Scalable and Accurate Type Analysis for JavaScript

Maxime Chevalier-Boisvert

March 22, 2012

1 Introduction

The use of dynamic programming languages, in particular JavaScript, has been rising dramatically over the last 10 to 20 years. Not long ago, implementing software in a dynamic language often meant trading efficiency for expressiveness. However, with the advent of client and server-side scripting in the browser, there has been a strong push to increase the performance of dynamic languages.

Various techniques have been discovered to compile more effective code and execute dynamic languages more effectively, including the use of tracing JIT compilers, inline type caches and type feedback systems. With these techniques, the performance of dynamic languages has greatly improved, but there is still room for improvement as programs written in dynamic languages are often still an order of magnitude slower than equivalent programs written in C.

What makes dynamic languages more expressive is often what also makes them slower. Implementing dynamic typing, late binding and dynamic dispatch conceptually requires the compiler to insert type checks into the generated code. Primitive values may also need to be boxed[5] because the types of variables are not constrained ahead of time and the generated code needs to be able to deal with multiple possibilities. This generality can incur a large penalty in terms of execution time.

In order to maximize the performance of dynamic languages, it is necessary to reduce the occurence of dynamic type checks to a minimum. Doing so effectively requires analyzing programs so as to attempt to determine potential variable types. Precise type information offers guarantees that make it possible to eliminate redundant type checks and otherwise specialize compiled code to maximize its efficiency.

Precise static analysis of dynamic programming languages is a challenging problem, as language constructs such as `eval` make it possible to load new code on the fly, essentially nullifying analysis results, forcing type analyses to make an open-world assumption. Another difficulty arises from the object model used in languages such as JavaScript, where objects are not necessarily initialized all at once, but rather extended on-the-fly through the progressive addition of new fields.

We examine static analysis strategies to obtain precise type information about JavaScript programs. Because objects are a central part of the language and efficient property accesses are crucial for good performance, special attention is paid to the issues pertaining to the analysis of object types.

The main contributions of this paper are:

- A survey of related work pertaining to the type analysis of JavaScript programs.
- A presentation of the current design of our type analysis.
- An examination of potential directions to improve the scalability and precision of our type analysis.

2 Related Work

LISP, a programming language designed by John McCarthy in 1958, was the first dynamic language. The first implementation of this language was interpreted. LISP quickly attracted much interest, particularly in artificial intelligence research. It became obvious, however, that a more efficient implementation which could compile LISP to machine code would yield significantly higher performance.

LISP dialects rely on late-binding, but in typical LISP programs, most function invocations always call the same function. Hence, to compile LISP programs efficiently, a compiler should ideally be able to determining which closures (lambdas) a given symbol may refer to. The k-CFA analysis was designed to try and answer this question[10]. This analysis relies on a fixed-point calculation which can take k-degrees of context into account to predict which values symbols may refer to at run-time.

A recent publication by Adams et al. outlines a flow-sensitive type-recovery algorithm which runs in log-linear time and is able to scale to hundreds of thousands of lines of Scheme code[1]. This algorithm is based on sub-0CFA and achieves its impressive scaling properties by limiting flow-sensitivity to singleton type sets only and by the use of cached skipping functions which allow type information to avoid threading through individual expressions when possible.

The algorithm designed by Adams et al. shows very impressive performance and outlines interesting strategies to maximize scaling of type inference algorithms. We are concerned, however, that their approach might be unsuitable for JavaScript programs. The limitation of flow-sensitivity to singleton type sets only is rather restrictive, considering that a significant proportion of call sites in typical JavaScript programs are polymorphic[9]. The type system used in this analysis is also rather simplistic and does not account for any kind of object system.

SELF is a minimalist dynamic programming language derived from Smalltalk. Like Smalltalk, it features a prototype-based object model, and every primitive value in the language behaves as an object which can receive messages. In SELF,

however the object inheritance hierarchy can change dynamically, making the language more difficult to optimize effectively.

In order to make the performance of SELF programs more competitive, a constraint-based type inference analysis was developed[2]. It is a whole-program analysis which represents constraints within a trace graph where nodes correspond to objects and methods and edges correspond to message sends. The analysis is flow-insensitive and runs in polynomial time.

The SELF type inference analysis seems to fare well for SELF programs, but it does not make use of flow-sensitivity, and thus cannot exploit useful optimization opportunities. Another real concern is that it is often assumed that SELF objects are constructed all at once, with all their properties being defined at the same time when the object is defined. This is generally not the case in JavaScript, where objects are often initially empty, and new properties are added later in their lifetime.

An important problem in object-oriented dynamic languages such as JavaScript is to gather precise information about object property types. This is complex because languages like SELF and JavaScript allow objects to grow dynamically. The concept of recency types, initially proposed for the analysis of heap-allocated data[3] has been adapted to dynamic languages[6]. Proponents suggest that in typical JavaScript programs, objects are initialized early on in their lifetime.

The idea behind recency types is then to represent objects using two different abstractions. The recent type represents the single most-recently allocated object at a program point. This type maps to a single object at a given time during execution, and so mutations to it can be accounted for using strong updates, allowing a precise representation of the object during initialization. After their initialization phase, recent object types are demoted into summary types which accounts for all objects allocated at given program points.

A recent trend in type analysis has been to build such analyses based on the may points-to analysis framework. In this framework, the state of a program is typically represented by a graph in which there are variable and value nodes. Edges go from variables to values and represent the fact that a given pointer variable may point to a given value[11]. This framework is convenient to reason with and adapts quite well to dynamic languages in which reference variables are analogous to pointers.

In 2006, Jensen et al. published a type analysis for JavaScript based on the points-to framework[7]. This analysis is flow-sensitive and path-sensitive. It represents whole-program type graphs including all variables and object properties at every program point. The analysis also makes use of recency types to analyze object types more precisely. The type abstraction used is multidimensional and able to represent objects, constants as well as integer ranges. This analysis suffers from scalability problems but it is the first analysis to try and tackle all of the JavaScript language.

This work was recently extended to try and improve the scalability of the analysis using lazy propagation techniques[8]. This improved analysis has better scalability but still takes minutes to complete on medium-size benchmarks.

While high precision may be the biggest strength of Jensen et al.'s analysis, scalability issues still remain its most important weakness. At this time, this analysis is entirely unsuitable for use in a JIT compiler, and may not be able to scale to very large JavaScript programs, even for offline analysis.

Limitations prevent static analysis from being able to cope with the full scope of dynamic languages. In a purely static setting, constructs such as `eval` must be ignored or limited in some way. One way to avoid such limitations is to design a hybrid analysis which can be incrementally updated as a program is running. In this way, when a construct the analysis cannot handle, such as `eval`, is encountered, the analysis can temporarily ignore it and defer its analysis until run-time, at which point the analysis results can be incrementally updated. Dataflow-based analysis frameworks make such incremental updates easy to perform by simply iterating until a new fixed-point is reached.

Brian Hackett, programmer and researcher at Mozilla, has designed a hybrid static and dynamic analysis which follows this principle. It is a flow-sensitive dataflow analysis. Its results are used to optimize real-world JavaScript programs running in the Firefox web browser. This analysis not only defers the analysis of some language constructs, it can also make optimistic assumptions about the contents of some variables and properties by inserting guards that dynamically verify the validity of these assumptions at run-time. If these dynamic checks report that some analysis results are no longer valid, the analysis results can be updated, and affected parts of programs recompiled.

Mozilla's type inference analysis shows impressive results and has already been integrated in a production web browser. It is able to scale to large programs and runs fast enough for its use to be justified, even within a JIT compiler. The main weakness of this analysis, however, may be its lack of precision when it comes to analyzing object property types. This lack of precision forces the inclusion of run-time guards to check that certain values do not occur at run-time. We believe that a more precise analysis may be able to definitively eliminate more guards and produce better performance on long-running programs.

3 Analysis Design

The type analysis we have implemented operates on a Static Single Assignment (SSA) intermediate representation. It is both flow-sensitive and path-sensitive. The analysis is a forward dataflow analysis which iterates until a fixed-point is reached. The analysis is sparse, that is, basic blocks and function bodies are only explored when the analysis determines they may be reached. A control-flow graph is built as the analysis runs. Type graphs representing the types of all variables and properties are available at all program points the analysis finds to be reachable. The following sections provide more details about the most salient aspects of our type analysis.

3.1 Flow-sensitivity

The SELF type analysis was a whole-program, flow-insensitive and path-insensitive analysis[2]. The JavaScript programming language takes some inspiration from SELF, but we are instead aiming to build an analysis which is is both flow-sensitive and path-sensitive. This will obviously incur a higher cost in terms of execution time, but we believe this tradeoff is necessary in order to gain more precise type information and optimize the language more effectively.

The fastest whole-program type analyses are of the flow-insensitive variety. In such analyses, the order of execution of statements is not taken into account. A fixed-point is instead computed over a graph of constraints representing all interactions between variables through function calls and operators. Such an analysis would undoubtedly reveal some information about JavaScript programs, but we feel it would most likely fail to take into account very useful information contained within the said programs.

Building a flow-sensitive analysis instead can allow us to more efficiently narrow the range of loop variables and take advantage of some built-in JavaScript constructs. It is fairly idiomatic in JavaScript code, for example, to make use of the `typeof` and `instanceof` operators in branch tests to perform different actions on values of different types (ex.: Listing 1). With the use of flow-sensitivity, it becomes possible to narrow the types of values on either side of a branch, which can be very useful for optimization.

Listing 1: Typical use of the `instanceof` operator

```
1  if (ast === null)
2  {
3     // no transformation
4     return ast;
5  }
6  else if (ast instanceof OpExpr)
7  {
8     ast.exprs = ast_walk_exprs(ast.exprs, ctx);
9     return ast;
10 }
11 else if (ast instanceof NewExpr)
12 {
13    ast.expr = ctx.walk_expr(ast.expr);
14    ast.args = ast_walk_exprs(ast.args, ctx);
15    return ast;
16 }
17 else if (ast instanceof CallExpr)
18 {
19    ast.fn = ctx.walk_expr(ast.fn);
20    ast.args = ast_walk_exprs(ast.args, ctx);
21    return ast;
22 }
```

3.2 Type Abstraction

The type abstraction we use for our analysis is that of a type set. At each program point, each variable and property has an associated set of values which it may have at run-time. This can be a set of object abstractions or primitive values. A separate object abstraction is associated with each program point

where an object can be created. That is, each program point where there is an object literal or a constructor call using the `new` keyword.

In the case of integer and string primitives, we limit ourselves to being able to represent an integer constant or an integer range as well as a separate string constant. We do this to try and prevent situations in which the type set could grow to contain a potentially infinite number of constants. We have deemed the ability to represent integer ranges potentially useful in order to prove that integer variables may be representable as machine integers.

Object abstractions are equivalence classes representing one or multiple objects created at the same program point (source code location). Object abstractions have a set of associated properties, each of which has its own type set. The prototype value of objects has its own associated type set as well. The global object and library prototype objects are treated in the same way as user-created objects.

3.3 Path-sensitivity and Strong Updates

One of the main difficulties in analyzing JavaScript code is that the language has no inherent concept of classes. The object system is instead prototype-based, like that of SELF. Unlike SELF, however, JavaScript objects are most often created empty, without properties of their own. Properties may then be added at any point during the execution of a program. This is problematic because when a property is read, if we cannot guarantee that it was previously defined, we must assume that the property may also take the special `undefined` value. The `undefined` value can easily spread through an analysis, polluting its results.

Much of our efforts are focused on analyzing object property types more precisely, so as to minimize the occurrence of superfluous `undefined` values. In order to guarantee that a given property is defined at a given program point, it is necessary to be able to analyze the initialization of objects with sufficient precision. More specifically, it is very useful to be able to analyze object types with path-sensitivity, that is, allowing properties to have different types at different program points, so that strong updates can be performed on the property types when properties are written to in the program being analyzed.

Performing strong updates on a property means discarding and replacing the previous type information associated with it, instead of performing the union of the type written with its current type. Doing so requires proving that, at run-time, all objects associated with a given object abstraction will receive this mutation. This is not always the case as object abstractions can represent an arbitrary number of runtime objects, and not all of these objects flow through the same paths at run-time. Thus, knowing that a property write applies to objects of a given abstraction does not necessarily imply that all objects of this abstraction will undergo this operation.

There are several cases in which it is obvious that an object is suitable for strong updates. One is when the object in question is a trivial singleton, such as the global object, a global function, or the default `prototype` property object

of a global constructor function. Another case is when an object was created in the current function and the mutation is being performed through a direct reference to this object. In this case, if, along any branch, we write to a property of this object through the reference, we know that we are writing to all objects of this type that can possibly go through that branch. Another useful case is when we are manipulating the `this` value of a function that is only ever called as a constructor.

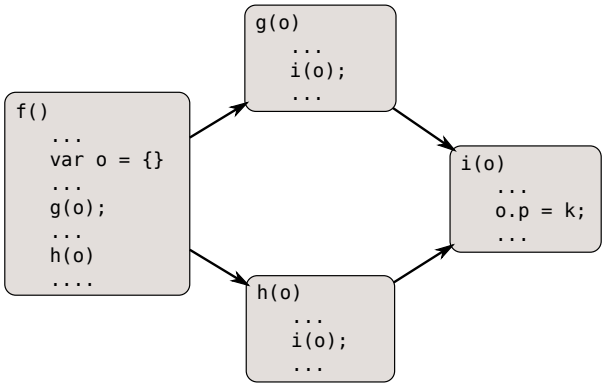

Figure 1: Strong Update with the Dominance Rule

We have observed that there seems to be a general rule, which we will call the dominance rule, that guarantees the safety of strong updates. It appears that when an object is created in the current function, in a caller of the current function or in a common ancestor of all callers of the current function, it is safe to perform strong updates on this object. When a property write applies to a single object abstraction, this rule appears to guarantee that we are operating on all objects which are part of that equivalence class. In Fig. 1, the object is created in `f` can be strongly updated in `i` because `f` is a common ancestor of all callers of `i`. We have not yet found a proof of this assumption, but this may be worth further examination, as finding such a proof could make our analysis more powerful.

3.4 Analysis of Property Accesses

JavaScript objects and arrays essentially behave as dictionaries where the keys are strings. Non-string values are implicitly converted to strings. Our analysis currently handles only constant strings and non-negative integers as keys. Strings are mapped to individual property type sets in object abstractions. All non-negative integer keys all map to the same indexed property type set. This is done under the assumption that integer keys most often refer to indexed homogeneous array properties.

When analyzing a property write `o.k = v` on an object property, four type sets are involved. Three of those sets are obvious: the type set of the object `o`,

that of the key `k` (property name), and that of the value `v` to be assigned to the property. The fourth is the type set associated with the property in the current type graph (associated with the current basic block).

There are only two cases to handle. If strong updates are possible (see Section 3.3), the current type set for the property is replaced by that of the value `v`. Otherwise, if strong updates are not possible, the type set of the property becomes the union of its current value and the type set for `v`.

The examples shown in Fig. 2 and Fig. 3 illustrate property write with and without strong updates, respectively. In the said examples, properties have solid round outlines, values have dashed square outlines and objects abstractions have dashed round outlines. Solid arrows represent definite set membership and dashed arrows represent possible set membership.

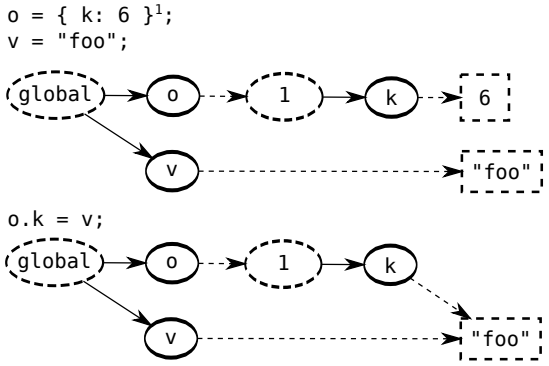

Figure 2: Property Write with Strong Update

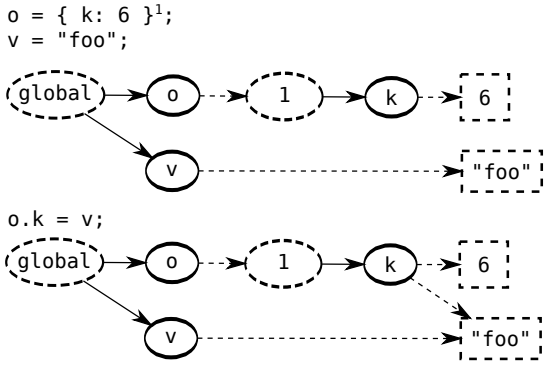

Figure 3: Property Write without Strong Update

When analyzing a property read `v = o.k` on an object property, multiple type sets are involved. Two of those sets are obvious: the type set of the object `o` and that of the key `k` (property name). However, there are at least two other

type sets involved: the current type set associated with the property and the type set for the object's prototype.

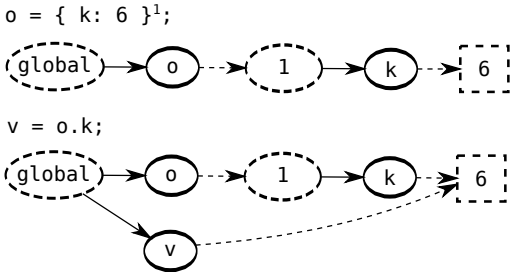

Figure 4: Trivial Property Read

In the simplest case, the property being read is defined on the object. Its type set is looked up in the current type graph and returned directly (see Fig. 4). However, if the property may not be defined on the object, the property look up will recurse on the object's prototype. Note that the prototype type set may actually contain multiple objects, in which case the lookup needs to be performed on each one. All the resulting types obtained are unioned into an output type set. The recursion terminates when the prototype type set for an object only contains `null`, as is the case with the object prototype object. In this case, the `undefined` type is returned (see Fig. 5).

3.5 Scalability

Path-sensitive analyses have the reputation of not scaling very well in terms of running-time and memory usage. In the case of our analysis, the implementation currently propagates type sets for each property and variable encountered thus far along all control-flow paths, and a separate type graph instance is kept for each basic block analyzed. The number of variables and properties increases with program size, and so does the number of basic blocks, which leads to an approximately quadratic scaling in terms of memory usage, and worse than cubic running time.

This issue of scaling has been a real problem in our experience. For one, the current implementation of our type analysis is too slow to be usable in many real-world use cases, with analysis times easily exceeding 5 minutes for moderate-size programs. On some benchmarks the analysis ran out of memory, on others, we could not get convergence within a reasonable time. It is clear to us that this is a problem that must be addressed.

4 Results

In this section, we report some preliminary analysis results. These results were gathered on some of the larger benchmarks from the SunSpider and V8

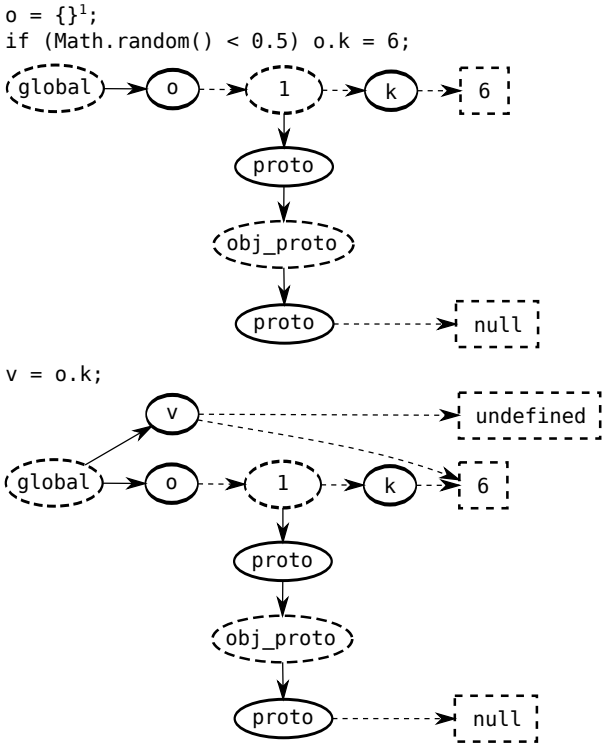

Figure 5: Property Read on Potentially Missing Property

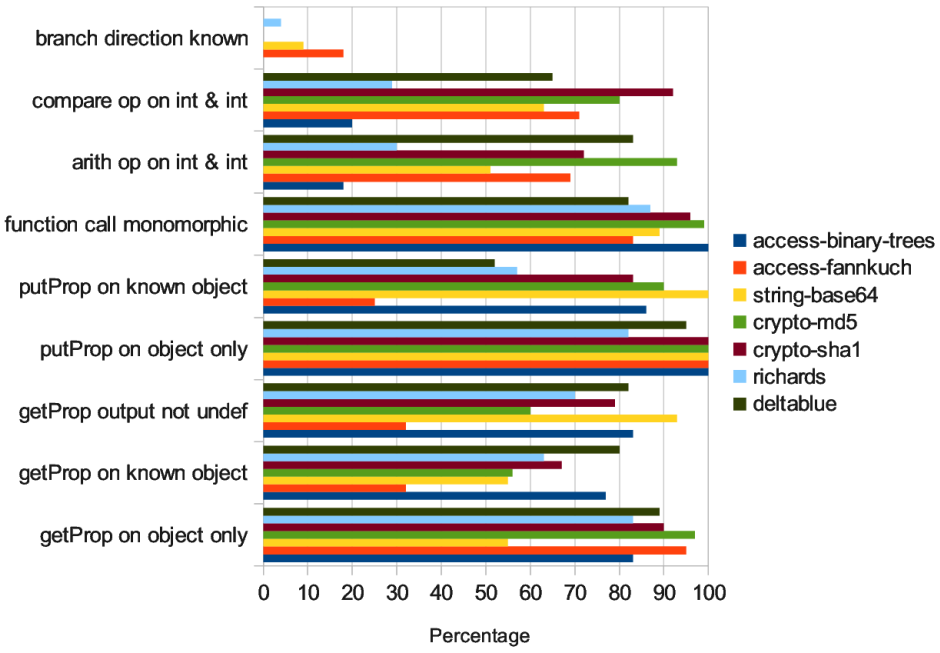

Figure 6: Analysis Accuracy Statistics

JavaScript benchmark suites. These are nontrivial and range from 50 to 900 lines of code. Based on the analysis results, we have computed several statistics approximating quantities we believe translate into run-time optimization potential.

The statistics are percentages computed on a per-instruction basis, and are as follows:

- getProp on Object only: proportion of property reads where the base of the read is known to always be an object.
- getProp on known object: proportion of property reads where the base is a known single object.
- getProp output not undef: proportion of property reads not returning the `undefined` value.
- putProp on Object only: proportion of property writes where the base is known to always be an object.
- putProp on known object: proportion of property writes where the base is a known single object.
- function call monomorphic: proportion of function calls determined to be monomorphic.

- arith op on int & int: proportion of arithmetic operations determined to always receive two integers as input.
- compare op on int & int: proportion of comparison operations determined to always receive two integers as input.
- branch direction known: proportion of branches whose direction has been statically determined.

The resulting statistics can be seen in Figure 6. In every benchmark, over 80% of function calls are found to be monomorphic, and in most cases, property reads and writes occur on known single objects. This is encouraging because both of these statistics are likely close to the theoretical maximum obtainable. We observe that the inference of property writes is better than that of property reads. This is likely because most property writes occur during the initialization of objects, near the object's creation site.

One of the areas where our analysis fares less well is that of arithmetic and comparison operations. In one specific benchmark (access-binary-trees), the analysis can only determine that a small fraction of these operations operate on integers, when in fact, it is easy to see that in practice, 100% of the values involved are integers. We believe this weakness occurs because the analysis of arithmetic operations is easily contaminated by `undefined` values.

In the case where one input to a multiplication operation may be `undefined`, we have that the output may be `NaN`. Subsequent operations using this result might also then produce `NaN`. Such erroneous values can easily propagate through large parts of a program being analyzed. As such, it is critical for the analysis to be able to eliminate as much superfluous `undefined` values as possible.

It is interesting to note, looking at our results, that in a few cases, we can determine branch directions ahead of time. This is no mistake. These cases correspond to sanity checks in the benchmark program which would throw exceptions in case of error. Our analysis is able to determine that these exceptions will never be thrown, and that these sanity checks are dead code.

5 Future Work

The most important problem with our current type analysis design is its poor scalability. We would like to make it possible for our analysis to scale up to a level where it can analyze the entirety of the Tachyon JavaScript compiler codebase[4] (approximately 80K lines of JavaScript code) within a few minutes of computation time on a modern PC. Making this possible will undoubtedly require important design changes.

We hope to find ways to make the analysis more scalable while also improving its overall precision. It seems fairly clear that the current design is rather wasteful. Type sets are propagated to regions of programs that do not make use of these. Type graphs are stored for every basic block, but these remain

largely unchanged between adjacent blocks, and much redundant information is stored.

The subsections below outline multiple areas which we believe are worth investigating to reduce the cost of our type analysis.

5.1 Lazy Propagation

As observed by Jensen et al., naïve path-sensitive analyses will propagate information to areas of programs which do not make use of it[8]. In the case of our analysis, the type sets of the properties of all previously encountered objects will be propagated. One possible solution to this is to propagate the type sets lazily, or on-demand. This is to say, type sets get propagated to successor blocks only when the analysis requires information about the type sets along that path.

In this way, it is possible to completely avoid propagating most object properties (including global variables) to most functions. Only the type sets used by a function or its callees need to enter the said function. A proper implementation of such a mechanism can also help improve the analysis precision in some cases. It so happens that propagating type sets to functions indiscriminantly contributes to polluting dataflow information (see Listing 2). This is because type sets are typically merged at the exit of function calls. Thus, not propagating unused type sets to a function will potentially avoid merging multiple type sets for a given variable when that variable is not even used in the said function.

Listing 2: Global Object Pollution

```
1
2  // foo is an empty function
3  function foo() {}
4
5  // the global variable n is implicitly undefined here
6
7  foo();
8
9  var n = 1;
10
11 // n is known to be 1 at this point
12
13 foo();
14
15 // n is inferred to be either 1 or undefined at this point
```

5.2 Selective Path-Sensitivity

Path-sensitivity is useful in order to be able to perform strong updates effectively, but it is very costly to implement. It is interesting to note that we only perform strong updates on singleton object sets. This suggests that general path-sensitivity applied to all object properties at every program point is probably unnecessary for our purposes. Instead, it may be advantageous to restrict our analysis to be path-sensitive for properties of objects that are only part of singleton sets.

Another possible approach would be to reduce path-sensitivity further. Our main use for it is to analyze the initialization of objects so that we can eliminate superfluous `undefined` values. However, once we know that a property cannot contain `undefined`, it may not be so useful to analyze this property in a path-sensitivite manner further along. The type set of this property could then be demoted to a summary type that is analyzed in a flow-insensitive manner. This bears some similarity to the concept of recenty types[3].

The current design of our type analysis not only analyzes object properties in a flow-sensitive manner, but local variables as well, so that their types can be narrowed based on branch tests. Should we alter the design of the analysis so that some variables and properties can selectively be analyzed without flow-sensitivity, it may be advantageous to make it so that most local variables are analyzed without it.

5.3 Type Lattice Depth

The hierarchy of type sets in our analysis can be described as a lattice built on the subset relationship. The original paper describing the SCCP analysis[12] explains that this analyses uses a three-level lattice where all constants sit between the top and bottom elements. This design was selected because it guarantees rapid convergence properties. The value inferred for an SSA temporary can only move up the lattice twice, and so a given instruction can provoke updates by changing its output at most twice.

In our current analysis design, we have chosen to support not only integer constants, but also integer ranges. When an integer range is extended upwards to have an upper bound of m, its upper bound moves to $2^n - 1$ such that n is the least integer with $m \leq 2^n - 1$. We allow this extension up to $2^{16} - 1$. This means there can easily be up to 15 range extension steps when inferring the range of a loop index variable. This causes many re-evaluations of the loop body and surrounding basic blocks. The situation can be compounded in the case of nested loops. This cost could be mitigated by the use of a type-flow graph (see Section 5.4).

Another possibility would be to redesign the range inference mechanism used. In terms of optimization, what we most care about is knowing whether an integer value is non-negative and whether it can fit within a machine integer range (e.g.: a signed or unsigned 32-bit integer). Being able to infer precise ranges in between is probably not all that useful. Hence, we could redesign our range system to infer either an exact bound, or the maximum value allowable by signed and unsigned 32-bit integers. This would allow integer ranges to reach a fixed-point much quicker.

5.4 Type Flow Graph

Our type analysis implementation is currently an abstract interpreter which iterates over instructions of basic blocks sequentially while updating type graph

data structures for each basic block. This is inefficient because during a fixed-point computation, a given basic block may be re-evaluated each time a type set flowing through it changes. This implies re-evaluating each instruction in the block, even if none of them touch the modified type set. Further overhead is incurred because manipulating type graphs implies many lookups and updates on complex data structures.

It may be possible to implement the core of the type analysis more efficiently by taking inspiration from Sparse Conditional Constant Propagation (SCCP). The SCCP analysis[12] is a constant propagation analysis on the SSA intermediate representation. It performs a fixed-point on dataflow edges rather than iterating on individual instructions in order, yielding a more efficient algorithm.

In the case of our type analysis, type flow edges do not directly correspond to def-use edges between SSA temporaries and instructions. This is because some variables can change type at uses, after definition, while object properties are not directly represented in the SSA IR. We could conceivably, however, build a type flow graph that would represent the flow of type sets between instructions. This would allow us to perform abstract interpretation more efficiently while doing away with the time and memory overhead associated with type graphs.

We believe that such an approach could yield much faster convergence while at the same time reducing the memory usage of the analysis to the point where it almost linearly scales with the program size. Another interesting advantage is that the construction of the type flow graph could be made to elegantly integrate with lazy propagation (see Section 5.1). All that is required for this is to allocate type flow edges so that they do not flow into function calls that do not make use of the types they represent.

6 Conclusion

We have presented a static inteprocedural type analysis for JavaScript based on the forward dataflow paradigm. This analysis is flow-sensitive and path-sensitive. Its main focus is to maximize the precision obtained on the analysis of object property types, as objects are a central aspect of JavaScript and its performance characteristics. Although the running time of our analysis currently makes it prohibitive for use in a JIT compiler, the results obtained so far are encouraging and lead us to believe that the analysis has potential for offline optimization of JavaScript code.

The current design of our analysis suffers from scalability issues. We remain confident that these can be fixed and have outlined multiple approaches for doing so. The most promising approach involves the reimplementation of the analysis to use a type flow graph data structure so that its memory usage is reduced and its convergence speed improved. We believe that implementing this approach could potentially reduce the footprint of our analysis by orders of magnitude while retaining most of its benefits.

7 Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) and Mozilla Corporation.

References

[1] M. D. Adams, A. W. Keep, J. Midtgaard, M. Might, A. Chauhan, and R. K. Dybvig. Flow-sensitive type recovery in linear-log time. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 483–498, New York, NY, USA, 2011. ACM.

[2] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type inference of self: Analysis of objects with dynamic and multiple inheritance. In *In ECOOP '93 Conference Proceedings*, pages 247–267. Springer-Verlag, 1993.

[3] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the 13th international conference on Static Analysis*, SAS'06, pages 221–239, Berlin, Heidelberg, 2006. Springer-Verlag.

[4] M. Chevalier-Boisvert, E. Lavoie, M. Feeley, and B. Dufour. Bootstrapping a self-hosted research virtual machine for javascript: an experience report. *SIGPLAN Not.*, 47(2):61–72, Oct. 2011.

[5] D. Gudeman. Representing type information in dynamically typed languages. Technical Report TR 97-27, University of Arizona, October 1993.

[6] P. Heidegger and P. Thiemann. Recency types for analyzing scripting languages. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 200–224, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *Proceedings of the 17th international conference on Static analysis*, SAS'10, pages 320–339, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[10] O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 164–174, New York, NY, USA, 1988. ACM.

[11] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[12] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, Apr. 1991.