Improved Type Specialization for Dynamic Scripting Languages

Madhukar N Kedlaya[†] Jared Roesch[†] Behnam Robatmili[⋆] Mehrdad Reshadi[⋆] Ben Hardekopf[†]

[†]University of California, Santa Barbara [⋆]Qualcomm Research Silicon Valley

{mkedlaya,jroesch,benh}@cs.ucsb.edu mcjs@qti.qualcomm.com

Abstract

Type feedback and type inference are two common methods used to optimize dynamic languages such as JavaScript. Each of these methods has its own strengths and weaknesses, and we propose that each can benefit from the other if combined in the right way. We explore the interdependency between these two methods and propose two novel ways to combine them in order to significantly increase their aggregate benefit and decrease their aggregate overhead. In our proposed strategy, an initial type inference pass is applied that can reduce type feedback overhead by enabling more intelligent placement of profiling hooks. This initial type inference pass is novel in the literature. After profiling, a final type inference pass uses the type information from profiling to generate efficient code. While this second pass is not novel, we significantly improve its effectiveness in a novel way by feeding the type inference pass information about the function signature, i.e., the types of the function's arguments for a specific function invocation. Our results show significant speedups when using these low-overhead strategies, ranging from 1.2× to 4× over an implementation that does not perform type feedback or type inference based optimizations. Our experiments are carried out across a wide range of traditional benchmarks and realistic web applications. The results also show an average reduction of 23.5% in the size of the profiled data for these benchmarks.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors–Run-time environments, Optimization

Keywords JavaScript, type specialization, type inference, profiling, language implementation

1. Introduction

In the past decade there has been a resurgence of dynamic languages, as shown by the growth and penetration of languages such as Python, Ruby, and JavaScript. The defining characteristic of these languages is their extreme dynamism, including dynamic types. This dynamism, while useful, presents a significant challenge for efficient language implementation. Because the types of values are statically unknown, the language runtime requires an extra level of indirection: instead of operating directly on values, it operates on special "dynamic values" that box actual values inside a data structure that records the enclosed value's type. To operate on these values the runtime must conditionally branch based on the enclosed value's type, unbox the enclosed value to perform the required operation (which sometimes involves complex type conversion operations), then re-box the result back into a dynamic value. This extra level of indirection not only imposes a large runtime overhead, but also inhibits other optimizations that could take place if the runtime knew the value types ahead of time.

Researchers and dynamic language implementors have spent considerable effort on creating efficient dynamic language runtimes. The main strategy employed is *type specialization*: replacing the generic code that manipulates dynamic values with code specialized to handle only specific types of values. Of course, this strategy is only effective if the runtime can be guaranteed that the specialized code will only be run on values of the appropriate types. The two differing methods that have historically been used in various language implementations to provide this guarantee are *type feedback* [20] and *type inference* [7].

These two methods have differing strengths and weaknesses. Type feedback uses online type profiling to find code that is (almost) always executed on specific types and specializes the code based on this information. Type profiling provides very precise information which enables many optimizations, but cannot guarantee that the code will never be executed with different types in the future; thus the specialized code must contain type checks that detect and recover when unexpected types are encountered. Type inference, in contrast, deduces value types that must necessarily be correct and specializes the code based on these deductions. While the resulting specialized code does not require any online checks or recovery, the dynamic nature of these languages means that type inference may miss many opportunities for specialization that would be discovered by type feedback.

1.1 Key Insights

A natural question to ask is which one of type feedback or type inference is the more effective method. Agesen et al [8] compare these two methods head-to-head in their Self language implementation and found that there was no clear winner. They suggest that future work should explore how to combine these two methods rather than choosing between them. Hackett et al [18] take up this idea to explore combining these two methods for an efficient JavaScript language implementation. However, their combination went in only one direction: they used the type feedback information to help increase the effectiveness of type inference.

Our work shows that there is even more to be gained from combining type feedback and type inference in novel ways. In particular, we present two new strategies for combining the two:

- We show that type feedback can do an even better job of supporting type inference by separating function invocations according to the functions' type signatures, i.e., the types of the function arguments at the time of function invocation.

DLS '13, October 28, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2433-5/13/10. . . $15.00.
http://dx.doi.org/10.1145/2508168.2508177

- We show that, besides using type feedback to aid type inference as has already been explored, type inference can actually be used to support type feedback by using the inferred type information to more intelligently place type profiling hooks, thus significantly reducing profiling overhead.

1.2 Contributions

Our specific contributions are:

- We propose a novel language-agnostic way of combining type inference and type feedback for dynamic language runtimes (Section 3).
- We improve upon previous schemes for using type feedback to aid type inference by using a function's type signature to distinguish different function invocations (Section 4).
- We introduce a new scheme that uses type inference to lower the overhead of type feedback by enabling more intelligent placement of profiling hooks (Section 4).
- We implement our proposed schemes in a research JavaScript engine, MCJS. We evaluate this implementation on both the standard performance benchmarks (including Sunspider [5], V8 [6], and Kraken [4]) and on real-world websites (including popular websites like Amazon and BBC) and the JS1k demos [2] (Section 5).

We find that this mechanism results in speedups ranging from 1.2× to 4× over an implementation that does not perform type inference and type feedback based optimizations, across standard benchmarks. For web-replay benchmarks, which represent the JavaScript code executed when loading a website, function signature based type inference gives an average speedup of 5%. In the case of the JS1k demo benchmarks, which run for a longer duration, we observe an average speedup of 1.6×. Finally, using the types inferred by type inference, the type feedback in this mechanism inserts 23.5% fewer type feedback sites in the code.

2. Background and Related Work

Type inference and type feedback for dynamic scripting languages have been a topic of research for a number of years. In this section, we define these terms and give a brief overview of the current state of the art approaches in this area.

2.1 Defining Terms

Type inference. Type inference enables type specialization without any instrumentation of the code at runtime. The types of some subset of the local variables in a function can be inferred statically before it is JITed and executed. For example, an assignment statement `var a = 0;` means that, according to the language semantics, variable `a` must be of type `int` immediately after that program point. Any further expressions using `a` can be type specialized based on this deduction as long as `a` is not redefined with a different type. Type inference is usually performed as a whole-program analysis in statically-typed languages (where type inference was first developed). However, whole-program type inference for a dynamic scripting language is not practical because the type inference is usually done *online* during program execution, and this requires that the type inference process must be extremely fast. Therefore, dynamic language runtimes usually perform type inference on a per-function basis only for hot functions, detected adaptively during execution.

Type feedback. Type feedback enables type specialization by instrumenting the code at runtime and observing the types actually seen during execution. This process involves instrumenting the runtime to collect and store the types that are observed during execution. For example, an expression `a + b` may imply string concatenation, integer addition, or dictionary update based on the types of `a` and `b`. By profiling the types of `a` and `b` during several executions of this expression, the runtime can type specialize the operation during the subsequent executions for those types that are most often seen. If, for instance, the observed types for `a` and `b` are usually integers then the runtime can insert type specialized code that first checks whether the types of `a` and `b` are `int` and then performs integer addition directly. This type specialization using type feedback comes at a cost. First, collecting type information during the initial execution phases creates overhead with respect to both time and memory. Researchers have explored various ways of optimizing this by using different strategies like collecting coarse-grained data using sampling [21] and counters [9, 10]. Secondly, types need to be checked during the course of execution of the program using *guard instructions*. To tackle this problem, in runtimes like Google's V8 engine [17, 25], the runtime performs a secondary pass over the generated code to detect and eliminate redundant checks.

2.2 State of the Art

Compiler developers for the language Self [15] pioneered the concept of using type feedback for optimization of object-oriented dynamic languages. The Self compiler used an instrumented version of the program being executed to observe the types of the objects or *receiver classes* for every message pass or function call. The program was then specialized for the most frequently observed receiver class. Hölzle et al [19] discuss the implementation of polymorphic inline caches and various strategies used to select the candidate code for specialization. These strategies helped shape the design of the dynamic scripting language runtimes that followed.

PyPy is a mature Python implementation written in a subset of Python called RPython [12, 13, 23]. PyPy contains a tracing JIT compiler that uses runtime profile information to guide its tracing and eventual compilation of code paths. In contrast to this approach our algorithm explores the interdependency of type inference and type feedback to perform type specialization online on a per-function basis. The two approaches are orthogonal to each other and can be combined to improve type specialization.

Rubinius [24] is a Smalltalk-80-style VM and JIT compiler for Ruby. Though not as mature as PyPy, it uses a more traditional method-based JIT compiler. The compiler uses a simple form of type feedback in which they just emit guards to validate type assumptions. They rely on LLVM to perform the bulk of their optimization. Rubinius has a fast compiler that emits bytecode, they then compile the bytecode directly to LLVM IR. By going directly to LLVM IR Rubinius is not able to use Ruby-level semantic reasoning in optimization, thus losing the opportunity to perform high-level optimizations such as type inference.

The Crankshaft compiler [17] in Google's V8 JavaScript engine heavily relies on type feedback to generate specialized code. During the generation of a high-level intermediate representation (Hydrogen), each operation in an expression is specialized based on the observed types. Once this is done a set of other optimizations are performed including a static type inference pass to eliminate unnecessary guards. In contrast to this approach our runtime performs static type inference in two stages, one of which happens before profiling to reduce the profiling overhead. This not only reduces the amount of unnecessary profile information that is collected but also makes sure that the number of guards is reduced in the generated code. Another distinction is that we supply the function argument types to the type inference pass, which greatly increases its effectiveness.

The Jaegermonkey compiler in Mozilla's Spidermonkey engine performs fast hybrid type inference [18] based on the observed types in the previous runs. The expressions that are not type inferred are encapsulated in a *type barrier* and monitored during runtime. These expressions include global variables, function arguments, object property accesses, array element accesses, and function calls. If the observed type differs from the type used to specialize the code, the whole code is invalidated. Our approach differs from this approach because our algorithm uses the function type signatures as one of the inputs to our type inference algorithm to gain greater precision. Thus, the generated code does not have type barriers or guards around function arguments. In contrast to their approach, our algorithm performs type inference in two stages to significantly reduce the profiling overhead. Another difference between our approaches is that Jaegermonkey's type inference algorithm attempts to infer the type of objects and their fields as well. Our algorithm relies on type feedback for specializing operations on objects and their fields.

3. High-Level Overview

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

Figure 1 describes the general workflow of the language runtime system when executing a function, indicating the places where our proposed methods fit in. Our first type inference pass (**First TI**) takes as input the function's signature, i.e., the types of the function arguments for this specific invocation. Different signatures for the same function are handled independently from each other. The phase **First TI** uses the function signature in combination with the standard techniques for type inference. The results are used to place type profiling hooks in the code, and in the phase **Profile** those hooks are used to collect type information that is specific to a given function signature (i.e., the type profiling information is collected and stored separately for each signature of a given function). The phase **Second TI** takes the original function signature along with the collected type profile information and performs a second, more aggressive type inference based on the new information. Finally, the result is used to specialize and optimize the code for further execution.

3.1 Function Signatures

Typically, type inference algorithms use the syntactic structure of a function, combined with certain semantic rules of the language, to deduce type information—for example, the result of a left-shift operation is guaranteed to be an integer. However, in a dynamic language there are many operations that do not give any clues about types. For example, the '+' operator is polymorphic and provides no information to the type inference algorithm. We can improve the available information by providing types for the function's arguments. This idea is inspired by existing schemes for specialized function dispatch based on type signatures, such as multimethods[11, 16]. Our innovation is to make the function signatures an additional input to the type inference algorithm.

Figure 2 provides a motivating example. Signature-based dispatch operates as follows: After `foo` becomes hot, during the first call to `foo` a type-specialized version of `foo`'s body is created and then specialized to handle arguments of signature `(int, int)`. During the second call to `foo`, another version of `foo`'s body is created that is specialized to handle arguments of type `(string, string)`. Finally, before the third execution of `foo` the runtime determines that there is a match between the current call's signature and a previously-seen signature. It then re-uses the specialized body for `(int, int)` as the target of the call.

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

Figure 2. Motivating example

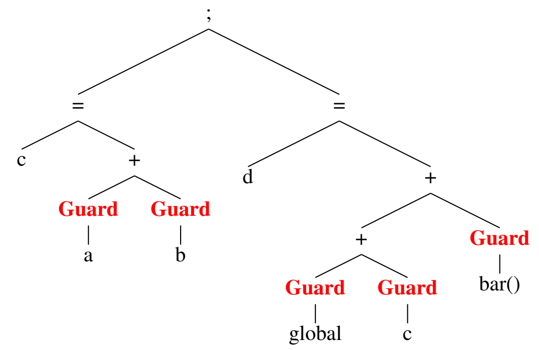

Figure 3. Intermediate representation of `foo` before type inference.

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

3.2 Phase First TI: Type Inference → Type Feedback

The goal of type feedback is to provide hints to the runtime and the JIT compiler about the types of variables. To do so, the runtime instruments the function's code with profile hooks that record type information observed during execution. These hooks are placed syntactically during the phase **Parse**, and show up as *guard nodes* in the abstract syntax tree (or IR) anywhere that type information may end up being useful (for example, on either side of a binary operator like '+'). See Figure 3 for an example of guard node placement for the function in Figure 2.

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

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

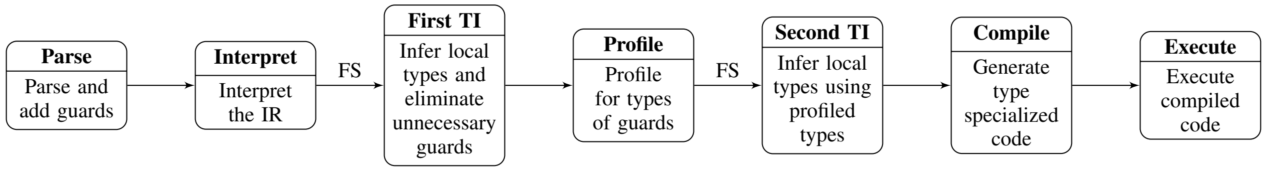
Parse
Parse and add guards
Interpret
Interpret the IR
FS
First TI
Infer local types and eliminate unnecessary guards
Profile
Profile for types of guards
FS
Second TI
Infer local types using profiled types
Compile
Generate type specialized code
Execute
Execute compiled code

Figure 1. A flow graph describing the execution phases of a function. FS stands for function signatures; TI stands for type inference.

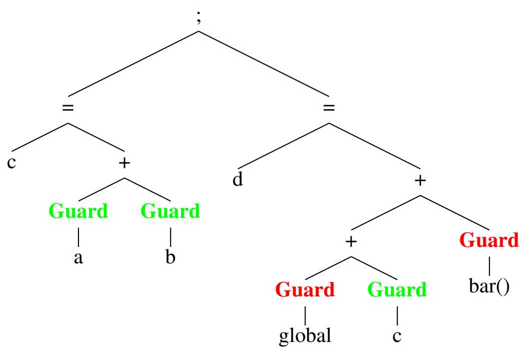

Figure 4. Intermediate representation of `foo` after type inference. Statically type inferred guards are in green.

For Figure 3, suppose that this function is called with a signature `(int, int)`. The first type inference pass is then able to infer types for the variables `a`, `b`, and `c`. Therefore, the guards around those variables are no longer useful and do not need to be profiled. Figure 4 shows the same function with eliminated guard nodes shown in green.

3.3 Phase Second TI: Type Feedback → Type Inference

In the last phase before code generation the runtime uses the type information generated by type feedback to perform a second, more aggressive type inference pass. This pass is identical to the first TI pass except that guard nodes have been annotated with type information supplied by type profiling, and the type inference algorithm uses those annotations instead of attempting to infer the types of the expressions under the guard nodes. This phase is similar to the existing work by Hackett et al [18] except that once again the type information is augmented by the function signature.

In Figure 2, suppose that type feedback shows that `global` is always of type `int` and `bar()` always returns a value of type `double`. The second type inference algorithm takes these types into consideration during type inference and thus can infer that variable `d` is type `double`. Since this assumption can be invalidated at any point in the future, the code generator places a type check to enforce the validity of the type feedback information. Consequently, almost all variables and expressions are type inferred at the end and only two guards are placed in this specialization of the function.

4. JavaScript Instantiation

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

4.1 MCJS JavaScript Engine

To evaluate our proposed ideas we use MCJS, a research JavaScript engine written in C♯. This subsection provides a summary of MCJS and its features. MCJS is a layered architecture, as shown in Figure 5. This means that the architecture splits responsibilities across a JavaScript-specific component and a language-agnostic lower-level VM. MCJS specifically uses the .NET Common Language Runtime (CLR) as the lower-level VM, as implemented by Mono [22]. The CLR provides traditional compiler optimizations such as instruction scheduling, register allocation, constant propagation, common subexpression elimination, code generation and machine specific optimizations. In addition, it provides managed language services such as garbage collection.

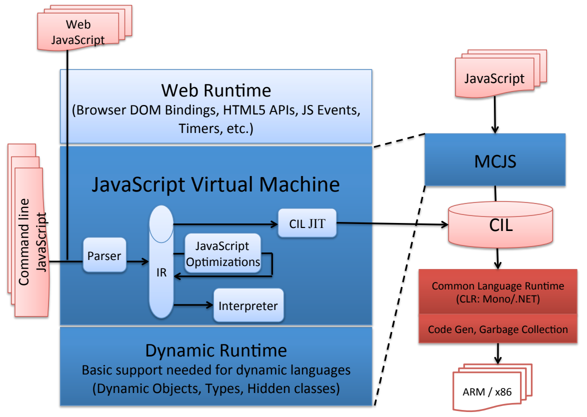
Web JavaScript
Web Runtime
(Browser DOM Bindings, HTML5 APIs, JS Events, Timers, etc.)
JavaScript
MCJS
JavaScript Virtual Machine
Command line JavaScript
Parser
IR
CIL JIT
JavaScript Optimizations
Interpreter
CIL
Common Language Runtime (CLR: Mono/.NET)
Code Gen, Garbage Collection
Dynamic Runtime
Basic support needed for dynamic languages (Dynamic Objects, Types, Hidden classes)
ARM / x86

Figure 5. MCJS JavaScript Engine Architecture. CIL = Common Intermediate Language, CLR = Common Language Runtime, and IR = MCJS Intermediate Representation

The JavaScript specific layer is a JavaScript VM implemented in C♯. The engine provides the standard dynamic language features such as dynamic values, objects, types, and hidden classes. Additionally, the engine includes the following major JavaScript-specific components and functionalities:

- **A JavaScript parser** takes in JavaScript code and generates a custom Intermediate Representation (IR) for each function in the code.
- **An interpreter** executes the IR directly for the cold functions during execution.
- **A JavaScript analysis engine** applies JavaScript-specific transformations and optimizations for hot functions. These JavaScript-specific optimizations include type analysis and type inference, array analysis, and signature-based specialization. These transformations augment the IR with extra information for more optimized code generation.
- **A Common Intermediate Language (CIL) bytecode generator** generates optimized CIL bytecodes for hot functions using the IR augmented by the previously mentioned transformations.

For this work, we extend the JavaScript-specific MCJS components to implement the ideas described in the previous section. MCJS already implements signature-based dispatch; the main changes we made were to add the type inference and type profiling phases (i.e., phases **First TI** through **Second TI** as described in the previous section).

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

Figure 6. `binb2b64` function from the crypto-sha1.js benchmark which is used to convert an array of big-endian words to a base-64 string.The **red** highlighting indicates the presence of Guard nodes around the expressions.

4.2 Parse Phase: Inserting Guard Nodes

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

Good candidates for type profiling include binary and unary operations, object property accesses, array element accesses, function calls, and the left-hand sides of assignments. Guard nodes are placed in all of these locations during parse time. However, recall that these are conservative placements—the initial type inference pass, described below, may statically infer types for some of these guarded expressions, in which case the associated guard nodes are marked so that the type profiler will ignore them. As an example, Figure 6 shows the function `binb2b64` from crypto-sha1.js. The **red** highlighting indicates the presence of guard nodes around the expressions.

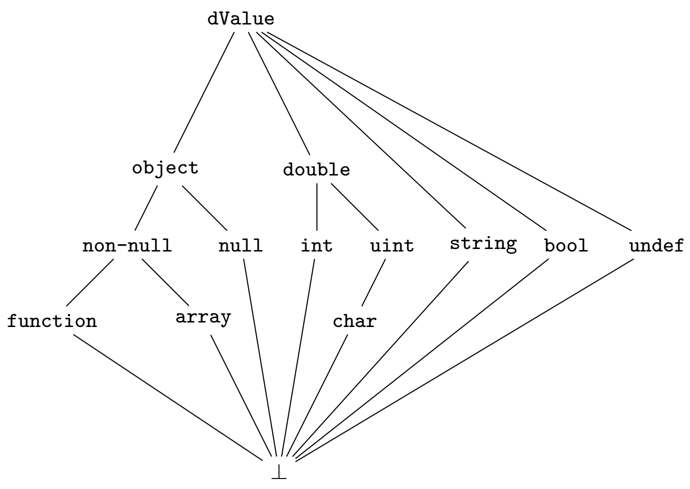

Figure 7. Type lattice used by our type inference algorithm.

4.3 First TI Phase: Initial Type Inference

Once a function with a particular type signature is deemed hot by the runtime, it is marked as a candidate for further optimization. The first step is an initial type inference pass. This pass will infer as many types as possible using the function signature and the type inference algorithm described by Figures 7 and 8 and Algorithm 1.

Figure 7 shows the type lattice used by the type inference algorithm. The most precise type is $\perp$, indicating an uninitialized value. For objects, we distinguish between function, array, null, and non-null values. For numbers, we distinguish between character, integer, unsigned integer, and double values. The least precise type is `dValue`, which stands for dynamic value—this is the default kind of value to use when the runtime has no static information about the value's type.

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

The types of the expressions in the worklist are inferred using a set of typing rules, a selected subset of which are given in Figure 8. This subset shows some of the more important inference rules used in the algorithm. The INT and BOOL rules show how constants in the code can be used to type an expression. Rules LSHIFT and GT show how type-specific operations can be used to guide type inference. The ADD rule uses a helper function **typeResolve** to determine the type of the add operation. The **typeResolve** function takes into consideration the implicit conversion rules of JavaScript and returns the appropriate resultant type of the operation. The VARASSIGN rule generates type constraints. Once all the constraints are collected, they are solved to assign types to the local variables. Finally, the GUARD rules correspond to the guard nodes inserted by the parser. The rules check the type of the expression it encloses; if the expression evaluates to `dValue` then the guard is marked to be profiled by the **Profile** phase, other-

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

```
worklist = []
for s in $\mathcal{S}$ do
    switch s.SymbolType do
        case local
            $\Gamma$(s) = $\bot$
            worklist.add(s.users)
        case parameter
            $\Gamma$(s) = $\tau$ from lookup($\mathcal{FS}$, s)
            worklist.add(s.users)
        case global
            $\Gamma$(s) = dValue
end for
while worklist.length $\neq 0$ do
    e = worklist.pop()
    typeEval(e)    ▷ Uses the rules from Figure 8 to infer types
end while
```

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

$$\tau \in \mathit{Type} = \{\texttt{dValue}, \texttt{object}, \texttt{double}, \texttt{non-null}, \texttt{function}, \texttt{array}, \texttt{null}, \texttt{int}, \texttt{uint}, \texttt{char}, \texttt{string}, \texttt{bool}, \texttt{undef}, \bot\}$$

$$\Gamma \in \mathit{Env} = \mathit{Variable} \to \mathit{Type}$$

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

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

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

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

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

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

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

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

Figure 8. Selected inference rules used in our type inference algorithm to generate type constraints. Algorithms 1 and 2 conflate the type constraint generation and constraint solving—in the algorithms, the VARASSIGN rule not only generates the type constraint, but also updates Γ with the resultant type and pushes the *users* of the variable x into the worklist. **typeResolve** is a helper function that takes into consideration the implicit conversion rules of JavaScript and returns the appropriate resultant type of the operation.

wise that guard node will be ignored by the **Profile** phase. This rule eliminates many unnecessary guard nodes, significantly increasing the profiler's performance.

As an example, Figure 9 shows the function from Figure 6 after the initial type inference with the inferred types and the eliminated guard nodes.

4.4 Profile Phase

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

- Disable profiling of IR nodes that are highly dynamic in nature: The profiler stops tracking the IR nodes that show highly dynamic nature, such as rapidly changing type information. We observe this behavior in some code snippets which iterate over the fields of an object. For such guard nodes, the profiler records the profiled type as `dValue` and stops profiling them.
- Efficient data structures: While designing the profiler we observed that the performance of the profiler depends heavily on the data structure that is used. In particular, we use an array-based implementation of the profiler which significantly outperforms a dictionary-based implementation.
- Selectively enabling the profiler: We observe that many functions execute only once during the initialization phase of the JavaScript application. Therefore, we enable the profiler only during the sixth invocation of the function code. By doing this we ensure that we only collect profiles for functions that are potentially hot.

4.5 Second TI Phase: Final Type Inference

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

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

The type inference algorithm uses the same inference rules as in Figure 8 except for the GUARD rule. The new GUARD rule is:

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

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

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

Figure 9. binb2b64 function after the first type inference pass. The **red** highlighting indicates the presence of guard nodes around the expressions that need to be profiled. The **green** nodes indicate that the guard nodes around these expressions are unnecessary and should not be profiled. The ⟨**type**⟩ indicates the type inferred by the type inference algorithm.

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

Figure 10. binb2b64 function after the first type inference pass. The **red** highlighting indicates the presence of Guard nodes which were profiled and ⟨**type**⟩ indicates the type profiled by the profiler. The ⟨**type**⟩ indicates the type inferred by the type inference algorithm after the first pass. The ⟨**type**⟩ indicates the type inferred by the type inference algorithm after second pass.

As an example, in Figure 10 we see that `str` is now type inferred to be a string based on the observed types of guards around the `b64pad` and `tab.charAt()` expressions.

4.6 Compile Phase: Specialized Code Generation

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

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

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

```
worklist = []
for s in $\mathcal{S}$ do
    switch s.SymbolType do
        case local
            if $\Gamma_1(s) \in$ PreciseTypes then
                $\Gamma$(s) = $\Gamma_1$(s)
            else
                $\Gamma$(s) = $\bot$
                worklist.add(s.users)
            end if
        case parameter
            if $\Gamma_1(s) \in$ PreciseTypes then
                $\Gamma$(s) = $\Gamma_1$(s)
            else
                $\Gamma$(s) = $\tau$ from lookup($\mathcal{FS}$, s)
                worklist.add(s.users)
            end if
        case global
            $\Gamma$(s) = dValue
end for
while worklist.length $\neq$ 0 do
    e = worklist.pop()
    typeEval(e)    $\triangleright$ Uses the rules from Figure 8 to infer types
end while
```

In the case of the example in Figure 10, since `str` is now type inferred as a `string`, the code generator does not add checks around it. With `str` being a local variable, we know its type is only influenced by the observed types of `b64pad` and `tab.charAt()`. Since we already have runtime checks around them, it is unnecessary to check for the type of `str` as well. This small optimization enables the runtime to reduce the number of unnecessary checks in the code. The total number of checks in the final CIL code for `binb2b64` is reduced from nine to seven.

5. Evaluation

In this section we describe our evaluation strategy and compare various combinations of our optimizations against a baseline MCJS implementation. Throughout this section, we abbreviate type inference as TI, type feedback as TF, function signatures as FS and guard elimination as GE. We indicate whether the optimizations are enabled or disabled using + or - respectively. Table 1 shows the MCJS configurations on which these experiments are carried out.

We choose the TI- TF+ FS- GE- MCJS configuration because it is in the same vein as V8's strategy for performing type specialization. In this configuration, MCJS performs pure type feedback without any type inference. Though V8's Crankshaft compiler performs various other optimizations and performs a variation of type inference based on the profiled types, we believe this configuration is a fair representation of Crankshaft's type specialization strategy based on the ordering of different phases.

We choose the TI+ TF+ FS- GE- MCJS configuration because, it is in the same vein as the SpiderMonkey's strategy of performing type specialization. In this configuration, MCJS performs type feedback based type inference without considering the types in function signatures. The types of function arguments are initialized to `dValue` during the type inference phase. Though SpiderMonkey's Jaegermonkey compiler performs various other optimizations such as single pass SSA transformation, we believe that MCJS in this configuration is a fair representation of Jaegermonkey's type specialization strategy based on the ordering of different phases.

We emphasize that we *are not* comparing MCJS with Crankshaft or Jaegermonkey directly. Rather, we compare different type specialization strategies that happen to be used by these engines, among many other optimizations that they implement. No direct conclusions can be drawn from our evaluation about the relative merits of these engines.

5.1 Experimental Methodology

We evaluate our optimizations on an AMD FX-6200 Hexa-Core 3.8GHz machine with 10GB RAM. We use Mono 3.0 as our underlying CLR implementation for MCJS. We choose popular JavaScript benchmark suites including Sunspider [5], V8[1] [6] and Kraken [4] as well as JavaScript code from 17 real world websites and web applications to evaluate our implementation. Each of the benchmarks is run 11 times and the data from the last 10 runs is averaged to compute mean performance. We describe the benchmarks in detail in the following subsections.

5.2 Standard Benchmarks

Figure 11 shows the relative speedup of different MCJS configurations with respect to the base configuration for the Sunspider, V8, and Kraken benchmarks. Since Sunspider benchmarks run for a relatively short period of time, each benchmark is repeated 20 times in a loop. Unlike Sunspider, the V8 and Kraken benchmarks run for a relatively longer period of time. Therefore, they are run without modification.

5.2.1 Sunspider

Figure 11 shows that MCJS with the TI+ TF+ FS+ GE+ configuration performs extremely well compared to other configurations for the Sunspider benchmarks, with an average speedup of **4.2**$\times$. The average execution time for the base configuration is 48.4 seconds, and the execution times range from 2 seconds for `bitops-bitwise-and` to 545.3 seconds for `string-tagcloud`.

Type inference provides a significant performance boost for these benchmarks, because they heavily rely on integer arithmetic. The compiler maps those JavaScript numbers that are inferred to be integers to CLR integer primitives. This optimization enables type specialized x86 integer operations in the generated code. This further enables x86 specific optimizations such as common subexpression elimination and faster integer arithmetic using bitwise shift operators. Therefore, benchmarks like `bitops-3bit-bits-in-byte`, `bitops-bits-in-byte` and `math-spectral-norm` perform an order of magnitude better using the TI+TF+ FS+ GE+ configuration.

When compared to the TI- TF+ FS- GE- strategy, all of the type inference based approaches perform better. This can be attributed to the constant boxing and unboxing of values required by the pure type feedback based approach. When compared to the TI+ TF+ FS- GE- strategy, the configurations with function signatures enabled perform better. This shows that use of function signatures during type inference improves application performance.

5.2.2 Kraken

We see an average speedup of **1.3**$\times$ for the TI+ TF+ FS+ GE+ configuration. The run times vary from 0.5 seconds to 559 seconds for the base configuration with an average execution time of 66.1 seconds.

In contrast to Sunspider, the Kraken benchmark suite heavily relies on global arrays and array element manipulation. The TI+ TF+ FS+ GE+ configuration performs well on the crypto subset

[1] MCJS does not support typed arrays. Therefore, we do not evaluate our implementation on Octane benchmarks.

MCJS configurations	Description
TI- TF- FS+ GE-	MCJS does not perform type inference or type feedback. We use this as our baseline for measuring speedup.
TI- TF+ FS- GE-	Enabling type feedback and disabling type inference in MCJS.
TI+ TF+ FS- GE-	Enabling type inference and type feedback and disabling function signature based TI in MCJS. In this configuration MCJS performs both the type inference passes with the types from function arguments set to `dValue`. These arguments are profiled during the profile phase.
TI+ TF- FS+ GE-	Enabling type inference and disabling type feedback in MCJS. In this configuration MCJS performs only the first pass type inference.
TI+ TF+ FS+ GE-	Enabling type inference and type feedback in MCJS. In this configuration the guard elimination is not enabled.
TI+ TF+ FS+ GE+	Enabling all of the type-inference-based optimizations in MCJS including guard elimination.

Table 1. Table describing various MCJS configurations which we perform our experiments on. TI = Type Inference, TF= Type Feedback, FS = Function Signature, GE = Guard Elimination, and +/- indicate whether the respective features are enabled or not.

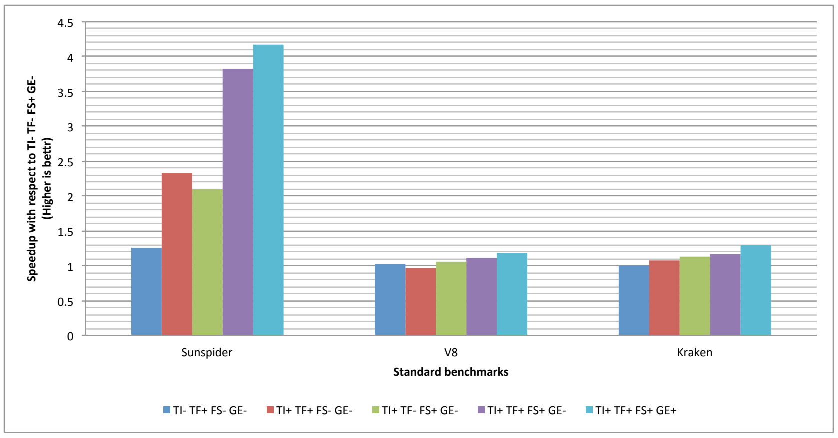

Figure 11. Speedup with respect to TI- TF- FS+ GE- configuration for standard benchmark suites: Sunspider, V8 and Kraken. TI = Type Inference, TF= Type Feedback, FS = Function Signature, GE = Guard Elimination, and +/- indicate whether the respective features are enabled or not.

of the benchmarks, giving an average speedup of **1.6**× over the base configuration. The optimizations are less effective for the rest of the benchmarks that rely heavily on array element manipulation. Since the type inference algorithm does not infer the types of global symbols, the global array access operations are not very optimized. We are currently investigating ways to extend our type inference algorithm to infer the types of arrays to optimize these benchmarks.

5.2.3 V8

We see an average speedup of around **1.2**× with the TI+ TF+ FS+ GE+ configuration for the V8 benchmarks. The run times vary from 6.1 seconds to 124.1 seconds with an average execution time of 36.1 seconds. The V8 benchmarks pose a different challenge from the Kraken benchmarks since most of them deal with global objects and property accesses. These expressions are also not type inferred by our algorithm. Though MCJS using TI+ TF+ FS+ GE+ performs well for the `splay`, `navier_stokes`, and `raytrace` benchmarks, with an average speedup of **1.5**× over the base configuration, it performs rather poorly on `regexp` and `deltablue`. This poor performance is mostly due to MCJS's inefficient regular expression and string library implementation. Our algorithm does not type the properties of an object and precisely tracking such information is difficult. We are currently working on extending our type inference algorithm to approximately infer types of object properties.

For both the V8 and Kraken benchmarks, MCJS with guard elimination and function signatures enabled does *not* perform significantly better than the other strategies. This is because MCJS engine spends most of the time executing inefficient string and regexp libraries. Therefore, the optimizations due to type specialization do not show any effect on the final execution time.

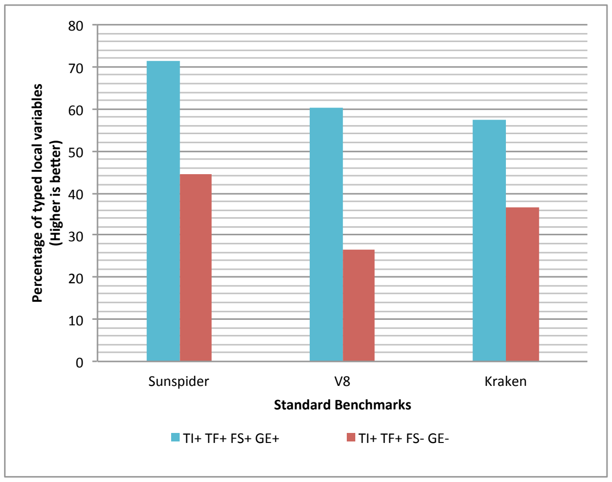

Figure 12. Percentage types inferred. TI = Type Inference, TF= Type Feedback, FS = Function Signature, GE = Guard Elimination, and +/- indicate whether the respective features are enabled or not.

5.2.4 Effect of Function Signatures

The percentage of type inferred variables is another important metric which shows the effectiveness of including function type signatures in our algorithm. Figure 12 shows the percentage of types inferred in the TI+ TF+ FS+ GE+ and the TI+ TF+ FS- GE+ configurations for various benchmark suites. The use of function signatures during type inference improves the percentage of types inferred. For Sunspider, the percentage of local variables that are type inferred increases from **44% to 74%**. There is a significant increase in this number from **26% to 60%** for the V8 benchmarks. The Kraken benchmarks also show an increase of **21%** in percentage of local variables that are type inferred.

The percentage of types inferred does not directly correspond to the speedup obtained for V8 and Kraken benchmarks, because these benchmarks spend a majority of the time in unoptimized parts of the MCJS engine. For example, the JavaScript standard libraries for strings and regular expressions are extensively used by these benchmarks.

5.3 Real-world Benchmarks

Apart from the standard benchmark suite, we test our implementation on 17 real-world websites and web applications. We use the record-and-replay feature of Zoomm [14], a research web browser, to collect the traces of JavaScript that are executed in real-world websites like Amazon, BBC, CNN, Google, Guardian and ESPNCricinfo at load time. These traces are then converted to pure JavaScript files by simulating the DOM objects and their properties in terms of JavaScript objects. Since most of the JavaScript execution happens at page load, the overhead of performing profiler based TI optimizations is not amortized for most of these benchmarks.

Therefore, we also use 11 benchmarks from demos submitted to the JS1k [2] competition. These benchmarks are relatively long running JavaScript applications when compared to the web-replay benchmarks. Though these benchmarks are relatively small in size, we believe that they are representative of core functionalities present in JavaScript heavy web-apps like games and animations. For these benchmarks, the DOM interactions are stubbed out and simulated using pure JavaScript objects. For the benchmarks that require user interaction, the events are simulated by providing them a fixed set of JavaScript event objects in a loop. The `setTimeout` and `setInterval` functions are replaced by loops that call the supplied function for a fixed number of iterations. We describe the nature of these benchmarks in Table 2.

5.3.1 Web-replay benchmarks

Figure 13 shows that our profiler based optimizations do not speedup the web-replay benchmarks, which are the first six benchmarks in the graph (indeed we see slowdown in some cases). This property is seen across all configurations which use type feedback, i.e., TF+. This is mainly because the specialized code is not executed long enough to amortize the overhead caused by profiling. Though web-replay benchmarks execute for an average of 4.1 seconds, most of the functions are executed only a few number of times. Therefore, the web-replay benchmarks are not optimized by type feedback.

But MCJS with function signature based type inference, i.e., TI+ TF- FS+ GE- (green bar) configuration shows speedup in most of the benchmarks with an average speedup of **5%**. This shows that a quick function signature based type inference performs well even during page load.

5.3.2 JS1k Demos

The final 11 benchmarks shown in Figure 13 are the JS1k demos. These benchmarks run for a relatively longer period of time with an average run-time of 11.3 seconds. Most of these benchmarks perform well with type feedback enabled, except for a few exceptions like `Conways` and `Sierpinski gasket` where type inference without type feedback performs better.

Like most of the JavaScript in popular websites, these benchmarks are minified using JavaScript minifiers like Google Closure Compiler [1] or JSCrush [3]. In addition to minifying, some of the benchmarks use global symbols in order to save space. The rest of them maintain local symbols in the functions. Therefore, we see varied behavior across configurations. `Kaboom`, `Spring pond`, `Tetris`, `Wave graph`, `Breakout`, and `Flying windows` use global variables heavily in their code. Therefore, type inference with type feedback performs better than type inference without type feedback.

In comparison to other strategies, our best strategy with all features enabled (TI+TF+FS+GE+) consistently performs better, especially on `Kaboom`, `Spring Pond`, `Breakout`, and `Flying Windows`. For `Breakout` and `Spring Pond`, type inference benefits both the TI+TF+FS-GE- strategy and our best strategy as compared to TI-TF+FS-GE-. However, for all these specific benchmarks, the main advantage of our strategy compared to other strategies stems from the combination of signature based type inference and guard elimination as can be seen on Figure 13.

5.3.3 Effect of Function Signatures

Figure 14 shows that the effect of using function signatures in type inference is similar to that observed for the standard benchmark suite. For the benchmarks that heavily rely on global variables, the use of function signatures during the type inference does not always give major benefit. For example, for benchmarks `Kaboom`, `Mandelbrot`, `Wave graph`, and `Conways`, even though the percentage of types inferred with function signatures is higher than without, we do not see much difference in execution time. This can be attributed to typing of variables that are non-critical for performance. In these benchmarks we observe that the the performance sensitive parts of the code like loops use a combination of global variables and local variables that are not typed in the first type inference phase.

5.4 Effect of Guard Elimination

The number of guards eliminated is an indicator of the effectiveness of the profiler. We measure this by collecting the number of unique guard nodes that are profiled during the execution of the program. We then compare the number of guard nodes eliminated due to the guard elimination technique.

Table 3 shows the percentage of guards eliminated due to guard elimination for each of the benchmark suites. On an average guard elimination results in **23.5%** reduction in guards profiled during the profiling phase. As Figure 11 and Figure 13 show, elimination of guards improves performance. It also helps reduce the amount of the information collected during runtime, thereby reducing the total memory used by the application. The elimination of guards (hence conditional control paths) from the CIL also creates more optimization opportunities for the underlying VM's code generator.

5.5 Effect of First TI Phase

There are two advantages of the first TI phase. First, it helps in reducing the number of guards that are profiled. This reduces the time spent during collection of type profiles as well as the time required to do the dynamic type checks. Secondly, it speeds up the operations involving the variables that are type inferred, during the profiling phase. Our best strategy (TI+ TF+ FS+ GE+) is $2.1\times$ and $1.1\times$ faster than the strategy without the first TI phase for Sunspider benchmarks and real-world benchmarks respectively.

Benchmarks	Description
Kaboom	JavaScript version of the classic arcade game Boom
Mandelbrot	Animation of classic mandelbrot with user clickable interface for zooming.
Spring pond	Algorithm that simulates the evolution of species of fishes in a pond and survival of the fittest.
Tetris	JavaScript version of the classic Tetris game.
Wave graph	Graph plotting application that plots continuous multicolored sinusoidal waves.
Breakout	JavaScript version of the paddle and ball game.
Conways	Animation simulating the Conway's game of life algorithm.
Flying windows	Animation showing flying windows.
Loading spinner	Spinner animation shown during page load.
Sierpinski gasket	3D representation of Sierpinski gasket fractal.
Analog clock	Analog clock written in pure JavaScript and HTML.

Table 2. Table describing the nature of the JS1k demos used as benchmarks.

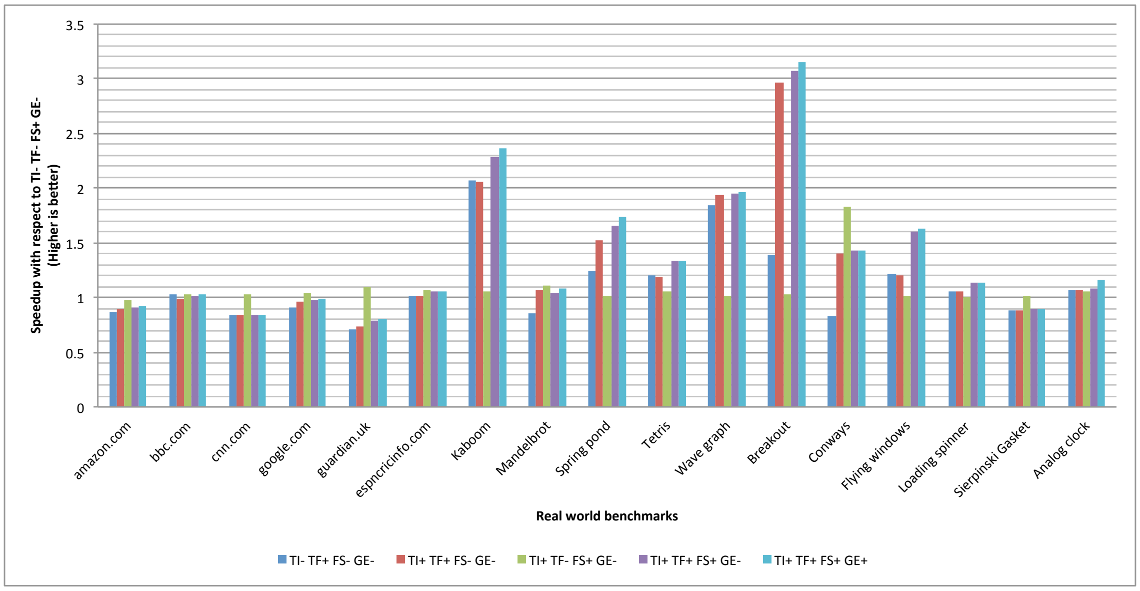

Figure 13. Speedup of various configurations of MCJS with respect to TI- TF- FS+ GE- for real world benchmarks. TI = Type Inference, TF= Type Feedback, FS = Function Signature, GE = Guard Elimination, and +/- indicate whether the respective features are enabled or not.

Benchmarks	GE-	GE+	% reduction in guards
Sunspider	1203	872	27.5
V8	2053	2004	2.4
Kraken	294	177	39.8
Web replay	62	50	19.4
JS1k Demo	177	127	28.2
Average			**23.5**

Table 3. Percentage of guards reduced due to guard elimination for each kind of benchmarks. GE+ indicates the configuration TI+ TF+ FS+ GE+ and GE- indicates the configuration TI+ TF+ FS+ GE- shown in Table 1. Numbers on columns two and three are absolute numbers of guards (type checks) across the corresponding benchmark suite and the configuration.

6. Conclusion

We explore the phase interdependency between the two most important methods used for type specialization of dynamic languages, type feedback and type inference. Our analysis shows that type feedback can improve the accuracy of type inference (as shown in previous work), but also that type inference can also significantly reduce the overhead of type feedback (during profiling) and type checks (during execution), resulting in overall more accurate and faster type analysis.

This paper proposes a novel strategy for combining type inference and type feedback in a way that reduces the overhead and improves the performance of both methods. In this strategy, two passes of type inference are applied, both before and after type feedback (profiling). The first type inference pass significantly reduces the profiling overhead during the type feedback phase. On the other hand, the reduced type feedback collected is then used by the second type inference pass to highly specialize the generated code. The key enabler for this multi-phase efficiency is syntactic guard instructions inserted at parse time into the IR, representing the possible profiling sites. These guards nodes are pruned and marked during the first type inference and type profiling phases. This combined strategy also employs function type signatures to further improve the accuracy and reduce the overhead of both type inference and type feedback methods.

We evaluate the proposed combined function signature based type inference and type feedback strategy on a large set of traditional benchmarks (including Sunspider, Kraken and V8) and re-

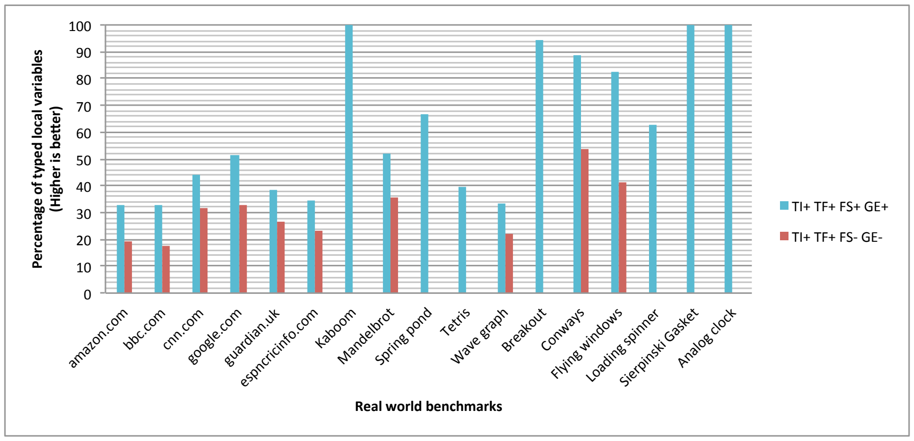

Figure 14. This graph shows percentage of typed local variables for configurations TI+ TF+ FS+ GE+ and TI+ TF+ FS- GE+ for real-world benchmarks. TI = Type Inference, TF= Type Feedback, FS = Function Signature, GE = Guard Elimination, and +/- indicate whether the respective features are enabled or not.

alistic web application benchmarks (including Amazon, BBC and JS1k demos). The results show that our proposed method speeds up the standard benchmarks by between 1.2× and 4.2× over the base implementation that does not perform type feedback or type inference based optimizations. For web-replay benchmarks, which represent the JavaScript code executed during website load, simple function signature based type inference gives an average speedup of 5%. In the case of JS1k demo benchmarks, which run for a longer duration, we observe an average speedup of 1.6×. Further more, this combined strategy is able to infer the types of symbols in the hot functions very accurately (between 60% and 80% of all variables) for both standard benchmarks and web applications. Moreover the combined strategy greatly reduces the overhead of both type profile sites during profiling and type checks during execution (by about 23.5%).

Acknowledgments

We would like to thank Călin Caşcaval and the MCJS team at Qualcomm Research Silicon Valley (QRSV), as well as the anonymous reviewers for their valuable input. The work described in this paper was significantly supported by Qualcomm Research.

References

[1] Google closure compiler. `https://developers.google.com/closure/compiler`.

[2] Js1k. `http://js1k.com`.

[3] Jscrush minifier. `http://www.iteral.com/jscrush`.

[4] Kraken benchmark suite. `http://krakenbenchmark.mozilla.org`.

[5] Sunspider benchmark suite. `http://www.webkit.org/perf/sunspider/sunspider.html`.

[6] V8 benchmark suite. `http://v8.googlecode.com/svn/data/benchmarks/v7/README.txt`.

[7] O. Agesen. *Concrete type inference: delivering object-oriented applications*. PhD thesis, Stanford, CA, USA, 1996. UMI Order No. GAX96-20452.

[8] O. Agesen and U. Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *ACM SIGPLAN Notices*, volume 30, pages 91–107. ACM, 1995.

[9] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. . URL `http://doi.acm.org/10.1145/378795.378832`.

[10] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. In *Proceedings of the IEEE, 93(2), 2005. special issue on program generatation, optimization, and adaptations*, 2004.

[11] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common lisp object system specification. *ACM Sigplan Notices*, 23(SI):1–142, 1988.

[12] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[13] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Runtime feedback in a meta-tracing jit for efficient dynamic languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, page 9. ACM, 2011.

[14] C. Cascaval, S. Fowler, P. Montesinos-Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar. Zoomm: a parallel web browser engine for multicore mobile devices. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 271–280, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. . URL `http://doi.acm.org/10.1145/2442516.2442543`.

[15] C. Chambers. *The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages*. PhD thesis, Stanford University, 1992.

[16] C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(6): 805–843, 1995.

[17] Crankshaft compiler. V8 engine. `http://www.jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler`, 2013.

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

[19] U. Hölzle. *Adaptive optimization for SELF: reconciling high performance with exploratory programming*. PhD thesis, Stanford University, 1995.

[20] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. *ACM SIGPLAN Notices*, 29(6):326–336, 1994.

[21] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, July 1996. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/233561.233562`.

[22] Mono. Xamarian Inc. Mono. `http://www.mono-project.com/Main_Page`, 2013.

[23] PyPy. PyPy Status Blog. `http://morepypy.blogspot.com`, 2013.

[24] Rubinius. Rubinius Blog. `http://rubini.us/blog`, 2013.

[25] v8. Google Inc. V8 JavaScript virtual machine. `https://code.google.com/p/v8`, 2013.