

📅 2017 ⌚ 24m 💬 4

An impressively low-level article that we hope gives you a good idea about what happens in V8 when it comes to optimization.

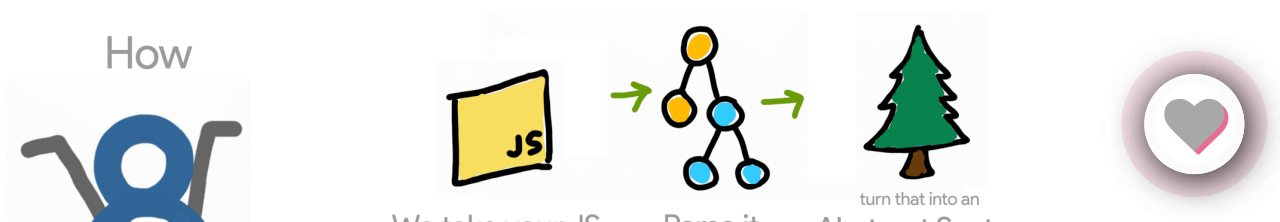
Once again, we have Benedikt Meurer in the house. Benedikt is the optimization lead in charge of V8 performance, and today he brings us a deep dive into how the TurboFan optimizing compiler in V8 works.

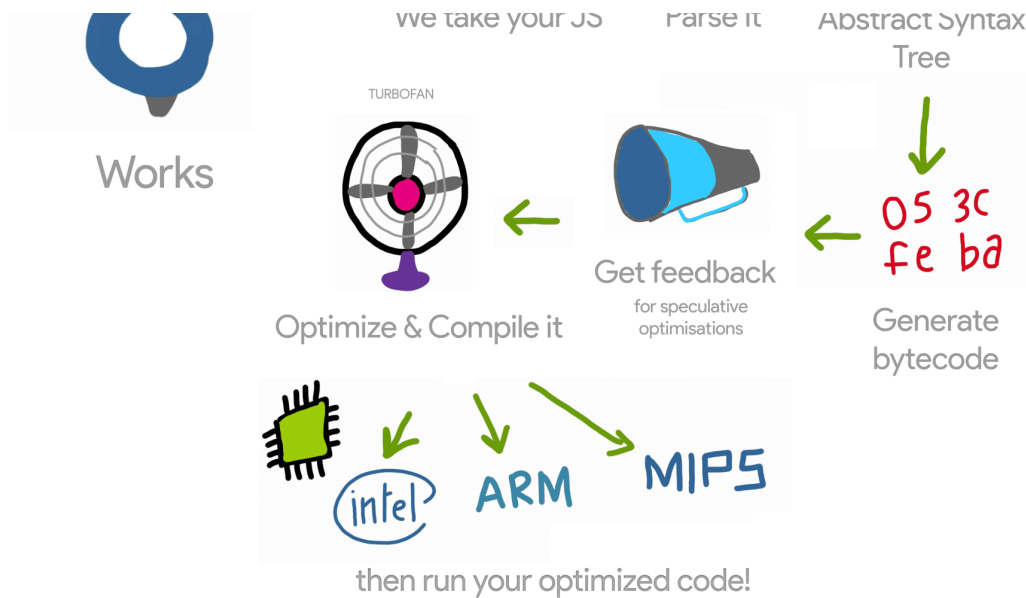
— Editor's note.

Following up on my talk [“A Tale of TurboFan” \(slides\)](#) at [JS Kongress](#), I wanted to give some additional context on how TurboFan, V8's optimizing compiler, works and how V8 turns your JavaScript into highly-optimized machine code. For the talk I had to be brief and leave out several details. So I'll use this opportunity to fill the gaps, especially how V8 collects and uses the profiling information to perform speculative optimizations.

Overview

Before we dive into the details of how TurboFan works, I'll briefly explain how V8 works on a high level. Let's have a look at this *simplified breakdown of how V8 works* (taken from the [“JavaScript Start-up Performance”](#) blog post by my colleague [Addy Osmani](#)):





By @addyosmani

How V8 Works

Whenever Chrome or Node.js has to execute some piece of JavaScript, it passes the source code to V8. V8 takes that JavaScript source code and feeds it to the so-called *Parser*, which creates an *Abstract Syntax Tree (AST)* representation for your source code. The talk “*Parsing JavaScript — better lazy than eager?*” from my colleague *Marja Hölttä* contains some details of how this works in V8. The AST is then passed on to the recently introduced *Ignition Interpreter*, where it is turned into a sequence of bytecodes. This sequence of bytecodes is then executed by Ignition.

During execution, Ignition collects *profiling information* or *feedback* about the inputs to certain operations. Some of this feedback is used by Ignition itself to speed up subsequent interpretation of the bytecode. For example, for property accesses such as `o.x`, where `o` has the same shape all the time (i.e. you always pass a value `{x:v}` for `o` where `v` is a String), we cache information on how to get to the value of `x`. Upon subsequent execution of the same bytecode we don't need to search for `x` in `o` again. The underlying machinery here is called *inline cache (IC)*. You can find a lot of details about how this works for property accesses in the blog post “*What's up with monomorphism?*” by my colleague *Vyacheslav Egorov*.

Probably even more important — depending on your workload — the *feedback* collected by the Ignition interpreter is consumed by the *TurboFan JavaScript compiler* to generate highly-optimized machine code using a technique called *Speculative Optimization*. Here the optimizing compiler looks at what kinds of values were seen in the past and assumes that in the future we're going to see the same kinds of values. This allows TurboFan to leave out a lot of cases that it doesn't need to handle, which is extremely important to execute JavaScript at peak performance.

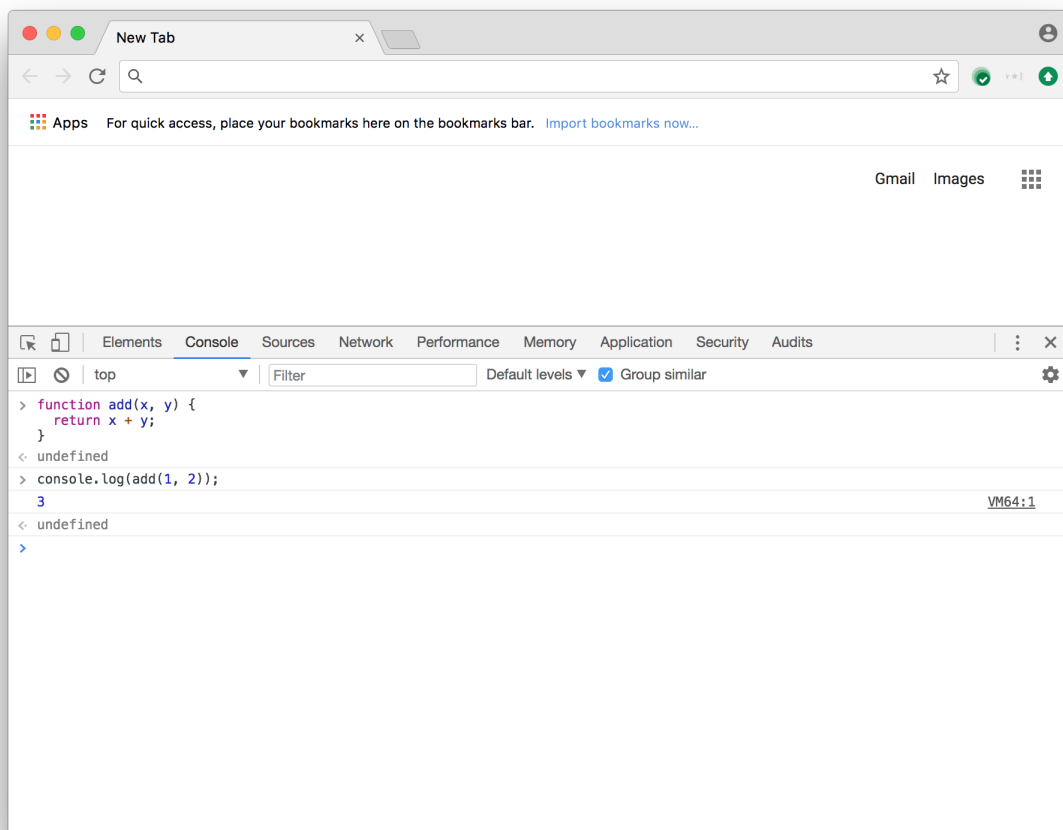


The Basic Execution Pipeline

Let's consider a reduced version of the example from my talk, focusing solely on the function `add`, and how this is executed by V8.

```
function add(x, y) {  
  return x + y;  
}  
  
console.log(add(1, 2));
```

If you run this in the Chrome DevTools console, you'll see that it outputs the expected result `3`:



Chrome DevTools

Let's examine what happens under the hood in V8 to actually get to these results. We'll do this step by step for the function `add`. As mentioned before, we first need to parse the function source code and turn that into an Abstract Syntax Tree (AST). This is done by the `Parser`. You can see the AST that V8 generates internally using the `--print-ast` command line flag



in a Debug build of the `d8 shell`.

```
$ out/Debug/d8 --print-ast add.js
```

```
...
```

```
--- AST ---
```

```
FUNC at 12
```

```
. KIND 0
```

```
. SUSPEND COUNT 0
```

```
. NAME "add"
```

```
. PARAMS
```

```
. . VAR (0x7fbd5e818210) (mode = VAR) "x"
```

```
. . VAR (0x7fbd5e818240) (mode = VAR) "y"
```

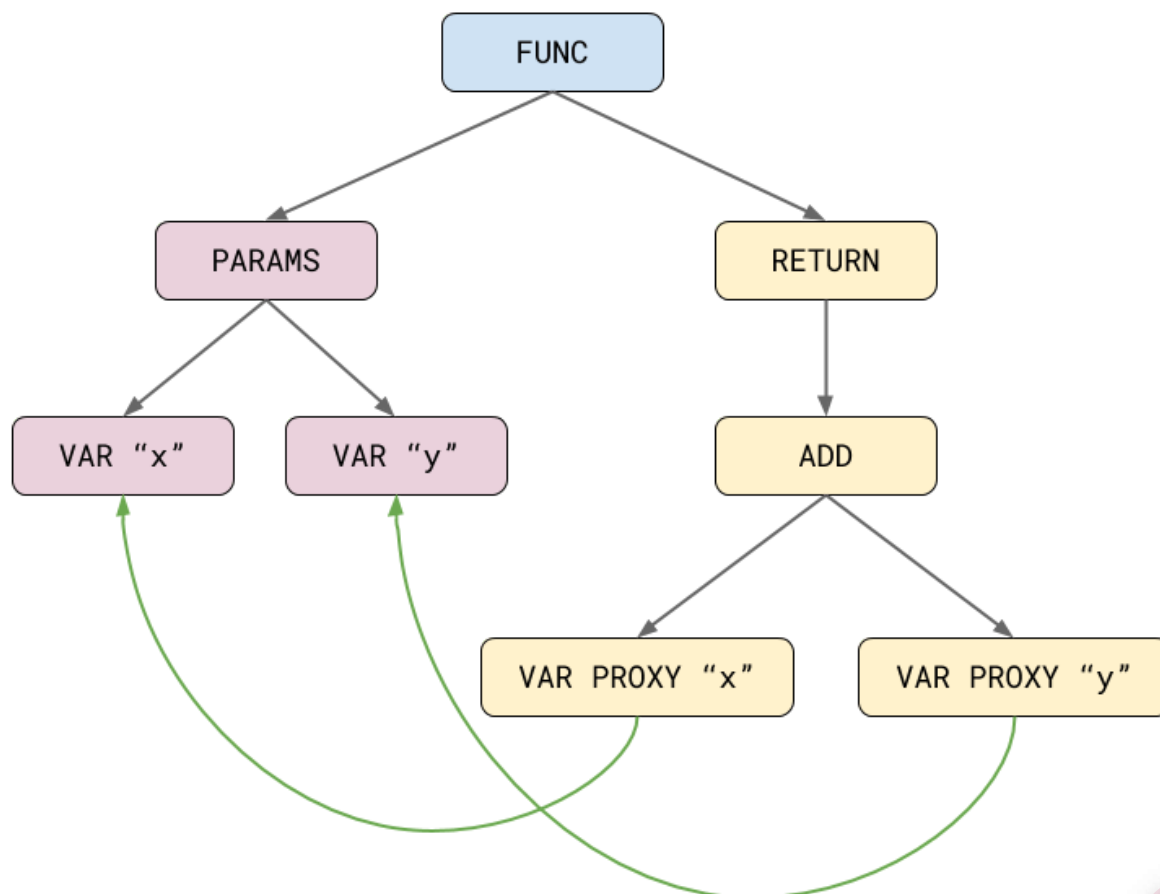
```
. RETURN at 23
```

```
. . ADD at 32
```

```
. . . VAR PROXY parameter[0] (0x7fbd5e818210) (mode = VAR) "x"
```

```
. . . VAR PROXY parameter[1] (0x7fbd5e818240) (mode = VAR) "y"
```

This format is not very easy to consume, so let's visualize it.



Abstract Syntax Tree



Initially the function literal for `add` is parsed into a tree representation, with one subtree for the parameter declarations and one subtree for the actual function body. During parsing it is impossible to tell which names correspond to which variables in the program, mostly due to the *funny var hoisting rules* and `eval` in JavaScript, but also for other reasons. So for every name the parser initially creates so-called `VAR PROXY` nodes. The subsequent scope resolution step connects these `VAR PROXY` nodes to the declaring `VAR` nodes or marks them as either *global* or *dynamic lookups*, depending on whether the parser has seen an `eval` expression in one of the surrounding scopes.

Once this is done we have a complete AST that contains all the necessary information to generate executable bytecode from it. The AST is then passed to the `BytecodeGenerator`, which is the part of the Ignition interpreter that generates bytecode on a per-function basis. You can also see the bytecode being generated by V8 using the flag `--print-bytecode` with the `d8` shell (or with `node`).

```
$ out/Debug/d8 --print-bytecode add.js
...
[generated bytecode for function: add]
Parameter count 3
Frame size 0
  12 E> 0x37738712a02a @    0 : 94          StackCheck
  23 S> 0x37738712a02b @    1 : 1d 02        Ldar a1
  32 E> 0x37738712a02d @    3 : 29 03 00      Add a0, [0]
  36 S> 0x37738712a030 @    6 : 98          Return
Constant pool (size = 0)
Handler Table (size = 16)
```

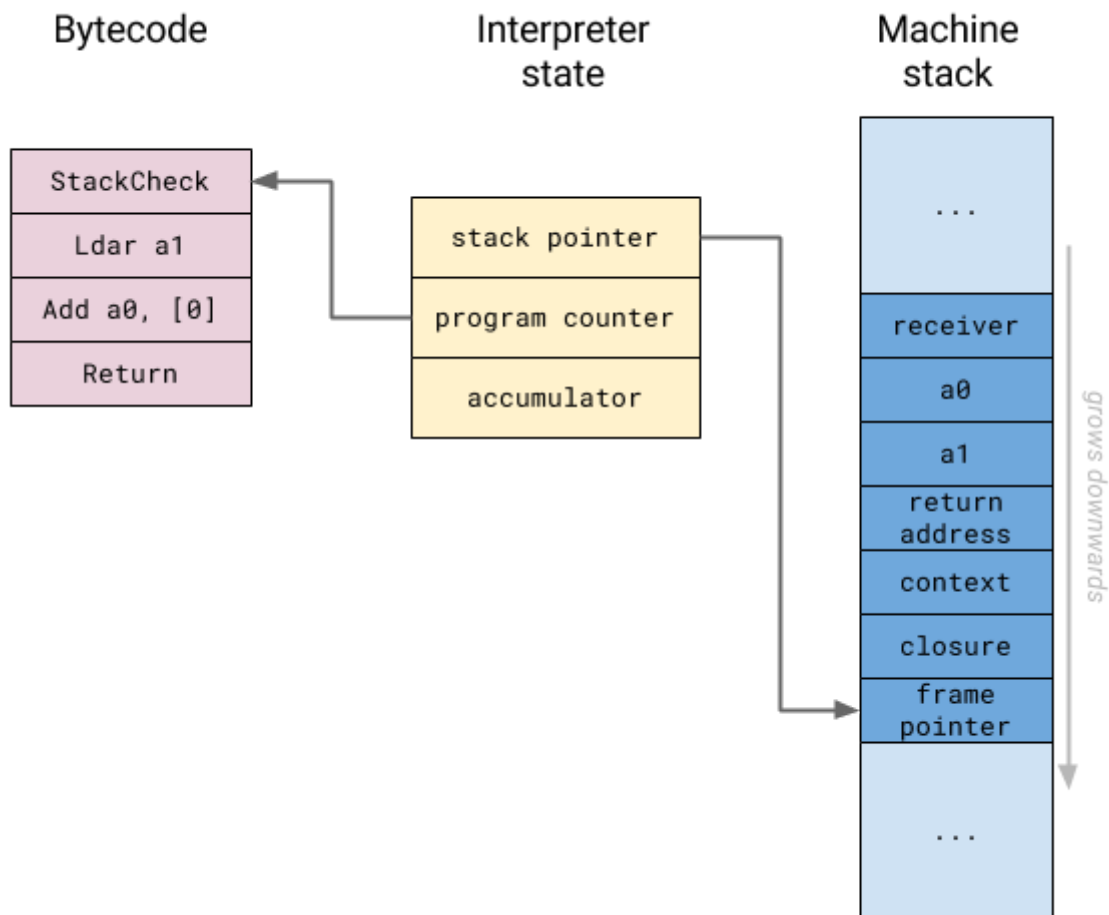
This tells us that a new bytecode object was generated for the function `add`, which accepts three parameters: the implicit receiver `this`, and the explicit formal parameters `x` and `y`. The function doesn't need any local variables (the frame size is zero), and contains the sequence of four bytecodes:

```
StackCheck
Ldar a1
Add a0, [0]
Return
```

To explain that, we first need to understand how the interpreter works on a high level. Ignition uses a so-called *register machine* (in contrast to the *stack machine* approach that was used by



uses a so-called *register machine* (in contrast to the *stack machine* approach that was used by earlier V8 versions in the FullCodegen compiler). It holds its local state in interpreter registers, some of which map to real CPU registers, while others map to specific slots in the native machine stack memory.



Interpreter overview

The special registers `a0` and `a1` correspond to the formal parameters for the function on the machine stack (in this case we have two formal parameters). Formal parameters are the parameters declared in the source code, which might be different from the actual number of parameters passed to the function at runtime. The last computed value of each bytecode is usually kept in a special register called the `accumulator`, the current *stack frame* or *activation record* is identified by the `stack pointer`, and the `program counter` points to the currently executed instruction in the bytecode. Let's check what the individual bytecodes do in this example:

- `StackCheck` compares the `stack pointer` to some known upper limit (actually a lower limit since the stack grows downwards on all architectures supported by V8). If the stack grows above a certain threshold, we abort execution of the function and throw a `RangeError` saying that the stack was overflowed.
- `Ldar a1` loads the value of the register `a1` into the `accumulator` register (the name stands for **Load Accumulator Register**).

- Add `a0, [0]` loads the value from the `a0` register and adds it to the value in the `accumulator` register. The result is then placed into the `accumulator` register again. Note that *addition* here can also mean string concatenation, and that this operation can execute **arbitrary JavaScript** depending on the operands. The `+` operator in JavaScript is really complex, and many people have tried to illustrate the complexity in talks. [Emily Freeman](#) recently gave a talk at JS Kongress titled “[JavaScript’s “+” Operator and Decision Fatigue](#)” on precisely this topic. The `[0]` operand to the `Add` operator refers to a *feedback vector slot*, where Ignition stores the profiling information about the values we’ve seen during execution of the function. We’ll get back to this later when we investigate how TurboFan optimizes the function.
- `Return` ends execution of the current function and transfers control back to the caller. The value returned is the current value in the `accumulator` register.

My colleague [Franziska Hinkelmann](#) wrote an article “[Understanding V8’s Bytecode](#)” a while ago that gives some additional insight into how V8’s bytecode works.

Speculative Optimization

Now that you have a rough understanding of how V8 executes your JavaScript in the baseline case, it’s time to start looking into how TurboFan fits into the picture, and how your JavaScript code can be turned into highly optimized machine code. The `+` operator is already such a complex operation in JavaScript which has to do a lot of checks before it eventually does the number addition on the inputs.

12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? [GetValue](#)(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? [GetValue](#)(*rref*).
5. Let *lprim* be ? [ToPrimitive](#)(*lval*).
6. Let *rprim* be ? [ToPrimitive](#)(*rval*).
7. If [Type](#)(*lprim*) is String or [Type](#)(*rprim*) is String, then
 - a. Let *lstr* be ? [ToString](#)(*lprim*).
 - b. Let *rstr* be ? [ToString](#)(*rprim*).
 - c. Return the [string-concatenation](#) of *lstr* and *rstr*.
8. Let *lnum* be ? [ToNumber](#)(*lprim*).
9. Let *rnum* be ? [ToNumber](#)(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below [12.8.5](#).

Runtime Semantics of the + operator

It’s not immediately obvious how this can be done in just a few machine instructions to reach peak performance (comparable to Java or C++ code). The magic keyword here is *Speculative*



peak performance (comparable to Java or C++ code). The magic keyword here is *Speculative Optimization*, which makes use of assumptions about possible inputs. For example, when we know that in the case of `x+y`, both `x` and `y` are numbers, we don't need to handle the cases where either of them is a string, or even worse — the case where the operands can be arbitrary JavaScript objects on which we need to run the abstract operation `ToPrimitive` first.

7.1.1 ToPrimitive (*input* [, *PreferredType*])

The abstract operation `ToPrimitive` takes an *input* argument and an optional argument *PreferredType*. The abstract operation `ToPrimitive` converts its *input* argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint *PreferredType* to favour that type. Conversion occurs according to the following algorithm:

1. **Assert:** *input* is an ECMAScript language value.
2. If `Type(input)` is Object, then
 - a. If *PreferredType* is not present, let *hint* be `"default"`.
 - b. Else if *PreferredType* is hint String, let *hint* be `"string"`.
 - c. Else *PreferredType* is hint Number, let *hint* be `"number"`.
 - d. Let *exoticToPrim* be ? `GetMethod(input, @@toPrimitive)`.
 - e. If *exoticToPrim* is not **undefined**, then
 - i. Let *result* be ? `Call(exoticToPrim, input, « hint »)`.
 - ii. If `Type(result)` is not Object, return *result*.
 - iii. Throw a **TypeError** exception.
 - f. If *hint* is `"default"`, set *hint* to `"number"`.
 - g. Return ? `OrdinaryToPrimitive(input, hint)`.
3. Return *input*.

ToPrimitive operation

Knowing that both `x` and `y` are numbers also means that we can rule out observable side effects — for example we know it cannot shut down the computer or write to a file or navigate to a different page. In addition we know that the operation won't throw an exception. Both of these are important for optimizations, because an optimizing compiler can only eliminate an expression if it knows for sure that this expression won't cause any observable side effects and doesn't raise exceptions.

Due to the dynamic nature of JavaScript we usually don't know the precise types of values until runtime, i.e. just by looking at the source code it's often impossible to tell the possible values of inputs to operations. That's why we need to speculate, based on previously collected *feedback* about the values we've seen so far, and then assume that we're going to always see similar values in the future. This might sound fairly limited, but it has proven to work well for dynamic languages like JavaScript.



```
function add(x, v) {
```

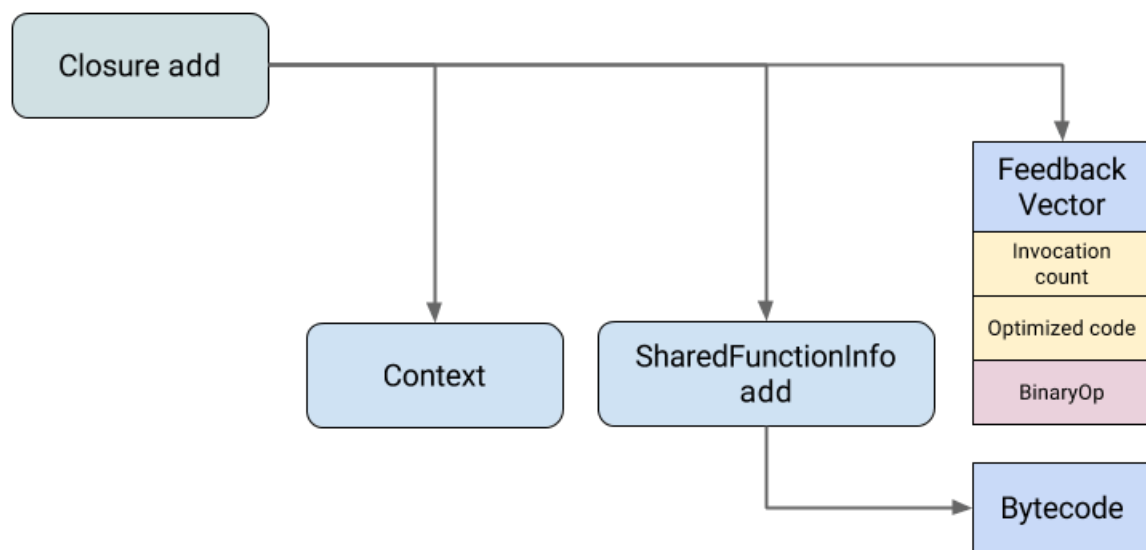


```

    return x + y;
}

```

In this particular case, we collect information about the input operands and the resulting value of the `+` operation (the `Add` bytecode). When we optimize this code with TurboFan and we've seen only numbers so far, we put checks in place to check that both `x` and `y` are numbers (in that case we know that the result is going to be a number as well). If either of these checks fail we go back to interpreting the bytecode instead — a process called *Deoptimization*. Thus TurboFan doesn't need to worry about all these other cases of the `+` operator and doesn't even need to emit machine code to handle those, but can focus on the case for numbers, which translates well to machine instructions.



Closure structure

The feedback collected by Ignition is stored in the so-called *Feedback Vector* (previously named *Type Feedback Vector*). This special data structure is linked from the closure and contains slots to store different kinds of feedback, i.e. bitsets, closures or hidden classes, depending on the concrete *inline cache (IC)*. My colleague [Michael Stanton](#) gave a nice presentation at [AmsterdamJS](#) earlier this year titled “V8 and How It Listens to You”, which explains some of the concepts of the Feedback Vector in detail. The closure also links to the *SharedFunctionInfo*, which contains the general information about the function (like source position, bytecode, strict/sloppy mode, etc.), and there's a link to the *context* as well, which contains the values for the free variables of the function and provides access to the global object (i.e. the `<iframe>` specific data structures).

In the case of the `add` function, the Feedback Vector has exactly one interesting slot (in addition to the general slots like the call count slot), and this is a `BinaryOp` slot, where binary operations like `+`, `-`, `*`, etc. can record feedback about the inputs and outputs that were seen so far. You can check what's inside the feedback vector of a specific closure using the

specialized `%DebugPrint()` intrinsic when running with the `--allow-natives-syntax` command line flag (in a Debug build of `d8`).

```
function add(x, y) {  
  return x + y;  
}  
  
console.log(add(1, 2));  
%DebugPrint(add);
```

Running this with `--allow-natives-syntax` in `d8` we observe:

```
$ out/Debug/d8 --allow-natives-syntax add.js  
DebugPrint: 0xb5101ea9d89: [Function] in OldSpace  
...  
- feedback vector: 0xb5101eaa091: [FeedbackVector] in OldSpace  
- length: 1  
SharedFunctionInfo: 0xb5101ea99c9 <SharedFunctionInfo add>  
Optimized Code: 0  
Invocation Count: 1  
Profiler Ticks: 0  
Slot #0 BinaryOp BinaryOp:SignedSmall  
...
```

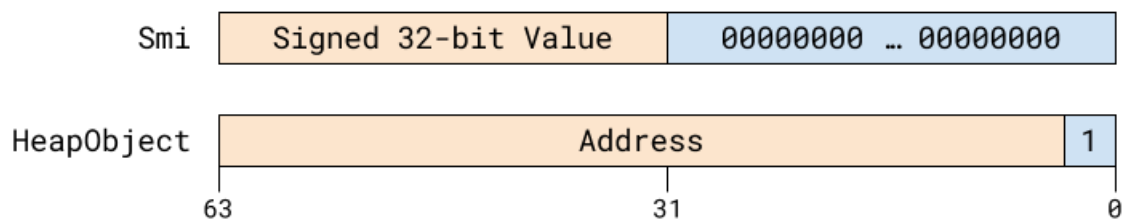
We can see the invocation count is 1, since we ran the function `add` exactly once. Also there's no optimized code yet (indicated by the arguably confusing `0` output). But there's exactly one slot in the Feedback Vector, which is a `BinaryOp` slot whose current feedback is `SignedSmall`. What does that mean? The bytecode `Add` that refers to the feedback slot 0 has only seen inputs of type `SignedSmall` so far and has also only produced outputs of type `SignedSmall` up until now.

But what is this `SignedSmall` type about? JavaScript doesn't have a type of that name. The name comes from an optimization that is done in V8 when representing small signed integer values that occur frequently enough in programs to deserve a special treatment (other JavaScript engines have similar optimizations).



Excuse: Value Representation

Let's briefly explore how JavaScript values are represented in V8 to better understand the underlying concept. V8 uses a technique called **Pointer Tagging** to represent values in general. Most of the values we deal with live in the JavaScript heap, and have to be managed by the garbage collector (GC). But for some values it would be too expensive to always allocate them in memory. Especially for small integer values that are often used as indices to arrays and temporary computation results.



Tagging Scheme

In V8, we have two possible *tagged representations*: A *Smi* (short for **Small Integer**) and a *HeapObject*, which points to memory in the managed heap. We make use of the fact that all allocated objects are aligned on word boundaries (64-bit or 32-bit depending on the architecture), which means that the 2 or 3 least significant bits are always zero. We use the least significant bit to distinguish between a *HeapObject* (bit is 1) and a *Smi* (bit is 0). For *Smi* on 64-bit architectures the least significant 32 bits are actually all zero and the signed 32-bit value is stored in the upper half of the word. This is to allow efficient access to the 32-bit value in memory using a single machine instruction instead of having to load and shift the value, but also because 32-bit arithmetic is common for bitwise operations in JavaScript.

On 32-bit architectures, the *Smi* representation has the least significant bit set to 0 and a signed 31-bit value shifted to the left by one stored in the upper 31-bit of the word.

Feedback Lattice

The `SignedSmall` feedback type refers to all values that have *Smi* representation. For the `Add` operation it means that it has only seen inputs represented as *Smi* so far and all outputs that were produced could also be represented as *Smi* (i.e. the values didn't overflow the range of possible 32-bit integer values). Let's check what happens if we also call `add` with other numbers that are not representable as *Smi*.

```
function add(x, y) {
```



```

    return x + y;
}

console.log(add(1, 2));
console.log(add(1.1, 2.2));
%DebugPrint(add);

```

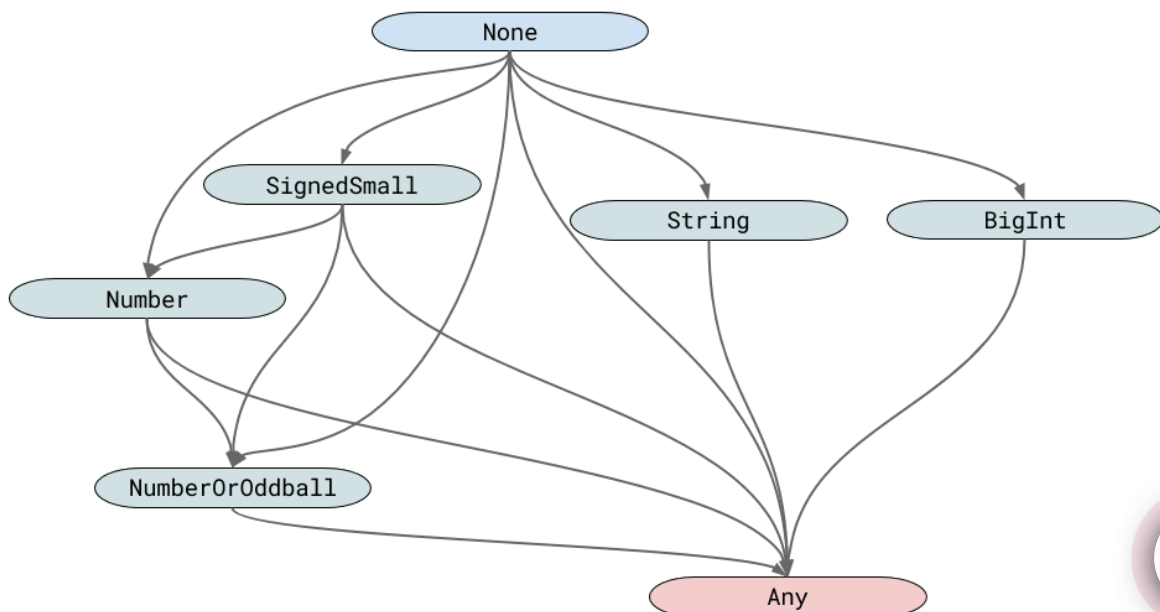
Running this again with `--allow-natives-syntax` in `d8` we observe:

```

$ out/Debug/d8 --allow-natives-syntax add.js
DebugPrint: 0xb5101ea9d89: [Function] in OldSpace
...
- feedback vector: 0x3fd6ea9ef9: [FeedbackVector] in OldSpace
- length: 1
SharedFunctionInfo: 0x3fd6ea9989 <SharedFunctionInfo add>
Optimized Code: 0
Invocation Count: 2
Profiler Ticks: 0
Slot #0 BinaryOp BinaryOp:Number
...

```

First of all, we see that the invocation count is now 2, since we ran the function twice. And then we see that the `BinaryOp` slot value changed to `Number`, which indicates that we've seen arbitrary numbers for the addition (i.e. non-integer values). For addition there's a lattice of possible states for feedback, which roughly looks like this:



Feedback Lattice

The feedback starts as `None`, which indicates that we haven't seen anything so far, so we don't know anything. The `Any` state indicates that we have seen a combination of incompatible inputs or outputs. The `Any` state thus indicates that the `Add` is considered *polymorphic*. In contrast, the remaining states indicate that the `Add` is *monomorphic*, because it has seen and produced only values that are somewhat the same.

- `SignedSmall` means that all values have been small integers (signed 32-bit or 31-bit depending on the word size of the architecture), and all of them have been represented as *Smi*.
- `Number` indicates that all values have been regular numbers (this includes `SignedSmall`).
- `NumberOrOddball` includes all the values from `Number` plus `undefined`, `null`, `true` and `false`.
- `String` means that both inputs have been string values.
- `BigInt` means that both inputs have been `BigInts`, see the current [stage 2 proposal](#) for details.

It's important to note that the feedback can only progress in this lattice. It's impossible to ever go back. If we'd ever go back then we risk entering a so-called *deoptimization loop* where the optimizing compiler consumes feedback and bails out from optimized code (back to the interpreter) whenever it sees values that don't agree with the feedback. The next time the function gets hot we will eventually optimize it again. So if we didn't progress in the lattice then TurboFan would generate the same code again, which effectively means it will bail out on the same kind of input again. Thus the engine would be busy just optimizing and deoptimizing code, instead of running your JavaScript code at high speed.

The Optimization Pipeline

Now that we know how Ignition collects feedback for the `add` function, let's see how TurboFan makes use of that feedback to generate minimal code. I'll use the special intrinsic `%OptimizeFunctionOnNextCall()` to trigger optimization of a function in V8 at a very specific point in time. We often use these intrinsics to write tests that stress the engine in a very specific way.

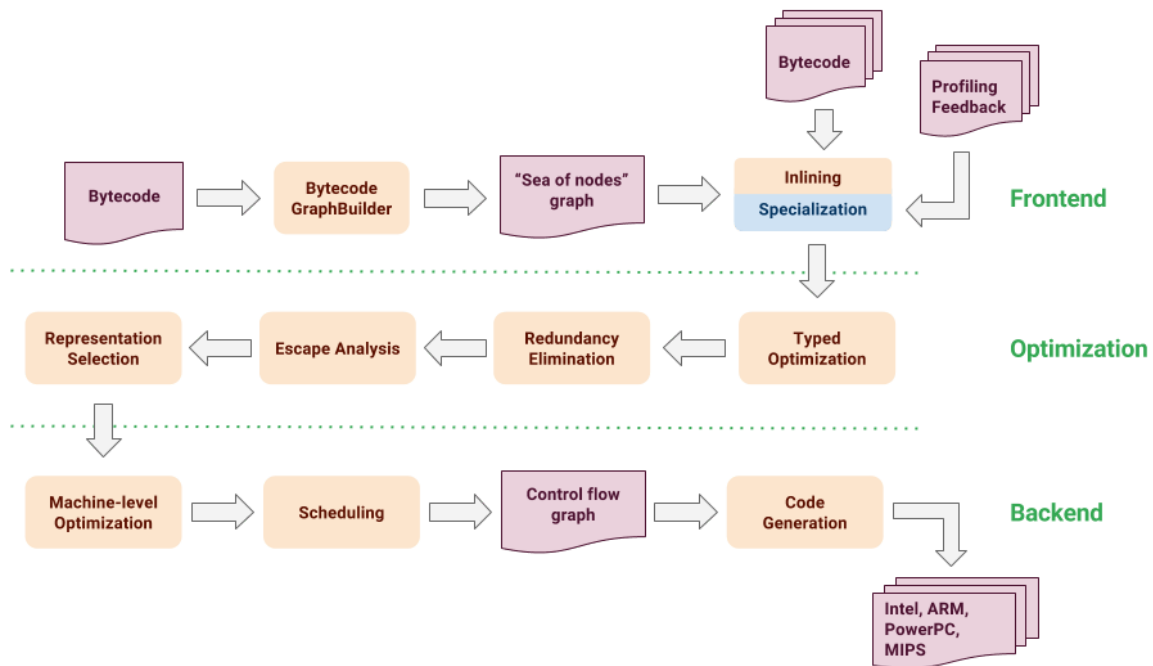
```
function add(x, y) {
  return x + y;
}
```

```
add(1, 2); // Warm up with SignedSmall feedback
```



```
add(1, 2); // warm up with SignedSmall feedback.
%OptimizeFunctionOnNextCall(add);
add(1, 2); // Optimize and run generated code.
```

Here we explicitly warm up the `x+y` site with `SignedSmall` feedback by passing in two integer values whose sum also fits into the small integer range. Then we tell V8 that it should optimize the function `add` (with TurboFan) when it's called the next time, and eventually we call `add`, which triggers TurboFan and then runs the generated machine code.



TurboFan

TurboFan takes the bytecode that was previously generated for `add` and extracts the relevant feedback from the `FeedbackVector` of `add`. It turns this into a graph representation and passes the graph through the various phases of the frontend, optimization and backend stages. I'm not going into the details of the passes here, that's a topic for a separate blog post (or a series of separate blog posts). Instead we're going to look at the generated machine code and see how the speculative optimization works. You can see the code generated by TurboFan by passing the `--print-opt-code` flag to `d8`.

```

Prologue {
    leaq rcx,[rip+0x0]
    movq rcx,[rcx-0x37]
    testb [rcx+0xf],0x1
    jnz CompileLazyDeoptimizedCode
    push rbp
    movq rbp,rsi
    push rsi
    push rdi
    cmpq rsp,[r13+0xdb0]
    jna StackCheck
    Check x is a small integer {
        movq rax,[rbp+0x18]
        test al,0x1
        jnz Deoptimize
    }
  
```



Check y is a small integer	{	movq rbx,[rbp+0x10]
		testb rbx,0x1
		jnz Deoptimize
Convert y from Smi to Word32	{	movq rdx,rbx
		shrq rdx, 32
Convert x from Smi to Word32	{	movq rcx,rax
		shrq rcx, 32
Add x and y (incl. overflow check)	{	addl rdx,rcx
		jo Deoptimize
Convert result to Smi	{	shlq rdx, 32
		movq rax,rdx
		movq rsp,rbp
Epilogue	{	pop rbp
		ret 0x18

Generated assembly code

This is the x64 machine code that is generated by TurboFan, with annotations from me and leaving out some technical details that don't matter (i.e. the exact call sequence to the Deoptimizer). So let's see what the code does:

```
# Prologue
leaq rcx,[rip+0x0]
movq rcx,[rcx-0x37]
testb [rcx+0xf],0x1
jnz CompileLazyDeoptimizedCode
push rbp
movq rbp,rsp
push rsi
push rdi
cmpq rsp,[r13+0xdb0]
jna StackCheck
```

The prologue checks whether the code object is still valid or whether some condition changed which requires us to throw away the code object. This was recently introduced by my intern [Juliana Franco](#) as part of her “[Internship on Laziness](#)”. Once we know that the code is still valid, we build the *stack frame* and check that there's enough space left on the stack to execute the code.

```
# Check x is a small integer
movq rax,[rbp+0x18]
test al,0x1
jnz Deoptimize
# Check y is a small integer
movq rbx,[rbp+0x10]
testb rbx,0x1
```



```

jnz Deoptimize
# Convert y from Smi to Word32
movq rdx,rbx
shrq rdx, 32
# Convert x from Smi to Word32
movq rcx,rax
shrq rcx, 32

```

Then we start with the body of the function. We load the values of the parameters `x` and `y` from the stack (relative to the frame pointer in `rbp`) and check if both values have *Smi* representation (since feedback for `+` says that both inputs have always been *Smi* so far). This is done by testing the least significant bit. Once we know that they are both represented as *Smi*, we need to convert them to 32-bit representation, which is done by shifting the value by 32 bit to the right.

If either `x` or `y` is not a *Smi* the execution of the optimized code aborts immediately and the `Deoptimizer` restores the state of the function in the interpreter right before the `Add`.

Side note: We could also perform the addition on the *Smi* representation here; that's what our previous optimizing compiler Crankshaft did. This would save us the shifting, but currently TurboFan doesn't have a good heuristic to decide whether it's beneficial to do the operation on *Smi* instead, which is not always the ideal choice and highly dependent on the context in which this operation is used.

```

# Add x and y (incl. overflow check)
addl rdx,rcx
jo Deoptimize
# Convert result to Smi
shlq rdx, 32
movq rax,rdx
# Epilogue
movq rsp,rbp
pop rbp
ret 0x18

```

Then we go on to perform the integer addition on the inputs. We need to test explicitly for overflow, since the result of the addition might be outside the range of 32-bit integers, in which case we'd need to go back to the interpreter, which will then learn `Number` feedback on `Add`. Finally we convert the result back to *Smi* representation by shifting the signed 32-bit value up by 32 bit and then we return the value in the accumulator register `rax`.



value up by 32 bits, and then we return the value in the accumulator register `rax`.

As said before, this is not yet the perfect code for this case, since here it would be beneficial to just perform the addition on *Smi* representation directly, instead of going to *Word32*, which would save us three shift instructions. But even putting aside this minor aspect, you can see that the generated code is highly optimized and specialized to the profiling feedback. It doesn't even try to deal with other numbers, strings, big ints or arbitrary JavaScript objects here, but focuses only on the kind of values we've seen so far. This is the **key ingredient** to peak performance for many JavaScript applications.

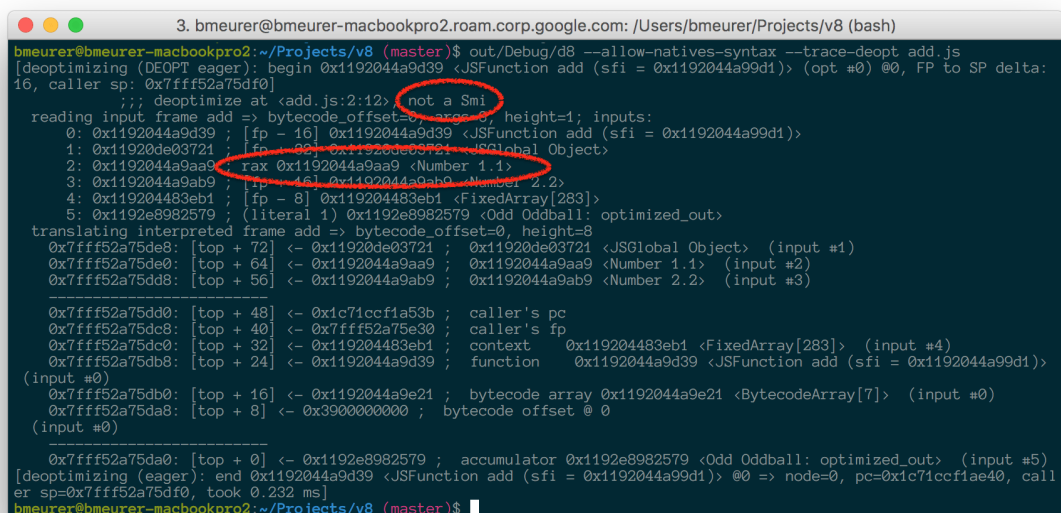
Making progress

So what if you suddenly change your mind and want to add numbers instead? Let's change the example to something like this instead:

```
function add(x, y) {
  return x + y;
}

add(1, 2); // Warm up with SignedSmall feedback.
%OptimizeFunctionOnNextCall(add);
add(1, 2); // Optimize and run generated code.
add(1.1, 2.2); // Oops?!
```

Running this with `--allow-natives-syntax` and `--trace-deopt` we observe the following:



```
3. bmeurer@bmeurer-macbookpro2.roam.corp.google.com: /Users/bmeurer/Projects/v8 (bash)
bmeurer@bmeurer-macbookpro2:~/Projects/v8 (master)$ out/Debug/d8 --allow-natives-syntax --trace-deopt add.js
[deoptimizing (DEOPT eager): begin 0x1192044a9d39 <JSFunction add (sfi = 0x1192044a99d1)> (opt #0) @0, FP to SP delta:
16, caller sp: 0x7fff52a75df0]
    ;; deoptimize at <add.js:2:12> not a Smi
reading input frame add => bytecode_offset=0, args=3, height=1; inputs:
0: 0x1192044a9d39; [fp - 16] 0x1192044a9d39 <JSFunction add (sfi = 0x1192044a99d1)>
1: 0x11920de03721; [fp - 20] 0x11920de03721 <JSGlobal Object>
2: 0x1192044a9aa9; [fp - 24] 0x1192044a9aa9 <Number 1.1>
3: 0x1192044a9ab9; [fp - 28] 0x1192044a9ab9 <Number 2.2>
4: 0x119204483eb1; [fp - 8] 0x119204483eb1 <FixedArray[283]>
5: 0x1192e8982579; (literal 1) 0x1192e8982579 <Odd Oddball: optimized_out>
translating interpreted frame add => bytecode_offset=0, height=8
0x7fff52a75de8: [top + 72] <- 0x11920de03721; 0x11920de03721 <JSGlobal Object> (input #1)
0x7fff52a75de0: [top + 64] <- 0x1192044a9aa9; 0x1192044a9aa9 <Number 1.1> (input #2)
0x7fff52a75dd8: [top + 56] <- 0x1192044a9ab9; 0x1192044a9ab9 <Number 2.2> (input #3)
-----
0x7fff52a75dd0: [top + 48] <- 0x1c71ccf1a53b; caller's pc
0x7fff52a75dc8: [top + 40] <- 0x7fff52a75e30; caller's fp
0x7fff52a75dc0: [top + 32] <- 0x119204483eb1; context 0x119204483eb1 <FixedArray[283]> (input #4)
0x7fff52a75db8: [top + 24] <- 0x1192044a9d39; function 0x1192044a9d39 <JSFunction add (sfi = 0x1192044a99d1)>
(input #0)
0x7fff52a75db0: [top + 16] <- 0x1192044a9e21; bytecode array 0x1192044a9e21 <BytecodeArray[7]> (input #0)
0x7fff52a75da8: [top + 8] <- 0x3900000000; bytecode offset @ 0
(input #0)
-----
0x7fff52a75da0: [top + 0] <- 0x1192e8982579; accumulator 0x1192e8982579 <Odd Oddball: optimized_out> (input #5)
[deoptimizing (eager): end 0x1192044a9d39 <JSFunction add (sfi = 0x1192044a99d1)> @0 => node=0, pc=0x1c71ccf1ae40, call
er sp=0x7fff52a75df0, took 0.232 ms]
bmeurer@bmeurer-macbookpro2:~/Projects/v8 (master)$
```



Deoptimization example

That's a lot of confusing output. But let's extract the important bits. First of all, we print a reason why we had to deoptimize, and in this case it's `not a Smi`, which means we baked in the assumption somewhere that a value is a *Smi*, but now we saw a *HeapObject* instead. Indeed it's the value in `rax`, which is supposed to be a *Smi*, but it's the number 1.1 instead. So we fail on the first check for the `x` parameter and we need to deoptimize to go back to interpreting the bytecode. That is a topic for a separate article though.

Takeaway

I hope you enjoyed this dive into how speculative optimization works in V8 and how it helps us to reach peak performance for JavaScript applications. Don't worry too much about these details though. When writing applications in JavaScript focus on the application design instead and make sure to use appropriate data structures and algorithms. Write idiomatic JavaScript, and let us worry about the low level bits of the JavaScript performance instead. If you find something that is too slow, and it shouldn't be slow, please [file a bug report](#), so we get a chance to look into that.

