Fast and Precise Hybrid Type Inference for JavaScript

Abstract

JavaScript performance is often bound by its dynamically typed nature. Compilers do not have access to static type information, making generation of efficient, type-specialized machine code difficult. To avoid incurring extra overhead on the programmer and to improve the performance of deployed JavaScript programs, we seek to solve this problem by inferring types. Existing type inference algorithms for JavaScript are often too computationally intensive and too imprecise—especially in the case of JavaScript’s extensible objects—to enable optimizations. Both problems arise from performing purely static analyses. In this paper we present a hybrid type inference algorithm for JavaScript based on points-to analysis. Our algorithm is fast, in that it pays for itself in the optimizations it enables. Our algorithm is also precise, generating information that closely reflects the program’s actual behavior, by augmenting static analysis with run-time type barriers.

We showcase an implementation for Mozilla Firefox’s JavaScript engine, demonstrating both performance gains and viability. Through integration with the just-in-time (JIT) compiler in Firefox, we have improved its performance on major benchmarks and JavaScript-heavy websites by up to 50%. This is scheduled to become the default compilation mode in Firefox 9.

1. The Need for Hybrid Analysis

Consider the example JavaScript program in Figure 1. This program constructs an array of Box objects wrapping integer values, then calls a use function which adds up the contents of all those Box objects. No types are specified for any of the variables or other values used in this program, in keeping with JavaScript’s dynamically-typed nature. Nevertheless, most operations in this program interact with type information, and knowledge of the involved types is needed to compile efficient code.

In particular, we are interested in the addition res + v on line 9. In JavaScript, addition coerces the operands into strings or numbers if necessary. String concatenation is performed for the former, and numeric addition for the latter. Without static information about the types of res and v, a JIT compiler must emit code to handle all possible combinations of operand types. Moreover, every time values are copied around, the compiler must emit code to keep track of the types of the involved values, using either a separate type tag for the value or a specialized marshaling format. This incurs a large runtime overhead on the generated code, greatly increases the complexity of the compiler, and makes effective implementation of important optimizations like register allocation and loop invariant code motion much harder.

If we knew the types of res and v, we can compile code which performs an integer addition without the need to check or to track the types of res and v. With static knowledge of all types involved in the program, the compiler can in many cases generate code similar to that produced for a statically-typed language such as Java, with similar optimizations.

We can infer possible types for res and v statically by reasoning about the effect the program’s assignments and operations have on values produced later. This is illustrated below (for brevity, we do not consider the possibility of Box and use being overwritten).

- On line 17, main passes an integer when constructing Box objects. On line 2, Box assigns its parameter to the result’s p property. Thus, Box objects can have an integer property p.
- Also on line 17, main assigns a Box object to an element of a. On line 15, a is assigned an array literal, so the elements of that literal could be Box objects.
- On line 18, main passes a to use, so a within use can refer to the array created line 15. When use accesses an element of a on line 8, per #2 the result can be a Box object.
- On line 8, property p of a at a[1] is assigned to v. Per #3 a[i] can be a Box object, and per #1 the p property can be an integer. Thus, v can be an integer.
- On line 6, res is assigned an integer. Since v can be an integer, res + v can be an integer. When that addition is assigned to res on line 9, the assigned type is consistent with the known possible types of res.

Figure 1. Motivating Example
This reasoning can be captured with inclusion constraints; we compute sets of possible types for each expression and model the flow between these sets as subset relationships. To compile correct output, we need to know not just some possible types for variables, but all possible types. In this sense, the static inference above is unsound: it does not account for all possible behaviors of the program. Few such behaviors are described below.

- The read of a[1] may access a hole in the array. Out of bounds array accesses in JavaScript produce the undefined value if the array’s prototype does not have a matching property. Such holes can also be in the middle of an array; assigning to just a[0] and a[2] leaves a missing value at a[1].

- Similarly, the read of a[1].v may be accessing a missing property and may produce the undefined value.

- The addition res + v may overflow. JavaScript has a single number type which does not distinguish between integers and doubles. However, it is extremely important for performance that JavaScript compilers distinguish the two and try to represent numbers as integers wherever possible. An addition of two integers may overflow and produce a number which can only be represented as a double.

In some cases these behaviors can be proven not to occur, but usually they cannot be ruled out. A standard solution is to capture these behaviors statically, but this is unfruitful. The static analysis must be sound, and to be sound in light of highly dynamic behaviors is to be conservative: many element or property accesses will be marked as possibly undefined, and many integer operations will be marked as possibly overflowing. The resulting type information would be too imprecise to be useful for optimization.

Our solution, and our key technical novelty, is to combine unsound static inference of the types of expressions and heap values with targeted dynamic type updates. Behaviors which are not accounted for statically must be caught dynamically, modifying inferred types to reflect those new behaviors if caught. If a[i] accesses a hole, the inferred types for the result must be marked as possibly undefined. If res + v overflows, the inferred types for the result must be marked as possibly a double.

With or without analysis, the generated code needs to test for array holes and integer overflow in order to correctly model the semantics of the language. We call dynamic type updates based on these events semantic triggers: they are placed on rarely taken execution paths and incur a cost to update the inferred types only if the first time that path is taken.

The presence of these triggers illustrates the key invariant our analysis preserves:

Inferred types must conservatively model all types for variables and object properties which currently exist and have existed in the past, but not those which could exist in the future.

This has important implications:

- The program can be analyzed incrementally, as code starts to execute. Code which does not execute need not be analyzed.
- This is necessary for JavaScript due to dynamic code loading and generation. It is also important for reducing analysis time on websites, which often load several megabytes of code and only execute a fraction of it.
- Assumptions about types made by the JIT compiler can be invalidated at almost any time. This affects the correctness of the JIT-compiled code, and the virtual machine must be able to recompile or discard code at any time, especially when that code is on the stack.

Dynamic checks and the key invariant are also critical to our handling of polymorphic code within a program. Suppose somewhere else in the program we have new Box("hello!"). Doing so will cause Box objects to be created which hold strings, illustrating the use of Box as a polymorphic structure. Our analysis does not distinguish Box objects created in different places, and the result of the a[i].v access in use will be regarded as potentially producing a string. Naively, solving the constraints produced by this analysis will mark a[i].v, v, res + v, and res as all producing either an integer or a string, even if use’s runtime behavior is actually monomorphic and only works on Box objects containing integers.

This problem of imprecision leaking across the program is serious: even if a program is mostly monomorphic, analysis precision can easily be poisoned by a small amount of polymorphic code.

We deal with uses of polymorphic structures and functions using runtime checks. At all element and property accesses, we keep track of both the set of types which could be observed for the access and the set of types which has been observed. The former will be a superset of the latter, and if the two are different then we insert a runtime check, a type barrier, to check for conformance between the resultant value and the observed type set. Mismatches lead to updates of the observed type set.

For the example program, a type barrier is required on the a[p].p access on line 8, and nowhere else. The barrier will test that the value being read is an integer. If a string shows up due to a call to use outside of main, then the possible types of the a[p].p access will be updated, and res and v will be marked as possibly strings by resolving the analysis constraints.

Type barriers differ from the semantic triggers described earlier in that the tests they perform are not required by the language and do not need to be performed if our analysis is not being used. We are effectively betting that the required barriers pay for themselves by enabling generation of better code using more precise type information. We have found this to be the case in practice (§4.1.1, §4.2.5).

### 1.1 Comparison with other techniques

The reader may question, “Why not use more sophisticated static analyses that produce more precise results?” Our choice for the static analysis to not distinguish Box objects created in different places is deliberate. To be useful in a JIT setting, the analysis must be fast, and the time and space used by the analysis quickly degrade as complexity increases. Moreover, there is a tremendous variety of polymorphic behavior seen in JavaScript code in the wild, and to retain precision even the most sophisticated static analysis would need to fall back to dynamic checks some of the time.

Interestingly, less sophisticated static analyses do not fare well either. Unification-based analyses undermine the utility of dynamic checks; precision is unrecoverable despite dynamic monitoring.

More dynamic compilation strategies generate type specialized code based on profiling information, without static knowledge of possible argument or heap types [9, 10]. Such techniques will determine the types of expressions with similar precision to our analysis, but will always require type checks on function arguments or when reading heap values. With knowledge of all possible types, we only need type checks at accesses with type barriers, a difference which significantly improves performance (§4.1.1).

We believe that our partitioning of static and dynamic analysis is a sweet spot for JIT compilation of a highly dynamic language. Our main technical contribution is a hybrid inference algorithm for the entirety of JavaScript, using inclusion constraints to unsoundly infer types extended with runtime semantic triggers to generate sound type information, as well as type barriers to efficiently and precisely handle polymorphic code. Our practical contributions include both an implementation of our algorithm and a realistic evaluation. The
In full JavaScript, we also have the primitive types \texttt{bool} and \texttt{null}.

The implementation is integrated with the JIT compiler used in Firefox and is of production quality. Our evaluation has various metrics showing the effectiveness of the analysis and modified compiler on benchmarks as well as popular websites, games, and demos.

The remainder of the paper is organized as follows. In §2 we describe the static and dynamic aspects of our analysis. In §3 we outline implementation of the analysis as well as integration with the JavaScript JIT compiler inside Firefox. In §4 we present empirical results. In §5 we discuss related work, and in §6 we conclude.

### 2. Analysis

We present our analysis in two parts, the static “may-have-type” analysis and the dynamic “must-have-type” analysis. The algorithm is based on Andersen-style (inclusion based) pointer analysis [6]. The static analysis is intentionally unsound with respect to the semantics of JavaScript. It does not account for all possible behaviors of expressions and statements and only generates constraints that model a “may-have-type” relation. All behaviors excluded by the type constraints must be detected at runtime and their effects on types in the program dynamically recorded. The analysis runs in the browser as functions to be executed: code is analyzed function-at-a-time.

Inclusion based pointer analysis has a worst-case complexity of \(O(n^3)\) and is very well studied. It has shown—and we reaffirm this with our evaluation—to perform and scale well despite its cubic worst-case complexity [22].

We describe constraint generation and checks required for a simplified core of JavaScript expressions and statements, shown in Figure 2. We let \(f, x\) range over variables, \(p\) range over property names, \(i\) range over integer literals, and \(s\) range over string literals. The only control flow in the core language is if, which tests for \(x\) definedness. We avoid talking about functions and function calls in our simplified core; the reader may think of functions as objects with special domain and codomain properties.

The types over which we are trying to infer are also shown in Figure 2. The types can be primitive or an object type. The \texttt{int} type indicates a number expressible as a signed 32-bit integer and is subsumed by number — \texttt{int} is added to all type sets containing string. Finally, we have sets of types which the static analysis computes.

#### 2.1 Object Types

To reason about the effects of property accesses, we need type information for JavaScript objects and their properties. Each object is immutably assigned an object type \(o\). When \(o \in T_o\) for some expression \(e\), then the possible values for \(e\) when it is executed include all objects with type \(o\).

For the sake of brevity and ease of exposition, our simplified JavaScript core only contains the ability to construct object-prototyped object literals via the \{\} syntax; two objects have the same type when they were allocated via the same literal.

In full JavaScript, types are assigned to objects according to their prototype: all objects with the same type have the same prototype. Additionally, objects with the same prototype have the same type, except for plain \texttt{Object} and \texttt{Function} objects. \texttt{Object} and \texttt{Array} objects have the same type if they were allocated at the same source location, and \texttt{Function} objects have the same type if they are closures for the same script. \texttt{Object} and \texttt{Function} objects which represent builtin objects such as class prototypes, the \texttt{Math} object and native functions are given unique types, to aid later optimizations (§2.4).

The type of an object is nominal: it is independent from the properties it has. Objects which are structurally identical may have different types, and objects with the same type may have different structures. This is crucial for efficient analysis. JavaScript allows addition or deletion of object properties at any time. Using structural typing would make an object’s type a flow-sensitive property, making precise inference harder to achieve.

Instead, for each object type we compute the possible properties which objects of that type can have and the possible types of those properties. These are denoted as type sets \(\text{prop}(o, p)\) and \(\text{index}(o)\). The set \(\text{prop}(o, p)\) captures the possible types of a non-integer property \(p\) for objects with type \(o\), while \(\text{index}(o)\) captures the possible types of all integer properties of all objects with type \(o\). These sets cover the types of both “own” properties (those directly held by the object) as well as properties inherited from the object’s prototype.

#### 2.2 Type Constraints

The static portion of our analysis generates constraints modeling the flow of types through the program. We assign to each expression \(e\):
a type set representing the set of types it may have at runtime. These constraints are unsound with respect to JavaScript semantics. Each constraint is augmented with triggers to fall in the remaining possible behaviors of the operation. For each rule, we informally describe the required triggers.

The grammar of constraints are shown in Figure 2. We have the standard subset constraint, $\subseteq$, and a barrier subset constraint, $\subseteq_X$. For two type sets $X$ and $Y$, $X \subseteq Y$ means that all types in $X$ are propagated to $Y$. On the other hand, $X \subseteq_X Y$ means that if $Y$ contains types that are not in $X$, then a type barrier is required which updates the types in $X$ according to values which are dynamically assigned to the location $X$ represents ($\subseteq$). The rules for the constraint generation functions, $\forall_c(e)$ for expressions (styled $\forall$) and $\forall_s(x)$ for statements (styled $\forall$), are shown in Figure 3. Staticaly analyzing a function takes the union of the results from applying $\forall_c$ to every statement in the method. The UNION, INT, STR, and OBJ rules for literals and the VAR rule for variables are straightforward.

The ADD rule is complex, as addition in JavaScript is similarly complex. It is defined for any combination of values, can perform either a numeric addition, string concatenation, or even function calls if either of its operands is an object (calling their valueOf or toString methods). Producing a number or string.

Additions in actual programs are typically used to add two numbers or concatenate a string with something else. We statically model these cases and use semantic triggers to monitor the results produced by other combinations of values, at little runtime cost.

Barriers could be used for other types of assignments, but we do not do so. Allowing barriers in new places is unlikely to significantly change the total number of required barriers — improving precision by adding barriers in one place can make barriers in another place unnecessary.

2.4 Supplemental Analyses

In many cases type information itself is insufficient to generate code which performs comparably to a statically-typed language such as Java. Statical triggers are generally cheap, but they nevertheless incur a cost. These checks should be eliminated in as many cases as possible.

Eliminating such checks requires more detailed analysis information. Rather than build additional complexity into the type analysis itself, we use supplemental analyses which leverage type information but do not modify the set of inferred types. We do several other supplemental analyses, but those described below are the most important.

Integer Overflow

In the execution of a JavaScript program, the overall cost of doing integer overflow checks is very small. On kernels which do many additions, however, the cost can become significant. We have measured overflow check overhead at 10-20% of total execution time on microbenchmarks.

Using type information, we normally know statically where integers are being added. We use two techniques on those sites to remove overflow checks. First, for simple additions in a loop (mainly loop counters) we try to use the loop termination condition to compute a range check which can be hoisted from the loop, a standard technique which can only be performed for JavaScript with type information available. Second, integer additions which are used as inputs to bitwise operators do not need overflow checks, as bitwise operators truncate their inputs to 32 bit integers.
definite then the access is on a map from property names to slots in an array of values. If a property
JavaScript objects are internally laid out as a Definite Properties to out-of-order writes, possibly invalidating JIT code.
object as possibly not packed. If an object type has never been packed elements in ascending order, with no gaps; we call these arrays
Packed Arrays are usually constructed by writing to their

We have implemented this analysis for SpiderMonkey, the Java-
Script engine in Firefox. We have also modified the engine’s JIT
compiler, JaegerMonkey, to use inferred type information when

### 3. Implementation

We have implemented this analysis for SpiderMonkey, the Java-
Script engine in Firefox. We have also modified the engine’s JIT
compiler, JaegerMonkey, to use inferred type information when
generating code. Without type information, JaegerMonkey gener-
ates code in a fairly mechanical translation from the original Spi-
der analysis consume even more. We reconcile this conflict by
observing how browsers are used in practice: to surf the web. The
web page being viewed, content being generated, and JavaScript
executing code being run are constantly changing. The compiler and analysis
need to not only quickly adapt to new scripts that are running, but
also to quickly discard regenerate data associated with old scripts
that are no longer running much, even if the old scripts are still
reachable and not subject to garbage collection.

We do this with a simple trick: on every garbage collection, we
throw away all JIT code and as much analysis information as pos-
sible. All inferred types are functionally determined from a small
core of type information: type sets for the properties of objects,
function arguments, the observed type sets associated with bar-
rier constraints and the semantic triggers which have been tripped.
All type constraints and all other type sets are discarded, notably
the type sets describing the intermediate expressions in a function
without barriers on them. This constitutes the great majority of
the memory allocated for analysis. Should the involved functions warm
back up and require recompilation, they will be reanalyzed. In com-
bination with the retained type information, the complete analysis
state for the function is then recovered.

In Firefox, garbage collections typically happen every several
seconds. If the user is quickly changing pages or tabs, unused JIT
code is marked for recompilation. We discard the JIT

### 3.1 Recompilation

As described in §1, computed type information can change as a result of
runtime checks, newly analyzed code or other dynamic behavior. For compiled code to rely on this type information, we
must be able to recompile the code in response to changes in types
while that code is still running.

As each script is compiled, we keep track of all type information
queried by the compiler. Afterwards, the dependencies are encoded
and attached to the relevant type sets, and if those type sets change
in the future the script is marked for recompilation. We represent
the contents of type sets explicitly and eagerly resolve constraints,
so that new types immediately trigger recompilation with little
overhead.

When a script is marked for recompilation, we discard the JIT
code for the script, and resume execution in the interpreter. We do
not compile scripts until after a certain number of calls or loop back
edges are taken, and these counters are reset whenever discarding
JIT code. Once the script warms back up, it will be recompiled
using the new type information in the same manner as its initial
compilation.

### 3.2 Memory Management

Two major goals of JIT compilation in a web browser stand in stark
counter to one another: generate code that is as fast as possible,
and use as little memory as possible. JIT code can consume a large
amount of memory, and the type sets and constraints computed
by our analysis consume even more. We reconcile this conflict by
observing how browsers are used in practice: to surf the web. The
web page being viewed, content being generated, and JavaScript
code being run are constantly changing. The compiler and analysis
need to not only quickly adapt to new scripts that are running, but
also to quickly discard regenerate data associated with old scripts
that are no longer running much, even if the old scripts are still
reachable and not subject to garbage collection.

We do this with a simple trick: on every garbage collection, we
throw away all JIT code and as much analysis information as pos-
sible. All inferred types are functionally determined from a small
core of type information: type sets for the properties of objects,
function arguments, the observed type sets associated with bar-
rier constraints and the semantic triggers which have been tripped.
All type constraints and all other type sets are discarded, notably
the type sets describing the intermediate expressions in a function
without barriers on them. This constitutes the great majority of
the memory allocated for analysis. Should the involved functions warm
back up and require recompilation, they will be reanalyzed. In com-
bination with the retained type information, the complete analysis
state for the function is then recovered.

In Firefox, garbage collections typically happen every several
seconds. If the user is quickly changing pages or tabs, unused JIT
code and attached to the relevant type sets, and if those type sets change
in the future the script is marked for recompilation. We represent
the contents of type sets explicitly and eagerly resolve constraints,
so that new types immediately trigger recompilation with little
overhead.

When a script is marked for recompilation, we discard the JIT
code for the script, and resume execution in the interpreter. We do
not compile scripts until after a certain number of calls or loop back
edges are taken, and these counters are reset whenever discarding
JIT code. Once the script warms back up, it will be recompiled
using the new type information in the same manner as its initial
compilation.

### 3.2 Memory Management

Two major goals of JIT compilation in a web browser stand in stark
counter to one another: generate code that is as fast as possible,
and use as little memory as possible. JIT code can consume a large
amount of memory, and the type sets and constraints computed
by our analysis consume even more. We reconcile this conflict by
observing how browsers are used in practice: to surf the web. The
web page being viewed, content being generated, and JavaScript
code being run are constantly changing. The compiler and analysis
need to not only quickly adapt to new scripts that are running, but
also to quickly discard regenerate data associated with old scripts
that are no longer running much, even if the old scripts are still
reachable and not subject to garbage collection.

We do this with a simple trick: on every garbage collection, we
throw away all JIT code and as much analysis information as pos-
sible. All inferred types are functionally determined from a small
core of type information: type sets for the properties of objects,
function arguments, the observed type sets associated with bar-
rier constraints and the semantic triggers which have been tripped.
All type constraints and all other type sets are discarded, notably
the type sets describing the intermediate expressions in a function
without barriers on them. This constitutes the great majority of
the memory allocated for analysis. Should the involved functions warm
back up and require recompilation, they will be reanalyzed. In com-
bination with the retained type information, the complete analysis
state for the function is then recovered.

In Firefox, garbage collections typically happen every several
seconds. If the user is quickly changing pages or tabs, unused JIT
code and attached to the relevant type sets, and if those type sets change
in the future the script is marked for recompilation. We represent
the contents of type sets explicitly and eagerly resolve constraints,
so that new types immediately trigger recompilation with little
overhead.

When a script is marked for recompilation, we discard the JIT
code for the script, and resume execution in the interpreter. We do
not compile scripts until after a certain number of calls or loop back
edges are taken, and these counters are reset whenever discarding
JIT code. Once the script warms back up, it will be recompiled
using the new type information in the same manner as its initial
compilation.

### 3.2 Memory Management

Two major goals of JIT compilation in a web browser stand in stark
counter to one another: generate code that is as fast as possible,
and use as little memory as possible. JIT code can consume a large
amount of memory, and the type sets and constraints computed
by our analysis consume even more. We reconcile this conflict by
observing how browsers are used in practice: to surf the web. The
web page being viewed, content being generated, and JavaScript
code being run are constantly changing. The compiler and analysis
need to not only quickly adapt to new scripts that are running, but
also to quickly discard regenerate data associated with old scripts
that are no longer running much, even if the old scripts are still
reachable and not subject to garbage collection.

We do this with a simple trick: on every garbage collection, we
throw away all JIT code and as much analysis information as pos-
sible. All inferred types are functionally determined from a small
core of type information: type sets for the properties of objects,
function arguments, the observed type sets associated with bar-
rier constraints and the semantic triggers which have been tripped.
All type constraints and all other type sets are discarded, notably
the type sets describing the intermediate expressions in a function
without barriers on them. This constitutes the great majority of
the memory allocated for analysis. Should the involved functions warm
back up and require recompilation, they will be reanalyzed. In com-
bination with the retained type information, the complete analysis
state for the function is then recovered.

In Firefox, garbage collections typically happen every several
seconds. If the user is quickly changing pages or tabs, unused JIT
code and attached to the relevant type sets, and if those type sets change
in the future the script is marked for recompilation. We represent
the contents of type sets explicitly and eagerly resolve constraints,
so that new types immediately trigger recompilation with little
overhead.

When a script is marked for recompilation, we discard the JIT
code for the script, and resume execution in the interpreter. We do
not compile scripts until after a certain number of calls or loop back
edges are taken, and these counters are reset whenever discarding
JIT code. Once the script warms back up, it will be recompiled
using the new type information in the same manner as its initial
compilation.

### 3.2 Memory Management

Two major goals of JIT compilation in a web browser stand in stark
counter to one another: generate code that is as fast as possible,
and use as little memory as possible. JIT code can consume a large
amount of memory, and the type sets and constraints computed
by our analysis consume even more. We reconcile this conflict by
observing how browsers are used in practice: to surf the web. The
web page being viewed, content being generated, and JavaScript
code being run are constantly changing. The compiler and analysis
need to not only quickly adapt to new scripts that are running, but
also to quickly discard regenerate data associated with old scripts
that are no longer running much, even if the old scripts are still
reachable and not subject to garbage collection.

We do this with a simple trick: on every garbage collection, we
throw away all JIT code and as much analysis information as pos-
sible. All inferred types are functionally determined from a small
core of type information: type sets for the properties of objects,
function arguments, the observed type sets associated with bar-
rier constraints and the semantic triggers which have been tripped.
All type constraints and all other type sets are discarded, notably
the type sets describing the intermediate expressions in a function
without barriers on them. This constitutes the great majority of
the memory allocated for analysis. Should the involved functions warm
back up and require recompilation, they will be reanalyzed. In com-
bination with the retained type information, the complete analysis
state for the function is then recovered.

In Firefox, garbage collections typically happen every several
seconds. If the user is quickly changing pages or tabs, unused JIT
code and attached to the relevant type sets, and if those type sets change
in the future the script is marked for recompilation. We represent
the contents of type sets explicitly and eagerly resolve constraints,
so that new types immediately trigger recompilation with little
overhead.

When a script is marked for recompilation, we discard the JIT
code for the script, and resume execution in the interpreter. We do
not compile scripts until after a certain number of calls or loop back
edges are taken, and these counters are reset whenever discarding
JIT code. Once the script warms back up, it will be recompiled
using the new type information in the same manner as its initial
compilation.
4.1 Benchmark Performance

As described in §3, we have integrated our analysis into the Jaegermonkey JIT compiler used in Firefox. We compare performance of the compiler both without the analysis (JM) and with the analysis (JM+TI). JM+TI adds several major optimizations to JM, and requires additional compilations due to dynamic type changes (§3.1). Figure 4 shows the effect of these changes on the popular SunSpider JavaScript benchmark.

The compilation sections of Figure 4 show the total amount of time spent compiling and the total number of script compilations for both versions of the compiler. For JM+TI, compilation time also includes time spent generating and solving type constraints, which is small: 4ms for the entire benchmark. JM performs 146 compilations, while JM+TI performs 224, an increase of 78. The total compilation time for JM+TI is 2.52 times that of JM, an increase of 579 ms, due a combination of recompilations, type analysis and the extra complexity of the added optimizations.

Despite the significant extra compilation cost, the type-based optimizations performed by JM+TI quickly pay for themselves. The ×1 and ×20 sections of Figure 4 show the running times of the two versions of the compiler and generated code on the benchmark run once and modified to run twenty times, respectively. In the single run case JM+TI is a 6.3% improvement over JM. One run of SunSpider completes in less than 250ms, which makes it difficult to get an optimization to pay for itself on this benchmark.

Figures 5 and 6 compare the performance of JM and JM+TI on two other popular benchmarks, the V8 and Kraken suites. These suites run for several seconds each, far longer than SunSpider, and show a larger speedup. V8 scores (which are given as a rate, rather than a raw time; larger is better) improve by 50%, and Kraken scores improve by a factor of 2.69.

Across the benchmarks, not all tests improved equally, and some regressed over the engine’s performance without the analysis. These include the date-format-xparb and string-tagcloud tests in SunSpider, and the RayTrace and RegExp tests in the V8. These are tests which spend little time in JIT code, and perform many side effects in VM code itself. Changes to objects which happen in the VM due to, e.g., the behavior of built-in functions must be tracked to ensure the correctness of type information for the heap. We are working to reduce the overhead incurred by such side effects.

4.1.1 Performance Cost of Barriers

The cost of using type barriers is of crucial importance for two reasons. First, if barriers are very expensive then the effectiveness of the compiler on websites which require many barriers (§4.2.2) is greatly reduced. Second, if barriers are very cheap then the time and memory spent tracking the types of heap values would be unnecessary.

To estimate this cost, we modified the compiler to artificially introduce barriers at every indexed and property access, as if the types of all values in the heap were unknown. For benchmarks, this is a great increase above the baseline barrier frequency (§4.2.2). Figure 7 gives times for the modified compiler on the tracked bench-

http://www.webkit.org/perf/sunspider/sunspider.html


http://krakenbenchmark.mozilla.org

2 http://www.webkit.org/perf/sunspider/sunspider.html

<table>
<thead>
<tr>
<th>Test</th>
<th>JM Compilation</th>
<th>JM+TI Compilation</th>
<th>×1 Times (ms)</th>
<th>×20 Times (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ms)</td>
<td>#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3d-cube</td>
<td>2.68</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3d-morph</td>
<td>0.55</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3d-raytrace</td>
<td>2.25</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>access-binary-trees</td>
<td>0.63</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>access-fannkuch</td>
<td>0.65</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>access-nsieve</td>
<td>0.28</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bitops-3bit-bits-in-byte</td>
<td>0.28</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bitops-bits-in-byte</td>
<td>0.29</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bitops-bitwise-and</td>
<td>0.24</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bitops-nsieve-bits</td>
<td>0.35</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>controlflow-recursive</td>
<td>0.38</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>crypto-aes</td>
<td>2.04</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>crypto-md5</td>
<td>1.81</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>crypto-sha1</td>
<td>0.88</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>date-format-tofte</td>
<td>0.93</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>date-format-xparb</td>
<td>0.88</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>math-cordic</td>
<td>0.45</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>math-partial-sums</td>
<td>0.47</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>math-spectral-norm</td>
<td>0.39</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>regexp-dna</td>
<td>0.00</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string-base64</td>
<td>0.87</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string-fasta</td>
<td>0.59</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string-tagcloud</td>
<td>0.54</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string-unpack-code</td>
<td>0.89</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string-validate-input</td>
<td>0.58</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Comparison of JM and JM+TI. The ratio of JM to JM+TI shows a larger improvement than a raw time; larger is better. The ×1 and ×20 columns show the running times of the two versions of the compiler and generated code on the benchmark run once and modified to run twenty times, respectively. In the single run case JM+TI is a 6.3% improvement over JM. One run of SunSpider completes in less than 250ms, which makes it difficult to get an optimization to pay for itself on this benchmark.

Figure 4. SunSpider-0.9.1 Benchmark Results
We modified Firefox to track several precision metrics while running, all of which operate at the granularity of individual operations. A brief description of the websites used is below. A full description of the tested websites and methodology used for each is available in the appendix of the full version of the paper.

<table>
<thead>
<tr>
<th>Test</th>
<th>JM</th>
<th>JM+TI</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richards</td>
<td>4497</td>
<td>7152</td>
<td>1.59</td>
</tr>
<tr>
<td>DeltaBlue</td>
<td>3250</td>
<td>9087</td>
<td>2.80</td>
</tr>
<tr>
<td>Crypto</td>
<td>5205</td>
<td>13376</td>
<td>2.57</td>
</tr>
<tr>
<td>RayTrace</td>
<td>3733</td>
<td>3217</td>
<td>0.86</td>
</tr>
<tr>
<td>EarleyBoyer</td>
<td>4546</td>
<td>6291</td>
<td>1.38</td>
</tr>
<tr>
<td>RegExp</td>
<td>1547</td>
<td>1316</td>
<td>0.85</td>
</tr>
<tr>
<td>Splay</td>
<td>4775</td>
<td>7049</td>
<td>1.48</td>
</tr>
<tr>
<td>Total</td>
<td>3702</td>
<td>5555</td>
<td>1.50</td>
</tr>
</tbody>
</table>

Figure 5. V8 (version 6) Benchmark Scores (higher is better)

<table>
<thead>
<tr>
<th>Test</th>
<th>JM (ms)</th>
<th>JM+TI (ms)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>ai-astar</td>
<td>889.4</td>
<td>137.8</td>
<td>0.15</td>
</tr>
<tr>
<td>audio-dlt</td>
<td>641.0</td>
<td>374.8</td>
<td>0.58</td>
</tr>
<tr>
<td>audio-dft</td>
<td>627.8</td>
<td>352.6</td>
<td>0.56</td>
</tr>
<tr>
<td>audio-fft</td>
<td>494.0</td>
<td>229.8</td>
<td>0.47</td>
</tr>
<tr>
<td>audio-oscillator</td>
<td>518.0</td>
<td>221.2</td>
<td>0.43</td>
</tr>
<tr>
<td>imaging-gaussian-blur</td>
<td>4351.4</td>
<td>730.0</td>
<td>1.70</td>
</tr>
<tr>
<td>imaging-darkroom</td>
<td>699.6</td>
<td>586.8</td>
<td>0.84</td>
</tr>
<tr>
<td>imaging-desaturate</td>
<td>821.2</td>
<td>209.2</td>
<td>0.25</td>
</tr>
<tr>
<td>json-parse-financial</td>
<td>116.6</td>
<td>119.2</td>
<td>1.02</td>
</tr>
<tr>
<td>json-stringify-tinderbox</td>
<td>80.0</td>
<td>78.8</td>
<td>0.99</td>
</tr>
<tr>
<td>crypto-aes</td>
<td>201.6</td>
<td>158.0</td>
<td>0.78</td>
</tr>
<tr>
<td>crypto-ccm</td>
<td>127.8</td>
<td>133.6</td>
<td>1.05</td>
</tr>
<tr>
<td>crypto-pbkdf2</td>
<td>454.8</td>
<td>350.2</td>
<td>0.77</td>
</tr>
<tr>
<td>crypto-sha256-iterative</td>
<td>153.2</td>
<td>106.2</td>
<td>0.69</td>
</tr>
<tr>
<td>Total</td>
<td>10176.4</td>
<td>3778.2</td>
<td>0.37</td>
</tr>
</tbody>
</table>

Figure 6. Kraken-1.1 Benchmark Results

<table>
<thead>
<tr>
<th>Suite</th>
<th>Time/Score</th>
<th>vs. JM</th>
<th>vs. JM+TI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunspider-0.9.1</td>
<td>262.2</td>
<td>1.00</td>
<td>1.06</td>
</tr>
<tr>
<td>Sunspider-0.9.1</td>
<td>4044.3</td>
<td>0.86</td>
<td>1.09</td>
</tr>
<tr>
<td>Kraken-1.1</td>
<td>7948.6</td>
<td>0.78</td>
<td>2.10</td>
</tr>
<tr>
<td>V8 (version 6)</td>
<td>4317</td>
<td>1.17</td>
<td>0.78</td>
</tr>
</tbody>
</table>

Figure 7. Benchmark Results with 100% barriers

In this section we measure the precision of the analysis on a variety of websites. The impact of compiler optimizations is difficult to accurately measure on websites due to confounding issues like differences in network latency and other browser effects. Since analysis precision directly ties into the quality of generated code, it makes a good surrogate for optimization effectiveness. We modified Firefox to track several precision metrics while running, all of which operate at the granularity of individual operations. A brief description of the websites used is below. A full description of the tested websites and methodology used for each is available in the appendix of the full version of the paper.

- Ten popular websites which use JavaScript extensively. Each site was used for several minutes, exercising various features.
- The membench50 suite, a memory testing framework which loads the front pages of 50 popular websites.
- The three benchmark suites described in §4.1.
- Six games and demos which are bound on JavaScript performance. Each was used for several minutes or, in the case of non-interactive demos, viewed to completion.

When developing the analysis and compiler we tuned behavior for the three covered benchmark suites, as well as various websites. Besides the benchmarks, no tuning work has been done for any of the websites described here.

We address several questions related to analysis precision, listed below. The answers to these sometimes differ significantly across the different categories of websites.

1. How polymorphic are values read at access sites? (§4.2.1)
2. How often are type barriers required? (§4.2.2)
3. How polymorphic are performed operations? (§4.2.3)
4. How polymorphic are the objects used at access sites? (§4.2.4)
5. How important are type barriers? (§4.2.5)

4.2.1 Access Site Polymorphism

The degree of polymorphism used in practice is of utmost importance for our analysis. The analysis is sound and will always compute a lower bound on the possible types that can appear at the various points in a program, so the precision of the generated type information is limited for access sites and operations which are polymorphic in practice. We draw the following distinction:

Monomorphic Sites that have only ever produced a single kind of value. Two values are of the same kind if they are either primitives of the same type or both objects with possibly different object types. Access sites containing objects of multiple types can often be optimized just as well as sites containing objects of a single type, as long as all the observed object types share common attributes (§4.2.4).

Dimorphic Sites that have produced either strings or objects (but not both), and also at most one of the undefined, null or a boolean value. Even though multiple kinds are possible at such sites, an untyped representation can still be used, as a single test on the unboxed form will determine the type. The untyped representation of objects and strings are pointers, whereas undefined, null and booleans are either 0 or 1.

Polymorphic Sites that have produced values of multiple kinds, and compiled code must use a typed representation which keeps track of the value’s kind.

The inferred precision section of Figure 8 shows the fractions of dynamic indexed element and property reads which were at a site inferred as producing monomorphic, dimorphic, or polymorphic sets of values. All these sites have type barriers on them, so the set of inferred types is equivalent to the set of observed types.

The category used for a dynamic access is determined from the types inferred at the time of the access. Since the types inferred for an access site can grow as a program executes, dynamic accesses at the same site can contribute to different columns over time.

Averaged across pages, 84.7% of reads were at monomorphic sites, and 90.2% were at monomorphic or dimorphic sites. The latter figure is 85.9% for websites, 97.3% for benchmarks, and

5http://gregar-wagner.com/tmp/mem50
precise type information is crucial for efficient compilation, and
of indexed accesses, respectively. These are operations for which
easily optimize their code.

if the barriers were not in place. We are building tools to identify
significant performance improvements but will perform worse than
leads to barriers being required. Per §4.1.1, such sites will still see
proportion of access sites which are, with only a small amount of
ers themselves means that we detect as monomorphic the very large
marks, games and demos were generally much lower, averaging
between 35% and 74% of accesses (averaging 50%), while bench-
barrier. Averaged across pages, barriers were required on 41.4% of
dexed and property accesses on sampled pages which required a
by the analysis.

Examining the frequency with which type barriers are required
gives insight to the precision of the model of the heap constructed
for the heap. Figure 8 shows the fractions of indexed
and property accesses on sampled pages which required a
barrier. Averaged across pages, barriers were required on 41.4% of
such accesses. There is a large disparity between websites and other
pages. Websites were fairly homogenous, requiring barriers on be-
tween 35% and 74% of accesses (averaging 50%), while bench-
marks, games and demos were generally much lower, averaging
13% except for two outliers above 90%.

The larger proportion of barriers required for websites indicates
that heap layouts and types tend to be more complicated for web-
sites than for games and demos. Still, the presence of the type barriers
themselves means that we detect as monomorphic the very large
proportion of access sites which are, with only a small amount of
barrier checking overhead incurred by the more complicated heaps.

The two outliers requiring a very high proportion of barriers
do most of their accesses at a small number of sites; the involved
objects have multiple types assigned to their properties, which
leads to barriers being required. Per §4.1.1, such sites will still see
significant performance improvements but will perform worse than
if the barriers were not in place. We are building tools to identify
hot spots and performance faults in order to help developers more
easily optimize their code.

4.2.2 Barrier Frequency

The arithmetic and indices sections of Figure 8 show the frequency
of inferred types for arithmetic operations and the index operand of
indexed accesses, respectively. These are operations for which
precise type information is crucial for efficient compilation, and
give a sense of the precision of type information for operations
which do not have associated type barriers.

In the arithmetic section, the integer, double, other, and unknown columns indicate, respectively, on known integers which give an integer result, operations on integers or doubles which give a double result, operations on any other type of known value, and operations where at least one of the operand types is unknown. Overall, precise types were found for 93.8% of arithmetic operations, including 90.2% of operations performed by websites. Comparing websites with other pages, websites tend to do far more arithmetic on non-numeric values — 16.8% vs. 1.6% — and considerably less arithmetic on doubles — 14.8% vs. 37.9%.

In the indices section, the integer, double, other, and unknown columns indicate, respectively, that the type of the index, i.e., the type of i in an expression such as a[i], is known to be an integer, a double, any other known type, or unknown. Websites tend to have more unknown index types than both benchmarks and games.

4.2.4 Access Site Precision

Efficiently compiling indexed element and property accesses requires knowledge of the kind of object being accessed. This information is more specific than the monomorphic/polymorphic distinction drawn in §4.2.1. Figure 9 shows the fractions of indexed accesses on arrays and of all property accesses which were optimized based on static knowledge.

In the indexed access section, the packed column shows the fraction of operations known to be on packed arrays (§2.4), while the array column shows the fraction known to be on arrays not known to be packed. Indexed operations behave differently on arrays vs. other objects, and avoiding dynamic array checks achieves some speedup. The “Uk” column is the fraction of dynamic accesses on arrays which are not statically known to be on arrays.

Static detection of array operations is very good on all kinds of sites, with an average of 75.2% of accesses on known packed arrays and an additional 14.8% on known but possibly not packed arrays. A few outlier websites are responsible for the great majority


<table>
<thead>
<tr>
<th>Test</th>
<th>Mono</th>
<th>Di</th>
<th>Poly</th>
<th>Int</th>
<th>Double</th>
<th>Other</th>
<th>Unknown</th>
<th>Int</th>
<th>Double</th>
<th>Other</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>gmail</td>
<td>78</td>
<td>5</td>
<td>17</td>
<td>47</td>
<td>62</td>
<td>9</td>
<td>7</td>
<td>21</td>
<td>44</td>
<td>0</td>
<td>47</td>
</tr>
<tr>
<td>googlemaps</td>
<td>81</td>
<td>7</td>
<td>12</td>
<td>36</td>
<td>66</td>
<td>26</td>
<td>3</td>
<td>5</td>
<td>60</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>facebook</td>
<td>73</td>
<td>11</td>
<td>16</td>
<td>42</td>
<td>43</td>
<td>0</td>
<td>40</td>
<td>16</td>
<td>62</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>flickr</td>
<td>71</td>
<td>19</td>
<td>10</td>
<td>74</td>
<td>61</td>
<td>1</td>
<td>30</td>
<td>8</td>
<td>27</td>
<td>0</td>
<td>70</td>
</tr>
<tr>
<td>grooveshark</td>
<td>64</td>
<td>15</td>
<td>21</td>
<td>63</td>
<td>65</td>
<td>1</td>
<td>13</td>
<td>21</td>
<td>28</td>
<td>0</td>
<td>56</td>
</tr>
<tr>
<td>mecbo</td>
<td>78</td>
<td>11</td>
<td>10</td>
<td>35</td>
<td>66</td>
<td>9</td>
<td>18</td>
<td>8</td>
<td>17</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>reddit</td>
<td>71</td>
<td>7</td>
<td>22</td>
<td>51</td>
<td>64</td>
<td>0</td>
<td>29</td>
<td>7</td>
<td>22</td>
<td>0</td>
<td>71</td>
</tr>
<tr>
<td>youtube</td>
<td>83</td>
<td>11</td>
<td>6</td>
<td>38</td>
<td>50</td>
<td>27</td>
<td>19</td>
<td>4</td>
<td>33</td>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td>ztype</td>
<td>91</td>
<td>1</td>
<td>9</td>
<td>52</td>
<td>43</td>
<td>41</td>
<td>8</td>
<td>8</td>
<td>79</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>280slides</td>
<td>79</td>
<td>3</td>
<td>19</td>
<td>64</td>
<td>48</td>
<td>51</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>91</td>
</tr>
<tr>
<td>membench50</td>
<td>76</td>
<td>11</td>
<td>13</td>
<td>49</td>
<td>65</td>
<td>7</td>
<td>18</td>
<td>10</td>
<td>44</td>
<td>0</td>
<td>47</td>
</tr>
<tr>
<td>sunspider</td>
<td>99</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>72</td>
<td>21</td>
<td>7</td>
<td>0</td>
<td>95</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>v8bench</td>
<td>86</td>
<td>7</td>
<td>7</td>
<td>26</td>
<td>98</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>kraken</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>61</td>
<td>37</td>
<td>2</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>angrybirds</td>
<td>97</td>
<td>2</td>
<td>1</td>
<td>93</td>
<td>22</td>
<td>78</td>
<td>0</td>
<td>0</td>
<td>88</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>gameboy</td>
<td>88</td>
<td>0</td>
<td>12</td>
<td>16</td>
<td>54</td>
<td>36</td>
<td>3</td>
<td>7</td>
<td>88</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bullet</td>
<td>84</td>
<td>0</td>
<td>16</td>
<td>92</td>
<td>54</td>
<td>38</td>
<td>0</td>
<td>7</td>
<td>79</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>lights</td>
<td>97</td>
<td>1</td>
<td>2</td>
<td>15</td>
<td>34</td>
<td>66</td>
<td>0</td>
<td>1</td>
<td>95</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>FOTN</td>
<td>98</td>
<td>1</td>
<td>1</td>
<td>20</td>
<td>39</td>
<td>61</td>
<td>0</td>
<td>0</td>
<td>96</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>monalisa</td>
<td>99</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>94</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Average</td>
<td>84.7</td>
<td>5.7</td>
<td>9.8</td>
<td>41.4</td>
<td>58.1</td>
<td>25.7</td>
<td>10.0</td>
<td>6.2</td>
<td>63.2</td>
<td>1.7</td>
<td>27.0</td>
</tr>
</tbody>
</table>

Figure 8. Website Type Profiling Results
of accesses in the latter category. For example, the V8 Crypto benchmark contains almost all of the benchmark’s array accesses, and the arrays used are not known to be packed due to the top down order they are initialized. Still, speed improvements on this benchmark are very large.

In the property access section of Figure 9, the “Def” column shows the fraction of operations which were resolved statically as definite properties (§2.4), while the PIC column shows the fraction which were not resolved statically but were matched using a fallback mechanism, polymorphic inline caches [14]. The “Uk” column is the fraction of operations which were not resolved either statically or with a PIC and required a call into the VM; this includes accesses where objects with many different layouts are used, and accesses on rare kinds of properties such as those with scripted getters or setters.

An average of 39.4% of property accesses were resolved as definite properties, with a much higher average proportion on benchmarks of 80.3%. The remainder were by and large handled by PICs, with only 5.5% of accesses requiring a VM call. Together, these suggest that objects on websites are by and large constructed in a consistent fashion, but that our detection of definite properties needs to be more robust on object construction patterns seen on websites but not on benchmarks.

### 4.2.5 Precision Without Barriers

To test the practical effect of using type barriers to improve precision, we repeated the above website tests using a build of Firefox where subset constraints were used in place of barrier constraints, and type barriers were not used at all (semantic triggers were still used). Some of the numbers from these runs are shown in Figure 10.

The precision section shows the fraction of indexed and property accesses which were inferred as polymorphic, and the arithmetic section shows the fraction of arithmetic operations where at least one operand type was unknown. Both sections show the ratio of the given fraction to the comparable fraction with type barriers enabled, with entries struck out when the comparable fraction is near zero. Overall, with type barriers disabled 42.1% of accesses are polymorphic and 37.4% of arithmetic operations have operands of unknown type; precision is far worse than with type barriers.

Benchmarks are affected much less than other kinds of sites, which makes it difficult to measure the practical performance impact of removing barriers. These benchmarks use polymorphic structures much less than the web at large.

### 5. Related Work

There is an enormous literature on points-to analysis, JIT compilation, and type inference. We only compare against a few here.

The most relevant work on type inference for JavaScript to the current work is Logozzo and Venter’s work on rapid atomic type analysis [16]. Like ours, their analysis is also designed to be used online in the context of JIT compilation and must be able to pay for itself. Unlike ours, their analysis is purely static and much more sophisticated, utilizing a theory of integers to better infer integral types vs floating point types. We eschew sophistication in favor of simplicity and speed. Our evaluation shows that even a much simpler static analysis, when coupled with dynamic checks, performs very well “in the wild”. Our analysis is more practical: we have improved handling of what Logozzo and Venter termed “havoc” statements, such as eval, which make static analysis results imprecise. As Richards et al. argued in their surveys, real-world use of eval is pervasive, between 50% and 82% for popular websites [19, 20].

Other works on type inference for JavaScript are more formal. The work of Anderson et al. describes a structural object type system with subtyping over an idealized subset of JavaScript [7]. As the properties held by JavaScript objects change dynamically, the structural type of an object is a flow-sensitive property. Thiemann and Jensen et al.’s typing frameworks approach this problem by using recency types [15, 23]. The work of Jensen et al. is in the context of better tooling for JavaScript, and their experiments suggest that the algorithm is not suitable for online use in a JIT compiler.
References


