], under development and optional for the current Dart, and planned as mandatory for Dart 2.0.

The snapshot format which serializes objects for message passing also packages code at compile time. *Script snapshots* are architecture-agnostic single-file images. *Application snapshots* include JIT-compiled code generated during a training run of the program on the VM. The goal of these two optional compilation tasks is to reduce VM startup time on the target program[4].

3 Runtime Internals

The Dart VM[5] is written in object-oriented C++. It's highly modular, comprising the components described later in this section, plus over 50 supporting source files and their corresponding headers. While typical file lengths range from hundreds to thousands of LOC, functions are usually 10-30 LOC, and lines are short and consistently indented. The style is quite literate, which compensates for the sparsity of comments in some files. Inline assertions and prolific unit test classes help document the code.

The runtime being object-oriented and modular, snippets are hard to understand when read in isolation. Custom classes are used extensively; and, literate though the coding style may be, the code is often opaque when you do not understand the supporting classes' interfaces. This fact may make Dart a unattractive subject for a seminar consisting of 75-minute case studies, although, with most of a semester available to study Dart, we may have time to understand these classes well, and the modularity may become an advantage.

Some important files and their line counts, including comments:

dart.cc
: [758 LOC] Bootstraps the Dart VM. Dispatches isolates to OS threads, including the mandatory main isolate. Configuration-dependent global processing (snapshot type, if any; type-checking mode; architecture-specific flags). Clear and well-commented.

debugger.cc
: [4378 LOC] Functions for breakpoint setting and handling, activation frame accounting, introspection and code printing. The functional decomposition is easy to understand and results in relatively short functions. Style is very literate, and the code is well-commented in places, and sometimes extensively commented.

exceptions.cc
: [962 LOC] Very well-documented exceptions class. The `ExceptionHandlerFinder` function could not be clearer or better-commented.

gc_marker.cc
: [764 LOC] The "mark" of mark-and-sweep garbage collection. Some differential handling of weak versus standard references.

gc_sweeper.cc
: [180 LOC] The GC sweeper.

snapshot.cc
: [2006 LOC] For the snapshot facility described above. Lots of fine pointer math and assembly code emission, all subject to architecture-dependent conditions, and other, cryptic `#ifdef`s. Mostly uncommented.

dwarf.cc
: [638 LOC] DWARF interface. Opaque, for the same reasons **snapshot.cc** is, but with the benefit of extensive comments.

unicode.cc
: [275 LOC] Unicode support library. Short and simple-looking. Probably not a very rich area for research, but it is interesting to know what it takes to support unicode in a runtime, and it may be practically all in one place here.

4 Research

I searched the ACM Digital Library for Dart articles, as well as the archives of OOPSLA/SPLASH. Type safety in Dart is apparently a popular topic — what you can prove at compile time, how an improved compiler might make stronger guarantees than checked mode provides, etc. A typical example is "Message safety in Dart" (Ernst, et al. 2015). (Perhaps Google's development of strong mode is a response to this trend.)

Unfortunately, I found practically no promising technical material on the runtime. "How Dart learned from past object-oriented systems" (Bak 2015) touches on snapshots, but it is very broad. "A SIMD programming model for dart, javascript,and other dynamically typed scripting languages" [sic] (McCutchan, et al. 2014) dives into the Dart VM, and might constitute some good material on JIT compilation, but it is much more experimental than descriptive.

5 Recommendation

Snapshots are interesting. So is Dart's concurrency model — although it seems similar to Erlang's, which I bet is very well understood. My suspicion is that Dart's runtime is otherwise not very innovative. It is marketed as a safe, familiar general-purpose language, so it'd make sense that the runtime architecture be conservative. So if the runtime is not widely studied, that may be why. (It is also a relatively new language.)

The source code being as nicely organized and readable as it is, the Dart VM may serve as a reference for competent implementation of classic runtime system concepts. There just isn't much literature on it yet, though, nor many ideas that cannot be studied in other runtimes on which more has been written.

$\Longrightarrow$ **Rating:** Weak con.

References

[1] Wikipedia, "Dart (programming language)", https://en.wikipedia.org/wiki/Dart_%28programming_language%29, retrieved September 25, 2017.

[2] Dart website, "Runtime Modes", https://www.dartlang.org/resources/dart-tips/dart-tips-ep-2, retrieved September 25, 2017.

[3] Dart website, "Strong Mode Dart: FAQ", https://www.dartlang.org/guides/language/sound-faq, retrieved September 25, 2017.

[4] Dart wiki, "Snapshots", https://github.com/dart-lang/sdk/wiki/Snapshots, retrieved September 28, 2017.

[5] Dart VM source code, https://github.com/dart-lang/sdk/tree/master/runtime/vm, as of commit ce427ac14ff1e261d56d0424794efc6cf2338541.

V8 Scouting Report

Karl Cronburg

September 21st, 2017

In this report I will answer the following:

- What papers would help us understand V8?
- What are the components of V8?
- Measured in lines of code, how big is each V8 component?
- What V8 components appear to be especially interesting or well done?
- What topics should we focus on if we study V8?

V8 Components & Interestingness

Component	Lines of Code (LoC)
Heap management	30k
TurboFan JIT Compiler	100k

The V8 mutator accesses properties of an object with near-zero overhead by dynamically creating "hidden" classes on demand whenever new properties are added to an object (similar to maps in Self).

V8 has dynamic code patching of mutator accesses to object properties based on the associated hidden class.

V8 has a stop-the-world generational garbage collector. From the description given in V8's source code documentation, the garbage collector is a fairly straightforward implementation of a generational collector with many of the standard optimizations present. Namely:

> V8 compiles and executes JavaScript source code, handles memory allocation for objects, and garbage collects objects it no longer needs. V8's stop-the-world, generational, accurate garbage collector is one of the keys to V8's performance.

Papers and Documentation

- Most useful for our purposes: `https://github.com/v8/v8/wiki/Design-Elements`
- JIT compiler: `https://v8project.blogspot.de/2015/07/digging-into-turbofan-jit.html`

From Andreas Rossberg, the tech lead of the V8 language team, said the following on stackoverflow:

> Unfortunately, there aren't many scientific papers about V8's implementation. The only proper one is a recent paper on ISMM about allocation folding. I wish there were more, but the reality is that the V8 team doesn't get paid to write papers, even if we'd like to.
>
> There also have been a couple of recent V8-related papers from other sources. Unfortunately, they have to rely on second-guessing V8's sources and underlying design decisions, and I would recommend taking the conclusions and results reached in such papers with several grains of salt.

(`https://stackoverflow.com/a/24593591/1542000`)

The following are recent (really the only) publications involving working on the V8 memory subsystem by Ben Titzer:

- Allocation Folding Based on Dominance (memory allocation optimization)
- Memento Mori: Dynamic Allocation-site-based Optimizations (garbage collection statistics collection and tenuring strategies)

V8 Components

If we were to study V8 we could focus on how the JIT compiler uses a "sea-of-nodes" representation for SSA form of a program. This is also used in Hotspot. A useful paper to read in this vain would be "A Simple Graph-Based Intermediate Representation". However, this work is more interesting on the compiler side of things (representation of an AST) though the approach does seem to require runtime support for features like "fluid" moving of code.

Overall, I do not recommend (strong con) we study V8 because:

- There are only a small number of academic-oriented contributions which focus on the novelty of very specific kinds of optimizations in the context of V8 rather than presenting the over-arching design philsophy of V8 from a strictly principled view.
- Documentation of the *design* of the system is largely lacking and itself defers to papers about Self.
- Andreas himself seems to admit the design philosophy of V8 is to pull in optimization ideas from other runtime systems and to get them to fit into V8. The engineering feats in pulling this off merit a course in their own right, but do not particularly match the focus of this class.

V8 Scouting Report

Ethan Pailes
Tufts University

1 Recommendation: Strong Con

V8 is a beast of a system with at least 3 different compiler pipelines, each of which uses 2 different compilers. Documentation and design documents run light, and what does exist tends to focus on the JIT pipeline rather than the run time system.

2 Resources for Learning about V8

The best resources for learning about v8 are found on the v8 wiki [1]. The most useful page being the Design Elements page [2], which outlines the key optimizations which deliver performance for v8. The page contains a few short paragraphs on the garbage collector, but otherwise very little information on the RTS. Discussion of how the event loop is implemented is conspicuously missing.

The rest of the Under the Hood section does contain several talks on the compiler pipeline. If we wanted to focus on how JITs work, v8 would make a stronger candidate based on the existence of this material.

The v8 source code is the other main resource for learning about the engine. The source is cleanly written in idiomatic C++, but the volume of source makes it difficult to develop a high level view of what is going on through source inspection alone. I was able to build from source on my mac quite easily, but v8's custom build system is sufficiently complex that the atypical python setup on my Arch Linux box made building difficult. I did not attempt to read the code with the aid of a debugger, but that might make the volume of source more manageable.

3 System Components

The v8 pipeline begins with parsing and then threads through two different stages of compilation. A fast-codegen stage produces unoptimized machine code, and a full compiler produces optimized machine code. Any stage of the compiler pipeline might cycle back to a previous stage (v8 uses a lazy-compilation strategy and so it has to re-parse code that gets invoked after the initial top-level pass). The garbage collector is a stop-the-world, generational garbage collection. One of the biggest services javascript brings to the table is the ability to sandbox different js vms, something that is required for executing in the unprivileged browser environment. The v8 runtime accomplishes this via the Isolate class. I was unable to easily find a good overview of the event loop, or the implementation of the core web APIs.

The whole project contains 2,435,720 SLOC, according to David A. Wheeler's 'SLOCCount', most of which is in the tools and test directories with the compiler following. Getting a more accurate per-component SLOC count for the components we care about would take more time than was supposed to be spent on scouting.

4 Things to Focus on

If we were to study v8 in detail, I think focusing on either the JIT or the sandboxing would be the way to go. v8 is a best-in-class JIT, so it has that going for it, but there are probably simpler JIT systems out there that would do a better job of teaching us the core insights of a JIT (@Self). On the other hand, javascript's sandboxing is not that common of a feature, so it might be cool to look at how the security invariants are maintained.

References

[1] https://github.com/v8/v8/wiki

[2] https://github.com/v8/v8/wiki/Design-Elements