

wingolog

[about](#) | [projects](#) | [photos](#)

- [subscribe](#) 

- **related**

- [what does v8 do with that loop?](#)
- [a register vm for guile](#)
- [on-stack replacement in v8](#)
- [just-in-time code generation within webassembly](#)
- [bigint shipping in firefox!](#)
- [heap object representation in spidermonkey](#)
- [design notes on inline caches in guile](#)
- [the half strap: self-hosting and guile](#)
- [two paths, one peak: a view from below on high-performance language implementations](#)
- [effects analysis in guile](#)

v8: a tale of two compilers

5 July 2011 2:31 PM ([v8](#) | [javascript](#) | [self](#) | [guile](#) | [igalia](#) | [compilers](#) | [hotspot](#))

Regular readers will have noticed my fascination with the [V8](#) JavaScript implementation. It is indeed an impressive piece of engineering.

When V8 was originally announced, Lars Bak [wrote](#):

I hope the web community will adopt the code and the ideas we have developed to advance the performance of JavaScript. Raising the performance bar of JavaScript is important for continued innovation of web applications.

How right he was! V8 has succeeded admirably, not only in adoption, but also in "raising the performance bar" among all JavaScript implementations.

But as William Gibson says, "The future is already here — it's just not very evenly distributed." Many parts of V8 are not documented at all, and perhaps understandably, given the pace at which things are changing. So as I'm getting up to speed with V8 for my work with [Igalia](#), I've been trying to document the interesting things that I find, so that all JavaScript implementations can learn and improve.

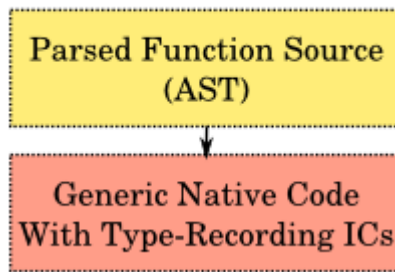
And indeed, with my selfish [Guile](#) hat on, this study in V8 is giving me lots of ideas and motivation. So perhaps V8's new motto should be, "shaming the world into faster code, one compiler at a time".

compiler the first: full-codegen

V8 compiles all JavaScript to native code. V8 has two compilers: one that runs fast and produces generic code, and one that doesn't run as fast but does try to produce optimized code.

The quick-and-simple compiler is known internally as the "full-codegen" compiler. It takes as its input the abstract syntax tree (AST) of a function, walks over the nodes in the AST, and emits calls to a macroassembler directly. Here's a picture:

V8 Full-Codegen Compiler



The boxes represent the flow of data in the compilation process. There's only two boxes because, as we said, it's a simple compiler. All local variables are stored either on the stack or on the heap, not in registers. Any variable referenced by a nested function is stored on the heap, in the context object associated with the function in which the variable was defined.

The compiler will emit loads and stores to pull these values into registers to actually do the work. The top of the stack of temporaries is cached in a register. Complicated cases are handled by emitting calls to runtime procedures. The compiler does track the context in which an expression is being evaluated, so that tests can just jump directly to the consequent blocks, instead of pushing a value, testing if it's zero or not, then branching. Small integer arithmetic is usually inlined. And that's all there is!

Actually I should mention one important optimization that is done even with the full-codegen compiler, and that is [inline caching](#). See the Hölzle, Chambers, and Ungar paper I link to there for all of the details. Inline caches are used for assignments, unary and binary operations, function calls, property accesses, and comparisons.

The inline caches also serve as sources of type information, used by the optimizing compiler. In the case of some statement types, like assignments, the only purpose of the IC is to record type information. *Ed.: see the comments for a correction.*

Here are some links to the source:

[ast.h](#)

~~The abstract syntax tree. Note that though the comment says that ASTs are allocated in separate zones, for constant-time deallocation, in practice the AST is kept around as long as the JavaScript functions are alive, for use as input to the optimizing compiler. Zone allocation is more of a win for the optimizing compiler, which produces more ephemeral garbage. Ed.: Vyacheslav Egorov writes in to say that what V8 keeps around is actually the function source, not the AST. It re-parses as needed. Makes sense. I recall Lars Bak saying in a [video](#) that source is the most compact IR there is, and indeed maybe that is the case.~~

[full-codegen.h](#)

[full-codegen.cc](#)

[full-codegen-ia32.cc](#)

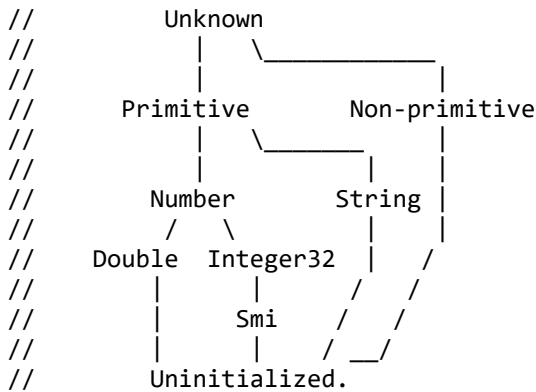
The full-codegen compiler. Most of the meat of the full-codegen compiler is in the target-specific directory (4257 lines vs 769+1323 lines). Currently supported targets are ia32, x64, arm, and mips.

type feedback

The first time V8 sees a function, it parses it to the AST but doesn't actually do anything with it. It only runs the full-codegen compiler when the function is first run. How's that for lazy! But after things have started up, it kicks off a profiler thread to see how things are going, and what functions are hot.

This lazy, sit-back-and-watch approach gives V8 time to record the type information flowing through it. So by the time it has decided that a function is hot, and could use a little more help, it has type information to give to the compiler.

Runtime type feedback information is recorded by and stored in the inline caches (ICs). Type feedback information is expressed internally as an 8-bit value constructed in such a way that it can detect a hierarchy of types, with a simple bitmask. At this point the best I can do is show the artwork in the source:



Everytime an IC stub sees a new kind of value, it computes the type of that value, and bitwise-and's it with the old type. The initial type value is uninitialized. So if an IC only sees integers in the Smi (small integer) range, the recorded type will indicate that. But as soon as it sees a double value, the type becomes Number; and if it then sees an object, the type becomes Unknown. A non-primitive IC will necessarily store the map of the receiver type in the IC, for dispatch purposes. Type feedback can parse the IC stub to get at this map, when needed. *Ed.: [more corrections](#). Thanks, Vyacheslav.*

Type feedback information is associated with a particular AST node (the assignment, property load, etc.). An integer identifier for the node is serialized into the IC so that when V8 decides that a function is hot, it can parse out the recorded type information from the full-codegen code, and associate it with the AST node.

Now, this process is a bit complicated. It needs support up and down in your compiler stack. You need to have inline caches. Your inline caches need to have support for the type information, both for operands and for the result. You need to be able to walk this data to find the values. Then you need to link it back to the AST, so that when you pass the AST to your optimizing compiler, the compiler is able to ask the right questions.

The concrete strategy that V8 takes is to parse the data into a `TypeFeedbackOracle` object, associating information with particular AST nodes. Then V8 visits all of the AST nodes with this oracle, and the nodes themselves parse out the data that they might find useful from the oracle.

In the end one is able to, for example, ask a Property node if it is monomorphic, and in any case what are the receiver types at that node. It seems that this works well for V8, because it reduces the number of moving parts in the optimizing compiler, given that it doesn't need to have the TypeFeedbackOracle itself.

So, source links.

[type-info.h](#)

The TypeInfo 8-bit data type, and the TypeFeedbackOracle declaration. I have to admit, I really like the use of C++ within V8. It's a nasty tool, but they wield it well.

type-info.cc

The implementation of `TypeFeedbackOracle`. See `ProcessTarget` down at the bottom of the file for where the bulk of the work happens.

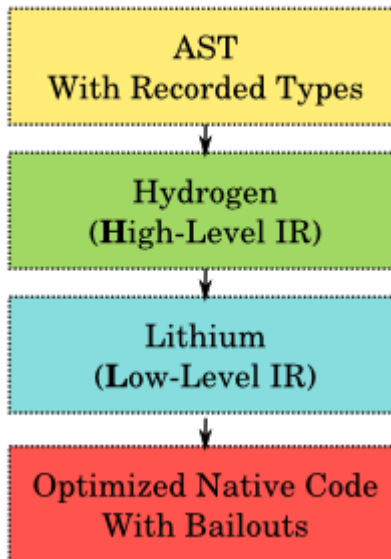
Also check the [ast.h](#) link before to see see how type feedback ties into the AST itself.

crankshaft = type feedback + hydrogen + lithium

Once V8 has identified that a function is hot and has collected some type feedback information, it tries to run the augmented AST through an optimizing compiler. This optimizing compiler is called Crankshaft in the marketing materials, though that name rarely appears in the source itself.

Instead, in the source, Crankshaft consists of the Hydrogen high-level intermediate representation (IR), the Lithium low-level IR, and their associated compilers. Like this:

V8 Crankshaft Compiler



(You probably caught on from my heavy-handed repetition above, but I believe that the names Hydrogen and Lithium come from High- and Low-level, respectively.)

It depends on your background, but you might have seen a diagram like that before:

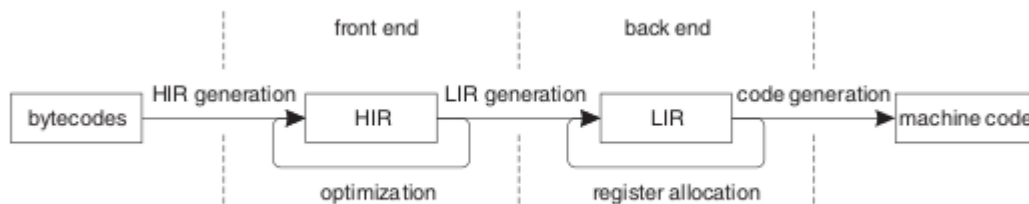


Fig. 2. Structure of the Java HotSpot™ client compiler.

Indeed I believe that Crankshaft was highly influenced by the changes that Sun introduced into the Hotspot Client compiler in Java 6. Let me quote from an illuminating 2008 paper by Kotzmann et al, [Design of the HotSpot Client Compiler for Java 6](#):

First, a high-level intermediate representation (HIR) of the compiled method is built via an abstract interpretation of the bytecodes. It consists of a control-flow graph (CFG), whose basic blocks are singly linked lists of instructions. The HIR is in static single-assignment (SSA) form, which means that for every variable there is just a single point in the program where a value is assigned to it. An instruction that loads or computes a value represents both the operation and its result, so that operands can be represented as pointers to previous instructions. Both during and after generation of the HIR, several optimizations are performed, such as constant folding, value numbering, method inlining, and null check elimination. They benefit from the simple structure of the HIR and the SSA form.

The back end of the compiler translates the optimized HIR into a low-level intermediate representation (LIR). The LIR is conceptually similar to machine code, but still mostly platform-independent. In contrast to HIR instructions, LIR operations operate on virtual registers instead of references to previous instructions. The LIR facilitates various low-level optimizations and is the input for the linear scan register allocator, which maps virtual registers to physical ones.

This statement describes Crankshaft very neatly, and the rest of section 2 of that paper applies in a general sense. Of course there are some differences. Crankshaft starts with the AST, not bytecodes. The HotSpot client runtime does not use type-feedback to help its compiler, as it is less necessary for Java, though it would still be helpful. Crankshaft doesn't do very much with exception handlers.

But the similarities are such that V8 can actually produce traces that can be read by [c1visualizer \(docs\)](#), a program used to visualize the internals of the HotSpot client compiler. (The client compiler appears to be known internally as [c1](#); the server compiler appears to be [opto](#)).

in our next installment

This post is getting quite long. I'm going to have to send my mom some flowers or something, because I'm sure she does check these things out, and it must be brain-numbing to slog through. *Another paragraph? When will he stop?*

So if you, dear reader, find yourself in this category, you're in good company, because my mom is a lovely lady.

But if you're a compiler geek, I think I have at least couple more articles to write on Crankshaft before I can get back to other hacks. Specifically, we'll go into the optimizations that Crankshaft does on the Hydrogen IR, and take a look on how it does the translation to Lithium. But until then, happy hacking!

14 responses

1. *John Cowan* says:
[5 July 2011 3:30 PM](#)

Note that SSA form is equivalent to our good old CPS form, by way of ANF form. Funny how the rest of the compiler community took decades to get to where Scheme was in 1975.

2. *John Cowan* says:
[5 July 2011 3:48 PM](#)

Oh, and looking at the definitions in type-info.h, a lovely pattern appears in the type bits. Starting from the most significant bit:

Bit 0 is not used,
Bit 1 is the non-primitive bit,
Bit 2 is the string bit,
Bit 3 is the known value bit,
Bit 4 is the double bit,
Bit 5 is the small-integer bit,
Bit 6 is the integer (of any size) bit,
Bit 7 is the number (of any size or type) bit.

If all 7 bits are set, that's the uninitialized value, which must be asserted away before the bits can be examined.

I wonder how the types of true, false, null, and undefined are represented.