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  function Box(v) {
2    this.p = v;
3  }
4
5  function use(a) {
6    var res = 0;
7    for (var i = 0; i < 1000; i++) {
8      var v = a[i].p;
9      res = res + v;
10   }
11   return res;
12 }
13
14 function main() {
15   var a = [];
16   for (var i = 0; i < 1000; i++)
17     a[i] = new Box(10);
18   use(a);
19 }
```

Figure 1. Motivating Example

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).

1. 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`.
2. 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.
3. 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.
4. On line 8, property `p` of a value at `a[i]` 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.
5. 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`.

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 code, 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. A few such behaviors are described below.

- The read of `a[i]` 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[i].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 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 the 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[i].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[i].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

$$
\begin{array}{ll}
v ::= \texttt{undefined} \mid i \mid s \mid \texttt{\{\}} & \text{values} \\
e ::= v \mid x \mid e + e \mid x.p \mid x[i] & \text{expressions} \\
s ::= \texttt{if}(x)\ s\ \texttt{else}\ s \mid x = e \mid x.p = e \mid x[i] = e & \text{statements} \\
\tau ::= \texttt{undefined} \mid \texttt{int} \mid \texttt{number} \mid \texttt{string} \mid o & \text{types} \\
T ::= \mathscr{P}(\tau) & \text{type sets} \\
C ::= T \supseteq T \mid T \supseteq_{\mathscr{B}} T & \text{constraints}
\end{array}
$$

Figure 2. Simplified JavaScript Core, Types, and Constraints

$$
\begin{array}{lll}
\vdash^e \texttt{undefined} : T_{\mathsf{u}} & \{\ T_{\mathsf{u}} \supseteq \{\texttt{undefined}\}\ \} & (\textsc{Undef}) \\
\vdash^e i : T_i & \{\ T_i \supseteq \{\texttt{int}\}\ \} & (\textsc{Int}) \\
\vdash^e s : T_s & \{\ T_s \supseteq \{\texttt{string}\}\ \} & (\textsc{Str}) \\
\vdash^e \texttt{\{\}} : T_{\texttt{\{\}}} & \{\ T_{\texttt{\{\}}} \supseteq \{o\}\ \}\ \text{where } o \text{ fresh} & (\textsc{Obj}) \\
\vdash^e x : T_x & \emptyset & (\textsc{Var})
\end{array}
$$

$$
\frac{\vdash^e x : T_x \quad \vdash^e y : T_y}{\vdash^e x + y : T_{x+y}}
\quad
\left\{
\begin{array}{l}
T_{x+y} \supseteq \{\texttt{int}\} \mid \texttt{int} \in T_x \wedge \texttt{int} \in T_y, \\
T_{x+y} \supseteq \{\texttt{number}\} \mid \texttt{int} \in T_x \wedge \texttt{number} \in T_y, \\
T_{x+y} \supseteq \{\texttt{number}\} \mid \texttt{number} \in T_x \wedge \texttt{int} \in T_y, \\
T_{x+y} \supseteq \{\texttt{string}\} \mid \texttt{string} \in T_x \vee \texttt{string} \in T_y
\end{array}
\right\}
\quad (\textsc{Add})
$$

$$
\frac{\vdash^e x : T_x}{\vdash^e x.p : T_{x.p}} \quad \{\ T_{x.p} \supseteq_{\mathscr{B}} \mathit{prop}(o,p) \mid o \in T_x\ \} \quad (\textsc{Prop})
$$

$$
\frac{\vdash^e x : T_x}{\vdash^e x[i] : T_{x[i]}} \quad \{\ T_{x[i]} \supseteq_{\mathscr{B}} \mathit{index}(o) \mid o \in T_x\ \} \quad (\textsc{Index})
$$

$$
\frac{\vdash^e x : T_x \quad \vdash^e e : T_e}{\vdash^s x = e : \bullet} \quad \{\ T_x \supseteq T_e\ \} \quad (\textsc{A-Var})
$$

$$
\frac{\vdash^e x : T_x \quad \vdash^e e : T_e}{\vdash^s x.p = e : \bullet} \quad \{\ \mathit{prop}(o,p) \supseteq T_e \mid o \in T_x\ \} \quad (\textsc{A-Prop})
$$

$$
\frac{\vdash^e x : T_x \quad \vdash^e e : T_e}{\vdash^s x[i] = e : \bullet} \quad \{\ \mathit{index}(o) \supseteq T_e \mid o \in T_x\ \} \quad (\textsc{A-Index})
$$

$$
\vdash^s \texttt{if}(x)\ s_1\ \texttt{else}\ s_2 : \bullet \quad \mathscr{C}_s(s_1) \cup \mathscr{C}_s(s_2) \quad (\textsc{If})
$$

Figure 3. Constraint Generation Rules

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 are trying to execute: 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 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 o.[1] The `int` type indicates a number expressible as a signed 32-bit integer and is subsumed by `number` — `int` is added to all type sets containing `number`. 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 $o \in T_e$ 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 `Object`, `Array` and `Function` objects. `Object` and `Array` objects have the same type if they were allocated at the same source location, and `Function` objects have the same type if they are closures for the same script. `Object` and `Function` objects which represent builtin objects such as class prototypes, the `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 $\mathit{prop}(o, p)$ and $\mathit{index}(o)$. The set $\mathit{prop}(o, p)$ captures the possible types of a non-integer property p for objects with type o, while $\mathit{index}(o)$ 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

[1] In full JavaScript, we also have the primitive types `bool` and `null`.

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 fill 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, $\supseteq$, and a *barrier subset constraint*, styled $\supseteq_{\mathscr{B}}$. For two type sets X and Y, $X \supseteq Y$ means that all types in Y are propagated to X. On the other hand, $X \supseteq_{\mathscr{B}} Y$ 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 (§).

Rules for the constraint generation functions, $\mathscr{C}_e(e)$ for expressions (styled $\vdash^e$) and $\mathscr{C}_s(s)$ for statements (styled $\vdash^s$), are shown in Figure 3. Statically analyzing a function takes the union of the results from applying $\mathscr{C}_s$ to every statement in the method.

The UNDEF, 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` members, producing a number or string).

Using unsound modeling lets us cut through this complexity. Additions in actual programs are typically used to add two numbers or concatenate a string with something else. We statically model exactly these cases and use semantic triggers to monitor the results produced by other combinations of values, at little runtime cost. Note that even the integer addition rule we have given is unsound: the result will be marked as an integer, ignoring the possibility of overflow.

PROP accesses a named property p from the possible objects referred to by x, with the result the union of $prop(o,p)$ for all such objects. This rule is complete only in cases where the object referred to by x (or its prototype) actually has the p property. Accesses on properties which are not actually part of an object produce `undefined`. Accesses on missing properties are rare, and yet in many cases we cannot prove that an object definitely has some property. In such cases we do not dilute the resulting type sets with `undefined`. We instead use a trigger on execution paths accessing a missing property to update the result type of the access with `undefined`.

INDEX is similar to PROP, with the added problem that any property of the object could be accessed. In JavaScript, `x["p"]` is equivalent to `x.p`. If x has the object type o, an index operation can access a potentially infinite number of type sets $prop(o,p)$. Figuring out exactly which such properties are possible is generally intractable. We do not model such arbitrary accesses at all, and treat all index operations as operating on an integer, which we collapse into a single type set $index(o)$. In full JavaScript, any indexed access which is on a non-integer property, or is on an integer property which is missing from an object, must be accounted for with triggers in the same manner as PROP.

A-VAR, A-PROP and A-INDEX invert the corresponding read expressions. These rules are complete, except that A-INDEX presumes that an integer property is being accessed. Again, in full JavaScript, the effects on $prop(o,p)$ resulting from assignments to a string index `x["p"]` on some x with object type o must be accounted for with runtime checks.

Our analysis is flow-insensitive, so the IF rule is simply the union of the constraints generated by the branches.

2.3 Type Barriers

As described in §1, type barriers are dynamic type checks inserted to improve analysis precision in the presence of polymorphic code. Propagation along an assignment $X = Y$ can be modeled statically as a subset constraint $X \supseteq Y$ or dynamically as a barrier constraint $X \supseteq_{\mathscr{B}} Y$. It is always safe to use one in place of the other; in §4.2.5 we show the effect of always using subset constraints in lieu of barrier constraints.

For a barrier constraint $X \supseteq_{\mathscr{B}} Y$, a type barrier is required whenever $X \not\supseteq Y$. The barrier dynamically checks that the type of each value flowing across the assignment is actually in X, and updates X whenever values of a new type are encountered. Thought of another way, the vanilla subset constraint propagates all types at analysis time. The barrier subset constraint does not propagate types at analysis time but defers with dynamic checks, propagating the types only if necessary during runtime.

Type barriers are much like dynamic type casts in Java: assignments from a more general type to a more specific type are possible as long as a dynamic test occurs for conformance. However, rather than throw an exception (as in Java) a tripped type barrier will despecialize the target of the assignment.

The presence or absence of type barriers for a given barrier constraint is not monotonic with respect to the contents of the type sets in the program. As new types are discovered, new type barriers may be required, and existing ones may become unnecessary. However, it is always safe to perform the runtime tests for a given barrier.

Recall our hypothetical situation from §1 where `Box` is used as a polymorphic structure containing either an integer or a string in the example program from Figure 1. The subset barrier constraint on line 8 is $T_{\texttt{a[i]}} \supseteq_{\mathscr{B}} T_{\texttt{Box}}$, with $T_{\texttt{a[i]}} = \{\texttt{int}\}$ and $T_{\texttt{Box}} = \{\texttt{int}, \texttt{string}\}$. Since $T_{\texttt{a[i]}} \not\supseteq T_{\texttt{Box}}$, a type barrier is required.

In the constraint generation rules in Figure 3 we present two rules which employ type barrers: PROP, and INDEX. In practice, we also use type barriers for call argument binding to precisely model polymorphic call sites where only certain combinations of argument types and callee functions are possible. 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. Semantic 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.

Packed Arrays Arrays are usually constructed by writing to their elements in ascending order, with no gaps; we call these arrays *packed*. Packed arrays do not have holes in the middle, and if an access is statically known to be on a packed array then only a bounds check is required. There are a large number of ways packed arrays can be constructed, however, which makes it difficult to statically prove an array is packed. Instead, we dynamically detect out-of-order writes on an array, and mark the type of the array object as possibly not packed. If an object type has never been marked as not packed, then all objects with that type are packed arrays.

The packed status of an object type can change dynamically due to out-of-order writes, possibly invalidating JIT code.

Definite Properties JavaScript objects are internally laid out as a map from property names to slots in an array of values. If a property access can be resolved statically to a particular slot in the array, then the access is on a *definite* property and can be compiled as a direct lookup. This is comparable to field accesses in a language with static object layouts, such as Java or C++.

We identify definite property accesses in three ways. First, if the property access is on an object with a unique type, we know the exact JavaScript object being accessed and can use the slot in its property map. Second, object literals allocated in the same place have the same type, and definite properties can be picked up from the order the literal adds properties. Third, objects created by calling `new` on the same function will have the same prototype (unless the function's `prototype` property is overwritten), and we analyze the function's body to identify properties it definitely adds before letting the new object escape.

These techniques are sensitive to properties being deleted or reconfigured, and if such events happen then JIT code will be invalidated in the same way as by packed array or type set changes.

3. Implementation

We have implemented this analysis for SpiderMonkey, the JavaScript 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 generates code in a fairly mechanical translation from the original SpiderMonkey bytecode for a script. Using type information, we were able to improve on this in several ways:

- Values with statically known types can be tracked in JIT-compiled code using an untyped representation. Encoding the type in a value requires significant memory traffic or marshaling overhead. An untyped representation stores just the data component of a value. Additionally, knowing the type of a value statically eliminates many dynamic type tests.
- Several classical compiler optimizations were added, including linear scan register allocation, loop invariant code motion and function call inlining.

  These optimizations could be applied without having static type information. Doing so is, however, far more difficult and far less effective than in the case where types are known. For example, loop invariant code motion depends on knowing whether operations are idempotent, while in general JavaScript operations are not, and register allocation requires types to determine whether values should be stored in general purpose or floating point registers.

In §3.1 we describe how we handle dynamic recompilation in response to type changes, and in §3.2 we describe the techniques used to manage analysis memory usage.

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 contrast 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 regenerable 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 possible. 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 barrier 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 combination 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 analysis information will be quickly destroyed. If the user is staying on one page, active scripts may be repeatedly recompiled and reanalyzed, but the timeframe between collections keeps this as a small portion of overall runtime. When many tabs are open (the case where memory usage is most important for the browser), analysis information typically accounts for less than 2% of the browser's overall memory usage.

4. Evaluation

We evaluate the effectiveness of our analysis in two ways. In §4.1 we compare the performance on major JavaScript benchmarks of a single compiler with and without use of analyzed type information. In §4.2 we examine the behavior of the analysis on a selection of websites which heavily use JavaScript to gauge analysis effectiveness in practice.

	JM Compilation		JM+TI Compilation			×1 Times (ms)			×20 Times (ms)		
Test	Time (ms)	#	Time (ms)	#	Ratio	JM	JM+TI	Ratio	JM	JM+TI	Ratio
3d-cube	2.68	15	8.21	24	3.06	14.1	16.6	1.18	226.9	138.8	0.61
3d-morph	0.55	2	1.59	7	2.89	9.8	10.3	1.05	184.7	174.6	0.95
3d-raytrace	2.25	19	6.04	22	2.68	14.7	15.6	1.06	268.6	152.2	0.57
access-binary-trees	0.63	4	1.03	7	1.63	6.1	5.2	0.85	101.4	70.8	0.70
access-fannkuch	0.65	1	2.43	4	3.76	15.3	10.1	0.66	289.9	113.7	0.39
access-nbody	1.01	5	1.49	5	1.47	9.9	5.3	0.54	175.6	73.2	0.42
access-nsieve	0.28	1	0.63	2	2.25	6.9	4.5	0.65	143.1	90.7	0.63
bitops-3bit-bits-in-byte	0.28	2	0.58	3	2.07	1.7	0.8	0.47	29.9	10.0	0.33
bitops-bits-in-byte	0.29	2	0.54	3	1.86	7.0	4.8	0.69	139.4	85.4	0.61
bitops-bitwise-and	0.24	1	0.39	1	1.63	6.1	3.1	0.51	125.2	63.7	0.51
bitops-nsieve-bits	0.35	1	0.73	2	2.09	6.0	3.6	0.60	116.1	63.9	0.55
controlflow-recursive	0.38	3	0.65	6	1.71	2.6	2.7	1.04	49.4	42.3	0.86
crypto-aes	2.04	14	6.61	23	3.24	9.3	10.9	1.17	162.6	107.7	0.66
crypto-md5	1.81	9	3.42	13	1.89	6.1	6.0	0.98	62.0	27.1	0.44
crypto-sha1	0.88	7	2.46	11	2.80	3.1	4.0	1.29	44.2	19.4	0.44
date-format-tofte	0.93	21	2.27	24	2.44	16.4	18.3	1.12	316.6	321.8	1.02
date-format-xparb	0.88	7	1.26	6	1.43	11.6	14.8	1.28	219.4	285.1	1.30
math-cordic	0.45	3	0.94	5	2.09	7.4	3.4	0.46	141.0	50.3	0.36
math-partial-sums	0.47	1	1.03	3	2.19	14.1	12.4	0.88	278.4	232.6	0.84
math-spectral-norm	0.54	5	1.39	9	2.57	5.0	3.4	0.68	92.6	51.2	0.55
regexp-dna	0.00	0	0.00	0	0.00	16.3	16.1	0.99	254.5	268.8	1.06
string-base64	0.87	3	1.90	5	2.18	7.8	6.5	0.83	151.9	103.6	0.68
string-fasta	0.59	4	1.70	9	2.88	10.0	7.3	0.73	124.0	93.4	0.75
string-tagcloud	0.54	4	1.54	6	2.85	21.0	24.3	1.16	372.4	433.4	1.17
string-unpack-code	0.89	8	2.65	16	2.98	24.4	26.7	1.09	417.6	442.5	1.06
string-validate-input	0.58	4	1.65	8	2.84	10.2	9.5	0.93	216.6	184.1	0.85
Total	21.06	146	53.13	224	2.52	261.9	246.4	0.94	4703.6	3700.3	0.79

Figure 4. SunSpider-0.9.1 Benchmark Results

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 used 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[2].

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 32ms, 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 $\times 1$ and $\times 20$ 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. JavaScript heavy webpages are typically viewed for longer than 1/4 of a second, and longer execution times better show the effect of type based optimizations. When run twenty times, the speedup given by JM+TI increases to 27.1%.

[2] `http://www.webkit.org/perf/sunspider/sunspider.html`

Figures 5 and 6 compare the performance of JM and JM+TI on two other popular benchmarks, the V8[3] and Kraken[4] 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 builtin 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-

[3] `http://v8.googlecode.com/svn/data/benchmarks/v6/run.html`

[4] `http://krakenbenchmark.mozilla.org`

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

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

Test	JM (ms)	JM+TI (ms)	Ratio
ai-astar	889.4	137.8	0.15
audio-beat-detection	641.0	374.8	0.58
audio-dft	627.8	352.6	0.56
audio-fft	494.0	229.8	0.47
audio-oscillator	518.0	221.2	0.43
imaging-gaussian-blur	4351.4	730.0	0.17
imaging-darkroom	699.6	586.8	0.84
imaging-desaturate	821.2	209.2	0.25
json-parse-financial	116.6	119.2	1.02
json-stringify-tinderbox	80.0	78.8	0.99
crypto-aes	201.6	158.0	0.78
crypto-ccm	127.8	133.6	1.05
crypto-pbkdf2	454.8	350.2	0.77
crypto-sha256-iterative	153.2	106.2	0.69
Total	10176.4	3778.2	0.37

Figure 6. Kraken-1.1 Benchmark Results

Suite	Time/Score	vs. JM	vs. JM+TI
Sunspider-0.9.1 $\times 1$	262.2	1.00	1.06
Sunspider-0.9.1 $\times 20$	4044.3	0.86	1.09
Kraken-1.1	7948.6	0.78	2.10
V8 (version 6)	4317	1.17	0.78

Figure 7. Benchmark Results with 100% barriers

marks. On a single run of SunSpider, performance was even with the JM compiler. In all other cases, performance was significantly better than the JM compiler and significantly worse than the JM+TI compiler.

This indicates that while the compiler will still be able to effectively optimize code in cases where types of heap values are not well known, accurately inferring such types and minimizing the barrier count is important for maximizing performance.

4.2 Website Performance

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[5], 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

[5] `http://gregor-wagner.com/tmp/mem50`

	Inferred Precision (%)				Arithmetic (%)				Indices (%)			
Test	Mono	Di	Poly	Barrier (%)	Int	Double	Other	Unknown	Int	Double	Other	Unknown
gmail	78	5	17	47	62	9	7	21	44	0	47	8
googlemaps	81	7	12	36	66	26	3	5	60	6	30	4
facebook	73	11	16	42	43	0	40	16	62	0	32	6
flickr	71	19	10	74	61	1	30	8	27	0	70	3
grooveshark	64	15	21	63	65	1	13	21	28	0	56	16
meebo	78	11	10	35	66	9	18	8	17	0	34	49
reddit	71	7	22	51	64	0	29	7	22	0	71	7
youtube	83	11	6	38	50	27	19	4	33	0	38	29
ztype	91	1	9	52	43	41	8	8	79	9	12	0
280slides	79	3	19	64	48	51	1	0	6	0	91	2
membench50	76	11	13	49	65	7	18	10	44	0	47	10
sunspider	99	0	1	7	72	21	7	0	95	0	4	1
v8bench	86	7	7	26	98	1	0	0	100	0	0	0
kraken	100	0	0	3	61	37	2	0	100	0	0	0
angrybirds	97	2	1	93	22	78	0	0	88	8	0	5
gameboy	88	0	12	16	54	36	3	7	88	0	0	12
bullet	84	0	16	92	54	38	0	7	79	20	0	1
lights	97	1	2	15	34	66	0	1	95	0	4	1
FOTN	98	1	1	20	39	61	0	0	96	0	3	0
monalisa	99	1	0	4	94	3	2	0	100	0	0	0
Average	84.7	5.7	9.8	41.4	58.1	25.7	10.0	6.2	63.2	1.7	27.0	7.7

Figure 8. Website Type Profiling Results

94.7% for games and demos; websites are more polymorphic than games and demos, but by and large behave in a monomorphic fashion.

4.2.2 Barrier Frequency

Examining the frequency with which type barriers are required gives insight to the precision of the model of the heap constructed by the analysis.

The barrier section of Figure 8 shows the frequencies 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 between 35% and 74% of accesses (averaging 50%), while benchmarks, 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 websites 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.

4.2.3 Operation Precision

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, operations 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

	Indexed Acc. (%)			Property Acc. (%)		
Test	Packed	Array	Uk	Def	PIC	Uk
gmail	90	4	5	31	57	12
googlemaps	92	1	7	18	77	5
facebook	16	68	16	41	53	6
flickr	27	0	73	33	53	14
grooveshark	90	2	8	20	66	14
meebo	57	0	43	40	57	3
reddit	97	0	3	45	51	4
youtube	100	0	0	32	49	19
ztype	100	0	0	23	76	0
280slides	88	12	0	23	56	21
membench50	80	4	16	35	58	6
sunspider	93	6	1	81	19	0
v8bench	7	93	0	64	36	0
kraken	99	0	0	96	4	0
angrybirds	90	0	10	22	76	2
gameboy	98	0	2	6	94	0
bullet	4	96	0	32	65	3
lights	97	3	1	21	78	1
FOTN	91	6	3	46	54	0
monalisa	87	0	13	78	22	0
Average	75.2	14.8	10.1	39.4	55.1	5.5

Figure 9. Indexed/Property Access Precision

	Precision		Arithmetic	
Test	Poly (%)	Ratio	Unknown (%)	Ratio
gmail	46	2.7	32	1.5
googlemaps	38	3.2	23	4.6
facebook	48	3.0	20	1.3
flickr	61	6.1	39	4.9
grooveshark	58	2.8	30	1.4
meebo	36	3.6	28	3.5
reddit	37	1.7	13	1.9
youtube	40	6.7	28	7.0
ztype	54	6.0	63	7.9
280slides	76	4.0	93	—
membench50	47	3.6	29	2.9
sunspider	5	—	6	—
v8bench	18	2.6	1	—
kraken	2	—	2	—
angrybirds	90	—	93	—
gameboy	15	1.3	7	1.0
bullet	62	3.9	79	11.3
lights	37	—	63	—
FOTN	28	—	57	—
monalisa	44	—	41	—
Average	42.1	4.3	37.4	6.0

Figure 10. Type Profiles Without Barriers

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 statically resolved as definite properties (§2.4), while the PIC column shows the fraction which were not resolved statically but were matched using a fall-back 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.

Again, these analyses do not perform well in the presence of statically uncomputable builtin functions such as `eval`.

Performing static type inference on dynamic languages has been proposed at least as early as Aiken and Murphy [4]. More related in spirit to the current work are the works of the the implementors of the Self language [24]. In implementing type inference for JavaScript, we faced many challenges similar to what they faced decades earlier [1, 25]. Agesen outlines the design space for type inference algorithms along the dimensions of efficiency and precision. We strived for an algorithm that is both fast and efficient, at the expense of requiring runtime checks when dealing with complex code. Our experience building tracing JIT compilers [11, 12] has demonstrated that solely using type feedback limits the optimizations that we can perform, and reaching peak performance requires static knowledge about the possible types of heap values.

Agesen and Hölzle compared the static approach of type inference with the dynamic approach of type feedback and described the strengths and weaknesses of both [2]. Our system tries to achieve the best of both worlds. The greatest difficulty in static type inference for polymorphic dynamic languages, whether functional or object-oriented, is the need to compute both data and control flow during type inference. We solve this by using runtime information where static analyses do poorly, e.g. determining the particular field of a polymorphic receiver or the particular function bound to a variable. Our type barriers may be seen as a type cast in context of Glew and Palsberg's work on method inlining [13].

Framing the type inference problem as a flow problem is a well-known approach [17, 18]; practical examples include Self's inferencer [3]. Aiken and Wimmers presented general results on type inference using subset constraints [5].

Other hybrid approaches to typing exist, such as Cartwright and Fagan's soft typing and Taha and Siek's gradual typing [8, 21]. They have been largely for the purposes of correctness and early error detection. While these approaches may also be used to improve performance of compiled code, they are at least partially *prescriptive*, in that they help enforce a typing discipline, while ours is entirely *descriptive*, in that we are inferring types only to help JIT compilation.

6. Conclusion and Future Work

We have described a hybrid type inference algorithm that is both fast and precise using constraint-based static analysis and runtime checks. Our production-quality implementation integrated with the JavaScript JIT compiler inside Firefox has demonstrated the analysis to be both effective and viable. We have presented compelling empirical results: the analysis enables generation of much faster code, and infers precise information on both benchmarks and real websites.

We hope to look more closely at type barriers in the future with the aim to reduce their frequency without degrading precision. We also hope to look at capture more formally the hybrid nature of our algorithm.

Acknowledgements. We thank the Mozilla JavaScript team, Todd Millstein, Jens Palsberg, and Sam Tobin-Hochstadt for draft reading and helpful discussion.

References

[1] O. Agesen. Constraint-Based Type Inference and Parametric Polymorphism, 1994.

[2] O. Agesen and U. Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *OOPSLA*, pages 91–107, 1995.

[3] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance. In *ECOOP*, pages 247–267, 1993.

[4] A. Aiken and B. R. Murphy. Static Type Inference in a Dynamically Typed Language. In *POPL*, pages 279–290, 1991.

[5] A. Aiken and E. L. Wimmers. Type Inclusion Constraints and Type Inference. In *FPCA*, pages 31–41, 1993.

[6] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[7] C. Anderson, S. Drossopoulou, and P. Giannini. Towards Type Inference for JavaScript. In *ECOOP*, pages 428–452, 2005.

[8] R. Cartwright and M. Fagan. Soft Typing. In *PLDI*, pages 278–292, 1991.

[9] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, Stanford, 1992.

[10] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language. In *PLDI*, 1989.

[11] M. Chang, E. W. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for Web 3.0: Trace Compilation for the Next Generation Web Applications. In *VEE*, pages 71–80, 2009.

[12] 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 *PLDI*, pages 465–478, 2009.

[13] N. Glew and J. Palsberg. Type-Safe Method Inlining. In *ECOOP*, pages 525–544, 2002.

[14] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP*, pages 21–38, 1991.

[15] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *SAS*, pages 238–255, 2009.

[16] F. Logozzo and H. Venter. RATA: Rapid Atomic Type Analysis by Abstract Interpretation. Application to JavaScript Optimization. In *CC*, pages 66–83, 2010.

[17] N. Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making Type Inference Practical. In *ECOOP*, 1992.

[18] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *OOPSLA*, 1991.

[19] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, pages 1–12, 2010.

[20] G. Richards, C. Hammer, B. Burg, and J. Vitek. The Eval That Men Do – A Large-Scale Study of the Use of Eval in JavaScript Applications. In *ECOOP*, pages 52–78, 2011.

[21] J. G. Siek and W. Taha. Gradual Typing for Objects. In *ECOOP*, 2007.

[22] M. Sridharan and S. J. Fink. The Complexity of Andersen's Analysis in Practice. In *SAS*, pages 205–221, 2009.

[23] P. Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *ESOP*, pages 408–422, 2005.

[24] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *OOPSLA*, pages 227–242, 1987.

[25] D. Ungar, R. B. Smith, C. Chambers, and U. Hölzle. Object, Message, and Performance: How they Coexist in Self. *Computer*, 25:53–64, October 1992. ISSN 0018-9162.