Ignition: V8 Interpreter

Authors: rmcilroy@, oth@
Last Updated: 2016/03/22

Background

The machine code generated by V8's full-codegen compiler is verbose, and as such, can contribute significantly to the amount of memory used by V8's heap for typical web-pages (a previous analysis showed that the code-space contributed to around 15-20% of the JS heap).

As well as causing memory pressure, this means that V8 tries very hard to avoid generating code that it thinks might not be executed. It implements *lazy* parsing and compiling, where functions are typically only compiled on first-run. This has a significant cost during webpage startup since the function's source needs to be reparsed during the lazy compilation (e.g., crbug.com/593477).

The aim of the Ignition project is to build an interpreter for V8 which executes a low-level bytecode, thus enabling run-once or non-hot code to be stored more compactly in bytecode form. Since the bytecode is smaller, compilation time is much reduced, and we will also be able to be more eager about initial compilation, which significantly improve startup time. An added advantage is that the bytecode can be fed into a Turbofan graph generator directly, thereby avoiding the need to reparse the JavaScript source code when optimizing a function in TurboFan.

The goals of the Ignition project are:

- Reduce the size of the code-space to around 50% of its current size.
- Have reasonable performance compared to full-codegen (somewhere in the range of 2x slower on peak performance benchmarks such as Octane, significantly less slowdown on real world websites).
    - Note: the slowdown overall will be significantly less, hopefully negligible, due to hot code being optimized by either Crankshaft or TurboFan.
- Full support for DevTools debugging and cpu profiling.
- Replace full-codegen for first-level compilation.
    - We can't replace full-codegen completely until Crankshaft is deprecated, since Crankshaft can't deoptimize to Ignition, it instead needs to deopt to full-codegen code.
- A new frontend to the Turbofan's compiler to enable optimized re-compilation without re-parsing the JS source code.
- Support for deoptimizing from TurboFan to the interpreter.

Explicit non-goals of the project (at least at this stage) are:

- Support for platforms not allowed to JIT code (e.g., iOS)
    - JIT code generation is still required for ICs and code stubs.
- Support non-JavaScript code executed by V8 (e.g., WASM).
- Equivalent performance to full-codegen compiler.
- Completely replacing the full-codegen compiler.
    - As outlined above, we need full-codegen both as a deoptimization target for Crankshaft, and to build some type feedback required by Crankshaft which Ignition cannot provide. As such, full-codegen will become a middle tier for hot code which will eventually be optimized by Crankshaft (any functions which the parser decides should be optimized by TurboFan will not be compiled via full-codegen though, instead they will be optimized directly by TurboFan.

Overall Design

This section outlines the overall design of the Ignition bytecode interpreter, with following sections giving more details on the specifics.

The interpreter itself consists of a set of bytecode handler code snippets, each of which handles a specific bytecode and dispatches to the handler for the next bytecode. These [bytecode handlers] are written in a high level, machine architecture agnostic form of assembly code, as implemented by the `CodeStubAssembler` class and compiled by Turbofan.

As such, the interpreter can be written once and uses TurboFan to generate machine instructions for each of the architectures supported by V8. When the interpreter is enabled, each V8 isolate contains a global interpreter dispatch table, holding a code object pointer to each bytecode handler, indexed by the bytecode value. These bytecode handlers can be included in the startup snapshot and deserialized when a new isolate is created.

In order to be run by the interpreter, a function is translated to bytecode by a `BytecodeGenerator` during its initial unoptimized compile step. The `BytecodeGenerator` is an `AstVisitor` which walks the function's AST emitting appropriate bytecodes for each AST node. This bytecode is associated with the function as a field on the `SharedFunctionInfo` object, and the function's code entry address is be set to the `InterpreterEntryTrampoline` builtin stub.

When the function is called at runtime, the `InterpreterEntryTrampoline` stub is entered. This stub set up an appropriate stack frame, and then dispatch to the interpreter's bytecode handler for the function's first bytecode in order to start execution of the function in the interpreter. The end of each bytecode handler directly dispatches to the next handler via an index into the global interpreter table, based on the bytecode.

The Ignition interpreter is a register-based interpreter. These registers are not traditional machine registers, but are instead specific slots in a *register file* which is allocated as part of a function's stack frame. Bytecodes can specify the input and output registers on which they operate through bytecode arguments, which immediately follow the bytecode itself in the `BytecodeArray` stream.

In order to reduce the size of the bytecode stream, Ignition has an accumulator register, which is used by many bytecodes as implicit input and output register. This register is not part of the *register file* on the stack, but is instead maintained in a machine register by Ignition. This minimize loading and storing repeatedly to memory for register operations. It also reduces the size of bytecode by avoiding the need to specify an input and output register for many operations. E.g., binary op bytecodes only require a single operand to specify one of the inputs, with the other input and the output registers being implicitly the accumulator register, rather than having to explicitly specify all three registers.

Generation of Bytecode Handlers

Bytecode handlers are generated by the TurboFan compiler. Each handler is its own code object and is generated independently. The handlers are written as a graph of Turbofan operations, via an `InterpreterAssembler`, which is a subclass of the `CodeStubAssembler`, with some extra high-level primitives required by the interpreter (e.g., `Dispatch`, `GetBytecodeOperand`, etc.). An example of the generator function for the `Ldar` (LoaD Accumulator from Register) bytecode handler is as follows:

```
void Interpreter::DoLdar(InterpreterAssembler* assembler) {
  Node* reg_index = __ BytecodeOperandReg(0);
  Node* value = __ LoadRegister(reg_index);
  __ SetAccumulator(value);
  __ Dispatch();
}
```

The bytecode handlers are not intended to be called directly, instead each bytecode handler dispatches to the next bytecode. Bytecode dispatch is implemented as a tail call operation in TurboFan. The interpreter loads the next bytecode, indexes into the dispatch table to get the code object of the target bytecode handler, and then tail calls the code object to dispatch to the next bytecode handler.

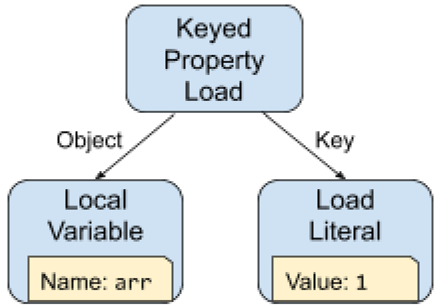
The interpreter also needs to maintain state across bytecode handlers in fixed machine registers. For example, a pointer to the `BytecodeArray`, the current bytecode offset and the value of the interpreter's accumulator. These values are treated as parameters by TurboFan, which are received as input from the previous bytecode dispatch, and passed to the next bytecode handler as parameters to the bytecode dispatch tail call. The bytecode handler dispatch calling convention specifies fixed machine registers for these parameters, which enables them to be threaded through interpreter dispatches without needing to be pushed / popped onto the stack.

Once the graph for a bytecode handler is produced it is passed through a simplified version of Turbofan's pipeline and assigned to the corresponding entry in the interpreter table.

Bytecode Generation

To compile a function to bytecode, the JavaScript code is parsed to generate its AST (Abstract Syntax Tree). The `BytecodeGenerator` walks this AST and generates bytecode for each of the AST nodes as appropriate.

For example, the snippet of Javascript "`arr[1]`" will be translated into the following AST tree:



The `BytecodeGenerator` will walk this tree, first visiting the `KeyedPropertyLoad` node, which will first visit the *Object* edge to generate code which evaluates the object on which the keyed load should be performed. In this case, the object is a local variable, which is already assigned to an interpreter register (e.g., `r3`), so code is generated to load this register into the accumulator (`Ldar r3`) and control returns to `KeyedPropertyLoad` node visitor. This visitor allocates a temporary register (e.g., `r6`) to save the object in, and generates code to store the accumulator into this register (`Star r6`)[1]. The *Key* edge is now visited to generate code to produce the key for the property load. In this case this node is an integer literal 1, and so a `LdaSmi #1` bytecode is output to load 1 into the accumulator. Finally the code to perform the keyed property load is output, resulting in a snippet of bytecode:

```
Ldar r3
Star r6
LdaSmi #1
KeyedLoadIC r6 <feedback slot>
```

The `BytecodeGenerator` uses a `BytecodeArrayBuilder` to generate well formed array of bytecodes for the interpreter. The `BytecodeArrayBuilder` provides flexibility in what raw bytecodes are emitted. For example, we can have multiple bytecodes which perform the same semantic operation, but with some having wider (e.g., 16 bit or 32 bit) operands. The `BytecodeGenerator` doesn't need to know any of this, instead it simply asks the `BytecodeArrayBuilder` to output a given bytecode with a set of operands, and the `BytecodeArrayBuilder` chooses the appropriate width for the operands.

Once the bytecode is generated it is stored on a field in the `SharedFunctionInfo`. As well as being used by the interpreter for code execution, this bytecode is representative enough to enable generation of the [TurboFan's compiler's graph] without having to regenerate the function's AST. This avoids the current requirement to reparse the function's JavaScript source before recompile.

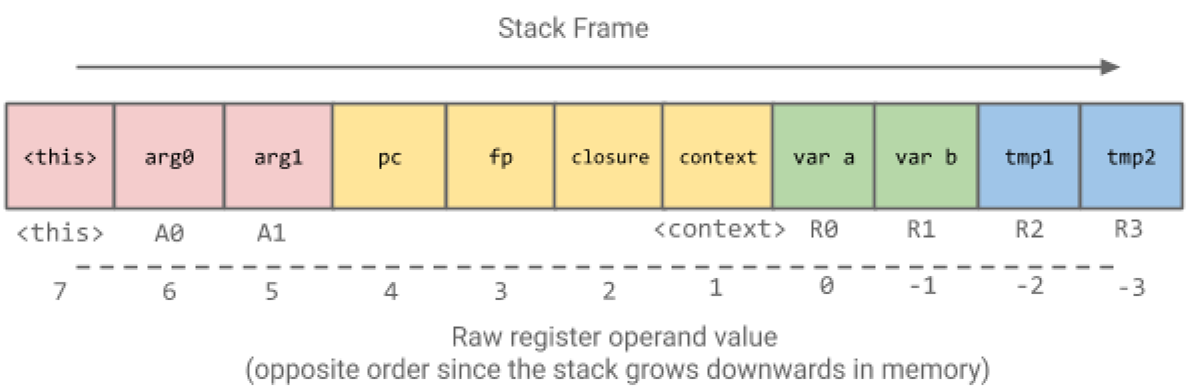
[1] Note: Since the object was already in a register, this move in and out of the accumulator is redundant, therefore we have register aliasing optimizations which avoid it, described in more detail in Section TODO.

Interpreter Register allocation

During bytecode generation, the `BytecodeGenerator` allocates registers in a function's register file for local variables, [context object pointers](#) (used to maintain state across function closures), and temporary values required for expression evaluation (the expression stack). During execution, space for the register file is allocated during the function's prologue as part of the function's stack frame. Bytecodes operate on these registers by specifying a specific register index as an bytecode operand, which the interpreter uses to load from or store to the specific stack slot associated with this register.

Since the register indexes map directly to the function stack frame slots, the interpreter can also directly access other slots on the stack as registers. For example, the function context and function closure pointers which are pushed onto the stack by the prologue can be accessed directly as `Register::current_context()` and `Register::function_context()` by any bytecode which has register operands. Similarly, the arguments passed to the function (including the implicit <this> parameter) can also be accessed by register.

The figure below shows an example stack frame for a function and the mapping of registers indexes and their raw operand values, to stack slots:



Due to this register file approach, Ignition doesn't dynamically push and pop values onto the stack during expression evaluation as full-codegen does (the only exception is pushing arguments for calls, however this is done in a separate builtin, not in the interpreter itself). This has the advantage the the stack frame can be allocated once during the function prologue and can remain aligned to architecture specific requirements (e.g., Arm64's 16 byte stack alignment requirement). However, it does mean that the `BytecodeGenerator` needs to calculate the maximum size of the stack frame during code generation.

Locals variables declared within the function are semantically hoisted to the top of the function by the parser. As such the number of locals is known ahead of time, and each local is assigned an index in the register file during the initial stage of AST walking. The number of extra registers required for inner contexts is also known ahead of time by the parser, and thus can be preallocated by the `BytecodeGenerator` during the initial stage of AST walking.

Temporaries are required during expression evaluation, where there may be one or more live registers holding intermediate values in the expression tree until they are consumed. The `BytecodeGenerator` allocates registers using a scoped `BytecodeRegisterAllocator`. On each statement, a new scoped allocator is created for the statement, which is used by inner expression nodes to allocate temporary registers. These temporaries are only live for the lifetime of the expression statement, therefore when `BytecodeGenerator` finishes visiting this statement, the scoped allocator will go out of scope, and all temporary registers allocated by it will be released. The `BytecodeGenerator keeps note of the` maximum number of temporary registers which were allocated over the whole function's code generation, and allocates enough extra slots in the register file for the largest number of temporaries required.

The layout of the register file will consist of the locals followed by the temporary registers, with no overlap between locals and temporary registers to keep access simple. Once the emitter has calculated the total size of the register file, it will use this to calculate the required frame size of the function, and will save this value in the `BytecodeArray` object.

On entry to the function, the interpreter prologue will increment the stack pointer amount required to allocate space for the register file on the stack frame. All entries in the register file will initially be tagged pointers to undefined_value (like the locals pushed by full-codegen's prologue). The interpreter will only every store tagged pointers in the entries of the register file, so the GC can walk over any register files on the stack visiting all entries as tagged pointers.

Context chains

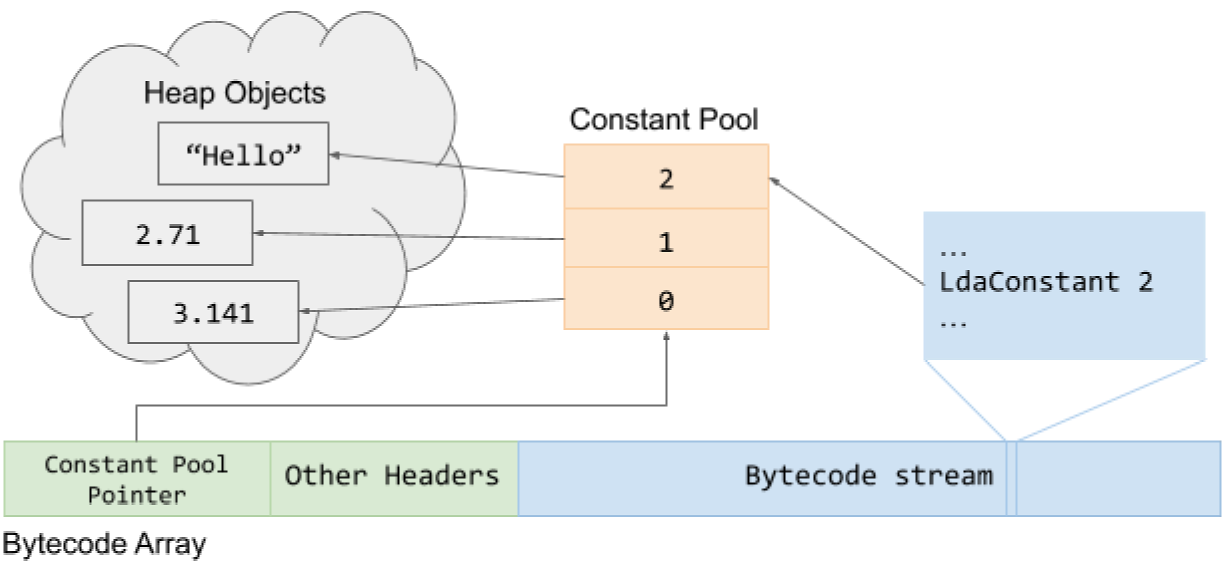
The interpreter tracks the current context object in the context stack slot (which is part of the function's fixed frame, and is accessed by bytecodes via `Register::current_context()`). When a new context is allocated, the `BytecodeGenerator` allocates a `ContextScope` object to track nested context chains. This allows the `BytecodeGenerator` to *unroll* nested context chain operations, enabling the interpreter to directly access any variables allocated within an inner context extension without having to walk the context chain.

When a `ContextScope` object is allocated, a `ContextPush` bytecode is emitted. This bytecode moves the current context object into a register allocated by the `BytecodeGenerator`, and pushes the new context into the current context register. When an operation is performed on a variable which was allocated to a local context, the `BytecodeGenerator` finds the associated `ContextScope`'s, and checks which register the context object is currently allocated in, and can then directly load the variable out of the correct context slot in the context in pointed to by this register, rather than having to walk up the context chain to find the correct context. When the `ContextScope` goes out of scope, a `ContextPop` bytecode is emitted, which restores the parent context into the current context register.

By maintaining the current context as an explicit register, rather than only relying on the `BytecodeGenerator` to keep track of which register holds the current context, enables the interpreter to perform operations which require the current context (e.g., passing the context to JS function calls or runtime builtin operations) without having to take an extra operand to specify which register holds the current context.

Constant Pool Entries

Constant pools are used to store heap objects and small integers that are referenced as constants in generated bytecode. Each `BytecodeArray` may have its own constant pool embedded in the `BytecodeArray` object. The constant pool is a `FixedArray` of pointers to heap objects. Bytecodes refer to constants in the constant pool by the index of each constant in the `FixedArray.` The indices are emitted as immediate unsigned operand values for bytecodes that take constant pool entries as inputs.



Strings and heap numbers are always placed into the constant pool as they live in the heap and are referenced by pointers. Small integers and the strong referenced oddball type’s have bytecodes to load them directly and do not go into the constant pool. The constant pool can also store forward jump offsets and this is discussed below.

The constant pool is built during bytecode generation and there is a `ConstantArrayBuilder` class that is responsible for this. When bytecode needs to reference a constant, for instance loading a string, it passes the object to the constant array builder and asks for a constant pool index for the object. The `ConstantArrayBuilder` checks if the object is already present in the array and returns the existing index if so, otherwise it adds the object to the end of the constant pool being built and returns the index for the object. At the end of bytecode generation, the `ConstantArrayBuilder` emits a fixed array with the necessary object pointers.

The logic described so far is relatively simple. Complexity comes from the use of the constant pool to store forward jump offsets. The issue with forward jumps is that the jump displacement is not known at the time the jump bytecode is emitted so a space is reserved for an jump displacement offset in the bytecode stream. The `BytecodeArrayBuilder` keeps track of this location and patches the jump when the displacement is known.

In a simplified scenario, the bytecode generator emits a forward jump with an immediate byte operand left blank in the stream. When the displacement is known, the jump is patched. If the displacements fits within a byte operand then the displacement can be written directly into the blank operand. If the displacement is larger than a byte, then the displacement is put into the constant pool and both the jump bytecode and the operand are updated. The jump bytecode is updated to be a jump with a constant pool operand and the operand is updated to be the constant pool entry.

In practice, the constant pool grows as code is generated and there are no restrictions on how large it can be. When emitting code, the `BytecodeArrayBuilder` needs to know what flavor of jump bytecode bytecode to emit for forward jumps: a jump taking a byte operand, or two byte operand, or four byte operand. Therefor the `ConstantArrayBuilder` class supports reserving constant pool entries. The reservation guarantees an index of a particular size will be available in future even though the value for the index is not currently known. The `BytecodeArrayBuilder` creates a reservation when it emits a forward jump and emits the bytecode having an operand size that matches the reservation made in the `ConstantArrayBuilder`. When the jump is patched, the reservation is cancelled if the displacement fits inside an immediate operand of the same size as the reservation. Otherwise, the constant pool reservation is committed to be the jump displacement value and the jump bytecode and operand patched to use the jump displacement value in the constant pool.

Local Control Flow

Javascript has the usual set of imperative control flow primitives at the language level such as `if` statements, `if..else` statements, `for` loops, `while` loops, `do` loops, and `switch` statements. It also has the the language specific construct `for..in` for iterating over object properties and also breakable named blocks. These control flow constructs only affect execution within a function compiled to bytecode.

The `BytecodeGenerator` converts the AST forms of Javascript's flow control statements into control flow builder class instances that generate the corresponding bytecode. Control flow at the bytecode level is implemented as a set of bytecodes for conditionally and unconditionally jumping. Supporting Javascript's for..in requires additional bytecodes to get the information to be iterated over and advancing to elements in the collection.

To facilitate building control flow structures, the `BytecodeArrayBuilder` supports the notion of labels that allow offsets to be specified. When the position of a label is known, the BytecodeGenerator binds the label to the current position with the `BytecodeArrayBuilder::Bind()` call. The `BytecodeGenerator` can refer to a label before it is bound when generating code for forward jumps. The `BytecodeArrayBuilder` checks that all labels referenced in bytecode are bound before the `BytecodeArray` is emitted. Because the Bind() call binds a label to the end of the bytecode being generated, labels are always constrained to target offsets within the bytecode array. The target offset for an emitted jump is present as either an immediate operand value in the bytecode or in the constant pool. There is no support for jumping to a dynamically computed value, for instance jumping to an offset in a register.

An example generating control flow is shown below for an `if..else` statement.

```
// Javascript
if (arg == true) {
  print("Hello World\n");
} else {
  print("Mmmm\n");
}
```

```
// BytecodeGenerator (simplified)
void VisitIfStatement(IfStmt* stmt) {
  BytecodeLabel else_label,
                end_label;
  VisitForAccumulatorValue(
    stmt->condition());
  builder()->JumpIfFalse(&else_label);
  Visit(stmt->then_statement());
  if (stmt->HasElseStatement()) {
    builder()->Jump(&end_label);
    builder()->Bind(&else_label);
    Visit(stmt->else_statement());
  } else {
    builder()->Bind(&else_label);
  }
  builder()->Bind(&end_label);
}
```

```
// Generated Bytecode
    LdaTrue
    TestEqual r0
    JumpIfFalse else_label
    ...
    Call r0, r1, #2, [1]
    Jump end_label
else_label:
    ...
    Call r0, r1, #2, [5]
end_label:
```

`If..else` is the simplest control flow structure emitted by the generator. For the more complex control flow structures like loops and breakable blocks, the `BytecodeGenerator` employs control flow builders and control scopes.

All loops use instances of the same control flow builder, the `LoopBuilder`. The `LoopBuilder` has a set of labels for the loop condition, the loop header, the target of continue/next, and the end of the loop. The visitor for the loop statement establishes the label positions of the loop header, loop condition, and the of end loop. It also visits the statements in the iteration body by calling `BytecodeGenerator::VisitIterationBody`. This visitor body instantiates a `ControlScopeForIteration` and pushes this onto a stack in the `BytecodeGenerator` and then visits the statements in the loop body. If a break statement or continue statement is hit, the visitor for these statements peeks at the current `ControlScopeForIteration` and signals a break (or continue). The control scope then invokes the appropriate method on its associated `LoopBuilder` which emits the appropriate jumps to its labels.

The `ControlFlowBuilder` classes ensure the emitted bytecode has the same control flow order as the source code. This natural ordering means the only time backwards branches are emitted in bytecode is in loop statements returning to the head of the loop. The ordering means no additional loop analysis need be performed when converting from bytecode form into TurboFan compiler graph form. The branch analysis for graph building need only identify the sites of jumps and their targets.

Exception Handling

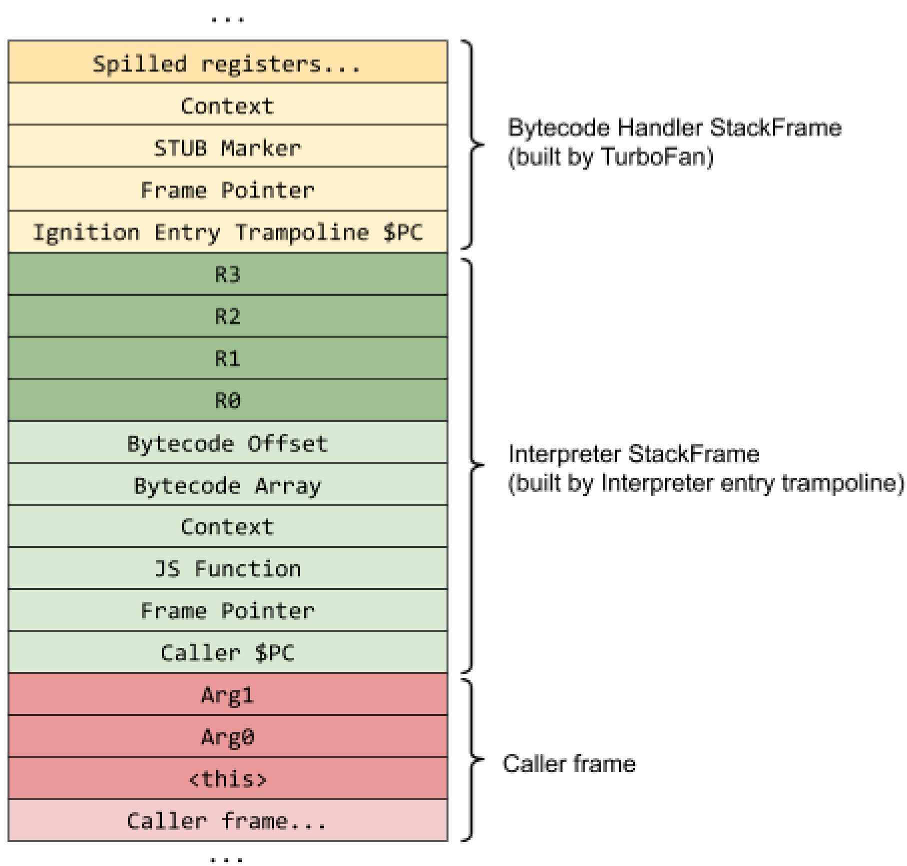
TODO

Interpreter Code Execution

The Ignition interpreter is a register-based, indirect threaded interpreter. The entry point to the interpreter from other JavaScript code is the `InterpreterEntryTrampoline` builtin stub. This builtin sets up an appropriate stack frame, and loads reserved machine registers (e.g., the bytecode pointer, interpreter dispatch table pointer) with the appropriate values before dispatching to the first bytecode of the called function.

Stack frame layout and reserved machine registers

An example of the layout of a stack frame for the interpreter is shown below:



The interpreter stack frame is built by the `InterpreterEntryTrampoline` builtin stub, which pushes the fixed frame values (Caller PC, Frame Pointer, JSFunction, Context, Bytecode Array and Bytecode Offset) and then allocates space in the stack frame for the register file (the `BytecodeArray` object contains an entry which tells the builtin how large a stack frame the function requires). It then writes `undefined` to all the registers in this register file, which ensures the GC doesn't see any invalid (i.e., non-tagged) pointers when it walks the stack.

The `InterpreterEntryTrampoline` builtin stub initializes some fixed machine registers which are used by the interpreter, namely:

- `kInterpreterAccumulatorRegister`: stores the implicit accumulator interpreter register
- `kInterpreterBytecodeArrayRegister`: points the the start of the `BytecodeArray` object which is being interpreted.
- `kInterpreterBytecodeOffsetRegister:` the current offset of execution in the `BytecodeArray` (in essense the bytecode PC).
- `kInterpreterDispatchTableRegister:` points to the interpreter's dispatch table, used to dispatch to the next bytecode handler.
- `kInterpreterRegisterFileRegister`: points to the start of the register file (will soon be removed now that TurboFan can use the parent frame pointer directly).
- `kContextRegister:` points to the current context object (will soon be removed and all access will be via interpreter register `Register::current_context()`

It then *calls* the bytecode handler for the first bytecode in the bytecode array stream. The registers initialized above are available to the bytecode handlers as TurboFan parameter nodes, due to the fact that the calling convention for bytecode dispatch specifies associated parameters for each of the fixed machine registers.

If the bytecode handler is simple and doesn't call any other function (other than the tail call in the dispatch operation) then TurboFan can elide stack frame creation for this bytecode handler. However, in more complex bytecode handlers, TurboFan will build a new `STUB` frame when execution enters the bytecode handler. Other than the required fixed frame, this frame only stores machine register values spilled by TurboFan (e.g., callee saved registers across a call). All the state associated with the interpreted function lives in the parent interpreted frame.

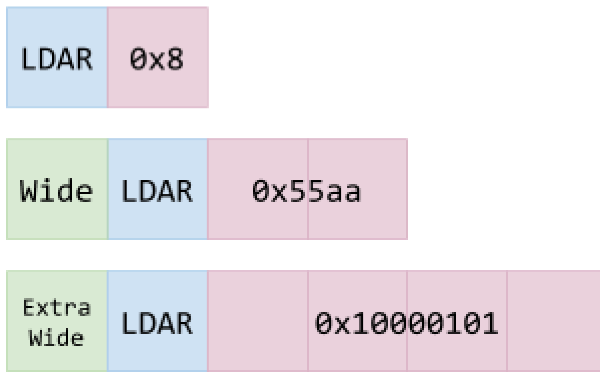
Interpreter Register Access

As described <u>above</u>, all local variables and temporaries will be allocated to specific registers within the function's *register file,* on its stack frame, during bytecode generation. These variables, as well as the function's arguments, can be accessed by bytecode handlers as interpreter registers. The register on which a bytecode handler should operate is specified as a bytecode operand in the bytecode stream. The interpreter loads the operand from the bytecode stream to get the register index on which it should operate. This index is a word-based offset from the start of the register file, which can be either positive (to access which live above the register file on the stack, such as function arguments) or negative (to access locals allocated by the function itself on its register file). The interpreter accesses a given register by scaling the operand's index to be a byte offset, and then loading from or store to the memory location at `kInterpreterRegisterFileRegister + offset`.

Wide operands

Minimizing the size of generated bytecode is a major motivator for Ignition so the bytecode format is able to support operands of different widths. Ignition uses prefix special bytecodes to support wider operands. Ignition supports fixed width operand types and scalable width operands. The scalable width operands increase in size proportional to the prefix bytecode.

The fixed width operands are for runtime call identifiers (16-bits) and flag operands (8-bits). Register operands, immediate value operands, index value operands, and register count operands all have a base size of 8-bits and are scalable up to 32-bits wide.



The effects of prefix bytecodes on the load accumulator bytecode

The `Wide` prefix bytecode doubles the width of scalable operands making them 16-bits wide, and the `ExtraWide` prefix quadruples the width to 32-bits.

To support dispatching prefixed, the bytecode dispatch table has entries for the scalable operands offset by 256 for each scaling so indices between 0-255 correspond to unscaled bytecodes, 256-511 correspond to bytecodes with the `Wide` prefix, and 512-767 correspond to bytecodes with the `ExtraWide` prefix.

Earlier Ignition designs, placed wide bytecodes into the same 8-bit space limiting the number of free bytecodes that could be used for other purposes such as bytecode specialization. The designs also had an elaborated register translation scheme for accessing wide register operands as it was not feasible to have scaled versions of all bytecodes. Using prefix bytecodes is cleaner and the overhead of 1-byte prefix is only incurred when used and the size overhead was less than 1% on the Octane-2.1 benchmark.

JS Calls

Calls to other JS functions are handled by the `Call` bytecode. The `BytecodeGenerator` ensures that the arguments which will be passed to the call are in a contiguous set of registers. It then emits `Call` bytecode with a register operand specifying the register which holds the callee, a second register operand specifying the register which holds the start of the arguments to be passed to the call, and a third operand which specifies the number of arguments to pass.

The interpreter `Call` bytecode handler updates the *bytecode offset* stack slot in the interpreted frame with the current bytecode offset. This allows stack walking code to calculate the bytecode pc for frames on the stack. It then calls the `InterpreterPushArgsAndCall` builtin, passing the callee, the memory location of the first argument register, and the number of arguments. The `InterpreterPushArgsAndCall` builtin then copies the argument values from the range <first_arg_register>...<first_arg_register + count> and pushes these values to the end of the stack. It then invokes the `Call` builtin, which is the same builtin used by full-codegen to evaluate the target address to call for the given callee. Calls to other interpreted functions are treated the same as calls to JITed functions - the `Call` builtin will load the `JSFunction`'s code entry field, which will point to the `InterpreterEntryTrampoline` builtin stub for interpreted functions, and thus the call will invoke the `InterpreterEntryTrampoline` and reenter the interpreter.

When an interpreted function returns, the interpreter tail calls the `InterpreterExitTrampoline` builtin stub, which tears-down the stack frame and restore control to the calling function, returning the value held in the accumulator at the time of the `return` bytecode as the return value.

Property loads / stores

Bytecodes which load or store property on JS objects do so via inline caches (ICs). The bytecode handlers call the same LoadIC and StoreIC code stubs as are used by JITed code (e.g., full-codegen). Since the Load/StoreICs no longer patch code (instead operating directly on the `TypeFeedbackVector`) this means the same ICs can be used for both JITed code and Ignition.

The bytecode handler passes the function's `TypeFeedbackVector`, to the appropriate IC stub along with the type feedback slot associated with the AST node of the operation. The IC stub can update entries in the `TypeFeedbackVector` in the same manner as it does now for the full-codegen compiler, enabling learning of type feedback for later use by the optimizing compiler.

Binary ops

For BinaryOps and other UnaryOps (e.g., `ToBoolean`) full-codegen currently employs ICs which patch machine code. We can't use these ICs in Ignition since they would patch the bytecode handler code (which is not a function specific call site). As such, we currently don't collect any type feedback for Binary or Unary operations.

At present, most Binary and Unary ops call runtime functions which perform the given operations. We are in the process of replacing these calls with calls to stubs written using the `CodeStubAssembler` which perform the common cases inline, only calling out to the runtime for more complex situations. Currently we call these stubs, however since they are written using the `CodeStubAssembler`, we will be able to inline the code directly into the interpreter's bytecode handlers easily.

As a future optimization, we may employ back-patching of the actual bytecodes to point to specialized bytecode handlers for a given operation (e.g., inlining `SmiAdd` as a specific bytecode), however this is dependent upon the decision we end up making regarding collection of type feedback for BinaryOps and UnaryOps in Ignition.

TurboFan Bytecode Graph Builder

The `BytecodeArray` built by the `BytecodeGenerator` contains all the information necessary to be fed into Turbofan and built a TurboFan compilation graph. This is implemented by the `BytecodeGraphBuilder`.

The `BytecodeGraphBuilder` starts by walking the bytecode to perform some basic branch analysis, which finds backward and forward branch target bytecodes. This is used to set up the appropriate loop header environments in the `BytecodeGraphBuilder` while building the TurboFan graph.

`BytecodeGraphBuilder` then walks the `BytecodeArray` again, visiting every bytecode and calling a bytecode specific visitor for each. The bytecode visitor functions read the bytecode operands from the `BytecodeArray`, and then add operations to the TurboFan graph to perform the operation of that bytecode. Many bytecodes have a direct mapping to an existing `JSOperator` in TurboFan.

The `BytecodeGraphBuilder` maintains an environment which tracks which nodes represent to value stored in each of the registers in the `BytecodeArray`'s register file (including the accumulator register). The environment also tracks the current context object (which is updated due to `Push/PopContext` bytecodes). When the visitors visit a bytecode which reads a value from register, it looks up the node in the environment associated with that register, and uses this as the input node to the `JSOperator` node being built for the current bytecode. Similarly, for operations which output a value to a register or the accumulator, the visitor will update the register in the environment with the new node when it is complete.

Deoptimization

Javascript operations can trigger deoptimizations from optimized code back to the unoptimized bytecode during execution. In order to support deoptimization, the `BytecodeGraphBuilder` needs to keep track of the state of the interpreter's stack frame so that it can rebuild the interpreter stack frame on deoptimization and re-enter the function at the deoptimization point.

The environment already tracks this, by keeping track of the nodes associated with each register at each point of the bytecode execution. The `BytecodeGraphBuilder` checkpoints this information in a `FrameState` node for `JSOperator` nodes which could trigger a deoptimization (either eager - with the checkpoint based on the state before the node executes, or lazy - with the checkpoint after the node has executed).

Since each bytecode maps to a single `JSOperator` node, this means that we will only ever deoptimize to a point immediately before or immediately after a bytecode. As such, we use the bytecode offset as the `BailoutId` in the deoptimization points. When TurboFan generates code to deal with potential deoptimization, it serializes a `TranslatedState` record which describes how to rebuild an interpreted frame for this deoptimization point. This is based on `FrameState` node for this deoptimization point. When this deopt point is hit, the interpreted frame is built using this `TranslatedState` record, and the runtime re-enters into the interpreter at the appropriate bytecode offset using the `InterpreterNotifyDeoptimized` builtin.

Debugging support

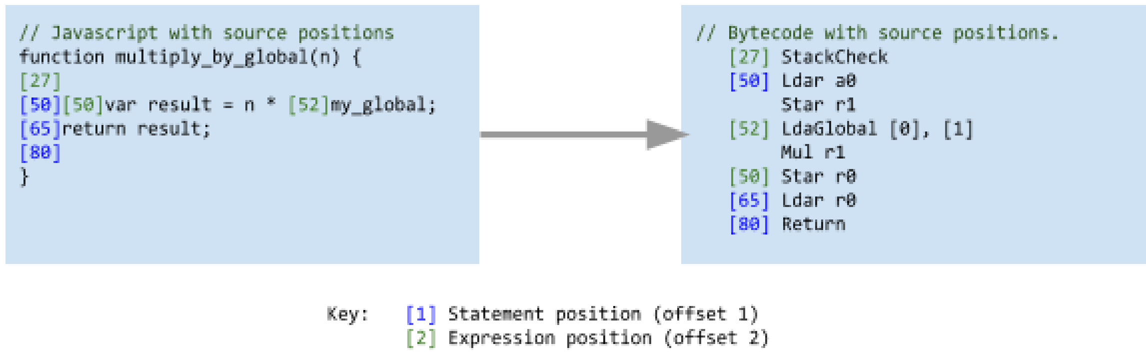
To enable the debugger breakpoint code executed by the interpreter, the debugger copies the `BytecodeArray` object of the function, and then replace any bytecode which is the target of a breakpoint with a special `DebugBreak` bytecode. Since all bytecodes are at least one byte, no extra space needs to be allocated in the bytecode stream to ensure breakpoints can be set. There are variants of the `DebugBreak` bytecode for each size of bytecodes to ensure that the `BytecodeArray` can still be iterated correctly. There are also `DebugBreakWide` and `DebugBreakExtraWide` bytecodes, which as well as acting as breakpoints, also prefix the next bytecode as being a `Wide` or `ExtraWide` variant (in the same manner as the `Wide` and `ExtraWide` bytecodes on non-breakpoint code).

Once a breakpoint is hit, the interpreter calls into the debugger. The debugger can resume execution by looking at the function's real `BytecodeArray` and dispatching to the actual bytecode which was replaced by the `DebugBreak` bytecode.

More details are available in Debugging support for Ignition design doc.

Source Positions

Source positions are emitted alongside bytecode when bytecodes are emitted by the bytecode generator. There are two types of source position: statement positions and expression positions. Statement positions are used for stepping between statements in the debugger. A statement position occurs a location where the debugger can step to before the bytecode it is associated with is dispatched. Expression positions are used to provide stack trace information when exceptions are raised in expressions. When an exception is raised the debugger scans back from the site of the exception looking for an expression position and this is used in the stack trace.



Source positions in source code and bytecode.

The bytecode generation process needs to maintain source positions for the debugger and exception reporting to work correctly. Any optimizations in the bytecode generation process need to ensure source positions have the same causal ordering. Some expression positions can be elided because they are associated with bytecodes that can not produce an exception or are duplicates.

Profiler Support

TODO

Future work

TODO - describe possible specialized bytecode patching optimizations.
TODO - describe possible super-bytecodes.
TODO - describe type feedback for binary ops.

Appendix A: Table of Bytecodes

Prefix Bytecodes
Wide
ExtraWide

Accumulator Load
LdaZero
LdaSmi
LdaUndefined
LdaNull
LdaTheHole
LdaTrue
LdaFalse
LdaConstant

Load/Store Globals
LdaGlobal
LdaGlobalInsideTypeof
StaGlobalSloppy
StaGlobalStrict

Context operations
PushContext
PopContext
LdaContextSlot
StaContextSlot

Unary Operators
Inc
Dec
LogicalNot
TypeOf
DeletePropertyStrict
DeletePropertySloppy

Control Flow
Jump
JumpConstant
JumpIfTrue
JumpIfTrueConstant
JumpIfFalse
JumpIfFalseConstant
JumpIfToBooleanTrue
JumpIfToBooleanTrueConstant
JumpIfToBooleanFalse
JumpIfToBooleanFalseConstant
JumpIfNull
JumpIfNullConstant
JumpIfUndefined
JumpIfUndefinedConstant
JumpIfNotHole
JumpIfNotHoleConstant

Load-Store lookup slots
LdaLookupSlot
LdaLookupSlotInsideTypeof
StaLookupSlotSloppy
StaLookupSlotStrict

Register Transfers
Ldar
Mov
Star

LoadIC operations
LoadIC
KeyedLoadIC

StoreIC operations
StoreICSloppy
StoreICStrict
KeyedStoreICSloppy
KeyedStoreICStrict

Binary Operators
Add
Sub
Mul
Div
Mod
BitwiseOr
BitwiseXor
BitwiseAnd
ShiftLeft
ShiftRight
ShiftRightLogical

For..in support
ForInPrepare
ForInDone
ForInNext
ForInStep

Stack guard check
StackCheck

Non-local flow control
Throw
ReThrow
Return

Illegal bytecode
Illegal

Calls
Call
TailCall
CallRuntime
CallRuntimeForPair
CallJSRuntime

Intrinsics
InvokeIntrinsic

New operator
New

Test Operators
TestEqual
TestNotEqual
TestEqualStrict
TestLessThan
TestGreaterThan
TestLessThanOrEqual
TestGreaterThanOrEqual
TestInstanceOf
TestIn

Cast operators
ToName
ToNumber
ToObject

Literals
CreateRegExpLiteral
CreateArrayLiteral
CreateObjectLiteral

Closure allocation
CreateClosure

Arguments allocation
CreateMappedArguments
CreateUnmappedArguments
CreateRestParameter

Debugger Support
DebugBreak0
DebugBreak1
DebugBreak2
DebugBreak3
DebugBreak4
DebugBreak5
DebugBreak6
DebugBreakWide
DebugBreakExtraWide

Appendix B: Reading Material

Threaded Code, in Wikipedia entry, https://en.wikipedia.org/wiki/Threaded_code

Revolutionizing Embedded Software, Kasper Verdich Lund and Jakob Roland Andersen, in Master's Thesis at University of Aarhus, http://verdich.dk/kasper/RES.pdf.

Inside JavaScriptCore's Low-Level Interpreter, Andy Wingo's blog, https://wingolog.org/archives/2012/06/27/inside-javascriptcores-low-level-interpreter

SquirrelFish, David Manelin's blog, https://blog.mozilla.org/dmandelin/2008/06/03/squirrelfish/

Context Threading, Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown, in CGO '05: Proceedings of the international symposium on Code generation and optimization, http://www.cs.toronto.edu/syslab/pubs/demkea_context.pdf (also Mathew Zeleski's thesis http://www.cs.toronto.edu/~matz/dissertation/)

Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters, M. Anton Ertl and David Gregg, in PLDI'03, http://www.eecg.toronto.edu/~steffan/carg/readings/optimizing-indirect-branch-prediction.pdf

The Case for Virtual Register Machines, Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron, in Interpreters, Virtual Machines, and Emulators (IVME'03), http://dl.acm.org/citation.cfm?id=858575

Virtual Machine Showdown: Stack vs Registers, Yuhne Shi, David Gregg, Andrew Beatty, and M. Anton Ertl, in VEE'05. https://www.usenix.org/legacy/events/vee05/full_papers/p153-yunhe.pdf

vmgen - A Generator of Efficient Virtual Machine Interpreters, M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan, in Software: Practice and Experience, 2002. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.16.7676&rep=rep1&type=pdf

The Self Bibliography, https://www.cs.ucsb.edu/~urs/oocsb/self/papers/papers.html

Interpreter Implementation Choices, http://realityforge.org/code/virtual-machines/2011/05/19/interpreters.html

Optimizing an ANSI C Interpreter with Superoperators, Todd A. Proebstring, in POPL'95, http://dl.acm.org/citation.cfm?id=199526

Stack Caching for Interpreters, M Anton Ertl, in SIGPLAN '95 Conference on Programming Language Design and Implementation, http://www.csc.uvic.ca/~csc485c/Papers/ertl94sc.pdf

Code sharing among states for stack-caching interpreter, Jinzhan Peng, Gansha Wu, Guei-Yuan Lueh, in Proceedings of the 2004 workshop on Interpreters, Virtual Machines, and Emulators, http://dl.acm.org/citation.cfm?id=1059584

Towards Superinstructions for Java Interpeters, K. Casey, D. Gregg, M. A. Ertl, and A. Nisbet, in Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems, http://rd.springer.com/chapter/10.1007/978-3-540-39920-9_23

Branch Prediction and the Perfomance of Interpreters - Don't Trust Folklore, Erven Rohou, Bharath Narasimha Swamy, Andre Seznec. International Symposium on Code Generation and Optimization, Feb 2015, Burlingame, United States. https://hal.inria.fr/hal-00911146/document