MuscalietJS: An Extensible Portable Layered JavaScript Engine

Behnam Robatmiil Calin Cascaval
Madhukar N. Kedlaya Mehrdad Reshadi
Michael Weber Seth Fowler
Vrajesh Bhavsar Ben Hardekopf

mcjs@cs.ucsb.edu

Contents

1 Building and Command line Arguments **3**
- 1.1 Building MCJS . . . 3
  - 1.1.1 Linux Builds . . . 3
  - 1.1.2 Windows Builds . . . 3
- 1.2 Command line Arguments . . . 4

2 MuscalietJS Architectural Overview **6**

3 Dynamic Runtime Components **9**
- 3.1 Value Representation . . . 9
- 3.2 Object Layout and Maps . . . 11
  - 3.2.1 Slow Path Lookup . . . 12
  - 3.2.2 Fast Path Lookup . . . 14

4 Parsing and Function Metadata **16**
- 4.1 Parsing and IR Generation . . . 16
- 4.2 Adaptive Function-level Execution . . . 18

5 Type Analysis **20**
- 5.1 High-Level Overview . . . 20
  - 5.1.1 Function Signatures . . . 21
  - 5.1.2 Phase **First TI**: Type Inference → Type Feedback . . . 22
  - 5.1.3 Phase **Second TI**: Type Feedback → Type Inference . . . 23
- 5.2 MuscalietJS Type Inference . . . 24
  - 5.2.1 **Parse** Phase: Inserting Guard Nodes . . . 24
  - 5.2.2 **First TI** Phase: Initial Type Inference . . . 25
  - 5.2.3 **Profile** Phase . . . 28
  - 5.2.4 **Second TI** Phase: Final Type Inference . . . 29
  - 5.2.5 **Compile** Phase: Specialized Code Generation . . . 31

6 CIL Code Generation for Hot Functions **35**
6.1 Operation Implementation 35
6.2 CIL Code Generation 36

7 Design Space of JavaScript Engines **41**
7.0.1 Traditional Monolithic Architectures 42
7.0.2 Layered Architectures 42

Chapter 1

Building and Command line Arguments

1.1 Building MCJS

After cloning the git repo for MCJS, change directory to the root MCJavascript folder. For the rest of the commands here we assume you are in the root directory.

1.1.1 Linux Builds

You must have *mono version 3.0* or higher already installed and in your *PATH*. You can use *xbuild* or *make* to build. The following command builds *./MCJavascript/bin/Release/MCJavascript.exe*:

```
make
```

or

```
make JS_BUILD_TYPE=Release
```

The following command builds *./MCJavascript/bin/Debug/MCJavascript.exe*:

```
make JS_BUILD_TYPE=Debug
```

1.1.2 Windows Builds

You can use *Visual Studio* or *Xamarin Studio* to open and build *MCJavascript*. In the root MCJavascript directory open *MCJavascript-VS.sln* solution and build.

1.2 Command line Arguments

Table 1.1: Important commandline arguments

Parameter	Shorthand	Description
interpreter	i	enable/disable interpreter (default is +) if not enabled it always jit the code discussed in Chapter 4.
jit	j	enable/disable JIT (default is +) if not enabled it always interprets the code CIL code generator discussed in Chapter 6.
profiler	pr	enable/disable profiler (default is -) discussed in Chapter 5.
type-inference	ti	enable/disable type inference (default is +) the original type inference discussed in Chapter 5.
gurad-elimination	ge	enable/disable Guard Elimination (default is +) discussed in Chapter 5.
speculative-jit	sj	enable/disable speculative JIT (default is -) full two-phase TI discussed in Chapter 5.
inline-cache	ic	enable/disable inline property cache (default is +)
recursive-interpreter	ri	enable/disable recursive interpreter (default is +) if not enable, an **experimental** inline cache based interpreter will be enabled.
deoptimization	dz	enable/disable deoptimization (default is -) This **experimental** feature must be enabled only when ri is disabled

Table 1.1 lists important command line arguments used by MCJS. To see a more complete lists, you can pass –help to MCJavascript.exe in ./MCJavascript/bin/Release or ./MCJavascript/bin/Debug.

Assuming running with *mono* on *Linux* the following command runs *test.js* just using interpreter (in Windows you can run ./MCJavascript/bin/Release/MCJavascript.exe directly using .NET):

```
mono ./MCJavascript/bin/Release/MCJavascript.exe -j- test.js
```

Assuming running with *mono* on *Linux* the following command runs *test.js* just using jit:

```
mono ./MCJavascript/bin/Release/MCJavascript.exe -i- test.js
```

The following command runs *test.js* with jit and type inference:

```
mono ./MCJavascript/bin/Release/MCJavascript.exe -i- –ti+ test.js
```

The following command runs *test.js* with jit, type inference and full specialization support (two phase type inference and speculative guards):

```
mono ./MCJavascript/bin/Release/MCJavascript.exe -i- –ti+ –pr+ –sj+ –ic+ test.js
```

Finally, the following command runs *test.js* with jit, type inference and full specialization and deoptimization support:

```
mono ./MCJavascript/bin/Release/MCJavascript.exe -i- –ti+ –pr+ –sj+ –ic+ –ri- –dz+ test.js
```

Chapter 2

MuscalietJS Architectural Overview

MuscalietJS is a layered architecture, shown in Figure 2.1. It splits responsibilities across two levels: a JavaScript-specific engine and a language-agnostic low-level VM. In principle, any managed language VM can serve as the low-level engine. Our current implementation uses the Common Language Runtime (CLR), as implemented by Mono [1]. The low-level VM provides traditional compiler optimizations: instruction scheduling, register allocation, constant propagation, common subexpression elimination, etc., as well as code generation and machine specific optimizations. In addition it provides managed language services such as garbage collection, allowing us to focus on the JavaScript-specifc aspects of the engine.

The JavaScript specific layer is further decomposed into several components:

- **JavaScript runtime.** The high-level runtime consists of a parser, an interpreter, a parallel JIT compiler, and a profiler. The parser takes in JavaScript code and produces a custom IR (see Sec. 4.1). The interpreter executes the IR directly, while the JIT compiler applies JavaScript-specific transformations and optimizations, and generates CIL bytecodes for hot functions. Some examples of performed optimizations are: type analysis and type inference, array analysis, and signature-based specialization; these are further discussed in Section 4.

  The combination of interpreter, JIT, and profiler provides adaptability in our design by deciding which compilation path is more appropriate given the workload. For latency-sensitive scenarios, like browser page load, we provide an interpreter which can directly execute the output of the parser without any intermediate bytecode generation. As functions are invoked multiple times and become hot, the JIT compiler

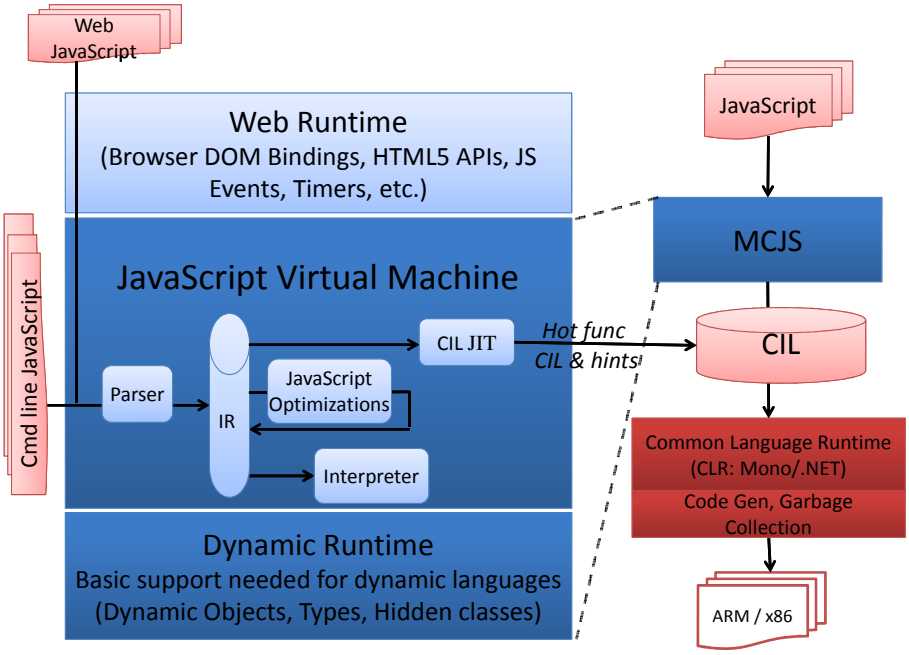

Figure 2.1: MuscalietJS Architecture

will optimize them since the compilation time can be amortized. For the hottest, most performance-sensitive code, we apply more expensive optimizations at both the high-level JavaScript engine layer and the low-level VM layer.

Source code location: *./MCJavascriptRuntime*

- **Dynamic runtime.** The dynamic runtime provides the necessary support to enable compilation for dynamic, prototype-based languages. This includes dynamic values, objects, types, and hidden classes.

Source code location: *./MCDynamicRuntime*

- **Web runtime.** The web runtime handles the integration with the browser. MuscalietJS was designed in concert with a browser architecture to optimize the bindings between the browser and the engine. The web runtime understands the semantics of the DOM and implements DOM bindings as well as other browser-related services like events and timers.

Source code location: *./MCWebRuntime*

Running the JavaScript engine inside another VM has performance implications. Our split design relies on JavaScript specific optimizations at the high-level to help mitigate the overhead of running on the CLR. The JavaScript engine code generator exploits advanced high-level techniques combined with type analysis and special hints to lead the low-level engine to generate high-quality optimized code. There are performance advantages to running on top of the CLR.

Chapter 3

Dynamic Runtime Components

We first discuss our representation of dynamic values and objects in the MuscalietJS Dynamic Runtime (DR), since several features involving performance and extensibility in this engine depend on these internal data structures. The next section presents the details of our JavaScript Runtime (JSR) and discusses how the input JavaScript program is processed, represented internally, optimized, JITed, and executed. We will also discuss how different JSR components make use of mechanisms provided by the DR.

3.1 Value Representation

Most dynamic languages are type-less, the type of variables becoming known only runtime. Therefore, an efficient representation of runtime type information and values is one of the fundamental features of the runtimes for these languages. LuaJIT, JavaScriptCore, and Mozilla's JavaScript engines currently all favor variations of *NaN-boxing*.[1] In contrast, V8 [2] uses a traditional *pointer tagging* scheme [3], in which the low bits of a value are used to distinguish integer and object types, at the expense of restricting the range of integers (Figure 3.1). Extracting the full type information in such runtimes may require multiple dependent memory accesses. For example, all the heap objects in V8 have *maps*. The actual type of a heap object is determined by comparing the map of the object against a set of predefined maps.

The tagging scheme employed by MuscalietJS is different: it uses *fat values*, i.e., an object reference consists of multiple machine words. The value type is directly encoded in the lower bits of the value. If the type is object or string, the rest of the value is a pointer to that object. Otherwise,

[1] encoding type information and values as IEEE-754 double-precision floating-point values.

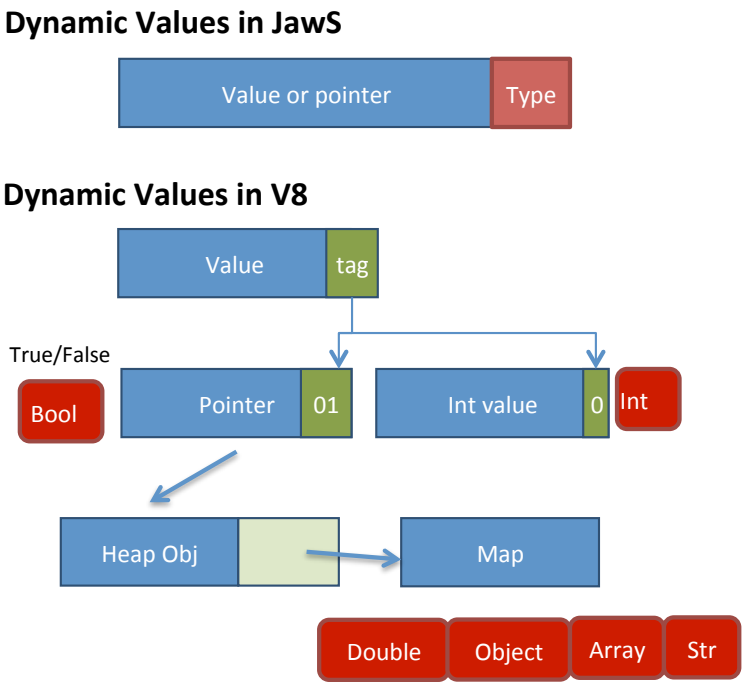

Figure 3.1: Dynamic values in MuscalietJS and V8.

the rest of value is the actual integer, boolean or double value. This design not only simplifies type lookups and comparisons, but also stores most value types inline, which reduces memory pressure on the garbage collector by reducing indirections and generating far less objects. However, it comes at the expense of having to copy fat values. Our choice here was motivated by memory subsystem traits of current mobile devices.

The fast and direct integration of types in dynamic values is exploited by MuscalietJS to implement a low overhead signature based function specialization and is explained in Section 4. This dynamic object model also makes dynamic boxing and unboxing fast in the generated code.

Source code location: *./MCDynamicRuntime/DValue.cs*

3.2 Object Layout and Maps

As in any dynamic runtime, the way objects and their fields are stored in memory affect the performance and efficiency of the memory allocation and object and field accesses [4]. In monolithic engines such as V8 [2], object layouts are complex and tightly coupled to the garbage collector, imposing a huge burden on extending the engine with new object types and features. For JavaScript, engines also have an internal data structure representing the implicit class of each object at a given point during execution, known as a *hidden class* or *map*. The hidden class of an object can change at any point given the dynamic nature of the language. This affects the way optimizing compilers generate efficient JITed code, and the JIT must take this layout into account in order to provide efficient property lookups using low-level optimizations such as inline caches. In MuscalietJS, both JavaScript objects and their hidden classes are normal CIL objects. Adding new features is easy, since it requires no translation between layers. This extensibility does not come at the cost of speed; the MuscalietJS representation is capable of fast property lookups using techniques like multi-level caching and property propagation. The same lookup mechanism is used directly by the interpreter and the JIT engine in the JSR (Section 4).

MuscalietJS employs a data structure for representing hidden classes that is customized for prototype-based languages. In such languages, objects act as multi-level dictionary, where each level adds or replaces properties and

inherits the rest from another object, its *prototype*. Since an object may both be a prototype and have a prototype of its own, a *prototype chain* is formed. The prototype chain is highly dynamic in nature because any of the objects forming the chain may be changed at any time. To address the challenge of inferring hidden classes from these chains of objects at runtime, we use a data structure with a tree of *property descriptors* at each level of inheritance or *prototype depth*. Each node in the tree represents an *owned* property – that is, a property which is set directly on the object rather than being inherited. Each property descriptor includes the property name and the offset at which the corresponding field is stored in the objects that share this prototype. Every path from a node to the root of the tree represents the fields of an object in the order they were added. Each node thus corresponds to a particular hidden class, shared by all objects whose fields were constructed in the same manner. This data structure makes it easy to keep the inferred hidden class of an object up to date as properties are added at runtime. The root of the tree has no property info. Therefore, the root node alone represents any object with no properties of its own. Note that such an object may still inherit properties from its prototype chain.

All of the property descriptors in a tree share a *map metadata* structure that includes a reference to the prototype object, a list of properties inherited through the prototype, and a cache for recently accessed property descriptors. Figure 3.2 shows an example of the two-level map data structure used by MuscalietJS for managing dynamic objects and properties. In this particular example, the prototype has one field called *"h"*. Objects *Obj1* and *Obj2* inherit *"h"* from that prototype and add { *"x", "y"* } and {*"x", "z"*}, respectively as sets of owned fields stored directly on each object.

3.2.1 Slow Path Lookup

The default process for looking up a property name in an object (e.g. *obj1.x* in Figure 3.2) includes walking the tree of property descriptors, starting at the node corresponding to the hidden class (map) of the object and continuing towards the root of the tree. If a property descriptor with the desired property name is found, the value is retrieved from the field at the corresponding offset in the object. Otherwise, the search is continued on the next object in the prototype chain. For instance, looking up *obj1.h* in Figure 3.2 will cause the property descriptors to be walked starting from *Obj1* until the root property descriptor is reached. Since the property won't be found, this will be followed recursively by a walk of *ProtoObj*'s property descriptors

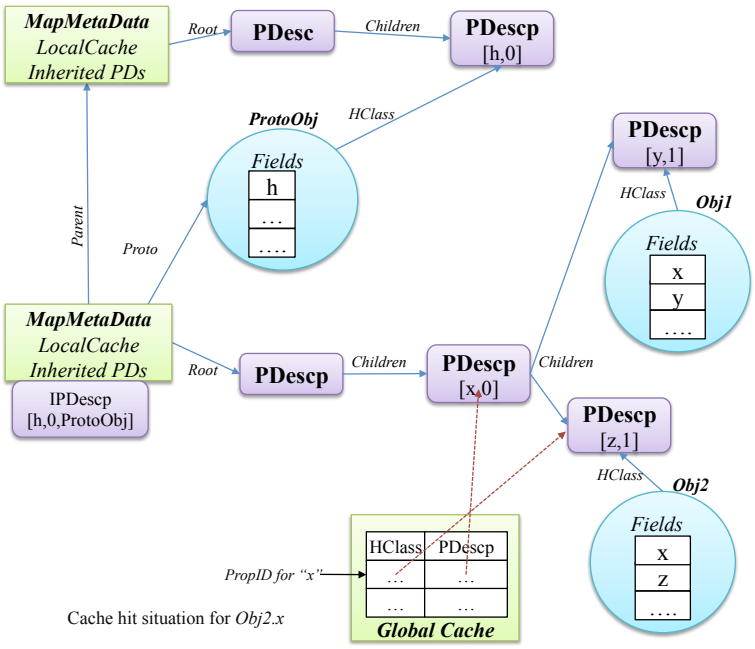

Figure 3.2: Property description data structure for representing hidden classes (maps) in MuscalietJS DR.

(since *ProtoObj* is the prototype of *Obj1*), eventually reaching the property descriptor for *"h"*. To reduce the cost of future lookups, when a lookup on an object results in an inherited property being found, an *inherited property descriptor* is added to the map metadata structure at the object's prototype depth. The inherited property descriptor indicates the prototype object that owns the property and the offset of the corresponding field in the prototype object.

3.2.2 Fast Path Lookup

MuscalietJS employs two levels of caching to speed up the property lookup process: a global cache shared between all hidden classes and objects and a cache at the map metadata level. For faster lookups in these caches, the MuscalietJS runtime globally assigns an integer ID (called a *propID*) to each statically known string property name and manages the propIDs during code generation and use these IDs to access these caches for known property names.

The global cache is a small array, indexed by the low-order bits of the propID, that contains tuples composed of a hidden class (identified by a node in the tree of property descriptors) and the last property descriptor retrieved for that propID. If the looked up hidden class matches the hidden class of the object on which we are performing the lookup, then the property descriptor is still correct and can be used to access the corresponding value. A global cache hit for *Obj2.x* (or any other object with the same hidden class accessing property *"x"*) is shown in Figure 3.2.

If there is no hit in the global cache, we check a second-level cache associated with the map metadata of that object (for example the cache in *Obj1.HClass.MapMetadata* for Obj1 in the example). This map metadata local cache similarly maps a property name to the last property descriptor matched for that name and the hidden class the match was for. While slower, this cache achieves better locality than the global cache given that it is limited to the hidden classes associated with a particular map metadata.

If the propID cannot be obtained for a property statically, the generated code bypasses the cache and uses the runtime property name as a string through the slow lookup path. Using these compile time assigned propIDs, the caching overhead is significantly reduced.

Some engines, such as V8 [2], use transient maps for representing hidden

classes. In this model, maps contain all of the fields of the class; changes in the prototype chain require affected maps to be rebuilt. While our hidden class model does not suffer from this problem, it does require a tree walk for uncached lookups. However, it is amenable to property caching at the runtime level.

Source code location: *./MCDynamicRuntime/DObject.cs, DPropertyMap.cs, DPropertyDescription.cs, PropertyCache.cs*

Chapter 4

Parsing and Function Metadata

This section discusses JavaScript-specific optimizations applied to the input JavaScript code in the JavaScript runtime component (JSR).

4.1 Parsing and IR Generation

For JavaScript function, the MuscalietJS engine uses a graph-based intermediate representation (IR) that describes JavaScript code, using simple operations that are easy to analyze – for example, all types of loops at the JavaScript level are represented using a single construct at the IR level. Our IR describes the flow of data through operations by placing edges in the graph between expressions that generate values (called *Writers* in our IR) and expressions that use those values (*Users*). It also represents implicit operations like type conversions explicitly so that they can be taken into account during later phases. This simplifies the implementation of analyses like type inference. The interpreter, optimization passes, and code generator all operate using this IR.

We depart from common practice in our approach when constructing the IR. Rather than building a temporary abstract syntax tree (AST) which is then used to build the IR in a separate pass, in our design the parser generates the IR directly. Because of the structure of the IR, constructing it has the effect of performing some of our analyses up front. Dataflow analysis, for example, is performed completely at parse time by connecting Writers and Users as they are constructed. We also build a symbol table for each function and determine useful metadata like whether the function uses *eval* or closes over its environment. This has significant advantages in the context of a latency-sensitive system like a web browser: it reduces the number of passes that are necessary, improves locality (since multiple passes are fused),

and eliminates the overhead of constructing temporary AST nodes that will later be discarded. The disadvantage of this approach is the complexity involved in implementing the semantic actions of the parser. This problem is approached by separating the parser's view of the IR from that of the *IR factory* which constructs the IR nodes. From the parser's perspective, we maintain the illusion, whenever possible, that we are building an AST that follows the structure of the ECMAScript 5 [5] grammar very closely. This is done by tagging the real IR node classes with empty interfaces corresponding to nonterminals in the grammar. These interfaces have no effect at runtime, but they provide static type-level constraints that make it easier to ensure the parser behaves correctly when constructing the IR graph. They also hide the complexity of the underlying graph nodes by presenting the parser with a simple, uniform interface. The IR factory, meanwhile, works with the concrete IR node types, and does not need to concern itself with the structure of the grammar at all.

Since there is such an impedance mismatch between the grammar and the IR, we cannot always hide the truth from the parser perfectly. However, by obeying two requirements when designing the IR and the factory code that constructs it, we were able to isolate these issues to a few small parts in the parser. One requirement was that the IR could be constructed in a recursive fashion, matching the parser's algorithmic structure. This was fundamentally needed to allow the construction of the IR without using a separate pass. We also needed to allow the parser to backtrack in certain circumstances. To support this, we required that the IR factory act only on local state stored within the IR nodes themselves, so that IR subgraphs could be thrown away without corrupting global data structures. The combination of these constraints also has another advantage: the designs of the parser and the IR factory lend themselves nicely to parallelism. We leave further exploration of that approach for future work. Once the IR for a parsed function is created, it can be used by the interpreter to run the function or by the CIL JIT engine to apply type optimizations and eventually generate optimized code.

Source code for the parser: *./MCJavascriptRuntime/JSParser/SequentialParser.cs*

Source code for IR: *./MCJavascriptRuntime/IR*

4.2 Adaptive Function-level Execution

To achieve adaptability and performance, traditional JavaScript engines support different modes of execution (interpretation, basic JIT and advanced JIT with specialization) at function granularity. For example, V8 first quickly generates an unoptimized version of the code for each function. After a certain number of executions a trampoline code section replaces the code pointer with a runtime function that generates a heavily optimized version of the code. The code pointer is then changed to point to the optimized code. It remains in this state unless the assumptions made during optimization prove false, in which case a *deoptimization* happens which restores the unoptimized code.

Providing this type of dynamic code adjustment (especially across the runtime function and JITed code) requires access to stack frames, control over stack semantics, and sometimes support for self-modifying code. This is hard to achieve in a layered design given the restrictions imposed by host VMs like the CLR. We therefore take a different approach to achieve the same degree of adaptability. Each JavaScript function object in MuscalietJS has a *codePtr*: a function pointer (C# delegate) that takes as its only argument a *CallFrame* object. The CallFrame includes a reference to the function object for that function as well as the actual input arguments and their types. A codePtr can point to different runtime functions that manage the execution and optimization of the function, or to different JITed specializations of the function. Each JavaScript function object also stores *function metadata*, which includes the current compilation state of the function (parse, analyze, JIT, specialized JIT), the IR graph of the function, and a code cache to store the function's JITed CIL code. The *codePtr* of a JavaScript function can point to one of the following functions:

- **FirstExecute (runtime function):** When the JavaScript function is called by the JavaScript code for the first time, this function performs pre-JIT analysis and the initialization of the JIT code cache for that function. At the end, the codePtr is changed to point to another runtime function responsible for normal execution.
- **NormalExecute (runtime):** Depending on the status of the JavaScript

function, the normal execution function either calls the interpreter (for "cold" functions) or JITs the function at one of several optimization settings.

- **Interpret:** The interpreter executes the function by traversing its IR graph and calling runtime functions that implement operations, property lookup, and other JavaScript semantics.
- **CIL JITed code:** For hot functions, normal execution starts the JIT to generate a specialized version of the function for the current arguments' types. The generated code will be added to the code cache and *codePtr* will be updated to point to it.

Tight coupling between the target and host runtime allows low overhead switching between interpreter and JIT code similar to traditional monolithic runtimes.

Source code location: *./MCJavascriptRuntime/JSFunctionMetadata.cs*

Source code for the interpreter: *./MCJavascriptRuntime/CodeGen/Interpreter.cs*

Chapter 5

Type Analysis

5.1 High-Level Overview

In this section, we provide a high-level, language-agnostic description of our proposed ideas. In the next section, we will make the discussion concrete for a specific language (JavaScript) and language implementation (MuscalietJS). We first discuss augmenting type feedback with function signatures to aid the effectiveness of type inference. We then discuss using type inference to aid the performance of type feedback.

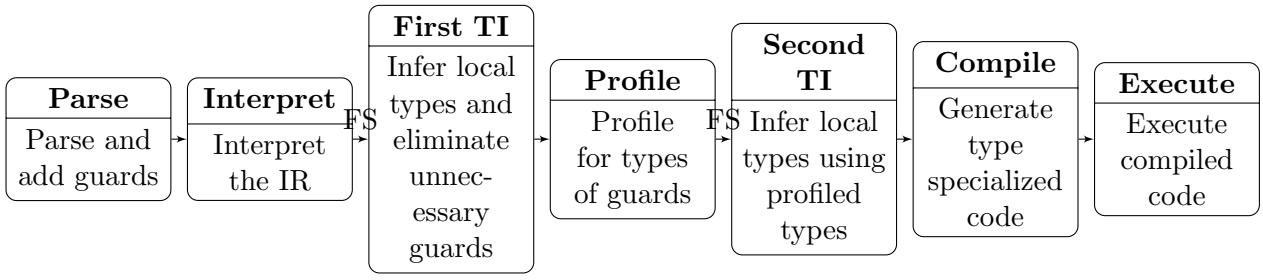

Figure 5.1: A flow graph describing the execution phases of a function. FS stands for function signatures; TI stands for type inference.

Figure 5.1 describes the general workflow of the language runtime system when executing a function, indicating the places where our proposed methods fit in. Our first type inference pass (**First TI**) takes as input the function's signature, i.e., the types of the function arguments for this specific invocation. Different signatures for the same function are handled independently from each other. The phase **First TI** uses the function signature in combination with the standard techniques for type inference. The results

are used to place type profiling hooks in the code, and in the phase **Profile** those hooks are used to collect type information that is specific to a given function signature (i.e., the type profiling information is collected and stored separately for each signature of a given function). The phase **Second TI** takes the original function signature along with the collected type profile information and performs a second, more aggressive type inference based on the new information. Finally, the result is used to specialize and optimize the code for further execution.

5.1.1 Function Signatures

Typically, type inference algorithms use the syntactic structure of a function, combined with certain semantic rules of the language, to deduce type information—for example, the result of a left-shift operation is guaranteed to be an integer. However, in a dynamic language there are many operations that do not give any clues about types. For example, the '+' operator is polymorphic and provides no information to the type inference algorithm. We can improve the available information by providing types for the function's arguments. This idea is inspired by existing schemes for specialized function dispatch based on type signatures, such as multimethods[6, 7]. Our innovation is to make the function signatures an additional input to the type inference algorithm.

Figure 5.2 provides a motivating example. Signature-based dispatch operates as follows: After `foo` becomes hot, during the first call to `foo` a type-specialized version of `foo`'s body is created and then specialized to handle arguments of signature `(int, int)`. During the second call to `foo`, another version of `foo`'s body is created that is specialized to handle arguments of type `(string, string)`. Finally, before the third execution of `foo` the runtime determines that there is a match between the current call's signature and a previously-seen signature. It then re-uses the specialized body for `(int, int)` as the target of the call.

We extend this idea to use the function type signature as an input to the type inference algorithm. Since the function dispatch mechanism ensures that the type signatures are always enforced (i.e., specialized code will never be called with the wrong types), we can rely on these signatures as always being correct and specialize the code for those types without requiring the type checks or recovery code that is necessary for normal type feedback mechanisms.

```
func foo(a, b)
{
  var c = a + b;
  var d = global + c + bar()
}
..............
foo(1, 3);
foo("bob", "alice");
foo(2, 5);
```

Figure 5.2: Motivating example

5.1.2 Phase First TI: Type Inference → Type Feedback

The goal of type feedback is to provide hints to the runtime and the JIT compiler about the types of variables. To do so, the runtime instruments the function's code with profile hooks that record type information observed during execution. These hooks are placed syntactically during the phase **Parse**, and show up as *guard nodes* in the abstract syntax tree (or IR) anywhere that type information may end up being useful (for example, on either side of a binary operator like '+'). See Figure 5.3 for an example of guard node placement for the function in Figure 5.2.

Type profiling can end up being quite expensive, and so reducing the number of guard nodes to be profiled can significantly improve performance of the profiling phase. Our key observation is that if type inference can already statically determine the type of an expression, then it is unnecessary to profile that expression. The phase **First TI** therefore uses the function signature and standard type inference rules to try and statically infer types for as many of the guard nodes as possible. Any guard node that is successfully typed is marked so that no profiling will be performed on that node during the phase **Profile**.

Of course, there will be many nodes that cannot have their types inferred (or inferring their types is only possible through very complex analysis), such as most object property accesses, untyped array indexing, and function call results. These guard nodes are left unmarked and will be profiled during the profiling phase; the resulting information will feed back into the second type inference pass as described in the next subsection.

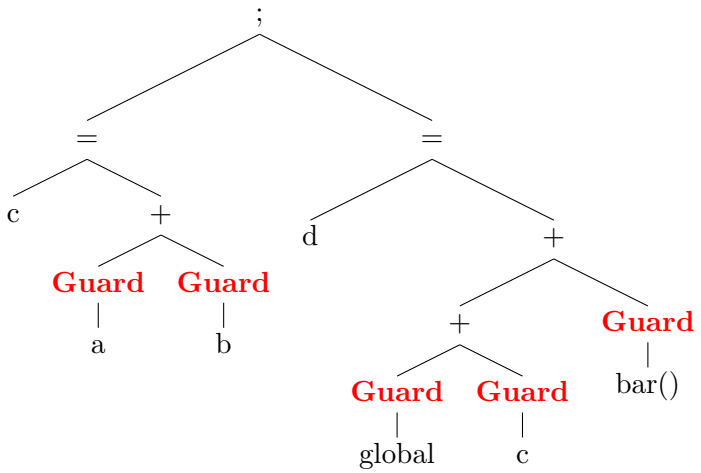

Figure 5.3: Intermediate representation of `foo` before type inference.

For Figure 5.3, suppose that this function is called with a signature `(int, int)`. The first type inference pass is then able to infer types for the variables `a`, `b`, and `c`. Therefore, the guards around those variables are no longer useful and do not need to be profiled. Figure 5.4 shows the same function with eliminated guard nodes shown in green.

5.1.3 Phase Second TI: Type Feedback → Type Inference

In the last phase before code generation the runtime uses the type information generated by type feedback to perform a second, more aggressive type inference pass. This pass is identical to the first TI pass except that guard nodes have been annotated with type information supplied by type profiling, and the type inference algorithm uses those annotations instead of attempting to infer the types of the expressions under the guard nodes. This phase is similar to the existing work by Hackett et al [8] except that once again the type information is augmented by the function signature.

In Figure 5.2, suppose that type feedback shows that `global` is always of type `int` and `bar()` always returns a value of type `double`. The second type inference algorithm takes these types into consideration during type inference and thus can infer that variable `d` is type `double`. Since this assumption can be invalidated at any point in the future, the code generator places a type check to enforce the validity of the type feedback information. Consequently, almost all variables and expressions are type inferred at the end and only two guards are placed in this specialization of the function.

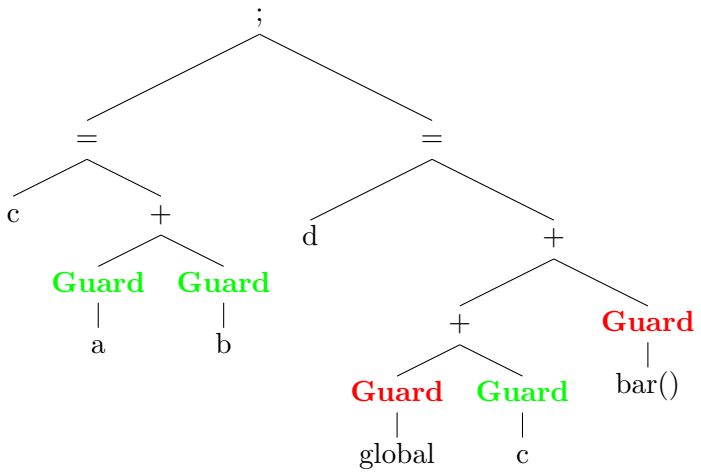

Figure 5.4: Intermediate representation of `foo` after type inference. Statically type inferred guards are in green.

5.2 MuscalietJS Type Inference

In this section we describe a specific instantiation of our proposed ideas for the JavaScript language, using the research JavaScript engine MuscalietJS.

5.2.1 Parse Phase: Inserting Guard Nodes

Guard nodes are inserted into a function to indicate where the type profiler should gather type information. Rather than requiring the runtime to transform the code midstream to insert these guard nodes, we have the function parser in the phase **Parse** conservatively inserts guard nodes into the function's IR at every point that may have a dynamic type and may benefit from type feedback. During interpretation these guard nodes are no-ops; their only purpose is to provide a hook for type profiling.

Good candidates for type profiling include binary and unary operations, object property accesses, array element accesses, function calls, and the left-hand sides of assignments. Guard nodes are placed in all of these locations during parse time. However, recall that these are conservative placements—the initial type inference pass, described below, may statically infer types for some of these guarded expressions, in which case the associated guard nodes are marked so that the type profiler will ignore them. As an example, Figure 5.5 shows the function `binb2b64` from crypto-sha1.js. The **red** highlighting indicates the presence of guard nodes around the expressions.

```
function binb2b64(binarray)
{
  var tab = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
  var str = "";
  for(var i = 0; i < binarray.length * 4; i += 3)
  {
    var triplet = (((binarray[i >> 2] >> 8 * (3 -  i %4)) & 0xFF) << 16)
                | (((binarray[i+1 >> 2] >> 8 * (3 - (i+1)%4)) & 0xFF) << 8 )
                |  ((binarray[i+2 >> 2] >> 8 * (3 - (i+2)%4)) & 0xFF);
    for(var j = 0; j < 4; j++)
    {
      if(i * 8 + j * 6 > binarray.length * 32)
      {
        str += b64pad;
      } else
      {
        str += tab.charAt((triplet >> 6*(3-j)) & 0x3F);
      }
    }
  }
  return str;
}
```

Figure 5.5: `binb2b64` function from the crypto-sha1.js benchmark which is used to convert an array of big-endian words to a base-64 string.The **red** highlighting indicates the presence of Guard nodes around the expressions.

5.2.2 First TI Phase: Initial Type Inference

Once a function with a particular type signature is deemed hot by the runtime, it is marked as a candidate for further optimization. The first step is an initial type inference pass. This pass will infer as many types as possible using the function signature and the type inference algorithm described by Figures 5.6 and 5.7 and Algorithm 1.

Figure 5.6 shows the type lattice used by the type inference algorithm. The most precise type is $\perp$, indicating an uninitialized value. For objects, we distinguish between function, array, null, and non-null values. For numbers, we distinguish between character, integer, unsigned integer, and double val-

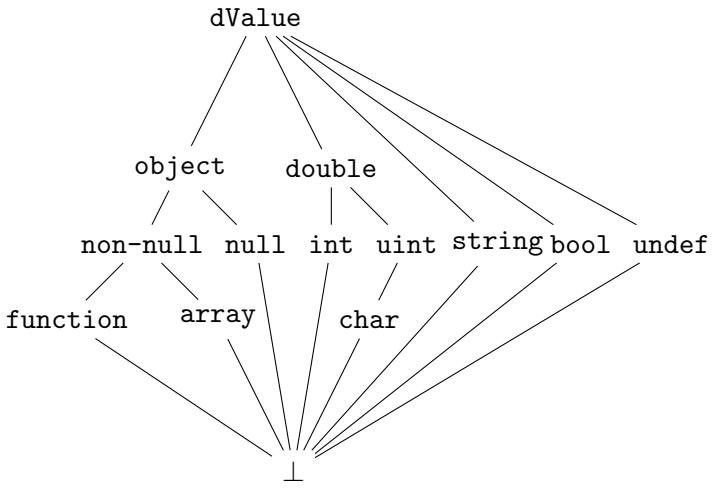

Figure 5.6: Type lattice used by our type inference algorithm.

ues. The least precise type is `dValue`, which stands for dynamic value—this is the default kind of value to use when the runtime has no static information about the value's type.

Algorithm 1 shows the initialization function for the type inference pass. The local variables are initialized to $\perp$, the parameter symbols are initialized to the types given by the function's type signature, and the global variables are initialized to `dValue` because there is no known information about the possible values of the global variables at this point in time. The algorithm then places the use sites of these symbols in the worklist.

The types of the expressions in the worklist are inferred using a set of typing rules, a selected subset of which are given in Figure 5.7. This subset shows some of the more important inference rules used in the algorithm. The INT and BOOL rules show how constants in the code can be used to type an expression. Rules LSHIFT and GT show how type-specific operations can be used to guide type inference. The ADD rule uses a helper function **typeResolve** to determine the type of the add operation. The **typeResolve** function takes into consideration the implicit conversion rules of JavaScript and returns the appropriate resultant type of the operation. The VARASSIGN rule generates type constraints. Once all the constraints are collected, they are solved to assign types to the local variables. Finally, the GUARD rules correspond to the guard nodes inserted by the parser. The rules check the

$$n \in \mathit{Num} \qquad b \in \mathit{Bool} \qquad x \in \mathit{Variable} \qquad e \in \mathit{Exp}$$

$$\begin{aligned}\tau \in \mathit{Type} = \{&\texttt{dValue}, \texttt{object}, \texttt{double}, \texttt{non-null}, \texttt{function}, \texttt{array},\\ &\texttt{null}, \texttt{int}, \texttt{uint}, \texttt{char}, \texttt{string}, \texttt{bool}, \texttt{undef}, \bot\}\\ \Gamma \in \mathit{Env} = \mathit{Variable} &\to \mathit{Type}\end{aligned}$$

$$\Gamma \vdash n : \texttt{int} \qquad \text{(INT)}$$

$$\Gamma \vdash b : \texttt{bool} \qquad \text{(BOOL)}$$

$$\Gamma \vdash e_1 \ll e_2 : \texttt{int} \qquad \text{(LSHIFT)}$$

$$\Gamma \vdash e_1 > e_2 : \texttt{bool} \qquad \text{(GT)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \sqsubseteq \Gamma(x)}{\Gamma \vdash x := e : \tau} \qquad \text{(VARASSIGN)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau = \textbf{typeResolve}(\tau_1, \tau_2)}{\Gamma \vdash e_1 + e_2 : \tau} \qquad \text{(ADD)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \sqsubset \texttt{dValue}}{\Gamma \vdash \text{guard } e : \tau} \qquad \text{(GUARD 1)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau = \texttt{dValue}}{\Gamma \vdash \text{guard } e : \textbf{profile}} \qquad \text{(GUARD 2)}$$

Figure 5.7: Selected inference rules used in our type inference algorithm to generate type constraints. Algorithms 1 and 2 conflate the type constraint generation and constraint solving—in the algorithms, the VARASSIGN rule not only generates the type constraint, but also updates Γ with the resultant type and pushes the *users* of the variable x into the worklist. **typeResolve** is a helper function that takes into consideration the implicit conversion rules of JavaScript and returns the appropriate resultant type of the operation.

Algorithm 1 `TypeInference(`$\mathcal{S}$, $\mathcal{FS}$`)`, where $\mathcal{S}$ gives the symbols in scope and $\mathcal{FS}$ gives the function's type signature.

```
worklist = []
for s in S do
    switch s.SymbolType do
        case local
            Γ(s) = ⊥
            worklist.add(s.users)
        case parameter
            Γ(s) = τ from lookup(FS, s)
            worklist.add(s.users)
        case global
            Γ(s) = dValue
end for
while worklist.length ≠ 0 do
    e = worklist.pop()
    typeEval(e)          ▷ Uses the rules from Figure 5.7 to infer types
end while
```

type of the expression it encloses; if the expression evaluates to `dValue` then the guard is marked to be profiled by the **Profile** phase, otherwise that guard node will be ignored by the **Profile** phase. This rule eliminates many unnecessary guard nodes, significantly increasing the profiler's performance.

As an example, Figure 5.8 shows the function from Figure 5.5 after the initial type inference with the inferred types and the eliminated guard nodes.

5.2.3 Profile Phase

The type profiling phase collects type information at the guard nodes inserted by the parser and marked by the type inference phase 3 as worth profiling. The type information collected by this phase is specific to a particular function *and* function signature. There is only a limited opportunity for profiling the code before it is JITed, therefore we chose to use exhaustive profiling rather than a sampling approach (though this configuration can be modified to use sampling if desired). We employ several heuristics to help minimize the profiling overhead:

- Disable profiling of IR nodes that are highly dynamic in nature: The

profiler stops tracking the IR nodes that show highly dynamic nature, such as rapidly changing type information. We observe this behavior in some code snippets which iterate over the fields of an object. For such guard nodes, the profiler records the profiled type as `dValue` and stops profiling them.

- Efficient data structures: While designing the profiler we observed that the performance of the profiler depends heavily on the data structure that is used. In particular, we use an array-based implementation of the profiler which significantly outperforms a dictionary-based implementation.

- Selectively enabling the profiler: We observe that many functions execute only once during the initialization phase of the JavaScript application. Therefore, we enable the profiler only during the sixth invocation of the function code. By doing this we ensure that we only collect profiles for functions that are potentially hot.

5.2.4 Second TI Phase: Final Type Inference

Once sufficient profile information is collected, the second pass type inference algorithm is performed in **Second TI** phase. In this pass the runtime tries to type the local variables that were not type inferred during the first pass, by using the collected profile information.

Algorithm 2 describes the initialization function of the second pass. This differs from the first pass because we reuse the types inferred by the first pass while initializing the types of the variables in this pass. We check whether the type of a variable is precise enough, i.e., if the type inferred in the first pass is in the set $\mathit{PreciseTypes} = \{$`function`, `array`, `null`, `bool`, `char`, `int`, `undef`, `string`$\}$. If it is, the algorithm initializes the variable to that type, otherwise the algorithm initializes the type of the variable to $\perp$ and adds its users to the worklist. This helps the algorithm converge to a fixpoint faster and avoid inferring types of variables that have already been typed.

The type inference algorithm uses the same inference rules as in Figure 5.7 except for the GUARD rule. The new GUARD rule is:

$$\frac{\tau = \mathcal{P}(\ell)}{\Gamma \vdash \text{guard}^{\ell} e : \tau} \qquad \text{(GUARD)}$$

Algorithm 2 TypeInference($\mathcal{S}$, $\mathcal{FS}$, Γ_1), where $\mathcal{S}$ gives the symbols in scope, $\mathcal{FS}$ gives the function's type signature and Γ_1 is the type environment from initial type inference.

```
worklist = []
for s in S do
    switch s.SymbolType do
        case local
            if Γ1(s) ∈ PreciseTypes then
                Γ(s) = Γ1(s)
            else
                Γ(s) = ⊥
                worklist.add(s.users)
            end if
        case parameter
            if Γ1(s) ∈ PreciseTypes then
                Γ(s) = Γ1(s)
            else
                Γ(s) = τ from lookup(FS, s)
                worklist.add(s.users)
            end if
        case global
            Γ(s) = dValue
end for
while worklist.length ≠ 0 do
    e = worklist.pop()
    typeEval(e)          ▷ Uses the rules from Figure 5.7 to infer types
end while
```

where $\mathcal{P}$ is a function that maps unique labels ℓ associated with guard nodes to the profiled type information. This new rule shows how the profiled type information is used to infer the type of the marked guard nodes. The guards that were not marked (i.e., were not used to gather type information during profiling) are treated as no-ops during this **Second TI** phase.

As an example, in Figure 5.9 we see that `str` is now type inferred to be a string based on the observed types of guards around the `b64pad` and `tab.charAt()` expressions.

5.2.5 Compile Phase: Specialized Code Generation

In this section we discuss the techniques used in generating type specialized Common Intermediate Language (CIL) code in MuscalietJS. After the second type inference pass, the runtime passes the intermediate representation (IR) of the code and the type environment Γ to the specialized code generator. The code generator maps primitive types such as `int`, `bool`, `double`, `char`, and `uint` to native CIL primitives. This ensures that the operations on them can be applied natively and are therefore faster.

After generating code for the expression enclosed in a tagged guard node, a check is added in the code to compare the observed type at execution time with the profiled type. The types inferred in the second pass are valid as long as the checks hold. If the observed type during the execution doesn't match the type for which the code was specialized, the runtime bails out and calls a deoptimization routine. The deoptimization routine captures the current state of the value stack and current values of the variables and reconstructs a new callframe. Once this is done, the execution shifts to the interpreter, which executes the function using the new callframe. This operation is expensive and must be avoided as much as possible. Therefore, capturing accurate profiles is very important.

In the case of the example in Figure 5.9, since `str` is now type inferred as a `string`, the code generator does not add checks around it. With `str` being a local variable, we know its type is only influenced by the observed types of `b64pad` and `tab.charAt()`. Since we already have runtime checks around them, it is unnecessary to check for the type of `str` as well. This small optimization enables the runtime to reduce the number of unnecessary checks in the code. The total number of checks in the final CIL code for `binb2b64` is reduced from nine to seven.

Source code location: *./MCJavascriptRuntime/CodeGen/TypeInferer.cs, TypeCalculator.cs, Types.cs, Profiler.cs*

```
function binb2b64(binarray⟨array⟩)
{
  var tab⟨string⟩ = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
  var str⟨dValue⟩ = "";
  for(var i⟨int⟩ = 0; i < binarray.length * 4; i += 3)
  {
    var triplet⟨int⟩ = (((binarray[i >> 2] >> 8 * (3 - i %4)) & 0xFF) << 16)
                     | (((binarray[i+1 >> 2] >> 8 * (3 - (i+1)%4)) & 0xFF) << 8 )
                     |  ((binarray[i+2 >> 2] >> 8 * (3 - (i+2)%4)) & 0xFF);
    for(var j⟨int⟩ = 0; j < 4; j++)
    {
      if(i * 8 + j * 6 > binarray.length * 32)
      {
        str += b64pad;
      } else
      {
        str += tab.charAt((triplet >> 6*(3-j)) & 0x3F);
      }
    }
  }
  return str;
}
```

Figure 5.8: `binb2b64` function after the first type inference pass. The **red** highlighting indicates the presence of guard nodes around the expressions that need to be profiled. The **green** nodes indicate that the guard nodes around these expressions are unnecessary and should not be profiled. The ⟨**type**⟩ indicates the type inferred by the type inference algorithm.

```
function binb2b64(binarray⟨array⟩)
{
  var tab⟨string⟩ = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
  var str⟨string⟩ = "";
  for(var i⟨int⟩ = 0; i < binarray.length⟨int⟩ * 4; i += 3)
  {
    var triplet⟨int⟩ = (((binarray[i >> 2]⟨int⟩ >> 8 * (3 - i %4)) & 0xFF) << 16)
                     | (((binarray[i+1 >> 2]⟨int⟩ >> 8 * (3 - (i+1)%4)) & 0xFF) << 8 )
                     |  ((binarray[i+2 >> 2]⟨int⟩ >> 8 * (3 - (i+2)%4)) & 0xFF);
    for(var j⟨int⟩ = 0; j < 4; j++)
    {
      if(i * 8 + j * 6 > binarray.length⟨int⟩ * 32)
      {
        str⟨string⟩ += b64pad⟨string⟩;
      } else
      {
        str⟨string⟩ += tab.charAt((triplet >> 6*(3-j)) & 0x3F)⟨string⟩;
      }
    }
  }
  return str;
}
```

Figure 5.9: `binb2b64` function after the first type inference pass. The **red** highlighting indicates the presence of Guard nodes which were profiled and ⟨**type**⟩ indicates the type profiled by the profiler. The ⟨**type**⟩ indicates the type inferred by the type inference algorithm after the first pass. The ⟨**type**⟩ indicates the type inferred by the type inference algorithm after second pass.

Chapter 6

CIL Code Generation for Hot Functions

In JavaScript, most operations are dynamically defined depending on the types of their operands. For example, a property load operation such as *object.a* can turn into a direct field access in *object*, a field access in *object*'s prototype chain, a call to a getter function call, or a call back to a browser function. A binary addition can translate to integer or double addition, a string concatenation or a complicated user-defined conversion on operands followed by a numerical addition or a concatenation, depending on the value types of the operands.

Given the heavy use of the dynamic operations in JavaScript (and most other dynamic languages), generating efficient code based on the inferred or profiled types is very important when JITing code for hot functions. With the large number of variants for each of these operations, producing low-level code for all of the variants, as is done in traditional engines, is challenging and hard to debug. MuscalietJS exploits the rich features of the host engine such as reflection and special code-generation hints to JIT efficient code while maintaining scalability and debuggability. This section discusses how MuscalietJS performs code generation for operations in hot functions using these techniques. It also discusses the special hints MuscalietJS passes to the host runtime (Mono or .NET) for guiding low-level code generation.

6.1 Operation Implementation

Initially, we generated the CIL bytecode implementing each operation manually; there were many variations depending on the operand type information extracted during type inference. However, this approach is very tedious and hard to maintain because the relationships between the input and output

types of the operations and their corresponding implementations were separated and implicitly captured in multiple places, including the type inference and JIT algorithms. Debugging was also very difficult.

To address these problems, we created an operation database. This is essentially a series of overloaded functions, each implementing one particular instance of an operation's behavior. During type inference, we use reflection to map operand types (function argument types) to operation result types (function return value types). During JIT we again use reflection to look up the appropriate operation implementation function based on the types of the operand and simply generate a call to that function. Figure 6.1 (B) shows a simplified code for binary addition operator for three of the possible many cases including *(Int, Int)*, *(Undefined, Bool)* and *(Bool, Float)* in the MuscalietJS operation code database.

This approach produces very robust, maintainable, and easy to debug code. Since we use reflection to take advantage of the overloading functionality of the host language, we do not even need to generate lookup code. By assigning proper CIL attributes to these operator implementation functions, the runtime is forced to inline them, and the compiled code achieves the same performance as the previous manually-generated CIL bytecode implementation. We developed a template-based code generator to generate most of the C# code for the various possible implementations of each operation, and used the "partial classes" feature of the C# language to manually implement the corner cases and integrate them with the rest of the auto-generated code.

Source code location: *./MCJavascriptRuntime/Operations*

6.2 CIL Code Generation

The MuscalietJS code generator (JIT engine) traverses the IR of the function being JITed and uses information added to the IR during type inference or other pre-JIT phases to generate efficient CIL code. This section discusses some of the techniques used by the engine and how it interacts with the low-level VM to achieve high performance.

The engine generates CIL for JavaScript code using the CLR's reflection API. The various JavaScript-level expressions are implemented by generating calls to appropriate operation functions specialized for the inferred types of their operands as shown in Figure 6.1. To show how our code generator uses

```
/**** (A) CodeGen for binary operations ***********/
Visit(BinaryExp node, OpCache operation)
{
  var ltype = Visit(node.Left);
  var rypet = Visit(node.Right);
  /* Using reflection to extract the expected type and call*/
  var methodInfo = operation.Get(ltype, rtype);
  var resultType = operation.ReturnType(ltype, rtype);
  GenerateCall(methodInfo, resultType);
}

/**** (B) Addition operation ***********/
public static class Add
{
  ...
  [MethodImplAttribute(MethodImplOptions.AggressiveInlining)]
  public static float Run(bool i0, float i1)
  { return Run((float)Convert.ToNumber.Run(i0), i1); }
  ...
  [MethodImplAttribute(MethodImplOptions.AggressiveInlining)]
  public static double Run(mdr.DUndefined i0, bool i1)
  { return double.NaN; }
  ...
  [MethodImplAttribute(MethodImplOptions.AggressiveInlining)]
  public static int Run(int i0, int i1) { return i0 + i1); }
  ...
}
```

Figure 6.1: CodeGeneration (JIT) code for binary operation IR nodes (A) and *Addition* operation implementation (B).

```
/********* (A) JavaScript Source ***********/
function doublePlusOne(a)
{
    return 2 * a + 1;
}
doublePlusOne(20)
.............
/*** (B) Generated CIL code for doublePlusOne ***/
.............
0.0 ldarg.0 ;loading call frame
1.1 ldflda ;loading arg0 in callframe (a)
2.1 call Int32 DValue:AsInt32 () ;unboxing arg0
2.2 stloc.1
3.0 ldarg.0
4.0 ldflda
5.1 ldc.i4 2
5.2 ldloc.1
5.3 call Int32 Binary.Mul:Run (Int32, Int32)
6.1 ldc.i4 1
6.2 call Int32 Binary.Add:Run (Int32, Int32)
7.0 call Void DValue:Set (Int32) ;boxing for return
.............
/*** (C) x86 assembly code generated by mono ***/
.............
0 addl $0x10,%esp
1 leal 0x34(%edi),%eax ;load arg0 (a)
2 movl 0x04(%eax),%ecx
3 movl %ecx,0xf4(%ebp)
4 leal 0x0c(%edi),%eax
5 shll %ecx ;inlined/optimized integer multiply
6 incl %ecx ;inlined/optimized integer addition
7 movl %ecx,0x04(%eax) ; boxing for return
8 movl $0x00000009,(%eax)
9 leal 0xfc(%ebp),%esp
```

Figure 6.2: JITed CIL and assembly for a sample code.

operand types and hints to the low-level VM to help the underlying platform generate efficient code, Figure 6.2 illustrates: (A) a sample JavaScript function, (B) its corresponding CIL code generated by MuscalietJS and, (C) the x86 code generated by the low-level engine. The call to the `doublePlusOne` function shown in the figure passes an *Int* value (a) as argument, and the literals in the function are all integer constants. Therefore, the type inference pass infers the type of the operands to both the multiplication and addition operations to be *Int*. During JIT compilation, as shown in Figure 6.1, the JIT engine uses reflection to look up multiplication and addition operation functions in the runtime appropriate for *(Int, Int)* operands. The resulting functions, which are called in the JITed code (CIL lines 5.3 and 6.2), are the basic integer addition and multiply functions `Binary.Mul:Run` and `Binary.Add:Run`, which are implemented in terms of CIL-level primitive operations. Since these functions return *Int* as well, the intermediate value passing the multiplication result to the addition operation is inferred to be an *Int* as well, and so it does not generate any boxing or unboxing operations. All the operations in the function are pure integer operations except the unboxing of the function argument read from the call frame (CIL line 2.1) and the boxing for the return (CIL line 7.0).

When the low-level VM sees this specialized version of `doublePlusOne`, it is guaranteed to inline the calls to `Binary.Mul:Run`, `Binary.Add:Run`, and the runtime functions that handle boxing, because we annotate them with CLR attributes that require this behavior. After inlining, the calls are replaced with simple integer `+` and `*` operations. The low-level VM then applies other optimizations such as dead code elimination, register allocation, and constant folding on the resulting code. Finally, optimized code is generated for the target hardware architecture. For example, as shown in Figure 6.2, the x86 generated code implements the multiplication and addition operations in the original JavaScript function with a `shift left` instruction and an `increment` instruction.

MuscalietJS engine implements *hidden classes* [2] as a mechanism for fast property access in MuscalietJS. In traditional engines, *hidden classes* or *maps* are internal data structures representing the implicit class of each object at a given point during execution. MuscalietJS hidden classes and property access algorithms are similar to V8 but implemented in C# [2] so we do not discuss them. We implement property access expressions (the `.` operator) at the JavaScript level with calls to the property lookup runtime function that uses *property caches* exploiting these *hidden classes* to implement fast prop-

erty access. These calls are also inlined by the low-level VM. For indexing operations (the `[]` operator), MuscalietJS performs special optimizations: if the inferred type of the expression that is being indexed is *Array*, and the indexing expression's inferred type is *Int*, then we can skip the property lookup code and cache accesses. Instead, the engine generates a simple array access, using the integer to directly index the internal CIL-level array used to implement the array object.

In traditional JavaScript engines, calls to runtime functions from the JITed code is usually very costly, given that the runtime is unmanaged while the JITed code exists in a managed environment. The primary source of overhead is the difference in stack frame models between the two environments. In the MuscalietJS runtime, however, JavaScript objects, the runtime, calls to operations, boxing and unboxing, and JITed code all exist in the same managed world and so communication between them has essentially no overhead. As shown, MuscalietJS effectively exploits this low-overhead during CIL code generation for hot functions.

Source code location: *./MCJavascriptRuntime/CodeGen/CodeGenerator.cs, CodeGeneratorWithSpecialization.cs, CodeGeneratorWithInlineCaches.cs*

Chapter 7

Design Space of JavaScript Engines

Given the pace of change experienced in web development and the growing ubiquity of JavaScript, we identify the following criteria for evaluating JavaScript engines:

- **Performance.** Performance is the key criterion by which production JavaScript engines are evaluated. Given the increase in dynamic execution of web pages [9] and the trend toward web apps, better JavaScript performance in browsers is increasingly important.

- **Adaptability.** A JavaScript engine must be adaptable enough to recognize different workloads, ranging from server-side [10] to latency-sensitive page-load and iterative hot-spot workloads, and react accordingly to provide the best possible quality of service.

- **Extensibility.** The JavaScript language is constantly evolving both formally, (e.g., the new ECMAScript standard [5]), and informally [11, 12, 13]. This means that JavaScript engines must evolve constantly and must be designed such that extensions can be added and integrated easily, while taking advantage of the full feature set of optimizations. Changes must be localized and modular, and avoid re-engineering the engine flow.

- **Portability.** Browsers, and hence JavaScript, are widely used on many platforms, from desktops to mobile devices; thus an engine should be easily portable to different hardware architectures. This implies that engines should be decoupled from the underlying hardware, treating the hardware interface as a separate concern. However, it is still critical for

performance that an engine take advantage of any available hardware-specific features.

7.0.1 Traditional Monolithic Architectures

The traditional engines are designed to achieve the best performance and adapt to different Web workloads. For example, the Google V8 JavaScript engine [2] is a native C++ application that has two just-in-time (JIT) compilers and no interpreter. One is a quick, simple compiler that generates very fast generic code using inline caches, and the second is a profile-based optimizing JIT compiler called Crankshaft. The optimizing compiler applies a wide range of low-level optimizations such as SSA (Static Single Assignment) redundancy elimination, register allocation, and static type inference when generating native code. It also uses type feedback from the inline caches from the simple compiler. V8 does not use a bytecode-based IR (intermediate representation); instead, it uses both high-level and low-level graph-based IRs for different levels of optimization. SpiderMonkey [14], Firefox's JavaScript engine, is another monolithic engine written in C/C++ and comprised of a bytecode-based interpreter, a baseline JIT compiler, and an optimizing JIT compiler (IonMonkey). The baseline compiler is similar to the V8 engine's simple compiler and uses inline caches to generate fast code.

While these runtimes are usually implemented and compiled in a low-level unmanaged language such as C++, they need to provide complete JIT compilers with several IRs and optimizations at different levels and the assembly code generators for all targeted machines. The compiler stack and multiple assembly code generators must be updated frequently to support newly added language features. Significant development effort is required to extend and maintain such engines and port them to new platforms. Researchers have been experimenting with various JIT compilers in the Mozilla framework, such as the tracing JIT in Gal et al. [15], and highlight the complexity of implementing such components in monolithic engines. Additionally, in these architectures, there is an impedance mismatch between the runtime call stack (generally C++) and the call stack of the executing generated code in the target language (JavaScript). Consequently, communication between the executing JITed code and the runtime is complex and has high overhead.

Table 7.1: Traditional vs. Layered JavaScript Engines

Architecture	Name	Host Language	Compilation methodology	LOC
Monolithic	V8	C++	Adaptive: Inline-cache based fast compiler wit no IR for cold code; an JavaScript optimizing compiler (HIR and LIR) for hot code	1M
Monolithic	SpiderMonkey	C++	Adaptive: A byte-code based interpreter for cold code; a trace JIT and method JIT for hot JavaScript code	300K
Layered	MuscalietJS	CLR (C#)	Adaptive: an IR-based interpreter implemented in CIL for cold code; an JavaScript optimizing compiler (implemented in CIL) and a CIL code generator for hot code	95K
Layered	SPUR	CLR (C#)	Translate JavaScript to CIL; leaves dynamic optimizations to tracing CIL JIT	N/A
Layered	IronJS	DLR (F#)	Translate JavaScript to DLR; leaves dynamic optimizations to DLR	23K
Layered	Rhino	JVM (Java)	Non-adaptive but can be reconfigured ahead of time (AOT); supports various degrees of optimizations (interpreter to optimized code)	115K

7.0.2 Layered Architectures

In layered JavaScript (also called Repurposed JIT) architectures, the target dynamic runtime is built on top of another runtime engine, rather than building an entire compiler from scratch. These designs are typically more portable than monolithic ones, as hardware-specific optimizations and functionality are provided by the host runtime. Additionally, the target runtime may take advantage of features and resources provided by the host runtime including object layout, garbage collection and code generators. Therefore, these designs can better fit the quick development cycles needed for evolving dynamic languages. For example, as shown in Table 7.1, the code size for these architecture is significantly smaller (up to 10x) than the code size of traditional engines.

Table 7.1 compares MuscalietJS against three common layered JavaScript engines and two traditional engines. MuscalietJS is a layered JavaScript engine architecture built on top of the .NET Common Language Runtime (CLR). It performs adaptive JavaScript-specific optimizations and uses CIL code generation to JIT optimized versions of hot function after applying high-level JavaScript optimizations. These high-level optimizations include type analysis, profiling and restricted dataflow analysis. MuscalietJS also exploits special features in CLR runtimes to generate high-performance optimized code. For example, MuscalietJS uses reflection and inlining hints to generate optimized code for dynamic JavaScript operations. MuscalietJS communicates special hints to the CLR engine to avoid array-bounds checks and object initialization for JavaScript property access and JavaScript object

creation.

SPUR [16] is a tracing JIT compiler for Common Intermediate Language (CIL) in which JavaScript is directly compiled to CIL and CIL is trace-compiled by SPUR for better optimization. The authors show that tracing CIL generated by compiling JavaScript programs gives similar performance gains to a JavaScript tracing compiler. Due to the modular nature of our engine, we believe that it is possible for MuscalietJS to run on top of any CIL compiler including SPUR. Given the effectiveness of SPUR over Microsoft's .NET runtime, MuscalietJS might benefit from it; unfortunately, SPUR is not available for general use and so we could not test this.

IronJS [17] translates JavaScript to Dynamic Language Runtime (DLR) expression trees [18] and leaves dynamic optimizations to the DLR. DLR uses the concept of *dynamic callsites* to generate type specialized versions of each operation. This is similar to polymorphic inline caches.

Rhino [19] is a JavaScript engine on top of the JVM which offers various levels of optimization. The default configuration of the runtime invokes a naïve bytecode-based interpreter written in Java. Other optimization levels invoke a code generator that generates Java class files. When set to the highest optimization level, Rhino performs optimizations such as detection of numerical operations, common sub-expression elimination, and function call target caching. Though the current implementation lacks adaptive compilation, optimizations can be enabled or disabled ahead of time.

Nashorn [20] is a more recent implementation of a JavaScript runtime on the JVM. Nashorn uses the *invokedynamic* bytecode instruction, which was added to recent versions of the JVM to enable efficient implementation of dynamic language runtimes. We were not able to find a performant version of Nashorn to compare against.

Bibliography

[1] "Mono Project." `http://www.mono-project.com`.

[2] "Google Inc. V8 JavaScript virtual machine." `http://code.google.com/p/v8/`.

[3] D. Gudeman, "Representing Type Information in Dynamically Typed Languages," TR 93-27, Dept. of Computer Science, University of Arizona, October 1993.

[4] C. Chambers, J. Hennessy, and M. Linton, "The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages," tech. rep., Stanford University, Department of Computer Science, 1992.

[5] "ECMAScript." `http://www.ecmascript.org/`.

[6] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon, "Common lisp object system specification," *ACM Sigplan Notices*, vol. 23, no. SI, pp. 1–142, 1988.

[7] C. Chambers and G. T. Leavens, "Typechecking and modules for multimethods," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 6, pp. 805–843, 1995.

[8] B. Hackett and S.-y. Guo, "Fast and precise hybrid type inference for javascript," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 239–250, ACM, 2012.

[9] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "Jsmeter: comparing the behavior of javascript benchmarks with real web applications," in

USENIX Conference on Web Application Development, WebApps'10, pp. 3–3, 2010.

[10] "Node.js." http://nodejs.org.

[11] "Typescript." http://www.typescriptlang.org/.

[12] "ECMAScript web workers." http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html#workers.

[13] "ECMAScript typed arrays." http://www.khronos.org/registry/typedarray/specs/latest/.

[14] "SpiderMonkey: Mozilla's JavaScript engine." https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[15] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, "Trace-based just-in-time type specialization for dynamic languages," in *ACM conference on Programming language design and implementation*, (Dublin, Ireland), pp. 465–478, 2009.

[16] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter, "SPUR: a trace-based JIT compiler for CIL," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, (Reno/Tahoe, Nevada, USA), pp. 708–725, 2010.

[17] "IronJS." https://github.com/fholm/IronJS.

[18] "Dynamic Language Runtime." http://msdn.microsoft.com/en-us/library/dd233052.aspx.

[19] "Rhino JavaScript engine." https://developer.mozilla.org/en-US/docs/Rhino.

[20] "Nashorn JavaScript engine." http://openjdk.java.net/projects/nashorn.