38 minutes — Written by amarekano

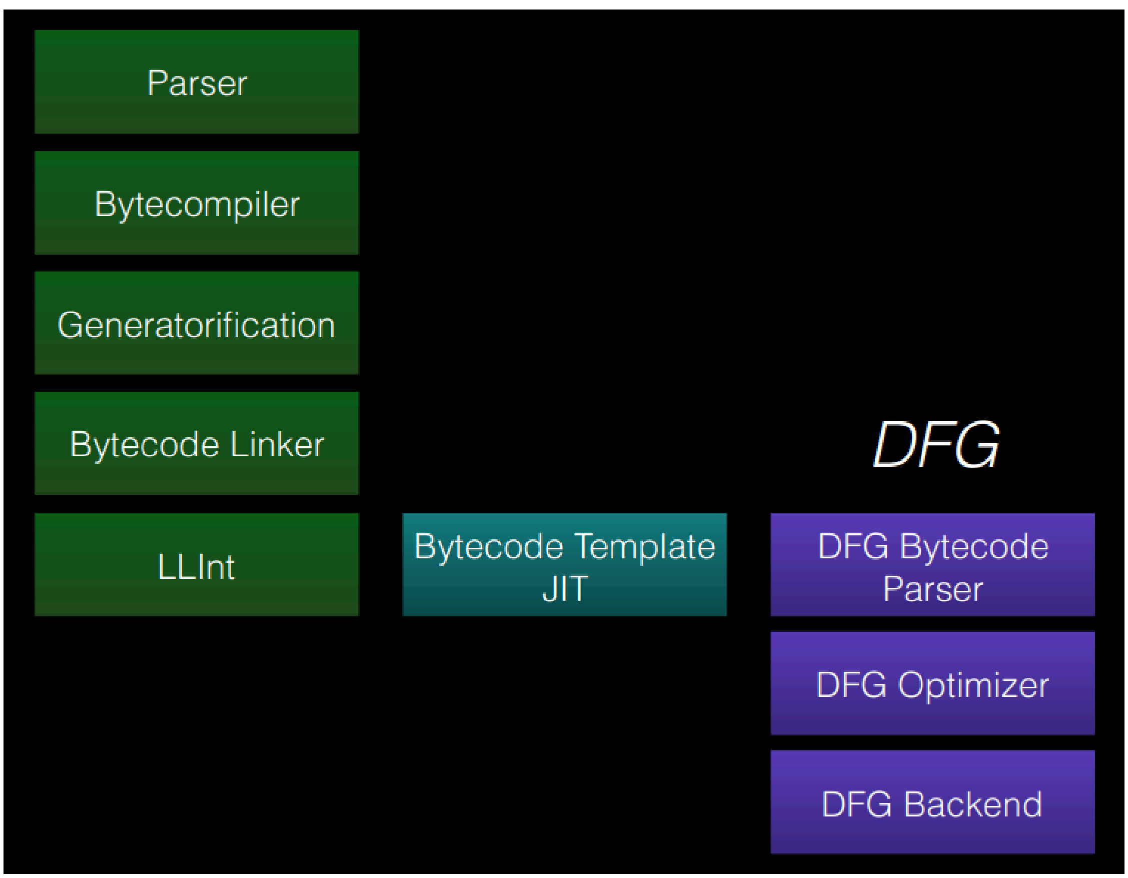
JavaScriptCore Internals Part III: The DFG (Data Flow Graph) JIT – Graph Building

Table of Contents

Introduction

The DFG (Data Flow Graph) and the FTL (Faster Than Light) are the two optimising compilers used by JavaScriptCore and have been the source of a number of JIT bugs that lead to type confusions, OOB (Out-Of-Bounds) access, information leaks, etc. Some of these have been successfully exploited as part of various Pwn2Own[1] [2] [3] competitions targeting Safari.

Part II examined the LLInt and Baseline JIT and explored how JavaScriptCore tiers up from one to the other and how the Baseline JIT optimises bytecode execution. This blog post will cover the DFG internals and focus on parsing bytecode to generate a DFG graph and reading DFG IR. The screenshot[4] below shows the previously explored stages of JSC and the DFG pipeline:



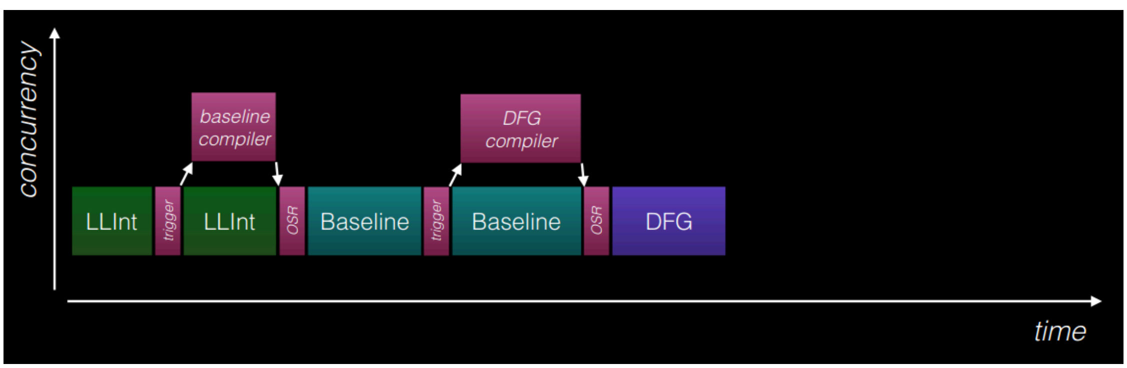
This blog begins with a review of how the Baseline JIT tiers up to the DFG and triggers compilation by the DFG. It then explores the process of parsing bytecode to generate an unoptimised graph and provides the reader with an understanding of the DFG IR syntax and semantics. The blog post concludes by examining various examples of JS program constructs and the DFG graphs generated.

Existing Work

As we've seen in Part II a key body of work that will be revisited is the WebKit blog: Speculation in JavascriptCore. The relevant sections from this blog are *Compilation and OSR* which discusses in great detail the theoretical aspects of the DFG, including its IR, optimisation phases and compilation.

Tiering Up

In Part II, explored how the LLInt Tiers up into the Baseline JIT. This section looks into how the Baseline JIT tiers up into the DFG. The graph[5] below shows the timeline on how this tiering up process pans out:



We begin by updating our debugging environment. The Baseline JIT tracing flags have been removed from `launch.json` and reset the internal flags in LLIntCommon.h and JIT.cpp. We've enabled the DFG compiler and also disabled the FTL compiler by adding the `--useFTLJIT=false` flag.

```
{
    "version": "0.2.0",
```

```
    "configurations": [
        {
            "name": "(gdb) Launch",
            "type": "cppdbg",
            "request": "launch",
            "program": "/home/amar/workspace/WebKit/WebKitE
            "args": [
                "--useConcurrentJIT=false",
                "--useFTLJIT=false",
                "--verboseOSR=true",

                "--thresholdForJITSoon=10",
                "--thresholdForJITAfterWarmUp=10",

                "--reportCompileTimes=true",
                "--dumpGeneratedBytecodes=true",

                "/home/amar/workspace/WebKit/WebKitBuild/De
            ],
            // truncated for brevity
        }
    ]
}
```

With the debugging environment setup, let's review the static execution count thresholds required to tier up from the Baseline JIT to the DFG. These static counter values are defined in OptionsList.h

```
v(Int32, thresholdForOptimizeAfterWarmUp, 1000, Normal, nul
v(Int32, thresholdForOptimizeAfterLongWarmUp, 1000, Normal,
v(Int32, thresholdForOptimizeSoon, 1000, Normal, nullptr) \
```

The ExecutionCounter object `m_jitExecuteCounter` is initialised in the generated CodeBlock to track the threshold values for DFG optimisation. The functions that set these values in the CodeBlock are defined in CodeBlock.h:

```
    // Call this to reinitialize the counter to its startir
    // forcing a warm-up to happen before the next optimiza
```

```
    // fires.
    void optimizeAfterWarmUp();

    // Call this to force an optimization trigger to fire c
    // a lot of warm-up.
    void optimizeAfterLongWarmUp();

    //... truncated for brevity
    void optimizeSoon();
```

For an unoptimised CodeBlock, the tier-up threshold is set by the call to `optimizeAfterWarmUp` . If a CodeBlock was previously compiled but execution in the optimised code either OSR exited or the optimised code entered into a loop, then the tier-up threshold for re-compilation is set with the call to `optimizeAfterLongWarmUp` . The function `optimizeSoon` , is used to set the tier-up threshold when there are call frames still executing a CodeBlock and determines that it's *cheaper* to delay tier-up and continue executing in a lower tier for a little longer.

Similar to the JSC flags that provides the ability to modify the static Baseline JIT thresholds at runtime (i.e. `--thresholdForJITAfterWarmUp` and `--thresholdForJITSoon`), JSC provides the following DFG JIT flags to do the same; which as we've seen above are: `--thresholdForOptimizeAfterWarmUp` , `--thresholdForOptimizeAfterLongWarmUp` , `--thresholdForOptimizeSoon` .

Let's now construct a test script `test.js` to observe DFG tiering up. The program below attempts to optimise the function `jitMe` using the DFG by executing it in a loop over, 1000 iterations:

```
$ cat test.js

function jitMe(x,y){
    return x + y;
}

let x = 1;

for(let y = 0; y < 1000; y++){
```

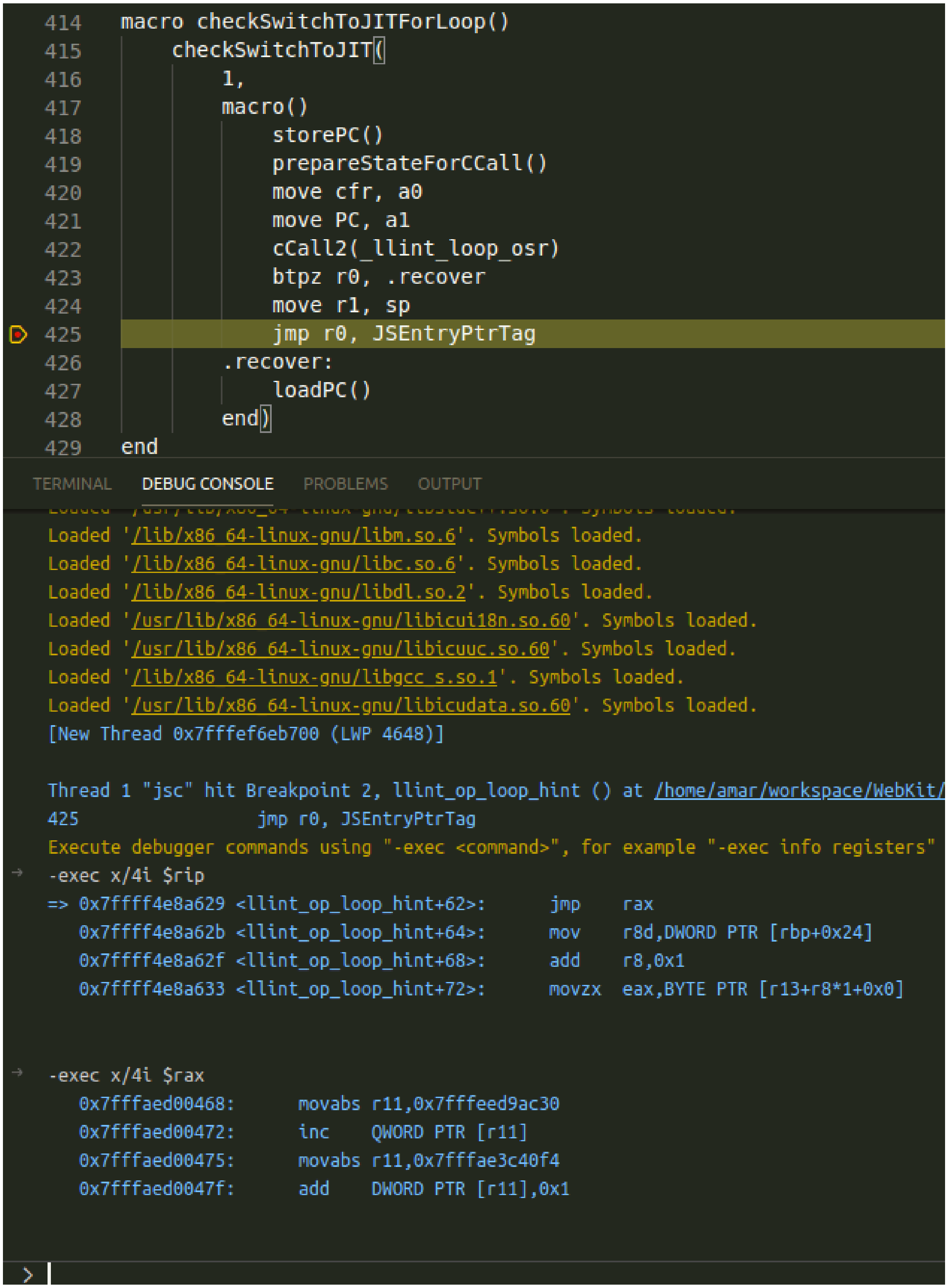
```
    jitMe(x,y)
}
```

The bytecode generated for the function `jitMe` is as follows:

```
jitMe#BcU0vO:[0x7fffae3c4130->0x7fffae3e5100, NoneFunctionC

bb#1
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4
[   6] check_traps
[   7] add                loc6, arg1, arg2, OperandTypes(12
[  13] ret                loc6
Successors: [ ]
```

When the opcode `enter` is compiled by the Baseline JIT, it emits an optimisation check to count executions of the CodeBlock in the Baseline JIT. One can inspect the emitted JIT code in the debugger by setting a breakpoint in the LLInt as described in Part II. In the screenshot below, a breakpoint is set at the jump instruction to the Baseline JIT compiled `on_enter` opcode pointed to by `rax` .



Examining the disassembly for `op_enter`, observe that `r11` stores the reference pointer to `m_jitExecuteCounter.m_counter` for the CodeBlock and is set to `-1000`:

```
-exec x/10i $rip
=> 0x7fffaed00474:  movabs r11,0x7fffae3c40f4
   0x7fffaed0047e:  add    DWORD PTR [r11],0x1
   0x7fffaed00482:  jns    0x7fffaed00809
```

```
    0x7fffaed00488:  movabs r11,0x7fffeed9ac38
    0x7fffaed00492:  inc    QWORD PTR [r11]
    0x7fffaed00495:  movabs r11,0x7fffaeb0bd20
    0x7fffaed0049f:  cmp    BYTE PTR [r11],0x0
    0x7fffaed004a3:  jne    0x7fffaed008c0
    0x7fffaed004a9:  movabs r11,0x7fffeed9ac40
    0x7fffaed004b3:  inc    QWORD PTR [r11]

-exec si

[Switching to thread 2 (Thread 0x7fffef6eb700 (LWP 6093))](
=thread-selected,id="2"
0x00007fffaed0047e in ?? ()
-exec x/wd $r11
0x7fffae3c40f4: -1000
```

The jump to address `0x7fffaed00809` in the snippet above is taken when the sign flag is unset (i.e. the value referenced by the pointer stored in `r11`) is incremented to `0` . This jump takes execution to the prologue to the function call `JSC::operationOptimize(JSC::VM*, uint32_t)` :

```
-exec x/20i 0x7fffaed00809
    0x7fffaed00809:  movabs r9,0x7fffaeb09fd0
    0x7fffaed00813:  mov    r9,QWORD PTR [r9]
    0x7fffaed00816:  add    r9,0xffffffffffffffd0
#... truncated for brevity
    0x7fffaed0083d:  mov    esi,0xf0
    0x7fffaed00842:  movabs rdi,0x7fffaeb00000
    0x7fffaed0084c:  mov    DWORD PTR [rbp+0x24],0x3c
    0x7fffaed00853:  movabs r11,0x7fffaeb09fd8
    0x7fffaed0085d:  mov    QWORD PTR [r11],rbp
    0x7fffaed00860:  movabs r11,0x7ffff5e1f50a
=> 0x7fffaed0086a:  call   r11
```

The call to `operationOptimize()` is a slow path call into the Baseline JIT which is the trigger to initiate DFG compilation of the CodeBlock and eventually performs OSR Entry into the compiled CodeBlock if the compilation succeeds. A truncated version of the function with the two key actions that it takes highlighted below:

```
SlowPathReturnType JIT_OPERATION operationOptimize(VM* vmPc
{
    //... truncated for brevity

    dataLogLnIf(Options::verboseOSR(), "Triggering optimize

    //... code truncated for brevity

    CodeBlock* replacementCodeBlock = codeBlock->newReplace
    CompilationResult result = DFG::compile(vm, replacement
        mustHandleValues, JITToDFGDeferredCompilationCallba

    //... code truncated for brevity

    CodeBlock* optimizedCodeBlock = codeBlock->replacement(
    //... code truncated for brevity

    if (void* dataBuffer = DFG::prepareOSREntry(vm, callFra
        //... truncated for brevity

        codeBlock->optimizeSoon();
        codeBlock->unlinkedCodeBlock()->setDidOptimize(TriS
        void* targetPC = vm.getCTIStub(DFG::osrEntryThunkGe
        targetPC = retagCodePtr(targetPC, JITThunkPtrTag, b
        return encodeResult(targetPC, dataBuffer);   <-- Re
    }

    //... code truncated for brevity

    return encodeResult(nullptr, nullptr);
}
```

In the snippet above, DFG compilation is initiated with the call to `DFG::compile` . Once compilation succeeds, a target address for OSR Entry into the optimised CodeBlock is acquired and returned to the BaselineJIT.

With this high level overview of now the Baseline JIT tiers-up to the DFG, let’s now dive into the details of DFG compilation. The sections that follow being by explaining how bytecode is parsed to generate a DFG, the DFG IR, and some examples typical JS programs that are represented in DFG IR.

Graph Building

Now that JSC has determined that the CodeBlock is *hot* enough to be compiled by the DFG, the next step is to parse the bytecodes generated for the CodeBlock into a graph that the DFG can then optimise and finally lower to machine code. Before getting started, there are some command line flags that should be enabled which will aid in debugging the DFG:

- `reportDFGCompileTimes` : We’ve previously seen the use of the `reportCompileTimes` flag and the statistics it prints about compilation in the Baseline JIT. Whilst `reportCompileTimes` would print compile times for all JIT tiers one can restrict logging to just the DFG compile times by using the `reportDFGCompileTimes` flag. This is mainly to tidy up our output to *stdout*,
- `dumpBytecodeAtDFGTime` : Enabling this flag will dump the bytecodes in the CodeBlock that will be compiled by the DFG,
- `verboseDFGBytecodeParsing` : Enabling this flag will print snippets of DFG IR as each bytecode in the CodeBlock is parsed,
- `dumpGraphAfterParsing` : Enabling this flag will print the generated unoptimised DFG to *stdout*

The debugging environment was updated by modifying `launch.json` as follows (Note that the DFG tier-up thresholds have now been reduced by adding the *thresholdForOptimize* flags):

```
{
    // truncated for brevity
```

```
    "program": "/home/amar/workspace/WebKit/WebKitBuild/Deb
    "args": [
        "--useConcurrentJIT=false",
        "--useFTLJIT=false",
        "--verboseOSR=true",

        "--thresholdForJITSoon=10",
        "--thresholdForJITAfterWarmUp=10",
        "--thresholdForOptimizeAfterWarmUp=100",
        "--thresholdForOptimizeAfterLongWarmUp=100",
        "--thresholdForOptimizeSoon=100",

        "--reportDFGCompileTimes=true",
        "--dumpBytecodeAtDFGTime=true",
        "--verboseDFGBytecodeParsing=true",
        "--dumpGraphAfterParsing=true",

        "/home/amar/workspace/WebKit/WebKitBuild/Debug/bin/
    ],
    //truncated for brevity
}
```

Let's now resume our debug tracing at DFG::Compile in JITOperations.cpp, which through a series of calls, ends up calling `DFG::Plan::compileInThreadImpl`. The call stack should resemble something similar to the one below:

```
libJavaScriptCore.so.1!JSC::DFG::Plan::compileInThreadImpl(
libJavaScriptCore.so.1!JSC::DFG::Plan::compileInThread(JSC:
libJavaScriptCore.so.1!JSC::DFG::compileImpl(JSC::VM & vm,
libJavaScriptCore.so.1!JSC::DFG::compile(JSC::VM & vm, JSC:
libJavaScriptCore.so.1!JSC::operationOptimize(JSC::VM * vmP
[Unknown/Just-In-Time compiled code] (Unknown Source:0)
libJavaScriptCore.so.1!llint_op_call() (/home/amar/workspac
[Unknown/Just-In-Time compiled code] (Unknown Source:0)
```

The function `compileInThreadImpl` is responsible for three main tasks:

1. Parsing the bytecodes in the CodeBlock to a graph,

2. Optimising the graph,
3. Generating machine code from the optimised graph.

These have been highlighted in the truncated code snippet below:

```
Plan::CompilationPath Plan::compileInThreadImpl()
{
    //... code truncated for brevity
    Graph dfg(*m_vm, *this);
    {
        parse(dfg);   <-- Bytecode parsed and unoptimised g
    }

    //... code truncated for brevity
    RUN_PHASE(performLiveCatchVariablePreservationPhase);
    RUN_PHASE(performCPSRethreading);
    RUN_PHASE(performUnification);
    //... code truncated for brevity
    switch (m_mode) {
    case DFGMode: {
        dfg.m_fixpointState = FixpointConverged;

        RUN_PHASE(performTierUpCheckInjection);
        //... truncated for brevity
        RUN_PHASE(performWatchpointCollection);
        dumpAndVerifyGraph(dfg, "Graph after optimization:'

        {
            JITCompiler dataFlowJIT(dfg);
            if (m_codeBlock->codeType() == FunctionCode)
                dataFlowJIT.compileFunction();
            else
                dataFlowJIT.compile();  <-- The optimised c
        }

        return DFGPath;
    }
    //... code truncated for brevity
    }
}
```

This function is particularly important and this blog will return to it periodically in blog posts to follow. Now that one has an overview of the three main activities performed by `compileInThreadImpl` , the rest of this blog post will focus on the first activity which is bytecode parsing and graph generation.

Bytecode Parsing

The first step in DFG compilation is parsing bytecode to generate a graph. This is done by creating a `Graph` object which is initialised with the CodeBlock to be parsed and then calling the `DFG::parse` function on this graph object.

The `DFG::parse` function which is essentially a wrapper to `ByteCodeParser::parse()` . The function begins parsing the CodeBlock with a call to `ByteCodeParser::parseCodeBlock` . The function `parseCodeBlock` collects information about the various *jump targets* that the bytecodes in the CodeBlock may have. This is done by the function `JSC::getJumpTargetsForInstruction` .

Once *jump targets* have been recorded, the function `parseCodeBlock` enters a loop that iterates over each *jump target* and performs the following actions; first it allocates a BasicBlock and appends it to the graph object. If this is the first block in the graph, it is marked as an OSR target and a reference to the block is appended to the list of root nodes in the graph:

```
for (unsigned jumpTargetIndex = 0; jumpTargetIndex <= jumpT
    unsigned limit = jumpTargetIndex < jumpTargets.size() ?

    // Loop until we reach the current limit (i.e. next jum
    do {
        if (!m_currentBlock) {
            m_currentBlock = allocateTargetableBlock(m_curr

            // The first block is definitely an OSR target.
            if (m_graph.numBlocks() == 1) {
                m_currentBlock->isOSRTarget = true;
                m_graph.m_roots.append(m_currentBlock);
            }
            prepareToParseBlock();
```

```
        }

        parseBlock(limit);

        //... code truncated for brevity
    } while (m_currentIndex.offset() < limit);
}
```

Once the BasicBlock has been allocated, it then calls `ByteCodeParser::parseBlock` with the argument `limit` . This argument determines the number of bytecode instructions within the CodeBlock that need to be included in the BasicBlock that was allocated.The function `parseBlock` is responsible for the parsing of this set of bytecode instructions and generating nodes and edges for the BasicBlock.

The function begins by evaluating if the basic block that is being parsed is the first block in the graph, if this is the case then each *argument value* to the basic block is appended as a `Node` to the graph. In the example script `test.js` above, *argument values* would be the parameters (i.e. `x` , `y` and `this`) passed to the function `jitMe` .

`Node` s represent a single DFG operation. With the *arguments* added to the graph, the function then loops over each opcode in the bytecode list and appends one or more `Node` s to the graph with the call to the overloaded function `addToGraph` . The truncated `parseBlock` with the key functionalities annotated with comments is shown below:

```
void ByteCodeParser::parseBlock(unsigned limit)
{
    auto& instructions = m_inlineStackTop->m_codeBlock->ins
    BytecodeIndex blockBegin = m_currentIndex;

    if (m_currentBlock == m_graph.block(0) && !inlineCallFr
        //... code truncated for brevity
        for (unsigned argument = 0; argument < m_numArgumer
            VariableAccessData* variable = newVariableAcces
                virtualRegisterForArgumentIncludingThis(arg
           //... code truncated for brevity
```

```
        /* Add argument to graph */
        Node* setArgument = addToGraph(SetArgumentDefir
        //... code truncated for brevity
    }
}

CodeBlock* codeBlock = m_inlineStackTop->m_codeBlock;

auto jumpTarget = [&](int target) {
    if (target)
        return target;
    return codeBlock->outOfLineJumpOffset(m_currentInst
};

/* Loop over each opcode in the list of bytecode instru
while (true) {
    //... code truncated for brevity

    // Switch on the current bytecode opcode.
    const Instruction* currentInstruction = instructior
    m_currentInstruction = currentInstruction; // Some
    OpcodeID opcodeID = currentInstruction->opcodeID();

    //... code truncated for brevity

    switch (opcodeID) {

    // === Function entry opcodes ===

    case op_enter: {
        Node* undefined = addToGraph(JSConstant, OpInfo
        // Initialize all locals to undefined.
        for (int i = 0; i < m_inlineStackTop->m_codeBlo
            set(virtualRegisterForLocal(i), undefined,

        NEXT_OPCODE(op_enter);
    }

    //... code truncated for brevity

    // === Arithmetic operations ===
```

```
        case op_add: {
            auto bytecode = currentInstruction->as<OpAdd>()
            Node* op1 = get(bytecode.m_lhs);
            Node* op2 = get(bytecode.m_rhs);
            if (op1->hasNumberResult() && op2->hasNumberRes
                set(bytecode.m_dst, makeSafe(addToGraph(Ari
            else
                set(bytecode.m_dst, makeSafe(addToGraph(Val
            NEXT_OPCODE(op_add);
        }

        //... code truncated for brevity

        case op_mov: {
            auto bytecode = currentInstruction->as<OpMov>()
            Node* op = get(bytecode.m_src);
            set(bytecode.m_dst, op);
            NEXT_OPCODE(op_mov);
        }

        //... code truncated for brevity
        }
    }
}
```

Each basic block in the CodeBlock is parsed by the function `parseBlock` and adds `Nodes` to an unoptimised graph before returning to its calling function `parseCodeBlock`. The function `parseCodeBlock` returns once all basic blocks in the CodeBlock have been parsed and BasicBlocks, Nodes and Edges have been appended to the unoptimised graph.

As the instructions in the CodeBlock are parsed, each parsing step is logged to *stdout* when the `--verboseDFGBytecodeParsing=true` is added. Using our test script `test.js` and initiating DFG compilation of the `jitMe` function, the following output is logged to *stdout*:

```
Parsing jitMe#BcU0vO:[0x7fffae3c4260->0x7fffae3c4130->0x7ff
Parsing jitMe#BcU0vO:[0x7fffae3c4260->0x7fffae3c4130->0x7ff
jitMe#BcU0vO:[0x7fffae3c4130->0x7fffae3e5100, BaselineFunct
```

```
bb#1
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4
[   6] check_traps
[   7] add                loc6, arg1, arg2, OperandTypes(12
[  13] ret                loc6
Successors: [ ]


Jump targets:
        appended D@0 SetArgumentDefinitely
        appended D@1 SetArgumentDefinitely
        appended D@2 SetArgumentDefinitely
    parsing bc#0: op_enter
        appended D@3 CountExecution
        appended D@4 JSConstant
        appended D@5 MovHint
        appended D@6 SetLocal

//... truncated for brevity

    parsing bc#7: op_add
        appended D@27 CountExecution
        appended D@28 GetLocal
        appended D@29 GetLocal
        appended D@30 ValueAdd
        appended D@31 MovHint
        appended D@32 SetLocal
    parsing bc#13: op_ret
        appended D@33 CountExecution
        appended D@34 Return
        appended D@35 Flush
        appended D@36 Flush
        appended D@37 Flush
Done parsing jitMe#BcU0vO:[0x7fffae3c4260->0x7fffae3c4130->
```

The snippet above lists the bytecodes that comprise the `jitMe` function and the DFG `Node` s that were generated for each bytecode. The values `D@1` , `D@2` , `D@3` , etc represent the `Node` and the values `SetArgumentDefinitely` , `CountExecution` , `JSConstant` , etc represent the DFG opcode.

The function `DFG::parse` returns when a complete unoptimised DFG for the function `jitMe` has been generated. With the `--dumpGraphAfterParsing=true` flag set, one can see the final graph printed to *stdout*:

```
Graph after parsing:

          : DFG for jitMe#BcU0vO:[0x7fffae3c4260->0x7fffae3c
          :   Fixpoint state: BeforeFixpoint; Form: LoadStor
          :   Arguments for block#0: D@0, D@1, D@2

      0   : Block #0 (bc#0): (OSR target)
      0   :   Execution count: 1.000000
      0   :   Predecessors:
      0   :   Successors:
      0   :   States: StructuresAreWatched, CurrentlyCFAUnre
      0   :   Vars Before: <empty>
      0   :   Intersected Vars Before: arg2:(FullTop, TOP, T
      0   :   Var Links:
  0   0   :    D@0:< 1:->        SetArgumentDefinitely(this(
  1   0   :    D@1:< 1:->        SetArgumentDefinitely(arg1(
  2   0   :    D@2:< 1:->        SetArgumentDefinitely(arg2(
  3   0   :    D@3:<!0:->        CountExecution(MustGen, 0x7
  4   0   :    D@4:< 1:->        JSConstant(JS|PureInt, Unde
//... truncated for brevity
 30   0   :   D@30:<!0:->        ValueAdd(Check:Untyped:D@28
 31   0   :   D@31:<!0:->        MovHint(Check:Untyped:D@30,
 32   0   :   D@32:< 1:->        SetLocal(Check:Untyped:D@30
 33   0   :   D@33:<!0:->        CountExecution(MustGen, 0x7
 34   0   :   D@34:<!0:->        Return(Check:Untyped:D@30,
 35   0   :   D@35:<!0:->        Flush(MustGen, arg2(C~/Flus
 36   0   :   D@36:<!0:->        Flush(MustGen, arg1(B~/Flus
 37   0   :   D@37:<!0:->        Flush(MustGen, this(a), R:S
      0   :   States: InvalidBranchDirection, StructuresAreW
      0   :   Vars After: <empty>
      0   :   Var Links: arg2:D@35 arg1:D@36 arg0:D@37 loc0:

          : GC Values:
```

```
        :     Weak:Object: 0x7fffeedbb068 with butterfly (
//... truncated for brevity
```

The graph above is represented in DFG IR and in the next section we explore the three main IR constructs (i.e. BasicBlocks, Nodes and Edges) and how to read and interpret graphs.

DFG IR

The DFG IR is the intermediate representation that is generated by the DFG bytecode parser. A DFG graph as seen in the example above is made up of one or more `BasicBlock` s and each `BasicBlock` is an ordered collection of Nodes. A `Node` in a BasicBlock represents a DFG instruction/opcode and can also be linked to child nodes via `Edges` . This IR is used by both the DFG and FTL tiers.

Let's attempt to generate DFG graphs for some common JavaScript language constructs and review the graph output. This will provide a more practical exploration of Nodes and Edges. To get familiar with the IR and its representation, let's being with the following test program that does trivial arithmetic and some load and store operations:

```
$ cat test.js

function jitMe(y){
    arr[y] += obj.sum;
    obj.sum += y
}

let arr = [];
let obj = {sum: 0}

for(let y = 0; y < 1000; y++){
    jitMe(y)
}
```

The goal of the program above is to DFG compile the `jitMe` function and this is achieved by invoking it a `1000` times in a for loop to trigger DFG compilation. The bytecodes generated for the function `jitMe` are listed below:

```
jitMe#EGS36l:[0x7fffae3c4130->0x7fffae3e5100, BaselineFunct

bb#1
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4

//... truncated for brevity

[  80] get_by_id          loc8, loc7, 2
[  85] add                loc8, loc8, arg1, OperandTypes(12
[  91] put_by_id          loc7, 2, loc8,
[  97] ret                Undefined(const0)
Successors: [ ]


Identifiers:
  id0 = arr
  id1 = obj
  id2 = sum
```

Let's attempt to print the unoptimised DFG generated from parsing these bytecodes. DFG printing is done by the function `Graph::dump`, one can set a breakpoint at this function to observe the graph output generation. The graph generated for this function is as follows:

```
Graph after parsing:

         : DFG for jitMe#EGS36l:[0x7fffae3c4260->0x7fffae3c
         :   Fixpoint state: BeforeFixpoint; Form: LoadStor
         :   Arguments for block#0: D@0, D@1

     0   : Block #0 (bc#0): (OSR target)
     0   :   Execution count: 1.000000
```

```
      0   :   Predecessors:
      0   :   Successors:
      0   :   States: StructuresAreWatched, CurrentlyCFAUnre
      0   :   Vars Before: <empty>
      0   :   Intersected Vars Before: arg1:(FullTop, TOP, T
      0   :   Var Links:
   0  0   :    D@0:< 1:->        SetArgumentDefinitely(this(
   1  0   :    D@1:< 1:->        SetArgumentDefinitely(arg1(
   2  0   :    D@2:<!0:->        CountExecution(MustGen, 0x7
   3  0   :    D@3:< 1:->        JSConstant(JS|PureInt, Unde
   4  0   :    D@4:<!0:->        MovHint(Check:Untyped:D@3,
   5  0   :    D@5:< 1:->        SetLocal(Check:Untyped:D@3,
   6  0   :    D@6:<!0:->        MovHint(Check:Untyped:D@3,
   7  0   :    D@7:< 1:->        SetLocal(Check:Untyped:D@3,
   8  0   :    D@8:<!0:->        MovHint(Check:Untyped:D@3,
   9  0   :    D@9:< 1:->        SetLocal(Check:Untyped:D@3,

//... truncated for brevity

  92  0   :   D@92:<!0:->        FilterPutByIdStatus(Check:U
  93  0   :   D@93:<!0:->        PutByOffset(Check:Untyped:D
  94  0   :   D@94:<!0:->        CountExecution(MustGen, 0x7
  95  0   :   D@95:< 1:->        JSConstant(JS|PureInt, Unde
  96  0   :   D@96:<!0:->        Return(Check:Untyped:D@95,
  97  0   :   D@97:<!0:->        Flush(MustGen, arg1(B~/Flus
  98  0   :   D@98:<!0:->        Flush(MustGen, this(a), R:S
      0   :   States: InvalidBranchDirection, StructuresAreW
      0   :   Vars After: <empty>
      0   :   Var Links: arg1:D@97 arg0:D@98 loc0:D@5 loc1:D

          : GC Values:
          :     Weak:Object: 0x7fffae3c0000 with butterfly (
          :     Weak:Object: 0x7fffeeda5868 with butterfly 0
          :     Weak:Object: 0x7fffeedbb068 with butterfly (
          :     Weak:Object: 0x7fffae3f59e0 with butterfly (
          : Desired watchpoints:
          :     Watchpoint sets: 0x7fffeeda7940, 0x7fffeeda7
          :     Inline watchpoint sets: 0x7fffae3f9478, 0x7f
          :     SymbolTables:
          :     FunctionExecutables: 0x7fffae3e5100
          :     Buffer views:
          :     Object property conditions: <Object: 0x7fffa
          : Structures:
```

```
        :     %BK:Function                      = 0x7fffae3f9
        :     %CM:Array,ArrayWithContiguous     = 0x7fffae3f9
        :     %Cv:Object                        = 0x7fffae3c8
        :     %DU:JSGlobalLexicalEnvironment = 0x7fffae3f9
```

The graph dump lists the various blocks that represent the bytecodes being optimised. Each DFG block represents a basic block in the bytecode dump. A DFG block consists of a *Head*, the block *Body* and a block *Tail*. The end of the graph dump lists information on the various JSC objects, watchpoints and structures that are referenced by the graph and their locations in memory. Let's now dissect the various sections of this graph and explore them in greater detail.

Graph Header

The start of the output prints the header information for the graph:

```
 : DFG for jitMe#EGS36l:[0x7fffae3c4260->0x7fffae3c4130->0x7
 :   Fixpoint state: BeforeFixpoint; Form: LoadStore; Unific
 :   Arguments for block#0: D@0, D@1
```

The first line prints details about the CodeBlock being parsed, the `codeType` which is `DFGFunctionCall` and the number of instructions in the CodeBlock. The second line prints the `OptimizationFixpointState` which can be thought of as a flag that tracks state changes during the various optimisation phases. the WebKit blog[6] describes *fixpoint* as follows:

> fixpoint: we keep executing every instruction until we no longer observe any changes.

The DFG `GraphForm` which can be one of three values `LoadStore` , `ThreadedCPS` and `SSA` , these are described in greater detail in the developer comments. The `GraphForm` gets modified depending on the various stages of optimisation that have occurred. For example, DFG optimisations are applied on graphs that are in `ThreadedCPS` form whereas majority of FTL optimisations require that the graph be in `SSA` form.

The [`UnificationState`] and [`RefCountState`] present additional statistics about the graph at various optimisation phases. The last line in the header lists the argument nodes that have been generated for the block `#0` which is the entry block into the graph. These argument nodes are `D@0` and `D@1` which represent the JavaScript values `this` and `num` .

Block Head

What follows the graph header is a dump of each [`BasicBlock`] in the graph. The block dump is comprised of a head, a block body and a tail. Let's examine the only block in this dump which is `Block #0` . In the snippet below the interesting details are the `Var` listings, this is essentially a dump of all the arguments, locals and temporary variables used by the basic block.

```
0   : Block #0 (bc#0): (OSR target)
0   :   Execution count: 1.000000
0   :   Predecessors:
0   :   Successors:
0   :   States: StructuresAreWatched, CurrentlyCFAUnre
0   :   Vars Before: <empty>
0   :   Intersected Vars Before: arg1:(FullTop, TOP, T
0   :   Var Links:
```

The `Vars Before` list in the Block Head represents a list of values at the head. This list is populated after the CFA optimisation phase and contains a list of `AbstractValue` s that were recorded at the start of the block. `AbstractValues` are speculated values that are inferred by the *Abstract Interpreter*

The `Intersected Vars Before` list is an intersection of assumptions we have made previously at the head of this block and assumptions we had used for optimizing the given basic block. Each operand (i.e. *args* and *locs*) is represented by a tuple of four values. The first value (e.g. `FullTop`) represents the speculated type of the argument or local variable. Since this is the graph generated after parsing the bytecodes, it hasn't been optimised yet to include predicted values. The second and third value in the tuple (e.g. `TOP`) represent the `AbstractHeap::Payload`, more on this will be covered in Part IV which is dedicated to graph optimisation. The final value in the tuple represents the ClobberState of the structure which can be one of two values:

```
enum StructureClobberState : uint8_t {
    StructuresAreWatched, // Constants with watchable struc
    StructuresAreClobbered // Constants with watchable stru
};
```

The `Var Links` list in the block head refer to the list of variables at head. This list maps arguments, locals and temporary variable to nodes in the various blocks of the DFG.

Note: The term `variable` and `operand` are used interchangeably in the context of the DFG IR. These should not be confused with JS variables defined in the example script.

`Predecessors` and `Successors` define the basic blocks that allow control and data to flow from and to the current block (in this instance the current block is `Block #0`). Since the bytecodes generated for the program above don't include any branching opcodes (e.g. comparison operators, loops, etc) the values of `Predecessor` and `Successors` are empty. Examples of branching code will be demonstrated in sections to follow.

Block Body

Now that we've reviewed the header, lets look into node representation. The developer comments provide a very helpful explanation on how to read this representation. This is shown in the code listing below:

```
// Example/explanation of dataflow dump output
//
//   D@14:   <!2:7>  GetByVal(@3, @13)
//     ^1     ^2 ^3     ^4       ^5
//
// (1) The nodeIndex of this operation.
// (2) The reference count. The number printed is the 'real
//     not including the 'mustGenerate' ref. If the node is
//     'mustGenerate' then the count it prefixed with '!'.
// (3) The virtual register slot assigned to this node.
// (4) The name of the operation.
// (5) The arguments to the operation. The may be of the fo
//         D@#  - a NodeIndex referencing a prior node in t
//         arg# - an argument number.
//         id#  - the index in the CodeBlock of an identifi
//         var# - the index of a var on the global object,
```

Nodes

Let's look at the nodes generated for the `add` opcode in bytecode `bc#85` from the graph dump above:

```
 87  0   :   D@87:<!0:->  CountExecution(MustGen, 0x7fffeec
 88  0   :   D@88:<!0:->  ValueAdd(Check:Untyped:D@84, Chec
 89  0   :   D@89:<!0:->  MovHint(Check:Untyped:D@88, MustG
 90  0   :   D@90:< 1:->  SetLocal(Check:Untyped:D@88, loc8
```

The column on the left indicates the node index followed by BasicBlock index. The values of the form `D@<number>` represent the Node. Let's examine the following node:

```
87  0   :   D@87:<!0:->  CountExecution(MustGen, 0x7fffeed9
```

`D@87` represents the 87th Node in the block `#0`. The expression `!0` indicates that the Node must be generated and has a reference count of zero and the `-` in `<!0:->` indicates that there isn't a virtual register that's assigned to the node.

Reference counts and spill registers will become more relevant during the discussion on graph optimisation covered in Part IV.

The DFG opcode or node type is `CountExecution`. The `NodeType` defines the DFG operation performed by the node itself. Various `NodeTypes` are defined in dfg/DFGNodeType.h along with helpful comments on their functions. The snippet below shows some of the `NodeTypes` that can be defined:

```
#define FOR_EACH_DFG_OP(macro) \
    /* A constant in the CodeBlock's constant pool. */\
    macro(JSConstant, NodeResultJS) \
    \
    /* Constants with specific representations. */\
    macro(DoubleConstant, NodeResultDouble) \
    macro(Int52Constant, NodeResultInt52) \
    \
    /* Lazy JSValue constant. We don't know the JSValue bit
    macro(LazyJSConstant, NodeResultJS) \
    \
    /* Marker to indicate that an operation was optimized e
    /* is to make one node alias another. CSE will later us
    /* though it may choose not to if it would corrupt prec
    macro(Identity, NodeResultJS) \
    /* Used for debugging to force a profile to appear as a

    //... truncated for brevity
```

The DFG opcode is followed by the node flags which is this case is `MustGen`. `NodeFlags` help define properties of the result from node computation. These flags are defined in dfg/DFGNodeFlags.h.

Tip: Another useful source to learn more about `NodeTypes` and `NodeFlags` is to grep through the several changelogs in the WebKit repo. The developer comments in the changelog provide helpful insight on nodes and their functionality within the graph.

`Node` s also define operands which are represented by the `OpInfo` structure. The developer comments describes the structure as follows:

> This type used in passing an immediate argument to Node constructor; distinguishes an immediate value (typically an index into a CodeBlock data structure - a constant index, argument, or identifier) from a Node*.

In the snippet above the value `0x7fffeed99440` is the raw pointer to the execution counter which is stored as an immediate value.

The `R` and `W` values determine what parts of the program state (i.e. *Abstract Heaps*) the node can read and write to. This is defined by the function `clobberize` which records aliasing information about a Node. `clobberize` is described in great detail in the WebKit blog[6].

The value `bc#85` indicates the bytecode index for which the node was generated for and is known as the `NodeOrigin` . This bytecode index is also the exit bytecode for the node unless an explicit exit bytecode is specified. The last value in the snippet above is `ExitValid` which indicates if the node is a valid point for OSR exit to occur. These three values define the `NodeOrigin` which are defined in dfg/DFGNodeOrigin.h and describe three properties of the Node.

Edges

Let's now examine a node with `Edge` s form the nodes generated for the `add` opcode:

```
88  0   :   D@88:<!0:->  ValueAdd(Check:Untyped:D@84, Chec
```

Here the DFG node `D@88` defines part of the `add` operation. `ValueAdd` has edges to two child nodes, `D@84` and `D@40` on which it performs an *addition operation*. It isn't specified what type of addition operation will be performed (e.g. arithmetic addition, string concatenation, etc.) on these two nodes and as a result, the DFG will add `Check` flags to these nodes. The value

`Check` indicates that the edge is unproven for the edge type, which is described by the value `Untyped`. The value `D@84` and `D@40` are child nodes associated with the edges from the `ValueAdd` node. The constructor of an `Edge` object is shown below:

```
explicit Edge(Node* node = nullptr, UseKind useKind = Untyp
        : m_encodedWord(makeWord(node, useKind, proofStatus
    {
    }
```

The constructor takes four parameters and generates an `encodedWord` to represent an edge in the graph. The `node` represents the child node that the parent links to, the `UseKind` parameter determines the representation of values used by the DFG IR. Essentially, the `UseKind` defines the type of Edge and determines the value type that is being propagated from one node to the other. The DFG has three value types that it uses which are all defined in dfg/DFGUseKind.h:

```
    // The DFG has 3 representations of values used:

    // 1. The JSValue representation for a JSValue that mus
    //    register (or a GP register pair), and follows rul
    //    that allow the JSValue to be stored as either ful
    //    unboxed Int32, Booleans, Cells, etc. in 32-bit as
    UntypedUse, // UntypedUse must come first (value 0).
    Int32Use,
    KnownInt32Use,
    AnyIntUse,

    //... code truncated for brevity

    // 2. The Double representation for an unboxed double v
    //    in an FP register.
    DoubleRepUse,
    DoubleRepRealUse,
    DoubleRepAnyIntUse,

    // 3. The Int52 representation for an unboxed integer v
```

```
//    in a GP register.
Int52RepUse,

LastUseKind // Must always be the last entry in the enu
```

One can inspect the Edge properties in more detail by setting a breakpoint at `Edge::dump` and stepping through the function as it dumps edge data. The last two parameters that define an edge are `ProofStatus` and `KillStatus` parameters indicate if a edge needs to be proved and if a node will be killed. These two properties of an edge will be revisited in Part IV on graph optimisation. Edges are bidirectional with DFG values flowing from parent to child nodes and execution control flowing from child nodes to parent nodes.

Block Tail

The block dump also includes details about `State` , `Vars After` and `Var Links` .

```
0   :   States: InvalidBranchDirection, StructuresAreW
0   :   Vars After: <empty>
0   :   Var Links: arg2:D@35 arg1:D@36 arg0:D@37 loc0:
```

The `Vars After` list represents the values at tail for the block. This is a list of `AbstractValue` s that are collected by the *Abstract Interpreter* after CFA.

`Var Links` is a list of variables at the end of the block. The *links* represent a mapping between DFG node and variable that helps the OSR Exit generator identify the node it would need to look up in order to reconstruct the state of the variable. More on this in the *OSR Exit* blog post.

Graph Footer

At the end of our graph dump lists the `GC Values` which is a list of references the CodeBlock has to objects on the heap. This instance lists references to four JSC Objects, two of which are defined in our JS script (i.e. `arr` and `obj`). The footer also prints information about the various `Watchpoints` and a list of structures associated with the CodeBlock.

Branching Instructions

Let's now look at an example of a program that generates branches:

```
function jitMe(num) {
    if(num % 2 == 0){
        arr[num] = num;
    }else{
        obj.sum += num;
    }
}

let arr = [];
let obj = {sum : 0}

for (let y = 0; y < 1000; y++) {
    jitMe(y)
}
```

The program above generates the following bytecodes:

```
jitMe#DFCAbp:[0x7fffae3c4130->0x7fffae3e5100, BaselineFunct

bb#1
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4
[   6] check_traps
[   7] mod                loc6, arg1, Int32: 2(const0)
[  11] jneq               loc6, Int32: 0(const1), 27(->38)
Successors: [ #3 #2 ]

bb#2
[  15] resolve_scope      loc6, loc4, 0, GlobalProperty, 0
```

```
[  22] get_from_scope     loc7, loc6, 0, 2048<ThrowIfNotFou
[  30] put_by_val         loc7, arg1, arg1, NotStrictMode
[  36] jmp                34(->70)
Successors: [ #4 ]

bb#3
[  38] resolve_scope      loc6, loc4, 1, GlobalProperty, 0
[  45] get_from_scope     loc7, loc6, 1, 2048<ThrowIfNotFou
[  53] get_by_id          loc8, loc7, 2
[  58] add                loc8, loc8, arg1, OperandTypes(12
[  64] put_by_id          loc7, 2, loc8,
Successors: [ #4 ]

bb#4
[  70] ret                Undefined(const2)
Successors: [ ]


Identifiers:
  id0 = arr
  id1 = obj
  id2 = sum

Constants:
   k0 = Int32: 2: in source as integer
   k1 = Int32: 0: in source as integer
   k2 = Undefined

Jump targets: 38, 70
```

The dump above lists four basic blocks that form the function `jitMe`. The block `bb#1` has a conditional opcode `jneq` at `bc#11`. Should the condition evaluate to `true`, execution jumps to bytecode `bc#50` which is in basic block `bb#3`. If the condition evaluates to `false`, execution continues on to `bc#15` in basic block `bb#2`. The dump also lists the two jump targets which are at `bc#38` and `bc#70`.

Let's now examine what the graph generated for each of these blocks would look like. The snippet below shows a truncated graph representation for the bytecodes generated for `bb#0`:

```
0    : Block #0 (bc#0): (OSR target)
     0    :   Execution count: 1.000000
     0    :   Predecessors:
     0    :   Successors: #1 #2
//... truncated for brevity
 28  0    :   D@28:< 1:->        JSConstant(JS|PureInt, Int3
 29  0    :   D@29:<!0:->        ValueMod(Check:Untyped:D@27
 30  0    :   D@30:<!0:->        MovHint(Check:Untyped:D@29,
 31  0    :   D@31:< 1:->        SetLocal(Check:Untyped:D@29
 32  0    :   D@32:<!0:->        CountExecution(MustGen, 0x7
 33  0    :   D@33:< 1:->        JSConstant(JS|PureInt, Int3
 34  0    :   D@34:<!0:->        CompareEq(Check:Untyped:D@2
 35  0    :   D@35:<!0:->        Branch(Check:Untyped:D@34,
//... truncated for brevity
```

There are two items of note in the dump above. `Successors` have now been populated with values `#1` and `#2`. These are references to DFG blocks `#1` and `#2` which represent `bb#2` and `bb#3` respectively. This indicates that the Block `#0` has edges to blocks `#1` or `#2` and as such allows control and data to transfer to these blocks. Note that DFG basic blocks don't necessarily have a 1:1 mapping with bytecode basic blocks.

The nodes `D@34` and `D@35` are responsible for evaluating the branching condition and determining how control and data should be directed.

```
34  0    :   D@34:<!0:->        CompareEq(Check:Untyped:D@29
35  0    :   D@35:<!0:->        Branch(Check:Untyped:D@34, M
```

The `CompareEq` operation is self explanatory, it has two edges to nodes `D@29` and `D@33` and defines flags to indicate the type of the operation result (i.e. either a boolean value or an Integer). The `Branch` operation evaluates the node `D@34` and uses the result to determine which DFG block to jump to. If the operation returns `true`, the jump to block `#1` is taken and if the operation returns `false`, the jump to block `#2` is taken.

For the sake of completion, DFG blocks `#1` and `#2` are listed below:

```
      1   : Block #1 (bc#15):
      1   :   Execution count: 1.000000
      1   :   Predecessors: #0
      1   :   Successors: #3
      1   :   States: StructuresAreWatched, CurrentlyCFAUnre
      1   :   Vars Before: <empty>
 //... truncated for brevity
  12  1   :   D@48:<!0:->        GetLocal(JS|MustGen|PureInt
  13  1   :   D@49:<!0:->        PutByVal(Check:Untyped:D@44
  14  1   :   D@50:<!0:->        CountExecution(MustGen, 0x7
  15  1   :   D@51:<!0:->        Jump(MustGen, T:#3, W:SideS
      1   :   States: InvalidBranchDirection, StructuresAreW
      1   :   Vars After: <empty>
      1   :   Var Links: arg1:D@48 loc4:D@39 loc6:D@41 loc7:

 //... truncated for brevity

      2   : Block #2 (bc#38):
      2   :   Execution count: 1.000000
      2   :   Predecessors: #0
      2   :   Successors: #3
      2   :   States: StructuresAreWatched, CurrentlyCFAUnre
 //... truncated for brevity
  20  2   :   D@72:<!0:->        ValueAdd(Check:Untyped:D@67
  21  2   :   D@73:<!0:->        MovHint(Check:Untyped:D@72,
  22  2   :   D@74:< 1:->        SetLocal(Check:Untyped:D@72
  23  2   :   D@75:<!0:->        CountExecution(MustGen, 0x7
  24  2   :   D@76:<!0:->        FilterPutByIdStatus(Check:U
  25  2   :   D@77:<!0:->        PutByOffset(Check:Untyped:D
  26  2   :   D@78:<!0:->        Jump(MustGen, T:#3, W:SideS
      2   :   States: InvalidBranchDirection, StructuresAreW
      2   :   Vars After: <empty>
      2   :   Var Links: arg1:D@71 loc4:D@55 loc6:D@57 loc7:
```

Note how both these blocks, listed above, list `#0` as a `Predecessor` and `#3` as the `Successor`.

Function Inlining

Another interesting construct utilised by the bytecode parser is function inlining. Consider the following program:

```
function jitMe(num) {
    obj.sum += num
    inlineFunc(obj.sum)
}

function inlineFunc(num){
    arr[num] = num;
}

let arr = []
let obj = {sum: 0}

for (let y = 0; y < 1000; y++) {
    jitMe(y)
}
```

In the program above, the function `jitMe` calls the function `inlineFunc` and when the DFG decides to optimise `jitMe` it evaluates the call to the function `inlineFunc` and determines that this function can be inlined into `jitMe` . The snippet below is the bytecode dump for the `jitMe` function and the bytecode of interest to this discussion is `bc#74` which is the call to the function `inlineFunc` .

```
jitMe#Bslwax:[0x7fffaf2c4130->0x7fffaf2e5100, BaselineFunct

bb#1
//... truncated for brevity
[  69] get_by_id          loc9, loc12, 1
[  74] call               loc6, loc6, 2, 16
[  80] ret                Undefined(const0)
Successors: [ ]
```

```
Identifiers:
  id0 = obj
  id1 = sum
  id2 = inlineFunc
```

When the DFGBytecodeParser encounters the bytecode `call`, it performs a number of checks to determine if the callee can be inlined. With the `verboseDFGBytecodeParsing` command line flag enabled one would be able to observe how the `call` bytecode is parsed and inlined:

```
parsing bc#74: op_call
    Handling call at bc#74: Statically Proved, (Function: O
        appended D@64 FilterCallLinkStatus
Handling inlining...
Stack: bc#74
    Considering callee (Function: Object: 0x7fffaf2f5a00 wi
Considering inlining (Function: Object: 0x7fffaf2f5a00 with
    Call mode: Call
    Is closure call: false
    Capability level: CanCompileAndInline
    Might inline function: true
    Might compile function: true
    Is supported for inlining: true
    Is inlining candidate: true
    Inlining should be possible.
        appended D@65 CheckIsConstant
        appended D@66 Phantom
   ensureLocals: trying to raise m_numLocals from 16 to 24
   ensureTmps: trying to raise m_numTmps from 0 to 0
        appended D@67 ExitOK
```

If the DFGBytecodeParser determines that the function can be inlined, it parses the function bytecode and appends appropriate nodes to the DFG graph. The snippet below shows the output generated by `verboseDFGBytecodeParsing` when a function is inlined.

```
Parsing inlineFunc#EfyD9C:[0x7fffaf2c4260->0x7fffaf2e5180,
inlineFunc#EfyD9C:[0x7fffaf2c4260->0x7fffaf2e5180, Baseline
```

```
bb#1
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4
[   6] check_traps
[   7] resolve_scope      loc6, loc4, 0, GlobalProperty, 0
[  14] get_from_scope     loc7, loc6, 0, 2048<ThrowIfNotFou
[  22] put_by_val         loc7, arg1, arg1, NotStrictMode
[  28] ret                Undefined(const0)
Successors: [ ]


Identifiers:
  id0 = arr

Constants:
   k0 = Undefined

Jump targets:
    parsing bc#74 --> inlineFunc#EfyD9C:<0x7fffaf2c4260> bc
        appended D@68 JSConstant
        appended D@69 MovHint

        //... truncated for brevity

        parsing bc#74 --> inlineFunc#EfyD9C:<0x7fffaf2c426(
        appended D@81 JSConstant
        appended D@82 JSConstant
        appended D@83 MovHint
        appended D@84 SetLocal
    parsing bc#74 --> inlineFunc#EfyD9C:<0x7fffaf2c4260> bc
        appended D@85 MovHint
        appended D@86 SetLocal
    parsing bc#74 --> inlineFunc#EfyD9C:<0x7fffaf2c4260> bc
        appended D@87 InvalidationPoint

        //... truncated for brevity
```

Once the bytecodes for the function `jitMe` and the inlined function `inlineFunc` have been parsed the following unoptimised graph is generated. Note the representation of the inlined function between nodes `D@67` to `D@101` :

```
Graph after parsing:

        : DFG for jitMe#Bslwax:[0x7fffaf2c44c0->0x7fffaf2c
        :   Fixpoint state: BeforeFixpoint; Form: LoadStor
        :   Arguments for block#0: D@0, D@1

    0   : Block #0 (bc#0): (OSR target)
    0   :   Execution count: 1.000000

   //... truncated for brevity

 0  0   :     D@0:< 1:->          SetArgumentDefinitely(this(
 1  0   :     D@1:< 1:->          SetArgumentDefinitely(arg1(
 2  0   :     D@2:< 1:->          JSConstant(JS|PureInt, Unde
 3  0   :     D@3:<!0:->          MovHint(Check:Untyped:D@2,

   //... truncated for brevity

66  0   :    D@66:<!0:->          Phantom(Check:Untyped:D@42,
    0   :    --> inlineFunc#EfyD9C:<0x7fffaf2c4260, bc#74,
67  0   :      D@67:<!0:->        ExitOK(MustGen, W:SideState
68  0   :      D@68:< 1:->        JSConstant(JS|PureInt, Unde
69  0   :      D@69:<!0:->        MovHint(Check:Untyped:D@68,
70  0   :      D@70:< 1:->        SetLocal(Check:Untyped:D@68
71  0   :      D@71:<!0:->        MovHint(Check:Untyped:D@68,

    //... truncated for brevity

93  0   :      D@93:< 1:->        JSConstant(JS|PureInt, Weak
94  0   :      D@94:<!0:->        MovHint(Check:Untyped:D@93,
95  0   :      D@95:< 1:->        SetLocal(Check:Untyped:D@93
96  0   :      D@96:<!0:->        PutByVal(Check:Untyped:D@93
97  0   :      D@97:<!0:->        Flush(MustGen, loc9(S~/Flus
98  0   :      D@98:<!0:->        Flush(MustGen, loc10(O~/Flu
99  0   :      D@99:< 1:->        JSConstant(JS|PureInt, Unde
100 0   :     D@100:<!0:->        MovHint(Check:Untyped:D@99,
101 0   :     D@101:< 1:->        SetLocal(Check:Untyped:D@99
    0   :    <-- inlineFunc#EfyD9C:<0x7fffaf2c4260, bc#74,
102 0   :   D@102:< 1:->          JSConstant(JS|PureInt, Unde
```

```
103  0    :  D@103:<!0:->        Return(Check:Untyped:D@102,
104  0    :  D@104:<!0:->        Flush(MustGen, arg1(B~/Flus

//... truncated for brevity
```

Conclusion

This post explored the various components that make up the DFG JIT tier in JavaScriptCore by understanding how bytecode is parsed to generate a graph in the DFG and reading DFG IR. Part IV of this blog series will dive into the details of graph optimisation that's performed by the DFG.

Appendix

1. https://github.com/saelo/pwn2own2018 ↩
2. https://www.zerodayinitiative.com/blog/2019/3/14/the-apple-bug-that-fell-near-the-webkit-tree ↩
3. https://www.thezdi.com/blog/2019/11/25/diving-deep-into-a-pwn2own-winning-webkit-bug ↩
4. http://www.filpizlo.com/slides/pizlo-speculation-in-jsc-slides.pdf#page=75 ↩
5. http://www.filpizlo.com/slides/pizlo-speculation-in-jsc-slides.pdf#page=61 ↩
6. https://webkit.org/blog/10308/speculation-in-javascriptcore/ ↩ ↩

#JSC #Safari #WebKit #DFG

8016 Words

2021-05-26 00:00 +0000