Some thoughts on LLVM vs. libjit (https://eli.thegreenplace.net/2014/01/15/some-thoughts-on-llvm-vs-libjit)

January 15, 2014 at 05:49 **Tags** Code generation (https://eli.thegreenplace.net/tag/code-generation) , Compilation (https://eli.thegreenplace.net/tag/compilation)

Having recently completed a series of articles (https://eli.thegreenplace.net/2014/01/07/getting-started-with-libjit-part-3/) on using libjit and understanding how it works, I couldn't stop comparing it to LLVM inside my head. This is hardly surprising, since LLVM has been a significant part of my professional life for the past 3.5 years, and will remain in this position in the foreseeable future.

So this post is some unfiltered thoughts on libjit and LLVM - what's similar, what's different, and which one is more suitable for new projects.

First, a tiny bit of background. LLVM hardly needs introduction for the readers of my blog, but just for completeness - it was first released (http://llvm.org/releases/) in late 2003. However, according to Chris Lattner's (one of the creators of LLVM and its main developer for many years) recent presentation, the first ideas for it came together in 2000 and the first prototype was hacked together in early 2001.

Today, LLVM is one of the most significant infrastructural backbones of modern computing, no less. It's the default compiler for the OS X and iOS platforms (FreeBSD is deprecating gcc in favor of LLVM & Clang too), and is heavily used in production by almost all major software companies. New things pop up all the time, like the (Windows-based, mind you) toolchain of the new Sony PS4 being completely built on top of LLVM.

libjit's first release was in April 2004 (http://www.dotgnu.org/), as part of the now-defunct DotGNU project. This was a much less complete release than LLVM 1.0, though (version 0.0.0f - doesn't sound very reassuring for running your nuclear reactor, does it?). Its goals, from the beginning, were quite similar to LLVM's, if somewhat more modest. It also aimed to provide the backend of a compiler from a target-independent IR.

Some internet archaeology brought up this fascinating mailing list thread (http://lists.gnu.org/archive/html/dotgnu-libjit/2004-05/msg00012.html) from May 2004, in which Chris Lattner asks Rhys Weatherley (the creator of libjit) whether the two projects can "join forces", since their goals are similar. If you have any interest in open-source, you can't miss this discussion - go read it now. After a few weak justification attempts, Chris and Rhys got to the real issue. In open-source, people often start new projects *just because it's fun*, as well as *to retain complete control*. There may be an existing project that does something similar. The effort to use this existing project is, in all likeness, smaller than the effort to roll your own. But what would you prefer - spending time adapting a large body of existing code you didn't write (and whose brace style you hate!) to your needs and quibbling over methodology on mailing lists, or gloriously hacking into the night - your fingers burning holes in the keyboard producing new code? Especially when this isn't a "day job", but something you do for fun... Yeah, I thought so. Joel's In Defense of the NIH syndrome (http://www.joelonsoftware.com/articles/fog0000000007.html) is a classic now (it's 13 years old, oh my) but it rings as true, and possibly truer, for open-source as for the corporate setting Joel first aimed it at.

But I digress... The short version of the paragraph above is that libjit and LLVM really do have similar goals, and if the mood of their overlords had been different on a few pleasant spring nights of 2004, we could've had just a single project today; that would, in all likeness, be LLVM. Could this also help the DotGNU project on the whole, which lost more and more steam over the years until its official abandonment in 2012? Who knows... If you have a time machine, let me know please.

But lo and behold - libjit is still alive today. It seems to live in a state of life support because there's not too much active development going on, but it's definitely not dead. Its current maintainer is fairly responsive and seems keen to fix things, including documentation.

As I mentioned, libjit's initial goals were much less ambitious than LLVM's, and it remained so. LLVM's philosophy of "everything is a pluggable library" kept driving its development, and if you look at LLVM today - everything is indeed quite pluggable, including newer parts like MC. LLVM supports pluggability in its very core. The so called mid-level IR infrastructure (arguably LLVM's best-designed part) is pluggability executed to perfection. You write generic passes that analyze and transform IR, and hook them together in any way you see fit (along with dozens of industrial-strength compilation passes already provided by LLVM). But the backend parts too: you can choose from more than one instruction selector (or write your own), multiple register allocators, schedulers, code emitters, assemblers, and so on.

libjit is not like that. While it was designed to be able to emit code for multiple architectures (it has both flavors of x86, some experimental ARM support as well as old "attic" code for Alpha), its internals are not modular; sure, you can add more optimizations (or a different register allocator) to libjit, but you'll have to hack it into libjit itself - the requisite APIs and data structures are not really exposed on the library level, the way they are with LLVM.

The libjit IR is also somewhat more limited, having been designed with a single front-end in mind (DotGNU). True, the designer aimed to make it fairly abstract, but some things are definitely missing compared to LLVM, which has been shape by years of use in multiple different front-ends. For example, the type system of libjit is far less flexible, supporting only a subset of integer and floating-point types (like `int32`, not the arbitrary LLVM jungle of `int73` if you want it).

But generality comes at a cost: LLVM is both relatively slow and its code is relatively difficult to grok. It's not slow for a regular (AOT - Ahead of Time) compiler, but it *is* slow for a JIT compiler, a common source of pain for some LLVM users.

Now, a disclaimer: I did not benchmark libjit's compilation speed vs. LLVM; I was simply too lazy to generate equivalent and large-enough inputs for both. I'm also not saying that libjit is faster than LLVM, perhaps it isn't. After all, the amount of engineering power expended on LLVM has been orders of magnitude larger than for libjit, which may very well have led to much more optimized code. All I want to imply here is that libjit *could* be faster than LLVM. Generality is almost always at odds with performance in software, a sad fact we all wish wasn't true but, oh, it is. Is this very important? Not for

most uses of LLVM today, but since libjit's main goal is JIT, then perhaps. But without real measurements and profiling, I don't have anything intelligent to contribute here.

libjit is also much simpler to understand. I've spent enough time with LLVM to appreciate its complexity, as well as see how confused many programmers are about it (http://stackoverflow.com/questions/tagged/llvm). libjit was dramatically easier to grok for me. This may have two reasons: one is definitely its significantly smaller size and simpler internal structure. There is simply much less code to read, and much less abstraction levels to keep in the head. The other reason is my (sometimes controversial) opinion about the choice of programming language. libjit is written in C, which IMHO is easier to understand than C++. But that's a whole other issue, unrelated to this post (you know where to find me for your hatemail).

There's another significant difference between LLVM and libjit. LLVM, despite the historical meaning of its acronym and the numerous attempts that have been made over the years to make it work for dynamic languages, is still in its core a backend for compiling static languages like C, C++ and Objective C. By this, I mean languages that don't really have a significant runtime, like Java or C#. libjit, on the other hand, was designed to serve as a backend for a .NET implementation. That said, like the rest of libjit, these parts are not really battle-proven, so it's hard to attest to their completeness and stability.

So, which one should you use or learn? As for use, there's no question; LLVM is an industrial strength, heavily tested and verified system. It's being used daily on millions (maybe billions) of devices; it's being hacked on by hundreds of programmers from dozens of companies. If it fits your needs, LLVM is the way to go. And these days it fits most needs. Not all of them, mind you. For example, as I've already mentioned above, LLVM is not great for fast JIT-ing. And to be honest, I don't think it's possible to create a really fast JIT within the framework of LLVM, *because* of its modularity. The faster the JIT, the more you'll have to deviate from the framework of LLVM. This is a serious problem for Portable Native Client (http://blog.chromium.org/2013/11/portable-native-client-pinnacle-of.html), for instance.

libjit, on the other hand, is much more limited, aimed at dynamic runtimes and can (potentially) JIT-compile faster. It's also not being used too much, and lacks credibility w.r.t. features and stability. libjit itself only comes with toy examples - not very reassuring if you want to bet your future project on it.

In terms of educational value, libjit is great. If you want to learn about compiler backends, I would definitely start with libjit and only move to LLVM later. I hope my series of articles (https://eli.thegreenplace.net/2014/01/07/getting-started-with-libjit-part-3/) will be useful here.

For comments, please send me ✉ an email (mailto:eliben@gmail.com).

↑ Back to top