Compiled Compiler Templates for V8

or:

How I Learned to Stop Worrying and Love JavaScript

Masterarbeit der
Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

Vorgelegt von

Olivier Flückiger

2014

Leiter der Arbeit
Prof. Dr. Oscar Nierstrasz
Dr. Mircea Lungu

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://scg.unibe.ch/

Acknowledgments

I am very grateful for all the kind support I received over the last months – most of it is probably not even mentioned here.
I'd like to thank Oscar Nierstrasz for supporting my ideas, pushing me forward and the empowering supervision.
Toon Verwaest you have been an inspiration and I really enjoyed our long evenings at the Isar.
I'm much obliged to the whole V8 team for making me feel so welcome and providing me with endless riddles in C++. I thank Daniel Clifford for his exceptional mentoring. And Ben Titzer, Hannes Payer, Michael Stanton, Michael Starzinger, Toon Verwaest, and all the others for the interesting time in Munich.
Merci to my good friend Marcel Härry for proofreading.
And of course babette who caught my fall.

Abstract

The performance of JavaScript virtual machines (VM) improved by several orders of magnitude in the last years, giving raise to ever more complex applications. Consequently there is a high demand for further compiler optimizations and therefore short VM development cycles. V8 the JavaScript engine of the Chrome web browser has a baseline compiler optimized for low compiler latency and an optimizing compiler called Crankshaft to compile hot functions. To achieve better maintainability and to improve robustness we aim at sharing compiler infrastructure and code generating back end between the two compilers. Our approach instruments Crankshaft to generate native code templates for the baseline compiler. We show that our method is practical, reduces complexity and maintains performance. As a case study to support our claims we ported the assembly level implementation of binary operations for baseline code to a version compiled by Crankshaft.

Contents

1 Introduction **1**

2 V8 in a Nutshell **3**
2.1 Overview . . . 3
2.1.1 Canonical Example . . . 5
2.2 Hidden Classes and Object Layouts . . . 6
2.3 Baseline Compiler . . . 6
2.3.1 Inline Caches . . . 8
2.4 Crankshaft . . . 10
2.4.1 Representations . . . 11
2.4.2 Compile-Phases and Optimizations . . . 11
2.4.3 Deoptimizer . . . 13
2.5 HydrogenCodeStubs . . . 14

3 Binary Operations **16**
3.1 BinaryOpStub . . . 16

4 Results **20**
4.1 Killed Lines of Code . . . 20
4.2 Reduced Complexity . . . 21
4.3 Performance . . . 22

5 Implementation **25**
5.1 ToBooleanStub . . . 25
5.2 BinaryOpStub . . . 28
5.2.1 Minor Key . . . 30
5.3 Supporting Infrastructure . . . 31
5.3.1 Back to the Future with X87 . . . 31
5.3.2 To Integer Conversions . . . 31

6 Related Work **33**

7 Conclusions **34**

Chapter 1

Introduction

V8 is an open source JavaScript engine developed by Google. It is used by several projects such as the Chrome web browser or the node.js platform. V8 features a state of the art JIT compiler to compile ECMAScript as specified in ECMA-262, 5th edition [8] to native code. The officially supported CPU architectures are IA-32, x68-64, ARM and MIPS.

The first released version of V8 featured a whole-method JIT with a simple code generator, no register allocator and with complex instructions. Core language semantics are realized by so called *CodeStubs* which can be understood as drop-in machine code fragments providing an implementation for specific AST nodes. They are generated by CodeStub builders, which are templates to generate a specific CodeStub variant e.g. specific to a certain register combination or primitive input type. CodeStubs are shared, thus executed in their own call frame.

Only later V8 was extended by an optimizing compiler called Crankshaft. Still JavaScript programs are first compiled using the aforementioned *baseline compiler* and only hot functions are optimized by Crankshaft.

Its high-level intermediate representation (HIR) is called Hydrogen. It is lowered to the platform dependent low-level IR Lithium which has its own back end to generate native code. Naturally Hydrogen instructions partially overlap in functionality with CodeStub builders.

A recent effort incrementally tries to unify the redundancy in implementation by (i) extending Crankshaft to support the full range of functionality as the corresponding CodeStubs builders do and then (ii) instrumenting Crankshaft to implement CodeStubs builders in Hydrogen. This approach was already applied in practice to a couple of Stubs (like the ones for field access and object allocation) by other V8 developers.

Thesis

We argue that implementing baseline compiler templates in high-level IR is practical, simplifies the implementation and is better maintainable. To validate our thesis we ported V8 CodeStubs for binary operations to Hydrogen. In particular we claim,

- Avoiding platform specific code.
- Reducing overall complexity.
- Unifying implementations and avoiding danger of diverging behavior.
- Maintaining the same level of performance.

Outline

The next chapter 2 provides an overview over V8 with the intention of making the reader familiar with the most important concepts, focusing on the parts central to understanding the contributions presented in this thesis. Subsequently chapter 3 focuses on binary operations in JavaScript and their existing hand-assembled implementation in V8. Finally in chapter 4 and chapter 5 we present our contributions – the former focusing on results and key figures for the hydrogenisation of binary operations whereas the later dwells in the details of implementation. Later chapters discuss related work and present conclusions.

Chapter 2

V8 in a Nutshell

In this chapter we first provide a dense overview of the compilation pipeline and then highlight the relevant parts independently.

2.1 Overview

JavaScript is a dynamic, object-oriented, prototype based language [8]. Definitions and declarations are late-bound, functions first-class. In contrast to languages with e.g. static class declarations, structure and behavior of JavaScript objects are not known at compile-time. In general it is hard to extract a meaningful type-like structure for various reasons.

At most objects might be identified to behave alike by observing the prototype chain. This would be practical in a limited number of simple situations in well-behaving programs, e.g. if no leaf objects in the prototype chain have methods and the prototype chain is never rewritten. In real world applications anonymous objects alone account for a relevant part of all live objects [15].

Additionally properties (later also called *fields*) are declared implicitly the first assignment. They can be declared in a dedicated constructor method[1] by a well-engineered program, but are found to be declared ad-hoc at a later point in time in a significant number of real world situations [15]. This makes it a hard problem for a compiler to define an efficient memory layout for objects.

V8 gathers information about properties and methods of objects at runtime and captures it with hidden classes [17, 2], also called *maps*. As an example the first step in Figure 2.1 shows the assignment of a new property x of type t to an object with initial map m. The object transitions to the (lazy) map m', a subtype of m having a field named x prepended. The field has a certain representation r capable of storing t.

Transitions and maps form the edges and vertices of the hidden class hierarchy, capturing state and structure of the emergent sub-type relations between

[1] A constructor in JavaScript is just a normal first-class JavaScript function having *this* bound to the newly allocated object. The constructor is invoked by the *new* keyword.

JavaScript objects. Objects are grouped by the names of properties and methods, and their order of declaration as shown in Figure 2.1.

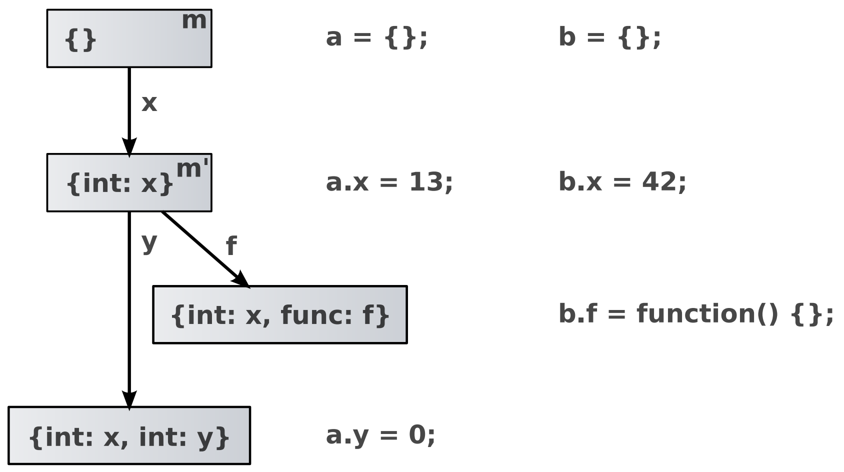

Figure 2.1: Hidden classes are adaptive models for JS objects. Two different code paths (from top down) generate different maps for objects a and b.

Functions are lazily compiled by the *baseline compiler*. Its execution model being a very simplistic stack machine, code can be generated fast and cheaply keeping compiler latency minimal. In contrast to a *virtual* stack machine baseline code runs on bare metal. Complex AST node functionality is provided by so called CodeStubs – shared pieces of native code implementing functionality as diverse as method invocation, string concatenation, math functions, object construction, and so on.

In general CodeStubs of a certain kind exist in several variants e.g. they are specific to the primitive type of arguments. A call site is specialized to a certain CodeStub variant by an inline cache (IC). The different variants of a certain CodeStub are generated by a common CodeStub builder.

After a warmup phase in baseline code functions worthwhile to optimize are marked for recompilation in a background thread by the optimizing compiler. This heuristic considers if the function is hot (i.e. a lot of time is spent in it), the number of instructions, how much room for optimization there is, and so on.

The baseline compiler described so far corresponds to the left hand code generation path in Figure 2.2. Following the right hand compilation path the optimizing compiler (*Crankshaft*) transforms AST into an IR called Hydrogen. Several optimization and inference phases are applied and the resulting graph is transformed into Lithium, a platform specific low-level IR. Then a modified linear scan register allocator [13] assigns registers and a final code generation phase assembles the code.

For this recompilation the AST is annotated with type feedback recorded

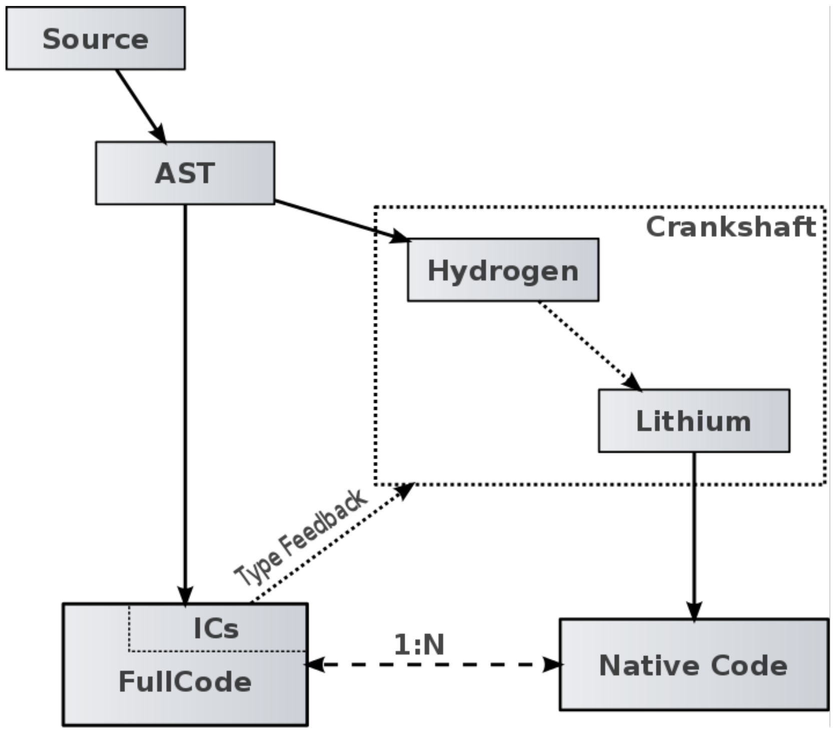

Figure 2.2: Overview over the V8 compiler architecture.

by the execution of baseline code. This e.g. includes maps of receiver for a certain call site, primitive types for operators, or field representation for loads and stores. The recorded types provide the basic assumptions for Crankshaft to select specific instructions or instruction sequences [19]. For sites with specific enough type feedback property access, function calls, array access, and other common patterns are compiled to native load, store and call instructions.

All assumptions about values, their maps and representations which cannot be proved at compile time[2] must be guarded by runtime checks. If one of them fails, the optimized code will jump through a jump table into the deoptimizer, which will rewind to the last safe point, rewrite the current call frame into baseline format and resume execution in unoptimized code.

2.1.1 Canonical Example

The following chapters will explain selected parts of V8 in more detail. Along the way we will analyze an exemplary program shown in Listing 2.1 which exercises the features of JavaScript we want to address.

[2]Some constraints can be asserted statically, e.g. by the semantics of the value producing instruction, but in general only few static guarantees are given by JavaScript semantics

Listing 2.1: A constructor method called *Duck* creates objects featuring a *height* property and a function to calculate the ducks *wingspan*.

```
function Duck(h) {
  this.height = h;

  this.wingspan = function() {
    return this.height * 4;
  }
}

var duck = new Duck(8.5);
```

2.2 Hidden Classes and Object Layouts

Hidden classes form a conservative approximation of the running JavaScript program's type structure. By ordering them using transitions as the order relation we extract a type lattice defining sub-type relations. In V8 this lattice defines the memory layout of objects.

JavaScript objects semantically behave like dictionaries but it would be very inefficient if they were implemented in the naive way. A dictionary lookup requires several indirections before the value can be retrieved, it is very inefficient regarding memory usage (since growing the dictionary is expensive) and therefore also poor at exploiting cache locality. A compact structural representation of objects is key to a fast dynamic language VM.

As mentioned before V8 objects are laid out in memory according to the transitions of the hidden class hierarchy. Assume an object o has a hidden class m_2 (defining property p_2) which has a parent class m_1 (defining p_1) rooting in m_0. Following the map transitions from the root class down we define the heap layout of o to be $[p_1, p_2]$.

The same definition is visualized in Figure 2.3 which also shows that V8 employs split object layouts. Objects corresponding to different leaves in a common hidden class hierarchy share the same field offsets for common properties.

In baseline code objects are allocated with a certain amount of slack space at the end, since they are expected to grow as more properties are defined and their final size is unknown. In optimized code however we expect to capture the system in a stable state thus allocations are performed to match the actual depth of the hidden class chain.

2.3 Baseline Compiler

The baseline compiler is a simple one-pass compiler backed by an extensive runtime infrastructure [21]. Primitive operations are delegated to CodeStubs and runtime functions written in assembly, Hydrogen, C++, or JavaScript. The target CPU is used solely as a stack machine, so no explicit register allocation

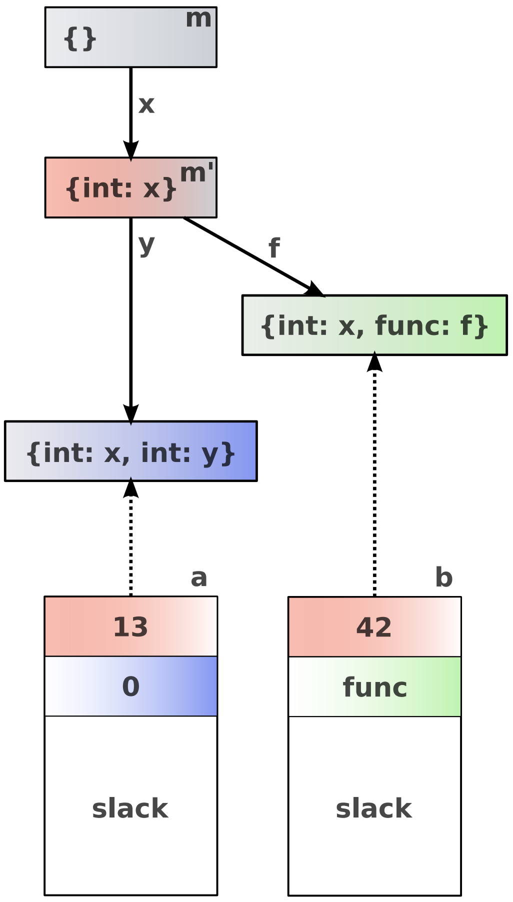

Figure 2.3: Objects as defined in Figure 2.1 are layed out in memory according to the order of properties given by the hidden class hierarchy.

takes place. Most AST nodes corresponding to non-trivial functionality are lowered to inline-cached calls to a corresponding CodeStub. As an example considering the function in Listing 2.1 with its corresponding AST in Figure 2.4 we identify two interesting nodes: *property access* and *binary operation.*

The property access relies on the object layout of the variable *this*, therefore the inline cache entries are selected according to the objects *hidden class.*

The binary operation is selected depending on the primitive type of operands. In the given example it multiplies the float 8.5 with the integer constant 4, producing an integer as result. The possible types for arguments and results of binary operations include numbers fitting into a tagged[3] or an unboxed integer, floats, the global $undefined$ object, generic objects, or (for addition) strings. In the case of binary operations the CodeStub is selected from the space spanned

[3]Tagged integers in V8 have 31 bits of payload and the least significant bit set to 0.

by $\langle left \times right \times result \rangle$ according to the type of left, right operand and result.

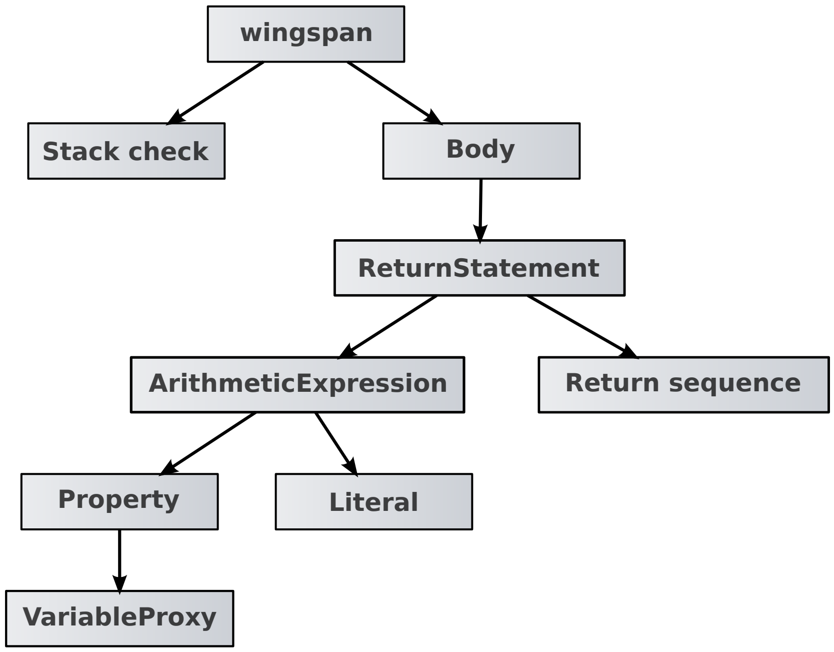

Figure 2.4: AST of the wingspan function as defined in Listing 2.1 line 4.

2.3.1 Inline Caches

The baseline compiler translates many AST nodes to CodeStub calls. Almost all of them are called through an inline cache which can contain entries for several possible types. First of all this provides good performance for all Stubs which can be specialized to certain maps. For example a Stub for a property access consists of just one indirect memory access if the memory layout (i.e. the map) of the receiver and the property offset are already known from a previous lookup. Secondly in the case of operations which are specific to primitive types (i.e. numbers, strings, floats) the inline cache ensures that the most specific (i.e. efficient) implementation is chosen at runtime. Those ICs specific to primitive types handle all cases up to the most generic one encountered so far. Thus previous IC entries are overwritten and they stay monomorphic. This concept is very similar to the idea of node rewriting [22] for interpreters.

Inline caches are implemented as small pieces of code separate from the call site method. As an example we consider the wingspan function in Listing 2.1. The baseline compiler emits a property access as a call to an instance of a *LOAD_IC*.

Listing 2.2: Loading a property in baseline code.

```
;; The VariableProxy 'this' was evaluated last,
;; therefore it's in the accumulator.
mov   edx, eax

;; Then we load the property name.
mov   ecx, 0x45b0a499      ;; object: <String[6]: height>

;; The actual call to the inline cache.
call  0x49d53e80           ;; debug: statement 63
                           ;; debug: position 74
                           ;; code: LOAD_IC, MONOMORPHIC,
                           ;; FAST (id = 8)
```

Following this call generated for the property access in Listing 2.2 we get to the load inline cache in Listing 2.3. We applied the *wingspan* function only to *Ducks*, therefore the IC is monomorphic. It performs one type check and then tail-calls a LoadFieldStub for the actual property access.

Listing 2.3: A monomorphic load IC.

```
      ;; Check for tagged integer.
      test_b dl, 0x1
      jz     miss

      ;; Typeguard: check the hidden class of the receiver
      mov    ebx, [edx+0xff]
      cmp    ebx, 0x23b17699   ;; object: <Map(elements=3)>

      ;; The actual property access.
      jz     0x49d53dc0        ;; code: HANDLER, LoadFieldStub,
                             ;; minor: 65654

      ;; The ic miss handler.
miss: jmp    LoadIC_Miss       ;; code: BUILTIN
```

Every CodeStub has a so called *minor key*, a dense 32 bit encoding of the Stub variant. In the case of the LoadFieldStub the minor key stores field type, field index, the store properties (in-object or in a backing store) and whether the field holds an unboxed double. This value is reused by the StubCache (a hash table for code objects) as a hash code. This ensures that every variant only has to be compiled once.

Inline caches are normal code objects on the V8 heap. They are cleared on garbage collection to allow the return to a monomorphic state if possible. Unnecessary polymorphic states can e.g. occur when a hidden class gets deprecated and replaced by a field representation change.

2.4 Crankshaft

The optimizing compiler (Crankshaft) is triggered to recompile and optimize hot methods in a background thread. Hot methods where most of the execution time is spent in are detected by counter based heuristics. Metrics to detect functions worthwhile to optimize include the number of invocations, the number of back edges taken in the control flow graph, or the amount of type feedback available.

Crankshaft features two intermediate representations (IR), the high-level Hydrogen and low-level, platform dependent Lithium [11]. Hydrogen is a graph based IR in SSA form [1]. The instruction graph representing data flow dependencies is superimposed on a control flow graph (CFG) [4]. Its nodes are basic blocks with an ordered list of Hydrogen instructions.

The example AST from Listing 2.1 is transformed by multiple passes to the Hydrogen graph in Listing 2.4. In the example the CFG consists of two sequential basic blocks, the first representing the method preamble and the second one the body. In *BasicBlock 1* the *CheckHeapObject* and *CheckMaps* (lines 12 and 13) are the typeguards for value *t*2 which is *Parameter 0* (*this*). If the typeguards succeed the double box is read out at the known offset (line 14) and then the double value is extracted from the box (line 15). The value is multiplied with a constant (line 17) and then boxed again (line 18) since only boxed values can be returned by a function.

Listing 2.4: Optimized Hydrogen graph produced by Crankshaft from the wingspan function as defined in Listing 2.1 line 4. Listing generated by the C1visualizer.

```
1   BasicBlock0:
2             BlockEntry
3   t1 <-     Context
4   t2 <-     Parameter 0
5             Simulate id=2 var[1] = t1, var[0] = t2
6             Goto B1
7
8   BasicBlockB1:
9             BlockEntry
10            Simulate id=3
11            StackCheck t1 changes[NewSpacePromotion]
12            CheckHeapObject t2 type:non-primitive
13            CheckMaps t2 [0x45617699]
14  t13 <-    LoadNamedField t2.[in-object]@12 type:heap-number
15  d14 <-    LoadNamedField t13.[in-object]@4
16  d21 <-    Constant 4  range:4_4
17  d16 <-    Mul d14 d21 !
18  t22 <-    Change d16 d to t, changes[NewSpacePromotion]
19                                type:heap-number
20  t23 <-    Constant 0  range:0_0 type:smi
21            Return t22 (pop t23 values)
```

2.4.1 Representations

Hydrogen values each are associated with one of the following representations, which form the lattice in Figure 2.5.

Smi A tagged integer (see [2]) with 31 bits of payload and the least significant bit (LSB) unset.

Integer32 An integer with 32 bit precision.

Double An unboxed double precision float value.

HeapObject A tagged pointer p. LSB set, real pointer value is $p \veebar 1$

Tagged A tagged value, either a tagged pointer or a tagged integer.

Instructions can specify constraints on the required input representation and adapt their output representation during compilation. Representations are not to be confused with types as e.g. the number 3 has type *float* in JavaScript, is stored as an *integer* inside V8 and can temporarily be kept in *Smi*, *Integer32*, *Double* and *Tagged* representation depending on the context.

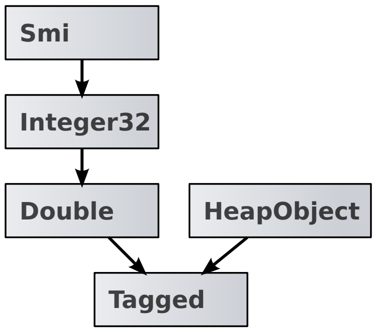

Figure 2.5: Value representations of Crankshaft.

In Listing 2.4 the representations are marked by mnemonic prefixes to the variable – t for tagged and d for double. Function parameters always have to be tagged, other representations are only used locally. For example in this listing the unboxed double result value has to be stored in a heap allocated number before it can be returned. Effectively the *Return* instruction on line 21 requires a tagged input value, necessitating the insertion of a *Change* instruction on line 18 to convert between the two representations.

2.4.2 Compile-Phases and Optimizations

When a function is to be recompiled by Crankshaft the first compilation step is to gather type feedback. As explained in subsection 2.3.1 while baseline code is executed, inline caches as a side-effect record the encountered maps of argument objects and the primitive types of arguments. Crankshaft's type oracle visits the AST and enriches all nodes with type information extracted from the inline caches of the corresponding baseline code.

Crankshaft builds a Hydrogen graph from the AST using this type feedback for instruction selection, representation inference and so on. At graph building time Crankshaft inlines functions if possible and considered advantageous by heuristics. Also constant folding is done at construction time [14].

Several optimizations and transformations [20] are then applied to the Hydrogen graph in sequential order including:

Dead code elimination Basic blocks which cannot possibly be reached are marked as unreachable and thus skipped on code generation.

Escape analysis Values never leaving the local context are detected. They do not have to be allocated on the heap.

Infer representations A simple cost approximating heuristic considers all usages of a certain value to determine the optimal representation. E.g. converting integers to doubles and vice-versa is an expensive operation, but subsequent operations with other integer/double operands might make it favorable to convert a value early.

Inferring types Type inference phase consults several facts about values to tighten the constraints on upper and lower bounds of possible types a value could hold. This includes a-priori knowledge about the value producing instruction, implicit and explicit type checks, and so on.

Representation changes Representation mismatch is fixed by inserting implicit representation conversion nodes.

Global value numbering A GVN [16] phase which needs to keep track of various side-effects certain JavaScript constructs exhibit.

Range analysis Possible numerical range of values is determined. The information is e.g. used to omit redundant overflow and bounds checks.

Compute change undefined to nan The JavaScript *undefined* and *nan* behave semantically very similar in a number of situations. Most importantly as argument of numerical operations they behave identical. Therefore it is sometimes favorable to use *nan* as a double value in combination with other double operands instead of the heap-object *undefined*.

Compute minus zero checks Since JavaScript only has a *float* number type the internal conversion to integer arithmetic has to be fully transparent. One notorious case is the difference between minus and plus zero: JavaScript on one hand deems them identical by means of any comparison operation but nevertheless a difference can be observed by dividing them by plus zero which results in either minus or plus infinity. Thus any integer operation which might result in the sign bit for zero being lost needs to check the condition at runtime.

Bounds checks elimination Bounds checks for array indices can be omitted if we can statically prove that the index lies within the array size. For example with loop unrolling it is possible to perform the array check in batches by already checking the bounds for a certain future offset.

2.4.3 Deoptimizer

As discussed in the previous chapter Crankshaft optimistically compiles code tailored to type-feedback from the warmup execution phase. At any point in time those assumptions can break, e.g. a type check fails, a variable contains an unexpected type. The optimized code becomes invalid and execution has to fall back to baseline code. The translation from optimized to non-optimized code is performed by a deoptimizer, which is an established mechanism [7, 9, 12] to provide generic mapping of program states between different compiled versions of the same program.

Every optimized code object contains meta-data (similar to debug data formats like DWARF encountered in ELF binaries). It defines a mapping of values to registers and stack slots. The meta-data contains safe-points within the function where the mapping between program counter (PC) values of optimized and baseline code is known. To bail out of optimized code the failed type-check calls into the deoptimizer, which will consult the meta-data to rewind to the last safepoint. It collects all live values, tears down the current call frame, then reconstructs the execution stack for the corresponding baseline code. Finally control is returned to baseline code by jumping to the safepoint PC value.

Many Hydrogen instructions have side effects observable by JavaScript code as defined in the ECMAScript standard. For example a property access might call into a user-defined getter method, a binary operation can cause ToString being called if non-primitive arguments are passed, or a certain operation might be intercepted by *Object.observe()*. When Crankshaft performs certain optimizations, like global value numbering (GVN), dead code elimination, or loop invariant code motion [10], it has to guarantee that the resulting program still behaves as if it fully adheres to the procedurally and precisely defined semantics of JavaScript.

Additionally the deoptimizer cannot rewind before an operation having observable side-effects as they would be observed twice in this case. Therefore every such instruction has to be followed by a guarding safe point. In Hydrogen the according instruction is called a *Simulate*. While building up the Hydrogen graph a virtual environment stack is used to keep values. This stack identically corresponds to the physical execution stack encountered in baseline code. A simulate records the state of this virtual stack; thereby storing a mapping of the actual values as they are represented in optimized code to the execution stack of baseline code.

2.5 HydrogenCodeStubs

As shown in section 2.3 the baseline compiler emits CodeStubs which implement the semantics of certain AST nodes. There are many ways to generate those Stubs. A very common situation in V8 is that they are written in V8 platform dependent macro assembly language.

On the other hand Crankshaft as noted in section 2.4 has its Lithium back end to lower Hydrogen instructions to machine code. Functionality of CodeStubs and the output of Hydrogen instructions partially overlap[4] or CodeStubs perform the equivalent of a sequence of Hydrogen instructions.

The same functionality being implemented several times poses problems. First the obvious one of having to maintain it several times, more code in general and similar artifacts in separate modules. Secondly the artifacts are in concept tightly coupled in several ways: Obviously the disjoint implementations have to agree on the semantics in all the corner cases – baseline and optimized code must behave identically. More subtly, both components have to produce matching meta-data for the deoptimizer or the stack state will be corrupted by deoptimization.

The goal must therefore be to unify the two disjoint implementations, which is exactly the idea of HydrogenCodeStubs: To instrument Crankshaft to generate CodeStubs for baseline code. The main advantages we hope to achieve with this approach being:

Ease of development It is easier to maintain CodeStubs since they are written in a higher level language.

Correctness CodeStubs are compiled – Crankshaft knows about object layouts, primitive types, bounds check and so on. Bugs are less likely than with unsafe assembly instructions.

Consistency Crankshaft can share the code between the CodeStub builder and the normal compiler, thus reducing the possibility of diverging behavior betweem baseline and optimized code.

Infrastructure Reuse HydrogenCodeStubs can reuse the deoptimizer infrastructure to handle IC failures: When a HydrogenCodeStub is not able to handle a certain argument type, it calls into the deoptimizer. By simply including the correct meta-data the deoptimizer is able to restore the initial Stub arguments from registers. It can then pass control to the IC patching mechanism. This function will trigger Crankshaft to compile a more generic Stub and patch the IC with it.

As an example consider the load IC in Listing 2.3 which delegates to LoadFieldStub. As can be seen in Listing 2.4 Crankshaft is fully able to inline an

[4] Although some overly complex Lithium instructions do not yet have an implementation of their own and therefore also emit a call to a CodeStub.

access to a field in the *wingspan* method. A LoadFieldStub can therefore be implemented by the means of a couple of Hydrogen instructions.

The LoadFieldStub is indeed not handwritten but generated on the fly by Crankshaft – the result is shown in Listing 2.5. It consists of the same instructions we can already see in Listing 2.4, but additionally boxes the result. This is not necessary in the former case since the loading is inlined, thus the internal double representation can be kept.

Listing 2.5: The full implementation of a LoadFieldStub called through an IC in Listing 2.3 to load the height property in Listing 2.1.

```
major_key = LoadFieldStub

; [Unboxing the number]
  ;;; <@10,#9> load_named_field
    mov eax, [edx+0xb]
  ;;; <@12,#10> load_named_field
    movsd xmm1, [eax+0x3]

; [Allocating a new double box]
  ;;; <@14,#13> number_tag_double
    mov ecx, [0x96fc590]
    mov eax, ecx
    add eax, 0xc

    ; [If pointer bump failed, need to grow heap or gc]
    jc 59  (Deferred allocation)
    cmp eax, [0x96fc594]
    ja 59  (Deferred allocation)

    mov [0x96fc590], eax
    inc ecx

    ; [populate the new double box]
    mov [ecx+0xff], 0x29808149    ;; <Map(elements=3)>
    movsd [ecx+0x3], xmm1

; [Returning the result]
  ;;; <@15,#13> gap
    mov eax, ecx

  ;;; <@16,#11> return
    ret
```

Chapter 3

Binary Operations

Binary operations in JavaScript are: bitwise OR, XOR, and AND, left shift, signed and unsigned right shift, plus, minus, multiplication, division, modulo [8]. The exact semantics for the different operations are defined in the ECMAScript standard. On primitive types the operators behave more or less straightforwardly.

The plus operator can either mean numerical addition or string concatenation – the latter being chosen if any of the two arguments *evaluate* to a string. Note that ECMAScript defines the evaluation of a non-primitive object operand o differently depending on the operation. In the case of binary plus the evaluation is to be $ToPrimitive(GetValue(o))$, where $ToPrimitive$ is a global function which converts JavaScript objects to primitive values and $GetValue$ is a global function defined to call the objects $valueOf$ method, which is user definable and can return values of any type. Therefore except in the case where all operands are primitive types it is not statically nor locally decidable if the plus operator represents addition or string concatenation.

The following section provides a short overview over the existing implementation of binary operations. Our own implementation is discussed in chapter 5.

3.1 BinaryOpStub

CodeStubs for binary operations are specific to the type of input arguments. In Figure 3.1 the state lattice of a binary operation IC can be seen. All states include support for all the less generic ones. E.g. if an IC is in state *Number* for the left hand operand the according CodeStub will be able to handle any numeric value from *Smi* to *HeapNumber* as left hand operand[1]. Binop ICs are always monomorphic, when a more generic type is encountered the IC is patched to a more generic CodeStub, depending on the type of operand that triggered the miss.

[1]See subsection 2.4.1 for an explanation of the different representations of numbers.

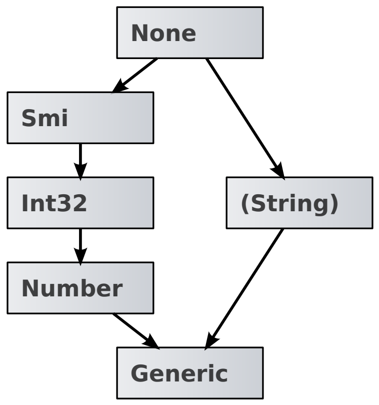

Figure 3.1: State machine of the BinaryOpStub IC. The *String* state is only recorded for addition, since a string argument changes the semantics of the plus operator.

The existing implementation of BinaryOpStub in V8 was implemented in V8 macro assembly using very few abstractions to perform the operations. It is completely separate from the corresponding implementation in Crankshaft. As an example the code shown in Listing 3.1 is responsible for generating the Smi handling part of a shift left CodeStub builder. The complete function is 373 LOC. Overall the ia32 CodeStub builder implementation for binary operations amounts to over 1200 LOC in this style.

Listing 3.1: Existing function to generate a shift left BinaryOpStub (Only relevant parts included). Lines starting with __ are macro assembler instructions and they emit native code accordingly.

```
static void BinaryOpStub_GenerateSmiCode(
    MacroAssembler* masm,
    Label* slow,
    SmiCodeGenerateHeapNumberResults allow_heapnumber_results,
    Token::Value op) {
  // 1. Move arguments into edx, eax except for DIV and MOD. [...]
  Register left = edx;
  Register right = eax;
  [...]

  // 2. Prepare the smi check of both operands by oring them.
  Label not_smis;
  Register combined = ecx;
  switch (op) {
    [...]
    case Token::SHL:
      // Move the right operand into ecx for the shift
      // operation, use eax for the smi check register.
      __ mov(ecx, right);
      __ or_(right, left);
      combined = right;
      break;
    [...]
  }

  // 3. Perform the smi check of the operands.
  __ JumpIfNotSmi(combined, &not_smis);

  // 4. Operands are both smis, perform the operation leaving
  // the result in eax and check the result if necessary.
  Label use_fp_on_smis;
  switch (op) {
    [...]
    case Token::SHL:
      // Remove tags from operands (but keep sign).
      __ SmiUntag(left);
      __ SmiUntag(ecx);
      // Perform the operation.
      __ shl_cl(left);
      // Check that the *signed* result fits in a smi.
      __ cmp(left, 0xc0000000);
      __ j(sign, &use_fp_on_smis);
      // Tag the result and store it in register eax.
      __ SmiTag(left);
      __ mov(eax, left);
      break;
    [...]
  }

  // 5. Emit return of result in eax. [...]
  switch (op) {
    [...]
    case Token::SHL:
      __ ret(2 * kPointerSize);
      break;
  }

  // 6. For some operations emit inline code to perform
  // floating point operations on known smis (e.g., if the
  // result of the operation overflowed the smi range).
  [...]

  // 7. Non-smi operands, fall out to the non-smi code with
  // the operands in edx and eax.
  __ bind(&not_smis);
  switch (op) {
    [...]
    case Token::SHL:
      // Right operand is saved in ecx and eax was destroyed by
      // the smi check.
      __ mov(eax, ecx);
      break;
    [...]
  }
}
```

The implementation in Listing 3.1 performs the following steps: (1) Set up a call frame for some operations or copy values to a backup register. (2,3) Perform a Smi check on both input arguments. As this part of the Stub compiler which only deals with Smi values, a jump target is provided for a later section to be compiled to deal with other kind of types. (4) The actual operation is performed. (5) The return statement if everything was successful. (6) Some special fast cases if there was an overflow in (4) and finally (7) fallback code if (2, 3) failed.

Chapter 4

Results

In this chapter we discuss the results and experience gained from implementing the CodeStub builder for binary operations using Hydrogen, the HIR of V8. We present this reimplementation as a representative case study of our approach. Additional changes which enabled and facilitated the hydrogenisation of binary operations, technical details, as well as the hydrogenisation of the ToBoolean operation are discussed in chapter 5. The semantics of binary operations in JavaScript and the previous assembly implementation of binop Stubs are discussed in chapter 3.

Results in this chapter are based on the Hydrogen BinaryOpStubs as they were commited to V8 trunk in SVN revisions 17104 and 17108 and additional bugfixes in 17143, 17144 and 17290. Benchmarks were performed using the standalone CLI version of V8, called D8.

4.1 Killed Lines of Code

As already mentioned in chapter 3 just the ia32 code for generating the BinaryOpStub was over 1200 LOC not counting some additional utility code for handling double values with the floating point unit. The other architectures had a bit less code (since historically more optimizations went into ia32): 1000 LOC for arm, 700 LOC for x64 and 949 LOC for mips.

Overall the hydrogenisation of binary operations removed 4726 LOC while introducing 1008 LOC. The data is summarized in Figure 4.1. Only 39 lines of platform dependent code used for declaring the calling conventions remain after the rewrite.

Overall 4 platform specific implementations were removed and unified with a single implementation which reuses for the most part the very same builder function used by the normal Crankshaft pipeline to compile binary operations.

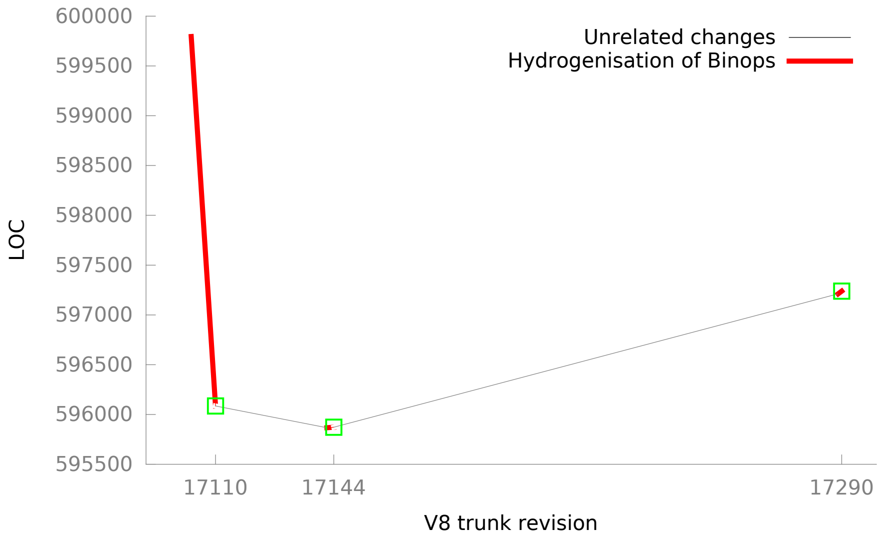

Figure 4.1: LOC in the src folder of V8 repository. Marked revisions are part of Hydrogenisation of binary operations.

4.2 Reduced Complexity

The Crankshaft implementation of binary operations for optimized code and the Hydrogen BinaryOpStubs implementation share most of the functionality. The only special cases the Stub handles, which are not part of the normal compiler pipeline are (i) a fast case for string concatenation in the generic add Stub and (ii) the possibility to reuse double boxes.

For more details on the first special case see section 5.2. The other special case covers chained operations. Consider the expression $a * b * c$ where the semantics of JavaScript guarantee the result of $a * b$ to be a number. Ad-hoc escape analysis shows us that the result is only used locally. We can therefore store the result of the second multiplication in the double box allocated by the first multiplication. This case however is only relevant for baseline code as in optimized code numbers are unboxed. For a detailed explanation of the implementation see section 5.2.

Except for those two special cases the implementation is fully shared, thus semantics of Stub code and the corresponding optimized code are guaranteed to be (and stay) equivalent. This unification makes the code base more maintainable and will facilitate future refactoring, like the hydrogenisation of string concatenation.

Though hard to measure the Hydrogen implementation seems to contain fewer implicit assumptions about the rest of the system. There are several examples to support this claim: Hard coded offsets to manipulate double boxes are replaced by Hydrogen field access instructions. Code generation for arithmetic operations is not ad-hoc and implemented redundantly.

Also a bug was revealed where the Stub implementation had a specific slow case to deal with a particular overflow flag situation, where the result would only seemingly overflow the Smi range. Thus the Stub IC could stay in Smi state in this situation – but of course the Lithium back end did not have the same behavior. This situation leads to a deopt-loop, where the baseline implementation would never pick up new type information, since it could perform the operation in Smi mode and the optimized code would always immediately deopt, since the generated code cannot perform the operation in Smi mode. This kind of diverging behavior is ruled out by sharing the code generation between baseline and optimized code.

4.3 Performance

Performance of the existing binary operations were compared to the new Hydrogen implementation using industry standard benchmarks, namely Kraken JavaScript Benchmark version 1.1 from Mozilla and Octane 1.0 from Google. We see in Figure 4.2 that no significant regressions for the overall score of both benchmarks exist. Additionally Figure 4.3 shows a breakdown of the Octane score, where none of the partial scores show a significant regression.

Although we do not consider the SunSpider benchmark to be adequate to measure the performance of modern JavaScript VMs[1], we note that according to Figure 4.2 no significant regression in SunSpider 1.0.2 overall score is observable.

An important property of Hydrogen CodeStubs is that they benefit from future improvements in Crankshaft. Therefore achieving performance on par with the assembly version (as we demonstrated it for BinaryOpStubs) only reflects the current abilities of Crankshaft at optimizing and compiling a certain Stub implementation and it is likely to improve over time.

[1]SunSpider consists of a collection of very short running micro benchmarks. Profiling the execution of SunSpider in V8 reveals that a significant amount of time is spent in parts of the system, which were not intent to be measured by the developers of SunSpider, e.g. parsing, warmup, initial cache lookups. Additionally results are poorly verified, if used at all, which makes the code to be benchmarked subject to dead code elimination. Since the benchmarks are so short, most time is spent in unoptimized code, which is not representative of modern JavaScript applications.

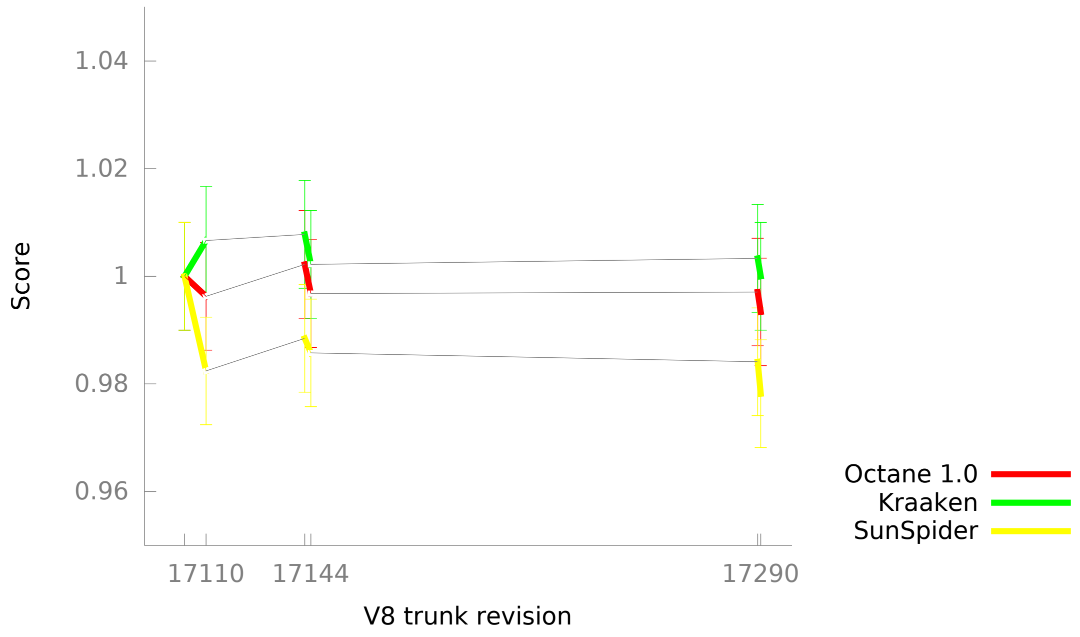

Figure 4.2: Kraken JavaScript Benchmark version 1.1 from Mozilla, Octane 1.0 from Google and SunSpider 1.0.2 performance relative to trunk revision 17103 (higher is better). Results are averaged over 10 runs on a Intel Core i5 and 10 runs on a Intel Core 2 machine. Thin line segments represent trunk revisions unrelated to BinaryOpStubs.

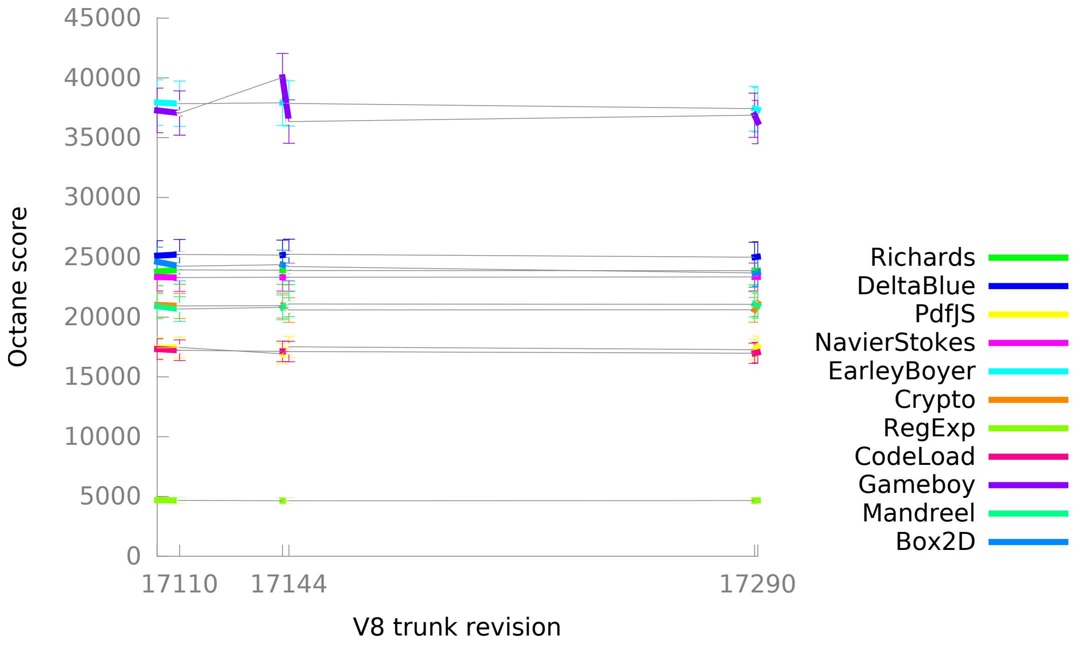

Figure 4.3: Breakdown of Octane 1.0 scores (higher is better). Results averaged over 10 runs on a Intel Core i5 machine. Thin line segments represent trunk revisions unrelated to BinaryOpStubs.

Chapter 5

Implementation

We discuss the details of two hydrogenisations of CodeStubs in this chapter. First the reimplementation of ToBoolean, a truth evaluating global function defined by ECMAScript. It served as a proof of concept and allowed us to explore different approaches on a simple test case. Secondly we present the implementation details of the Hydrogen BinaryOpStub whose performance is evaluated in depth in chapter 4 in support of our thesis.

5.1 ToBooleanStub

The global *ToBoolean* function is used when a JavaScript value is to be used in a boolean context e.g. as predicate for a conditional statement. Baseline code invokes a ToBooleanStub to evaluate the truth value. The Stub returns with 0 or 1 since it is mostly used for conditional jumps where the comparison against a global object (true/false) is more expensive. If used in a value context the true or false object must be explicitly loaded after the Stub call. The ToBooleanStub is also an IC Stub, only ever covering the types encountered so far at a specific call site. The types and values representing ToBooleanStub IC states cover the full semantics of truth in JavaScript:

Boolean the global true/false objects,

Undefined the global *undefined* object is $\perp$,

Null the global *null* object is $\perp$,

Smi or HeapNumber a numerical value x is $\top$ iff $x \neq 0$,

Strings strings of any kind are $\top$ iff they are non-empty.

Object an object o is $\top$ unless it is undetectable[1].

[1]A V8 specific internal notion to mark certain global objects defined to evaluate to $\perp$ by ECMAScript standard.

The ToBooleanStub in V8 used to be implemented by around 100 lines of macro assembly per platform. In this particular example the almost literally same code was produced by the HBranch Hydrogen control flow instruction already. The instruction has an input value used as boolean predicate and two output basic blocks as jump targets. HBranch in optimized code is type specific and corresponds to a ToBooleanStub in baseline code which provides the type feedback.

Using a simple helper class called IfBuilder containing some accounting code to deal which control flow graphs we reimplemented the ToBooleanStub in Hydrogen with just seven lines of code, as shown in Listing 5.1. The Hydrogen ToBooleanStub was committed to V8 trunk in revision 14886.

Listing 5.1: The Hydrogen implementation of the ToBooleanStub.

```
1  IfBuilder if_true(this);
2  if_true.If<HBranch>(GetParameter(0), stub->GetTypes());
3  if_true.Then();
4  if_true.Return(graph(->GetConstant1()));
5  if_true.Else();
6  if_true.Return(graph(->GetConstant0()));
7  if_true.End();
```

Note the second argument given to the HBranch instruction in Listing 5.1 line 2, the set of supported types. Peeking into the corresponding Lithium LBranch back end in Listing 5.2 we notice the instruction ending in an unconditional deopt point. As the comment in this listing already states this point is reached if none of the previous test and compare instructions were successful and it is unreachable (thus omitted) if the instruction is generic meaning all types are covered. Since in Hydrogen CodeStubs we use the deoptimizer for IC miss handling (additionally to the normal fallback to baseline code story) this LBranch implementation can be fully shared between Stub code and optimized code.

In the ToBooleanStub the *expected* set of types passed to the HBranch instruction represents exactly the IC state. The corresponding set of types for optimized code generation is gathered by the type oracle querying aforementioned baseline IC state.

Listing 5.2: Deoptimizer call in the implementation of the Lithium LBranch instruction.

```
void LCodeGen::DoBranch(LBranch* instr) {
  [···]

      ToBooleanStub::Types expected =
          instr->hydrogen()->expected_input_types();

      [···]

      if (expected.Contains(ToBooleanStub::BOOLEAN)) {
        // true -> true.
        __ cmp(reg, factory()->true_value());
        __ j(equal, instr->TrueLabel(chunk_));
        // false -> false.
        __ cmp(reg, factory()->false_value());
        __ j(equal, instr->FalseLabel(chunk_));
      }
      if (expected.Contains(ToBooleanStub::NULL_TYPE)) {
        // 'null' -> false.
        __ cmp(reg, factory()->null_value());
        __ j(equal, instr->FalseLabel(chunk_));
      }

      [···]

      if (!expected.IsGeneric()) {
        // We've seen something for the first time -> deopt.
        // This can only happen if we are not generic already.
        DeoptimizeIf(no_condition, instr->environment());
      }
    }
  }
}
```

Subsequently we did some experiments splitting the HBranch implementation into smaller instructions. The experimental builder function in Listing 5.3 implements the same functionality in Lithium as Listing 5.2 in assembly. We completely replaced the usage of HBranch by this builder function and verified correctness using the V8 test suite. Unfortunately we were not able to avoid all performance regressions, thus the implementation was not committed.

The main reason is that Crankshaft architecture assumes primitive operations to be implemented by one Hydrogen instruction. E.g. representation inference, instruction selection and optimizations are performed on a per-instruction basis. Thus having a single Hbranch instruction with a flexible representation is more suitable for the current architecture than several instructions possibly interfering with representation inference and causing boxing and unboxing operations. The approach could be facilitated by supporting nodes which can be expanded to a whole subgraph at a later compile phase.

Listing 5.3: Experimental HBranch split into smaller instructions.

```
void HGraphBuilder::BuildToBoolean(
    HValue* value,
    ToBooleanStub::Types types,
    int position,
    HIfContinuation* continuation) {

  [···]

  if (types.Contains(ToBooleanStub::BOOLEAN)) {
    test.TrueIf<HCompareObjectEqAndBranch>(
        value, graph()->GetConstantTrue());
    test.FalseIf<HCompareObjectEqAndBranch>(
        value, graph()->GetConstantFalse());
  }

  if (types.Contains(ToBooleanStub::NULL_TYPE)) {
    test.FalseIf<HCompareObjectEqAndBranch>(
        value, graph()->GetConstantNull());
  }

  [···]

  test.CaptureContinuation(continuation);

  end->FinishExitWithDeoptimization(HDeoptimize::kUseAll);
}
```

5.2 BinaryOpStub

As discussed in section 4.2 BinaryOpStub mainly delegates to *BuildBinaryOperation*, the Crankshaft builder function for binary operations. In Listing 5.4 lines 11 to 26 we see that the Stub can handle one additional special case which is not present in the Crankshaft builder yet. When the Stub is generic there are two different branches generated: next to generic addition there is a fast case for string concatenation.

The reason for the string fast case is the frequent occurrence of string concatenations where one of the operands is sometimes numeric. This case is only generated in the CodeStub builder and not inlined in optimized code because the additional branch increases code size causing performance regressions. At the current state of development it is advantageous to isolate this special case in the generic Stub. With the hydrogenisation of string concatenation the situation is likely to change since concatenation will be decomposable and its instructions accessible to Crankshaft's optimizations.

Listing 5.4: HydrogenCodeStub implementation of BinaryOpStubs.

```

template <>
HValue* GraphBuilder<BinaryOpStub>::BuildCodeInitializedStub() {
  [···]

  if (stub->operation() == Token::ADD  &&
      left_type->Maybe(Type::String()) &&
      !right_type->Is(Type::String())) {
    // For the generic add stub a fast case for String
    // addition is performance critical.
    IfBuilder if_leftisstring(this);
    if_leftisstring.If<HIsStringAndBranch>(left);
    if_leftisstring.Then();
      Push(BuildBinaryOperation(
                  stub->operation(), left, right,
                  handle(Type::String(), isolate()), right_type,
                  result_type, stub->fixed_right_arg()));
    if_leftisstring.Else();
      Push(BuildBinaryOperation(
                  stub->operation(), left, right,
                  left_type, right_type, result_type,
                  stub->fixed_right_arg()));
    if_leftisstring.End();
    result = Pop();
  } else if ([···]) {
    [···]
  } else {
    result = BuildBinaryOperation(
            stub->operation(), left, right,
            left_type, right_type, result_type,
            stub->fixed_right_arg());
  }
  [···]

  // Reuse the double box of one of the operands if we are
  // allowed to (i.e. chained binops).
  if (stub->CanReuseDoubleBox()) {
    HValue* operand = (stub->mode() == OVERWRITE_LEFT)
                        ? left : right;
    [···]


  return result;
}
```

The code generated from the Hydrogen implementation of BinaryOpStubs is very similar to the old assembly version. Of course there are differences, e.g. the Stub used to check if both operands l and r are Smis at once by testing the Smi tag on $l \,\hat{}\, r$. We abstained from teaching Crankshaft this micro optimization.

5.2.1 Minor Key

Compiled CodeStubs are stored an retrieved from a cache since they are shared between call sites. CodeStubs have a major key designated to the type of Stub (e.g. BinaryOp, Load, Store, ...) and a minor key encoding the variant. Together they form the identity hash used for the StubCache hashmap.

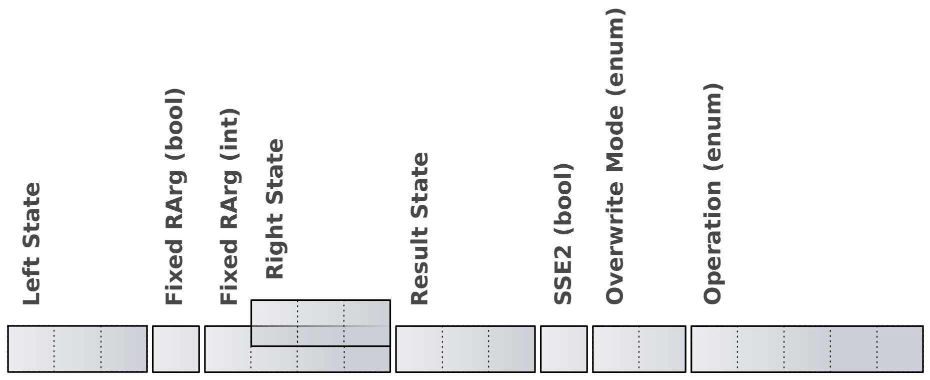

Figure 5.1: The minor key of a BinaryOpStub from left to right: (i) left argument type according to Figure 3.1, (ii) boolean indicating whether the right argument was constant so far (only for mod), (iii) the constant right argument (only used if (ii) is true), (iv) the right argument type (only used if (ii) is false), (v) the result argument type, (vi) whether the stub was compiled for an SSE architecture, (vii) whether the left or right hand double box can be reused to store the result, (iix) the binary operation.

As already hinted in chapter 3 there exist many variants of BinaryOpStubs. In Figure 5.1 we show how the 19 bits in the minor key are used to encode all possible variants. First of all the Stub is specific to the binary operation of course – five bits are used to encode the 11 different operations. Then we specialize according to both argument- and the result types – each being in one of the states from Figure 3.1.

There is a catch: integer modulo arithmetic of the form $a \,\%\, 2^n$ which is often used e.g. in cryptographic functions, can be simplified to $a \,\&\, (2^n - 1)$. Since this expression is much less expensive it even pays off to check for constant arguments in the form 2^n at runtime. Thus there are special modulo Stubs to capture and remember constant rvalues. Since the minor key bitfield size is limited we use an ad-hoc compression: if the right hand argument was constant so far we set bit 3 (Fixed RArg) and use bit 4 to 7 to encode the exponent n. We do not need to store the type of the right argument, since we can recover it (Smi or Integer32) from the precise numerical value 2^n. If the right hand argument is not constant we clear bit 3 and use bit 5 to 7 for the normal right-state encoding.

The remaining 3 bits are used as follows: bit 12 marks BinaryOpStubs compiled for architectures supporting the SSE2 instruction set. This is necessary so we can safely include SSE2 stubs in the precompiled snapshot image. Bits 13/14 specify Stubs which can reuse the left/right argument double box to return the result.

5.3 Supporting Infrastructure

Especially for the hydrogenisation of BinaryOpStubs several functional extensions to Crankshaft had to be made. We discuss some of those in the following sections.

5.3.1 Back to the Future with X87

The Chrome browser and V8 still officially support Intel CPUs without the Streaming SIMD Extension (SSE) instruction set. For Crankshaft however the design decision was made not to support those platforms; only the baseline compiler does. Of course the problem for Hydrogen CodeStubs being that they need to be available on all supported platforms, since they are called mainly from baseline code.

For the hydrogenisation of arithmetic operations we therefore had to add the necessary support in Crankshaft to emit non-SSE code for all required Hydrogen instructions. In particular floating point arithmetic has to be performed using the x87 floating point-related subset of the IA-32 instruction set. Due to the very low number of affected users and the high complexity of precisely tracking the x87 stack registers a very simple but practical approach was chosen. Two virtual registers are given to the normal register allocator to store double values. We keep track of those two registers in the Lithium backend and whenever the order of the physical stack registers does not match the virtual registers we emit an *fxch* instruction to swap stack position 0 and 1. This fits into the existing architecture since Crankshaft already allocates gap inter-instructions to resolve phi nodes by moving values between registers. The implementation is efficient since the BinaryOpStubs in the fast case don't have more than two double values live at the same time, thus no values have to be spilled. Additionally *fxch* can be paired with many fpu instructions or removed completely from the execution pipeline thus its latency is significantly below 1 cycle [5, 3].

5.3.2 To Integer Conversions

Additionally to the introduction of x87 stack values discussed in subsection 5.3.1 we had to implement the various conversions of such values to and from integer registers and boxed values. We support truncating conversions which round down to the next integer value e.g. used for shift operations and non truncating conversions which bail out if precision is lost due to the conversion.

Double to integer conversion is highly dependent on the CPU generation for Intel architectures. E.g. with the introduction of the *fisttp* instruction with SSE3 the truncating conversion has become much faster and easier to implement. For older models its even cheaper to perform bit arithmetic than to use the designated x87 instructions.

As a simplification we encapsulated the four possible conversions[2] and the existing SSE versions into macro assembly instructions and replaced all ad-hoc occurrences of such conversions in V8 by them. Thereby we ensured that the supported value ranges are always the same thus reducing a possible source of diverging behavior. For simple conversions the macro assembler inlines them; more lengthy ones are in deferred code or in their own CodeStub to increase code density and reduce memory footprint.

We also applied instruction level optimizations to those conversions in the Lithium back end, since they are performance critical to BinaryOpStubs. As an example on x64 we made the following improvement to the unboxing operation. The logical instruction ordering (in pseudocode) being

```
1  function UnboxHeapNumber(maybe_number, mmx_register)
2    deoptimize if IsSmi(maybe_number)
3    deoptimize unless IsHeapNumber(maybe_number)
4    mmx_register <- maybe_number[HeapNumberValueOffset]
5  end
```

By reversing lines 3 and 4 producing

```
1  function UnboxHeapNumber(maybe_number, mmx_register)
2    deoptimize if IsSmi(maybe_number)
3    mmx_register <- maybe_number[HeapNumberValueOffset]
4    deoptimize unless IsHeapNumber(maybe_number)
5  end
```

we remove a data dependency between the *mmx_register* and the map check $IsHeapNumber$ and facilitate preloading. The reordered function is still correct and safe since (i) *maybe_number* is guaranteed to be an object on the heap after the Smi check on line 2 and (ii) we know that there is no object smaller than a HeapNumber, thus line 3 will never read out of bounds. In the case the following map check fails it just produces a random value that can be discarded. There is no harm in loading that value in vain since deoptimizing is the slow case.

[2]Tagged to double and vice-versa, double to integer and vice-versa.

Chapter 6

Related Work

The Maxine Java VM follows a similar approach for code generation. The baseline compiler (T1X) is a template-based compiler optimized for fast compilation time. The templates for each bytecode are specified in an annotated Java subset and compiled ahead of time by the optimizing compiler. The approach is explained by Wimmer et al. [18]. The template language is more coarse and high-level than Hydrogen and provides less control over the generated code. Maxine templates have uniform calling conventions; every bytecode fully loads and restores its arguments from the stack. They provide more guarantees, e.g. type safety for the compiled code.

Würthinger et al. present a high-level language-agnostic VM implementation [23], which uses similar concepts for baseline code performance and type-feedback. Guest languages are implemented as AST interpreters. After an interpreter warmup phase the host system compiles the user space program together with the guest language interpreter using partial evaluation [6]. The interpreters feature *node rewriting*, i.e. the AST nodes are specialized to the most specific arguments encountered so far and are transitioned in-place to a more generic version if needed. Thus they provide a tailored implementation in the interpreter and type-feedback for partial evaluation. The functionality of node rewriting is the same as inline cache stubs in V8. However in their system the baseline interpreter implementation is the template for the optimized code and not the other way round. It is an open question if this design is able to yield similar performance as a custom single language compiler.

Chapter 7

Conclusions

In conclusion we can say that implementing compiler templates for a baseline compiler in the HIR of the optimizing compiler is a practical approach. In our case study we successfully replaced the assembly implementation of binary operations in the baseline compiler of V8 by an implementation in Hydrogen. Our work is extensively evaluated and used in production Chrome.

We were able to show that our approach does not depend on platform specific code anymore. Complexity is reduced, we implement the same functionality with 1/4th of the previous source code. We respect encapsulation by accessing values with Hydrogen instructions instead of manipulating them in unsafe assembly. This makes the code easier to maintain as assumptions about types and objects are localized and encapsulated.

Additionally we show that most code duplication between baseline and optimizing compiler can be avoided. The resulting unification is more robust as the source for diverging behavior is drastically reduced.

Meanwhile we verify that there are no performance regressions with our changes.

The approach depends on a feature complete and integrated compiler. The refactoring itself is non-trivial and uncovered existing bugs in the system. We therefore suggest when starting from scratch to pursue the approach from the beginning and first implement a compiler with small and simple instructions to generate templates.

List of Figures

2.1 Hidden classes are adaptive models for JS objects. Two different code paths (from top down) generate different maps for objects a and b. 4

2.2 Overview over the V8 compiler architecture. 5

2.3 Objects as defined in Figure 2.1 are layed out in memory according to the order of properties given by the hidden class hierarchy. 7

2.4 AST of the wingspan function as defined in Listing 2.1 line 4. . . 8

2.5 Value representations of Crankshaft. 11

3.1 State machine of the BinaryOpStub IC. The *String* state is only recorded for addition, since a string argument changes the semantics of the plus operator. 17

4.1 LOC in the src folder of V8 repository. Marked revisions are part of Hydrogenisation of binary operations. 21

4.2 Kraken JavaScript Benchmark version 1.1 from Mozilla, Octane 1.0 from Google and SunSpider 1.0.2 performance relative to trunk revision 17103 (higher is better). Results are averaged over 10 runs on a Intel Core i5 and 10 runs on a Intel Core 2 machine. Thin line segments represent trunk revisions unrelated to BinaryOpStubs. 23

4.3 Breakdown of Octane 1.0 scores (higher is better). Results averaged over 10 runs on a Intel Core i5 machine. Thin line segments represent trunk revisions unrelated to BinaryOpStubs. 24

5.1 The minor key of a BinaryOpStub from left to right: (i) left argument type according to Figure 3.1, (ii) boolean indicating whether the right argument was constant so far (only for mod), (iii) the constant right argument (only used if (ii) is true), (iv) the right argument type (only used if (ii) is false), (v) the result argument type, (vi) whether the stub was compiled for an SSE architecture, (vii) whether the left or right hand double box can be reused to store the result, (iix) the binary operation. 30

Bibliography

[1] Bowen Alpern, Mark. N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[2] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF — a dynamically-typed object-oriented language based on prototypes. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 49–70, October 1989.

[3] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`, 2013. Version from July 2013.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13 (4):451–490, 1991. ISSN 0164-0925. doi: 10.1145/115372.115320.

[5] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. `http://www.agner.org/optimize/`, 2013. Version from 2013-10-07.

[6] Yoshihiko Futamura. Partial evaluation of computation process: An approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, 1999. ISSN 1388-3690. doi: 10.1023/A:1010095604496.

[7] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. *SIGPLAN Not.*, 27(7):32–43, July 1992. ISSN 0362-1340. doi: 10.1145/143103.143114. URL `http://doi.acm.org/10.1145/143103.143114`.

[8] Ecma International. ECMAScript Language Specification, 5.1 Edition, 2012.

[9] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 111–120, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. doi: 10.1145/1064979.1064996.

[10] Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, January 1969. ISSN 0001-0782. doi: 10.1145/362835.362838. URL `http://doi.acm.org/10.1145/362835.362838`.

[11] marja@google.com. Crankshafting from the ground up. `https://docs.google.com/document/d/1hOaE7vbwdLLXWj3C8hTnnkpE0qSa2P--dtDvwXXEeD0/pub`, 2013. Accessed: 2014-01-06.

[12] Rei Odaira and Kei Hiraki. Sentinel pre: Hoisting beyond exception dependency with dynamic deoptimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 328–338, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. doi: 10.1109/CGO.2005.32. URL `http://dx.doi.org/10.1109/CGO.2005.32`.

[13] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21:895–913, sep 1999. ISSN 0164-0925. doi: 10.1145/330249.330250.

[14] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 104–118, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512961. URL `http://doi.acm.org/10.1145/512950.512961`.

[15] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. *SIGPLAN Not.*, 45(6):1–12, June 2010. ISSN 0362-1340. doi: 10.1145/1809028.1806598. URL `http://doi.acm.org/10.1145/1809028.1806598`.

[16] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73562. URL `http://doi.acm.org/10.1145/73560.73562`.

[17] Google V8. V8 Design Elements. `https://developers.google.com/v8/design`, 2014. Accessed: 2014-01-06.

[18] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4):

30:1–30:24, January 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400689. URL `http://doi.acm.org/10.1145/2400682.2400689`.

[19] Andy Wingo. v8: a tale of two compilers. `http://wingolog.org/archives/2011/07/05/v8-a-tale-of-two-compilers`, 2011. Accessed: 2014-01-06.

[20] Andy Wingo. a closer look at crankshaft, v8's optimizing compiler. `http://wingolog.org/archives/2011/08/02/a-closer-look-at-crankshaft-v8s-optimizing-compiler`, 2011. Accessed: 2014-01-06.

[21] Andy Wingo. inside full-codegen, v8's baseline compiler. `http://wingolog.org/archives/2013/04/18/inside-full-codegen-v8s-baseline-compiler`, 2013. Accessed: 2014-01-06.

[22] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384587. URL `http://doi.acm.org/10.1145/2384577.2384587`.

[23] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming &#38; Software*, Onward! '13, pages 187–204, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509581. URL `http://doi.acm.org/10.1145/2509578.2509581`.