# Efficient Just-In-Time Execution of Dynamically Typed Languages Via Code Specialization Using Precise Runtime Type Inference

Mason Chang Michael Bebenita Alexander Yermolovich Andreas Gal

Michael Franz

Department of Computer Science
University of California, Irvine
Irvine, CA 92697-3450

**Abstract**

Dynamically typed languages such as JavaScript present a challenge to just-in-time compilers. In contrast to statically typed languages such as JVML, in which there are specific opcodes for common operations on primitive types (such as `iadd` for integer addition), all operations in dynamically typed language such as JavaScript are late-bound. Often enough, types cannot be inferred with certainty ahead of execution. As a result, just-in-time compilers for dynamically typed languages have tended to perform worse than their statically-typed counterparts. We present a new approach to compiling dynamically typed languages in which code traces observed during execution are dynamically specialized for each actually observed run-time type. For most benchmark programs, our prototype JavaScript virtual machine outperforms every other JavaScript platform known to us.

## 1 Introduction

Due to the popularity of languages such as JavaScript, Python, Ruby, and PHP, the *dynamically typed* programming language has emerged as the paradigm of choice for client-server computing. With the introduction of Asynchronous JavaScript And XML (AJAX), JavaScript in particular has emerged as the dominant language on the web. Many of the top sites on the web present a rich user experience that relies heavily on JavaScript support, and JavaScript is also the language behind Adobe's Flash technology.

This development has caught the developers of just-in-time compilers somewhat off-guard. In the original roadmap for the web, generally accepted until quite recently, executable content was supposed to be provided by *statically typed* languages such as Java Virtual Machine Language and the .NET Common Intermediate Language. Over the past decade, powerful just-in-time compiler frameworks have been developed that handle these statically typed languages surprisingly well.

Unfortunately, in the very recent past, there has been almost no activity in executable web content based on statically typed intermediate languages. Instead, there has been an explosion of executable content in dynamically typed languages. According to Mozilla's Chief Technical Officer, Brendan Eich [1], a typical web browser today executes orders of magnitude more JavaScript than Java, both in code volume as well as in dynamic instructions. Yet the irony is that most browsers contain highly optimized and ambitious JVML execution engines and only fairly mediocre JavaScript execution engines, many of which still only use interpretation.

While recent implementations of JavaScript such as Tamarin [12] and Rhino [10] have begun to use just-in-time compilation, these compilers perform significantly worse than their counterparts for statically typed languages such as JVML. The main reason for this discrepancy is precisely the untyped nature of JavaScript. While a JVML program contains primitive operations such as `iadd` that tell the virtual machine directly that two integers need to be added (and which can be translated directly into a corresponding machine instruction), every operation in a JavaScript program is

late-bound, i.e., depends on the dynamic type of the argument which can change even between successive executions of the exact same instruction. Hence, a costly dynamic dispatch is needed that cannot be optimized away completely.

What can, however, be done is to *specialize* the code for the most common dynamic types of receiver arguments. For example, we can create a special version of a program fragment in which we assume that a certain variable is an integer. In that special version, we can then map an `add` operator directly to integer addition, and can even perform arithmetic simplifications on the code. In theory, every possible combination of types for all variables involved might occur, but compilers use *type inference* to reduce the number of such type combinations quite effectively. However, there are always cases in which a run-time type cannot be inferred statically.

Our contribution in this paper is a new technique to efficiently generate specialized code fragments for dynamically typed languages. Our method is a trace-based compilation technique that only ever compiles hot loops and that is able to handle alternative paths through such loops. Using this technique, we treat every possible combination of run-time type in a JavaScript program as a possible path through the loop. Our dynamic compiler will detect the few cases that actually do appear with significant frequency and create specialized code fragments for them. All not-so-common cases will be ignored and executed only by interpretation if and when they do occur.

Our implementation achieves this by first translating the untyped JavaScript code into a strongly typed intermediate representation. In particular, we translate JavaScript into a specific form of JVML that uses double dispatch [8] to choose the correct semantics based on the run-time types of *both* operands of dyadic operations. Such double dispatch would normally incur a high execution time penalty. However, our underlying just-in-time compiler is able to dynamically detect and optimize frequently executed code traces and inline all dynamic invocations that occur along such a trace.

This has the net effect that for specific cases of dynamic type combinations that occur frequently in a program, *both* of the dynamic dispatches involved in our "double dispatch" scheme will lie "on trace", enabling them to be optimized away completely. As an end result, our compiler generates code that is virtually as efficient as that for statically-typed languages.

In the following sections, we first briefly explain how traditional compilers handle dynamically typed languages using type inference (Section 2). We then present our "double dispatch" approach, which appears cumbersome at first sight but whose inefficiencies can be optimized away completely by subsequent trace-based compilation (Section 3). We explain how our trace compiler handles alternative paths through the same loop (Section 4). If variables can have several dynamic types in a loop, each combination of types represents one potential path through the loop. We briefly outline our implementation (Section 5). After a presentation of related work (Section 6), we present highly encouraging benchmark results (Section 7). We conclude the paper with an outlook to future work.

# 2 Static Type Inference

In traditional compilers for dynamically typed languages, static type inference is used to predict the runtime type of untyped variables. Figure 1 shows a simple JavaScript code example. The value of the loop variable $i$ is accumulated in variable $x$. Since $i$ starts out as an integer (constant 0), and is incremented by an integer ($i++$), $i$ will always be an integer in this loop. Since $x$ also starts out as an integer (constant 0) and only integers are added to it, it also can be represented as an integer.

To infer this type information, many compilers for dynamically typed languages use iterative data-flow analysis. First, all known types are flagged. The types of constants, for example, are usually known, and thus in our example we would flag the results of the initial assignments to $x$ and $i$ as integers. Through subsequent analysis it can then be discovered whether these initial type assignments are preserved by every operation that executes on the variables.

To guarantee correct semantics, this type inference process must be *precise*. If the analysis infers an integer type for a variable, for example, but that variable can become a floating point value at some point, the generated machine code would not be able to handle this case correctly since it would use an integer register to hold the value.

The precision requirement often interferes with generating efficient machine code. In JavaScript, for example, the addition of two integers ($+_{\text{INT}}$) actually returns a floating point DOUBLE value if the additions overflow the range representable by integers. This significantly complicates static type inference, since we can no longer safely infer that the result of an addition of two integers will always produce an integer. It will produce an integer only as long no overflow occurs. Otherwise, a floating point value will be returned.

```
var x = 0;
for (i = 0; i < 1000000000; i++) {
    x = x + i;
    if (x == 500000000) x = "hello";
}
```

Figure 1: A simple accumulation loop in JavaScript. Through the initial assignment of 0, the variable $x$ has the type `int` at the loop entry. Variable $x$ will remain of type `int` until it eventually exceeds the value range of the integer type. At this point, the addition operator will automatically produce a result of type `double`, which then changes the type of $x$ into a `double`.

In these cases, when static type inference cannot precisely predict a single type for a variable, it assigns the *any* type, which at runtime is implemented using an object that can store any type. Operating on variables of this type is of course significantly more expensive at runtime since the appropriate operation has to be selected dynamically depending on the dynamic type of the variable at that particular point in the program execution. Thus, using standard static type inference techniques, it would be impossible to compile this loop into efficient code.

# 3 Deferred Runtime Type Inference Via Double Dispatch

We use an alternative approach for the efficient compilation of dynamically typed programming languages such as JavaScript. Instead of attempting to infer type information statically ahead-of-time, we translate the dynamically typed code into a statically typed intermediate representation in which the semantics of dynamic typing are modeled precisely by a "double dispatch" scheme [8]. For every operation (i.e. $+$), "double dispatch" considers *both* operands to select the correct semantics to be applied.

At first sight, double dispatch thereby pushes the burden of dynamic typing entirely onto the runtime system. However, in conjunction with trace-based compilation, this method actually results in better code because the overhead of both dispatches can be optimized away entirely in all cases where this would be worthwhile.

Double dispatch involves boxing all dynamic types of the source language (in our case JavaScript) as Java objects, i.e. *JsInt* to store integers and *JsDouble* to store floating point values. Each class used to represent JavaScript values derives form an abstract interface *JsAny*. The interface of *JsAny* defines an (abstract) virtual method for each operation that can be performed on a value. Each operation is implemented for the generic case [i.e. *add*(*JsAny*)], which is invoked when the type of the 2nd operand is not known, and a series of specific implementations for each concrete JavaScript type (Figure 2).

To select the concrete implementation of an operation based on the types of both operands, we use two subsequent virtual method calls (Figure 3). Initially, the declared type of both operands $x$ and $y$ is *JsAny*, which means no static type information is available or required at this point. To execute an addition between $x$ and $y$, for example, we first invoke $x$.*add*(*JsAny* $y$). Since the type of $y$ is not known precisely at this point, only the generic signature of the *add* method matches this invocation. Invoking *JsAny.add*(*JsAny*) will execute the implementation of *add*(*JsAny*) in *JsInt*, since even though $x$ is only known to be of (sub-) type of *JsAny*, its precise type is actually *JsInt* since it is a boxed integer value. Through the virtual method dispatch we have effectively redirected execution to a specific implementation based on the type of the first operand.

The concrete type of the 2nd operand is still only known as a (sub-) type of *JsAny* at this point. Thus, we perform a 2nd virtual method call inside *JsInt*::*add*(*JsAny* y), which will invoke *add* on $y$. In contrast to the first invocation, however, $x$ which we pass as parameter to *add* is now known by its concrete type due to the fact that we perform the invocation from inside a method implementation of $y$'s concrete *JsInt* type. Thus, we pass $x$ as *this* to the 2nd invocation of *add*, which reveals the precise type of $x$ and thus matches one of the concrete signatures (in this case *JsAny.add*(*JsInt*)).

The actual implementation of the integer addition is located in *JsInt.add*(*JsInt*), which is executed as a result of the 2nd virtual method call.

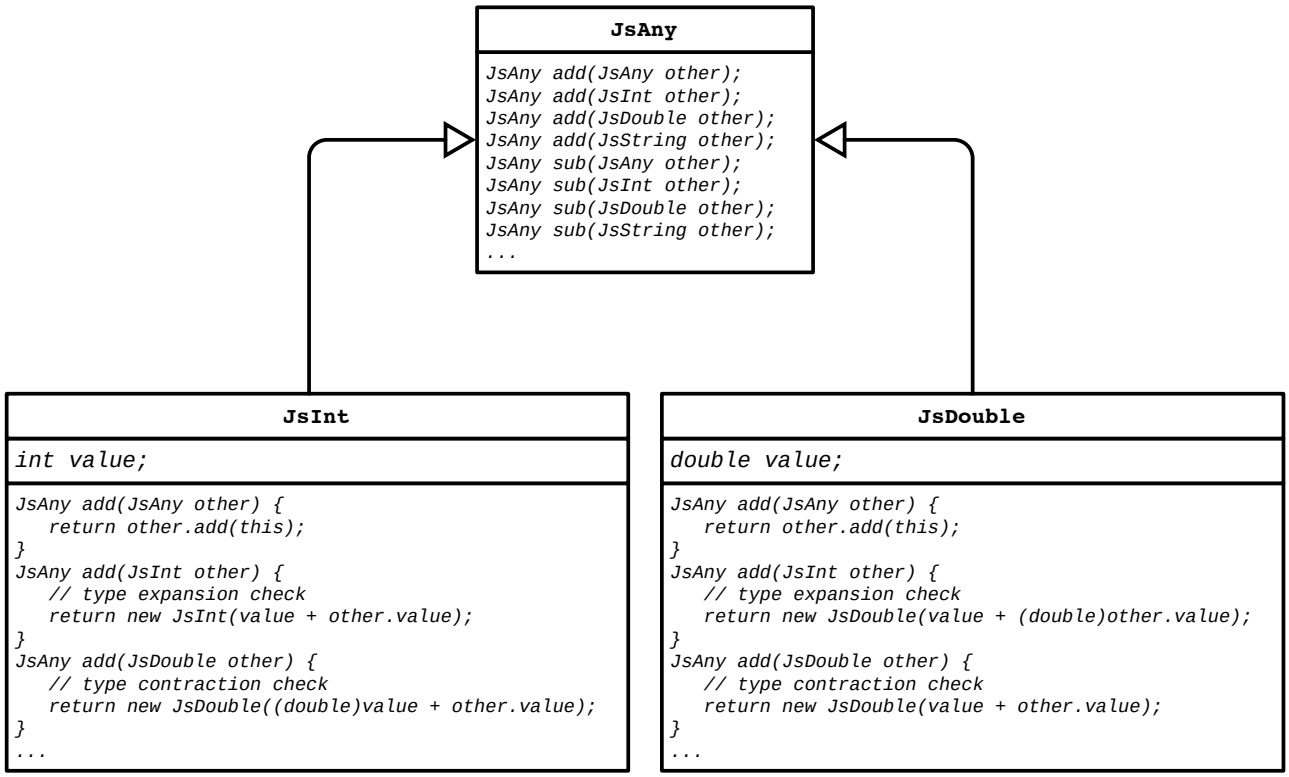


Figure 2: UML class diagram of the object hierarchy used to represent JavaScript values at the JVML level.

When translating JavaScript code into the statically typed JVML representation, all variables are declared as type *JsAny*, and all operations are translated to method calls on the abstract *JsAny* interface. Figure 4 shows the JVML code we generate for the simple accumulator loop in Figure 1.

The JavaScript specification does not strictly prescribe that integer values have to be stored as integers. Instead, it is conceivable to store all numbers as double internally and to forgo using the *JsInt* class to box integers as actual JVML integers altogether. From a performance perspective, however, this is a poor design choice since integer numbers can be used much more efficiently when treated as actual integers without having to convert them back from floating point values. This is particularly true for certain operations that absolutely require the value in integer form such as bitwise operations.

In our implementation, integer constants (i.e. 0) always create objects of type *JsInt*. The concrete implementation of operations operating on integers always try to box the result into a *JsInt* object except if the result exceeds the value range of JVML integers, at which point the value would be converted and returned as a boxed floating point double (Figure 5). The same optimization is performed for floating point double values. When the result of a floating point operation is an integer value, it will automatically be boxed to a *JsInt* object. Since values are computed and manipulated dynamically, these checks have to be performed at runtime and in most cases it is impossible to infer statically whether an overflow would occur even if we were to perform static analysis (which we do not). On the other hand, our trace compiler can add additional code paths specializing for additional dynamic types that are seen in the course of an ongoing computation.

This lazy type expansion and contraction technique is essential to achieve good performance for code with extensive integer arithmetic.

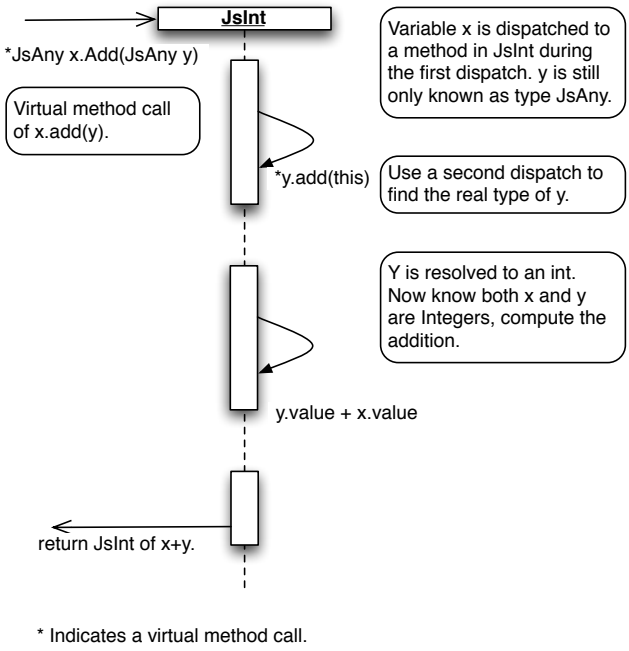


Figure 3: Double dispatch sequence diagram for an add instruction.

```
JsAny x = new JsInt(0);
for (JsAny i = new JsInt(0); new JsInt(1000000000).less(limit); i.increment())
    x = x.add(i);
```

Figure 4: Statically typed JVML code generated for the JavaScript example code in Figure 1. For readability the JVML code is shown as Java source code here.

# 4 Runtime Code Specialization with Trace Trees

The statically typed JVML code that we generate for the JavaScript input is completely self-contained and can be executed without additional optimization on a Java virtual machine [9]. The resulting performance, however, is sub-ideal because of the large amounts of dynamic virtual method calls being executed to perform the operand type resolutions. This is particularly undesirable because, even in dynamically typed code, the actual runtime type of a value tends to be constant at any given point in the program. This property can be exploited when *specializing* the JVML intermediate code. This specialization can be performed either based on a static prediction of types across the program, or based on runtime type observation.

As we have discussed in Section 2, it is often difficult and sometimes impossible to statically predict the precise type of values in dynamically typed program code. Our solution relies on dynamic type resolution using double dispatch and we then use dynamic type observation to drive the specialization process. For this, the generated JVML code is initially executed unmodified on top of a standard Java Virtual Machine (JVM). The code is allowed to execute but is profiled to identify frequently executed cyclic code regions. Using trace recording and trace compilation, we start to record trace trees covering most or all frequently executed code cycles through such a ”hot” code region. Since trace recording follows the virtual machine as it executes the JVML intermediate code to record the trace code, the trace recorder records the specific implementation of operations that is invoked as a result of the double dispatch that was used to resolve the operand types. The resulting trace thus contains a specialized sequence of JVML instructions

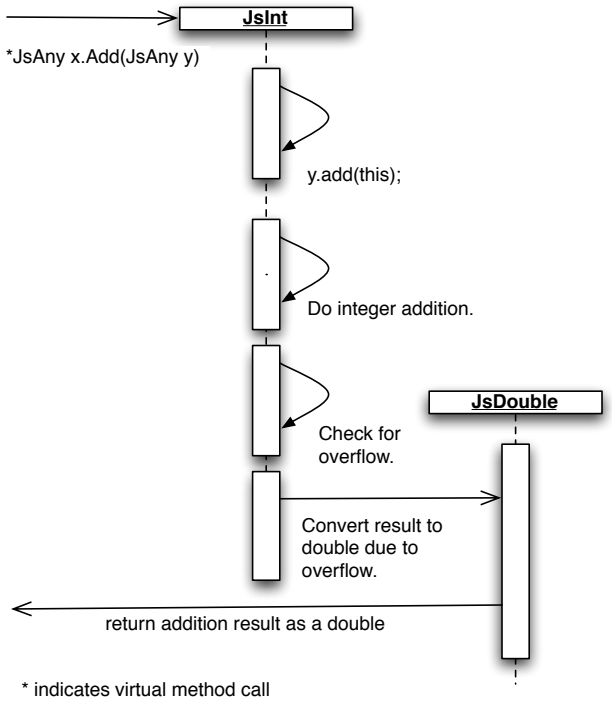


Figure 5: Expansion of type when result overflows an integer.

that execute the operations of the JavaScript program based on a specific typing scenario, which was observed during that particular iteration through the loop code region. Since subsequent executions of such loops are overwhelmingly likely to execute with the same typing scenario, such a trace is likely to be applicable to future iterations as well, and we can run it instead of executing the original JVML intermediate code.

Just as it is statically difficult to predict the type of values in a JavaScript program, it is also of course impossible to guarantee that future executions of the JVML code will always execute with the same types. While this is in fact very likely, certain scenarios such as value range overflows from integer to double, for example, can cause the results of operations to be boxed into a different type than during the recorded execution of the loop. To ensure proper semantics, we have to *guard* against such conditions by leaving the corresponding value range checks in the recorded trace. Since traces only represent a single path through the code, if such a guard fails we can no longer continue along the prerecorded trace code. Instead, we generate compensation code that is triggered in such cases and returns execution to the original non-optimized JVML intermediate code.

The idea of compiling a trace across a frequently execute cyclic code region can be extended to a tree of traces, with each trace sharing a portion of code with all the other traces. Each individual trace will contain the set of JVML instructions that represent the code executed when for a specific path. The first trace to occur is known as the main branch of a trace tree. Individual traces are attached to the main branch of a tree when a guard instruction fails, i.e. when a value range check causes the type of a value to change. In this case a new trace that is generated starting from the failed guard, then compiled, and finally attached to the main branch. A collection of these individual traces for a entry point into the cyclic region (loop header) is called a tree of traces, or trace tree.

Utilizing trace trees, we can speculate what the type of a variable will be during the execution of a loop. By deferring compilation until runtime at which time the types of all parameters are known, we can generate code which assumes that the current observed type will be the same for extended periods of execution across the entire loop. In the example with a simple for loop addition, the type of variable x can be contained and treated as an integer for many loop iterations. Thus we can compile the code assuming that variable x is an integer type, with a guard instruction to

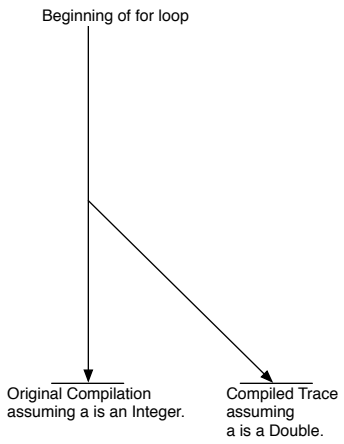


Figure 6: Trace tree covering the accumulator loop across the entire value range from integer over doubles to finally string objects holding the result value. Along each trace the code is specialized to deal with that particular type scenario only.

check that variable x is of type *JsInt*.

Using trace-based compilation, the loop is compiled for the first observed type constellation, and new traces are attached whenever types change. In the for loop example above, at a certain point, the addition will box the result into a *JsDouble* because it no longer fits into a 32-bit integer. The check in the *add* method to ensure that the result can fit into an integer will fail, and a new trace will be recorded starting at this guard instruction. In subsequent iterations the addition will be done in the implementation of the *add* operation in the *JsDouble* class instead of the *JsInt* class, because the left operand is now a double, and no longer an int. The newly recorded trace follows this new code path and will contain optimized code for dealing with the double type scenario.

The two traces compiled together represent a tree of traces, one for each of the possible types variable x can be: an integer and a double. An illustration of the tree of traces can be found in Figure 6.

Since along any given trace, types are assumed to be constant (and a guard instruction would exit the trace if this condition no longer holds), we can optimize the trace code based on the concrete types we have observed during recording. In particular, we can optimize away most of the type check guards associated with the underlying virtual method calls that were used by the double dispatch mechanism. All that remains in the trace code is the actual implementation of each operator, specialized for the concrete type scenario associated with that particular trace.

Eliminating the type check guards that were inserted while recording across the double dispatch virtual method invocations is possible because most of these operations execute on temporary values that were created inside the loop. Since we box such temporary values into Java objects, they are expressed as object allocations with a concrete type at the JVML level [i.e. NEW *JsInt*()]. If such a boxed temporary value is used as part of a double dispatch, the type check we usually would have to perform to ensure that the value is actually of type *JsInt* before we can continue along the trace and execute the specialized code dealing with operating on *JsInt*s can be eliminated, because the object allocation firmly established that the concrete type is *JsInt* (Figure 7).

While simplifying the type guard elimination process, the object allocations associated with boxing return values have their own significant runtime cost. We address this issue by performing a simple loop-local escape analysis, which checks to see if any values within a loop ever escape the loop, and are used before or after the loop has executed. This allows to bypass object allocation, while very cheap in Java, is still a considerable performance penalty if thousands of objects are created. This is the case during any operation, as the result of any method is a new object which represents the result. In a for loop with thousands of iterations, each addition will create a new object with the value of the addition result. By creating one instance of a new JsInt named result, which is created prior to the loop executing, the operation of an addition will be converted to a field write within the result, instead of a new object allocation every

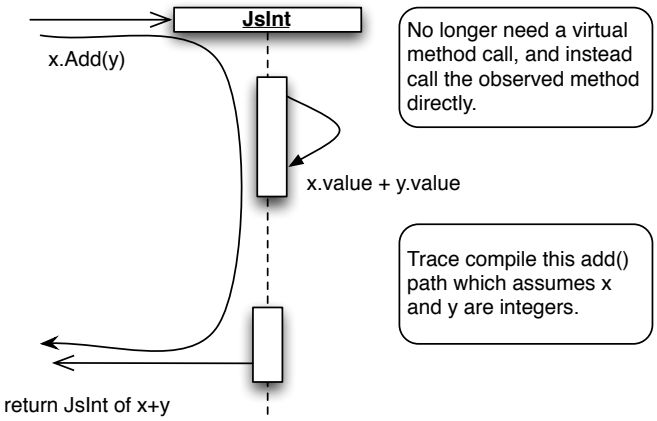


Figure 7: Eliminating redundant type guards that were added to the trace while recording virtual method invocations.

iteration of the loop. The resulting final trace code is shown in Figure 8.

# 5 Implementation

To evaluate the performance of our runtime code specialization approach we have implemented a JavaScript to statically typed JVML intermediate code compiler and a virtual machine that runs the JVML intermediate code, detects "hot" code regions and records trace across them. The recorded traces and the trees they are assembled to are then optimized and compiled to JVML code, which is not only our input code format but also our output/backend code format. Since our JVML execution framework records traces at runtime, we use runtime class loading to load and immediately execute the newly generated and optimized JVML bytecode every time we compile a trace tree.

Instead of implementing a JavaScript parser in our compiler, JavaScript programs are fed to our compiler in Adobe's ActionScript Virtual Machine 2 [13] binary file format, which is a bytecode representation for ECMAScript 4-compatible JavaScript programs. Compiling from this bytecode representation to JVML bytecode is much easier than first having to parse and semantically analyze the JavaScript source code since these steps are already performed by the ActionScript compiler.

JavaScript programs are translated to JVML code method by method, and each method is translated by iterating through all bytecodes in the method in a single pass emitting equivalent JVML instructions. The JavaScript to JVML compiler performs essentially no analysis and blindly translates all variables into JVML variables of type *JsAny*, and all operations on JavaScript values into double dispatches based on the boxed operands involved.

This approach has a number of advantages. On the one hand the complexity of the JavaScript to JVML translator is minimal as it is not concerned with program analysis or even the details of implementing JavaScript operations. These details are hidden the library classes we use to box the JavaScript types (i.e. *JsInt*, *JsDouble*), since all the generated JVML code does is invoke virtual methods on these library classes. The actual implementation is inside the corresponding class used to box the value.

The code our compiler generates for the JavaScript expression **var** *result* $= 3 + 4$; is shown in Figure 9. Both constants, 3 and 4 are boxed into *JsInt* objects and the add method of the left operand (3) is invoked, passing the right operand (4) as a parameter. Note that we invoke *JsAny*.add(*JsAny*) to add the two values, even though both are clearly of type *JsInt*. However, since our compiler doesn't perform any static analysis, it always invokes the generic operand method, which then will select the appropriate implementations at runtime using double dispatch.

If the expression above is part of a frequently executed loop, the resulting JVML code will eventually be recorded and trace-compiled by our execution engine. In this case the trace recorder will follow the constructor calls and inline the constructor code directly into the trace code. For both constructors that are invoked, a PUTFIELD instruction is

```
// Guard instruction
if variable a is not of type JsInt
    breakout of compiled code and retrace

// Guard instruction
if variable i is not of type JsInt
    breakout of compiled code and retrace

// Only do one object creation
JsAny result = new JsInt(previousVarAvalue);

// Guard instruction to check for if condition
for i is less than 100000000
    result = a.value + i.value;
    a = result;

    // Guard instruction
    if a overflows
        breakout of compiled code and retrace
end loop
```

Figure 8: Performing simple localized escape analysis to avoid the object allocation overhead associated with boxing JavaScript values into Java objects.

executed which stores the boxed value (3 and 4 respectively) in the wrapper object. The *add* method implementation is also inlined, and it reads the values from the wrapper object with GETFIELD. The trace compiler is able to eliminate these loads and directly connect them with the original source (the corresponding load constant instructions) using escape analysis and load propagation. Since both operands of the integer addition are thus constant, the final code that is emitted for this example by the trace compiler is actually a constant 7.

# 6 Related Work

Static type inference has been studied extensively in the context of functional programming languages. Thatte used static type inference in Lisp to detect type errors [14]. Aiken et al. proposed using static type inference to not only detect errors in dynamically typed languages, but also for program optimization [3]. Static type inference for optimizing object-oriented programming languages has been studied extensively in the SELF system [2].

As an alternative to static type inference, SELF also supports run-time type feedback-based optimization [7]. This approach is closely related to our work. SELF collects run-time type feedback by profiling the application and recording the actual receiver types of message dispatches. The type of the corresponding objects is then dynamically inferred based on this profiling information. Specializing code through trace recording is very similar to this approach. The main difference is that we combine the run-time type feedback collection and subsequent optimization step in one. The moment we speculatively decide to follow a particular trace, we automatically commit to specialize along the same path. This basically performs the optimization (specialization) for free as part of the analysis (trace recording).

Our trace-compiling JVML code execution environment is based on prior work by Gal et al. on trace compilation [6] and trace trees [5]. Novel in this context is that we apply trace compilation as a method to speculatively specialize code along frequently executed code paths, which allows us to neglect optimizing in the frontend translation process (and thus greatly simply the frontend compiler) without sacrificing performance.

Our prototype implementation of a trace-based JavaScript execution engine runs on top of the Java Virtual Machine [9] and we use the JVM as our backend code generator. This is closely related to the approach taken by the Rhino [10] JavaScript VM and JavaScript to Java compiler. In contrast to our system, Rhino does not statically

```
// Create a new JsInt representing 3
new     #15; //class JsInt
dup
ldc     #30; //int 3
invokespecial   #19; //Method JsInt."<init>":(I)V

// Create a new JsInt representing 4
new     #15; //class JsInt
dup
ldc     #30; //int 4
invokespecial   #19; //Method JsInt."<init>":(I)V

// Call virtual method to add
// syntactically is JsInt(3).add(JsInt(4))
invokevirtual   #33; //Method JsAny.add:(LJsAny;)LJsAny;

// Store the result
astore_2
```

Figure 9: JVML intermediate code generated for the JavaScript expression **var** *result* $= 3 + 4$;.

compile JavaScript to JVML. Instead it merely embeds the JavaScript program as a string in a class file. The actual JavaScript compilation process is then triggered at runtime. Rhino does not attempt to perform type inference, and does incurs a significant type checking overhead for every operation. As we show in the benchmark section we significantly outperform Rhino for most test cases.

Tamarin [12] is the next generation JavaScript interpreter and just in time compiler for Mozilla products. It is also embedded within the Adobe Flash Player 9. It has numerous features such as a garbage collector, and support for the ECMAScript Edition 4 standard. Tamarin performs limited static type inference and attempts to distinguish between strings and numbers. Our main advantage over Tamarin is our more fine grained type system that also distinguishes integer types, which then can be executed cheaper.

SpiderMonkey [11] is a production quality interpreter for JavaScript, and is embedded in current Mozilla products. It does not perform just-in-time compilation or any code optimization.

JScript [4] is Microsoft's implementation of ECMAScript 3, and is currently the JavaScript interpreter in Internet Explorer. JScript is being advanced to JScript.NET, which contains a compiler targeting Microsoft's Common Language Runtime.

# 7 Benchmarks

There is no established benchmark suite for JavaScript. To evaluate the performance of our prototype system, we created a number of micro benchmarks and report the performance of our prototype system in comparison to the performance of existing JavaScript interpreters and just-in-time compilers. The microbenchmarks are designed to exercise simple dynamically typed language features of JavaScript. *Function Call* loops a set number of times, and calls a function which does some mathematical calculations on the input number. This number is then returned and added to a total value, which is printed at the end. There are two possible functions to be called, depending on the value of the loop counter. Each function applies a different mathematical calculation on the input variable. *Prime Number* finds the number of prime numbers that is less than a certain limit. This test case was provided by Adobe, as one of their functional tests. *Random Number* tests the generation and summation of calling a native method. The results of this benchmark have to be interpreted with some care since the random generator of our system is implemented in pure Java (our "native" language) whereas Rhino and Tamarin perform actual native machine code calls to obtain the next random number. *Sum* iterates over a range of integers, adding the counter to a *total* variable, and subtracted a constant

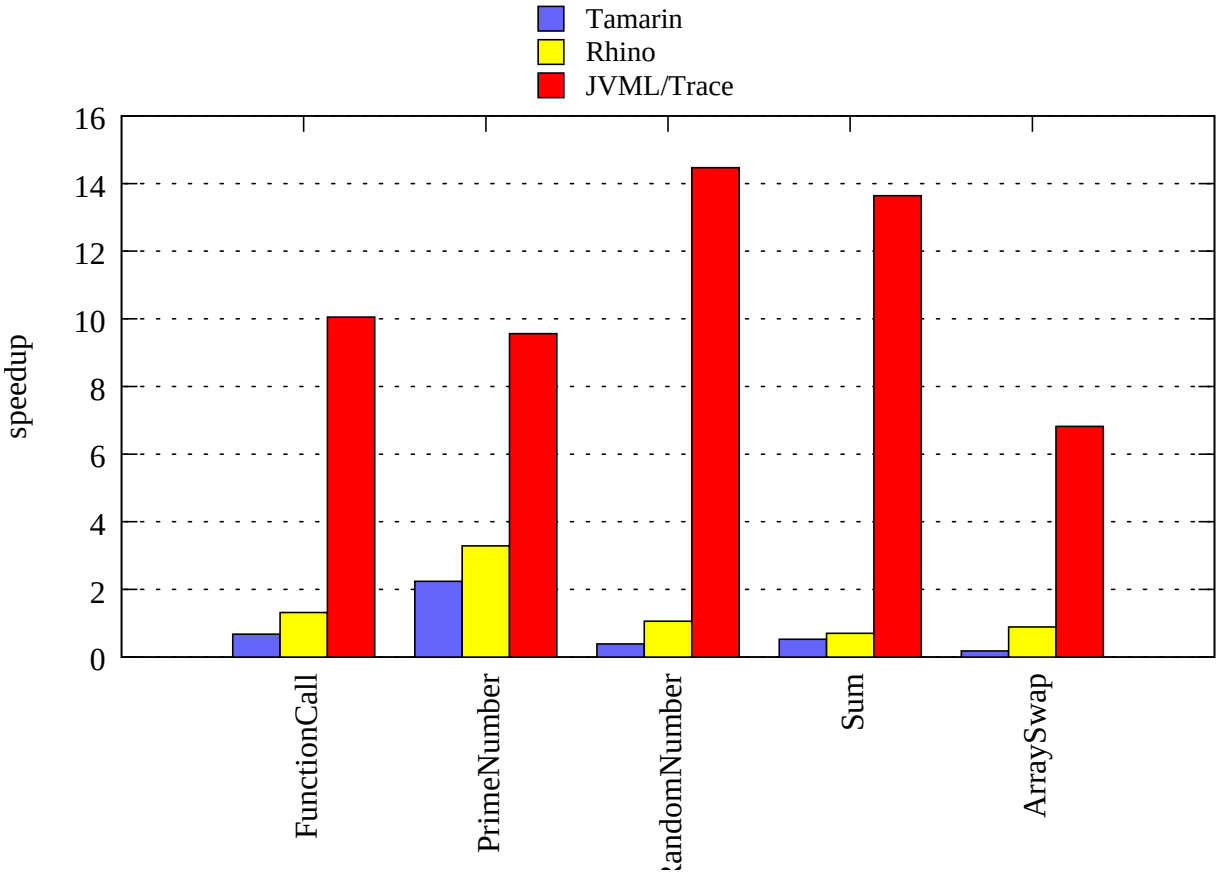


Figure 10: Speedup for Micro Benchmarks relative to SpiderMonkey, an interpretation-only JavaScript VM.

from the *total* at every step. *Array swap* loops over a large array and fills each array element with the loop counter. A second loop is then executed which swaps the $N$th element with $N-1$st element.

In addition, to test the performance of larger codes, we ported the JavaGrande suite to JavaScript, and tried to mimic the Java semantics as much as possible. For example, since many of the JavaGrande section 2 tests use the same seed for the random number generator, and JavaScript does not have a random generator which accepts a seed, we developed our own random number generator in the actual JavaScript source file. All methods requiring a random number bypass JavaScript's native random number generator, and called our own instead. This is required in order to mimic the JavaGrande semantics of using the same random numbers every time the benchmark was executed. Finally, the JavaScript ports contain much more initialization code, which may include multiple loops iterating over the size of the dataset. This is due to the fact that Java will initialize objects to a default value, including all indexes in an array. JavaScript on the other hand, does not have a default value, and instead will return a type of "Undefined" or throw an exception when accessing an uninitialized object. Therefore, code had to be included to initialize all elements of an array.

Our benchmarks represent the relative speedup of our solution, Tamarin, and Rhino against Spidermonkey. All execution times were measured as real time executed. For the frameworks that use JVM as backend compiler (Rhino and our prototype) we subtracted the time spent in the Hotspot compiler from the execution time reported. This was necessary because the Hotspot JIT compiler is heavily optimized to compile code generated by *javac*. For any other code (i.e. code emitted by Rhino and our system) the performance characteristics of the JIT are somewhat unpredictable. We have observed 8 seconds being spent in the register allocator of Hotspot's JIT for a benchmark running 15 seconds. As a result of this observation we have decided to abandon JVML as backend code, and we will

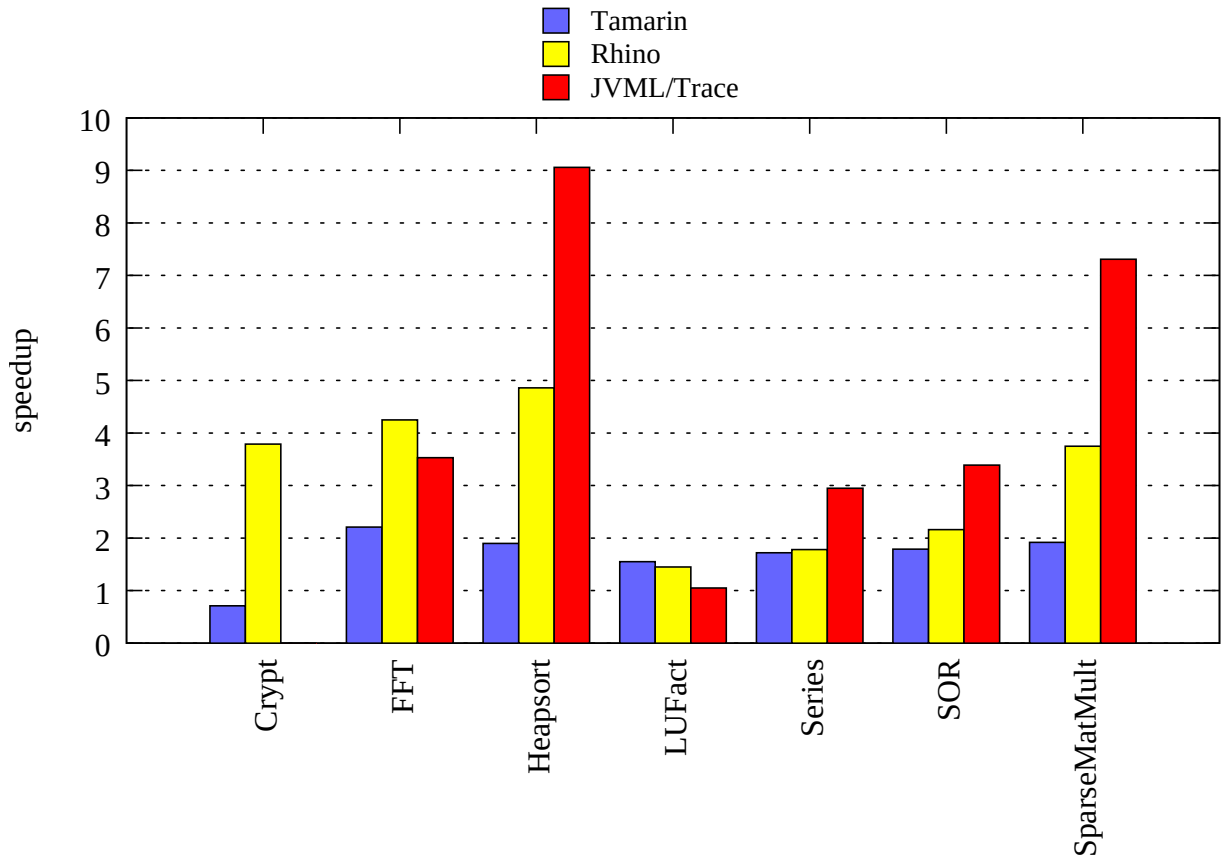


Figure 11: Speedup for the JavaScript port of JavaGrande section 2 size A, relative to SpiderMonkey.

attempt to report numbers based on a native x86 backend for the camera-ready version.

Figure 10 shows the results for the micro benchmarks. We are able to significantly outperform all existing JavaScript VMs, including Rhino, which executes on top of the same Java VM. The speedup over interpretation is 10-14 for our system, which is comparable to the speedup just-in-time compilers achieve for statically typed languages.

In the benchmark graphs, a result of 5 means that the particular application was executed five times faster than running under a pure interpreter (Spidermonkey). The higher the number, the better the performance. All benchmarks were run on a Macbook Pro, 2.4GHz Intel Core 2 Duo, with 2 GB of PC2-5300 RAM. All Java tests were run on the Java 6 developer version provided by Apple, with the -server flag, and an increased heap size of 1.5GB. Benchmarks had to be executed on Intel MacOSX as Tamarin is untested for Linux and our system is untested on Windows. For the size B set, Rhino and our implementation require than 1.5 GB of RAM for the CRYPT test, and therefore could not be executed on this machine. Spidermonkey could not properly execute the FFT size B test, and therefore no baseline results were obtained. As can be seen, except for FFT and CRYPT bench, our solution beats all available solutions by a good margin. We have no benchmark numbers for CRYPT due to a regression in our trace optimization framework which we were unable to resolve by the submission deadline.

The results for the ported JavaGrande benchmark set are shown in Figure 11 and Figure 12. As mentioned before, we are not reporting numbers for some benchmark programs for size B due to an out of memory condition in the Java environment. The results are more mixed for the JavaGrande benchmark set. We attribute this mostly to using Hotspot's JIT as our backend code generator. For these more complex test cases, we still significantly outperform the other compilers, but the trace specialization generates a less dramatic speedup. Analysis of the "optimized" JVML

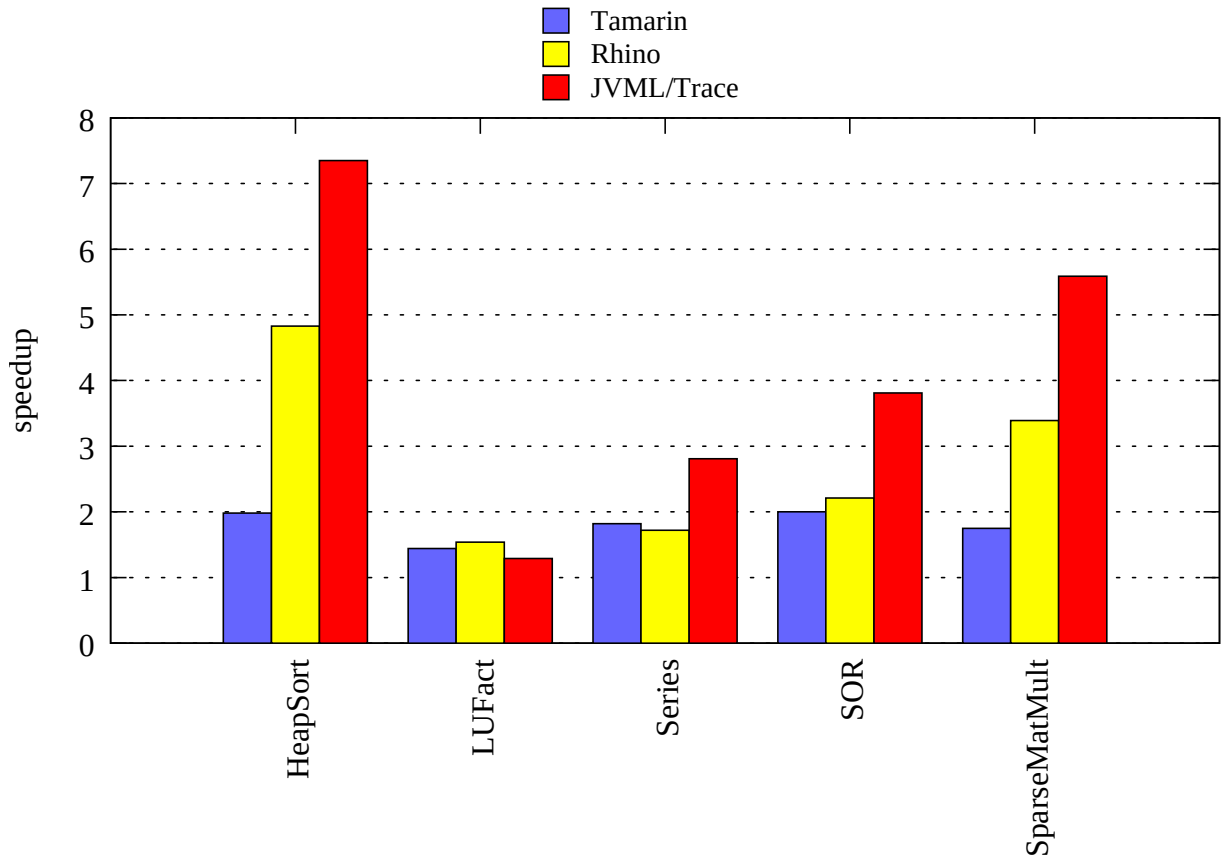


Figure 12: Speedup for the JavaScript port of JavaGrande section 2 size B, relative to SpiderMonkey.

code generated by the trace compilation framework has shown that it looks even more "unnatural" than the artificial JVML code generated by the JavaScript to JVML compiler. In particular, our trace framework generates JVML methods that are very long (up to 64kBytes) and have a large number of very far branches. Hotspot seems to be not optimized for this kind of workload and generates code of very poor quality, and in some cases also takes an extraordinary amount of time to do so as discussed above. More definitive benchmark results for complex test cases such as JavaGrande will only be possible through replacing the Hotspot JIT backend with a true x86 native machine code backend. We expect that with such a native code backend, the speedup for complex cases such as JavaGrande will be similar to the speedup observed in the micro benchmarks.

# 8 Conclusions & Future Work

We have presented a new approach to compiling dynamically typed languages, which is based on observing the actual types of variables at run-time rather than trying to infer them beforehand. We model the precise semantics of dynamic typing using a double dispatch mechanism. Our underlying compilation engine is able to completely optimize away the overhead of this double dispatch mechanism along critical paths. This results in surprisingly good performance compared not only Rhino, which executes in the JVM, but also to a full Just in Time compiler written in native C++. We achieve all the benefits of writing in Java, such as type safety and memory management, and still achieve better performance than C++ code.

Our prototype currently does not yet handle the full ECMAScript 4 specification. In the coming months, we will

be expanding it to cover the whole set of features provided by the new language that is currently being standardized. We also will extend our trace compiler to compile directly to x86 native code, instead of relying on the Java virtual machine to compile the traced execution. This should make our compiler even more competitive. Finally, we are looking into expanding our compilation technique to other dynamic languages such as PHP and Python.

# References


[1] Private Communcation with Brendan Eich, June 2007.

[2] O. Agesen, J. Palsberg, and M.I. Schwartzbach. Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. *Software - Practice and Experience*, 25(9):975–995, 1995.

[3] Alex Aiken and Brian Murphy. Static type inference in a dynamically typed language. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 279–290, New York, NY, USA, 1991. ACM Press.

[4] Microsoft Corporation. Microsoft Corporation - JScript User's Guide - http://msdn2.microsoft.com/en-us/library/4yyeyb0a.aspx - September 14, 2007.

[5] A. Gal. *Efficient Bytecode Verification and Compilation in a Virtual Machine*. PhD thesis, University of California, Irvine, 2006.

[6] A. Gal, C.W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. *Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153, 2006.

[7] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 326–336, New York, NY, USA, 1994. ACM Press.

[8] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 347–349, New York, NY, USA, 1986. ACM Press.

[9] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[10] Mozilla. Rhino - JavaScript for Java - http://www.mozilla.org/rhino/ - August 30, 2007.

[11] Mozilla. SpiderMonkey(JavaScript-C) Engine - http://www.mozilla.org/js/spidermonkey/ - February 17, 2006.

[12] Mozilla. Tamarin Project - http://www.mozilla.org/projects/tamarin/ - August 1, 2007.

[13] Adobe Systems. Adobe Systems - ActionScript Virtual Machine 2 (AVM2) Overview - http://www.adobe.com/devnet/actionscript/articles/avm2overview.pdf - May 2007.

[14] S. Thatte. Type Inference with Partial Types. *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, pages 615–629, 1988.