TypeCastor: Demystify Dynamic Typing of JavaScript Applications

Shisheng Li
China Runtime Tech Center
Intel China Research Center
Beijing, China
shisheng.li@intel.com

Buqi Cheng
China Runtime Tech Center
Intel China Research Center
Beijing, China
bu.qi.cheng @intel.com

Xiao-Feng Li
China Runtime Tech Center
Intel China Research Center
Beijing, China
xiao.feng.li@intel.com

Abstract
Dynamic typing is a barrier for JavaScript applications to achieve high performance. Compared with statically typed languages, the major overhead of dynamic typing comes from runtime type resolution and runtime property lookup. Common folks' belief is that the traditional static compilation techniques are no longer effective for dynamic languages. The best known JavaScript engines such as Mozilla TraceMonkey and Chrome V8 have developed non-traditional techniques to reduce the runtime overhead. This paper describes TypeCastor, a new JavaScript engine that tries to investigate where and how much the dynamism really is in JavaScript applications, thus to demystify their dynamic typing behavior. To verify our findings, we evaluate TypeCastor with SunSpider benchmark. For type resolution, we find 99% of all the primitive type instances can be statically identified before the program execution. For object property lookup, more than 97% of all runtime property accesses can be satisfied by inline cache. These data mean that the representative JavaScript applications are not that dynamic as people expect, although the language provides the flexible dynamism supports. Though not developed for pure performance, TypeCastor achieves 5.6% and 12.7% higher scores compared to current Chrome V8 and Mozilla TraceMonkey engines respectively.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors---optimization, runtime environments

General Terms
Algorithms, Performance

Keywords
Dynamic typing, JavaScript, type analysis, inline caching

1. Introduction
JavaScript is a popular programming language for client-side web development. Many popular applications, such as Google Desktop Gadgets, Adobe's Flash, and the latest HTML5 standard, use JavaScript as the development language. The flexibility and expressiveness make it easy for web designers to work with. Dynamic typing is one of its major enabling factors.

However, dynamic typing can degrade the performance of the programs. This drawback could impact JavaScript language's applicability in computation intensive applications, which are increasingly common in new web applications. Hence some big industry players remain using statically typed languages for their web applications.

The overhead of dynamic typing comes mainly from the runtime type resolution and runtime property lookup. The community has developed various optimization techniques to solve the performance problem. The techniques mainly include type inference, type specialization, and inline caching. Type inference identifies the variable types based on program analysis or a type system summarized from the language. Type specialization tries to speculate on the variables types based on the runtime type profiling. Inline caching records the results of previous property lookups at the call site, assuming that the objects types are not changed frequently.

The common folks' belief is that JavaScript applications are quite dynamic at runtime hence the traditional compilation techniques may no longer be effective. In this paper, we try to demystify the dynamic behavior of JavaScript programs by identifying where and how much the dynamism really is. The main contributions of the paper include:

1. We design an effective and efficient type analysis algorithm that speeds up the variable type resolution. Type inference and type prediction can be applied to the program variables during the analysis. In our evaluation with SunSpider benchmark, more than 99% of all the primitive type instances can be correctly identified. This finding is significantly different from the traditional impression on the type dynamism of JavaScript programs.

2. We propose "position inline caching" mechanism to speed up the object property access. Different from other inline caching techniques, position inline caching can improve the property access even if the object type is changed at runtime. In our evaluation with SunSpider benchmark, among all the runtime property access instances, 97% of them can successfully hit in the cache. This finding is also different from the traditional impression on the object dynamism of JavaScript programs.

3. We develop the TypeCastor JavaScript engine from the scratch and validate our findings with SunSpider benchmark. Although it is not designed for absolute performance,

TypeCastor gets 5.6% and 12.7% higher scores compared to Chrome V8 and Mozilla TraceMonkey engines respectively.

The paper is organized as follows. Section 2 discusses the related work. Section 3 describes the infrastructure of TypeCastor engine. Section 4 introduces TypeCastor type analysis algorithm and the associated optimizations. Section 5 presents TypeCastor inline caching mechanisms. Section 6 introduces the developed memory optimizations. Section 7 analyzes JavaScript dynamism in SunSpider benchmark with TypeCastor. Section 8 summarizes the paper and discusses future work.

2. Related work

Thiemann [1] proposes a type system to check JavaScript programs statically. It is focused on the type system design and soundness proof. In later work, Heidegger and Thiemann [7] propose a recency-based type system and sketch an inference algorithm in the system. The usage of the recency abstractions [8] for the objects turns the weak object updates in the JavaScript program into strong updates, and increases the analysis accuracy significantly [3] .

Anderson et al [2] design a type system with an inference algorithm for the primitive subset of the JavaScript types. The type system allows objects to be inferred in a controlled manner by classifying the properties as being definite or potential. Their system does not model type change, and the transition between the presence and absence of a property is harder to be predicted than in a recency-based system.

Jang and Choe [9] develop a point-to analysis for JavaScript to enable some program optimizations. They demonstrate their result by applying partial redundancy elimination to the property references. The analysis is flow and context insensitive, and is limited to a subset first-order language.

Jensen et al [3] introduce a lattice model for the static type analysis of the full JavaScript language. Recency abstraction is used for the type representation in the analysis. The algorithm complexity is high in modeling object property.

Chrome V8 [5] is a high performance JavaScript engine developed by Google. Hidden class and method inline caching are the major optimization techniques. Hidden class uses a common data structure to represent the type of the objects that have same properties.

Apple's JavaScript engine SFX (SquirrelFish Extreme) [10] compiles JavaScript code into intermediate bytecode sequence and then interprets the sequence with context-threaded JIT. It develops polymorphic inline caching to speed up the property access. The StructureID [10] concept for inline caching is similar to the hidden class of Chrome V8.

Mozilla TraceMonkey [6] is a trace-JIT extension to SpiderMonkey JavaScript engine [12] . It profiles the program at runtime and identifies the hot traces for cross-procedural type specialization. "Shape" object [19] is used to represent the object type to speed up the property access.

Mozilla Rhino [4] is a JavaScript engine written in Java. It compiles the scripts into Java class files and runs them on a JVM. Rhino's performance is obviously lower than Chrome V8 and Mozilla TraceMonkey in our evaluations with SunSpider benchmark.

3. TypeCastor design overview

We develop TypeCastor to demystify the dynamic behavior of JavaScript program. We also want to understand the implications of dynamic typing and object inheritance to compiler design. To serve the purpose, a phased compilation and optimization model is used in TypeCastor. In the first phase, the high level and sophisticated optimizations are performed with static compilation. In the second phase, the low level and platform specific optimizations are conducted at runtime in an execution engine. The first phase generates statically typed intermediate representation (IR), thus in the second phase a traditional runtime engine such as JVM can be used.

This infrastructure design has following advantages. Firstly, it enables both ahead-of-time (the first phase) and just-in-time (the second phase) compilations, so that we have opportunity to tradeoff the merits and drawbacks between the phases. Secondly, the design does not lose any benefits of type safety and program portability of JavaScript, because the second phase only accepts type safe IR. As a natural result, TypeCastor can be easily ported to other runtime systems. Thirdly, this design matches well with current web-service computing model, where the computation is partitioned between the server and client. In our implementation, we choose Java bytecode as the TypeCastor IR for convenience since it is well designed and we have a runtime engine developed for it already.

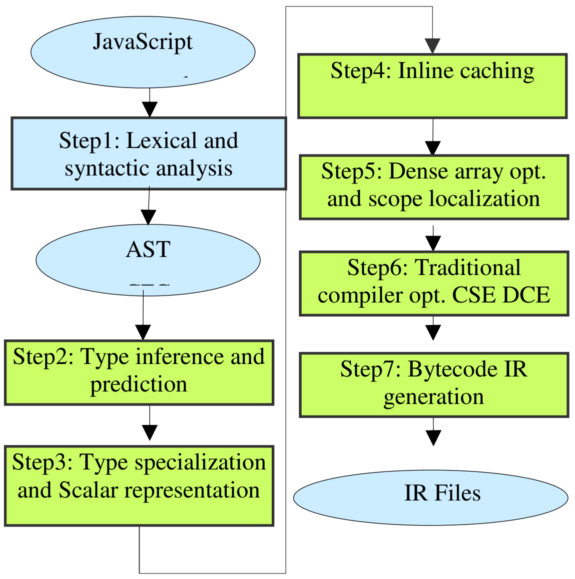

JavaScript
Step1: Lexical and syntactic analysis
AST
Step2: Type inference and prediction
Step3: Type specialization and Scalar representation
Step4: Inline caching
Step5: Dense array opt. and scope localization
Step6: Traditional compiler opt. CSE DCE
Step7: Bytecode IR generation
IR Files

Figure 1. Framework of TypeCastor

Figure 1 illustrates the static compilation part of the TypeCastor framework. In Step 1, the JavaScript source code is transformed into AST and the corresponding control flow graph (CFG). Then in the followed steps, TypeCastor conducts a few analyses and optimizations to tackle the dynamism of JavaScript language. Step 2 (type inference and prediction) and Step 3 (type specialization and scalar representation) are analyses and optimizations for variable type resolution. Step 4 (inline caching) is optimization for object property lookup. Step 5 (dense array optimization and scope localization) are memory optimizations for special objects' property access. Before emitting bytecode IR in Step 7, TypeCastor conducts simple traditional optimizations (CSE and DCE) in Step 6 to reduce the redundancy in the data and code.

4. Type analysis to reduce type resolution overhead

The type analysis in TypeCastor tries to infer the primitive types of the variables before the program execution. For a variable whose type cannot be inferred exactly, TypeCastor predicts its most possible type. Based on the type analysis result, the optimizations like type specialization and scalar representation are applied.

4.1 Constraints for type analysis

TypeCastor develops a constraint-based type analysis algorithm. We extract the expression-based set constraints [13] from the JavaScript language specification [14] , and then use the constraints as the typing rules in the analysis.

Based on the nature of JavaScript language, we divide the expression-level rules into two kinds: type inference rules for result variables and type prediction rules for operand variables. (Table 3 in Appendix gives the concrete constrains for all the expressions in JavaScript.)

An observation of JavaScript language is that, the result type is mostly static, without any runtime dynamism. For example, the result type of "delete" operation is known to be always Boolean. For a few cases where the result type is not statically defined, such as "add" operation, the type still can be inferred from the source operand types.

As a comparison, the types of source operands are usually ambiguous in compilation time. TypeCastor tries to predict their types by using available type information in the expression and by applying best guess based on the common programming practice. For example, if "a" is known to have string type in expression "a + b", TypeCastor predicts that it is highly possible for "b" to have string type as well. This is reflected in the operand typing rule for "e1 + e2": "e2's type set is {String} if e1's type set is {String}".

TypeCastor performs type analysis for all the variables with possible types of "Undefined, Null, Boolean, Number, String, JSObject". This is different from all other known JavaScript type analyses [4] [5] [6] , which only distinguish the Number type and non-Number type, without telling the exact primitive types of those non-Number variables. In our evaluation with SunSpider benchmark, the non-Number primitive types cover 31% of all the type instances; thus it is highly useful to refine their types with extra details.

TypeCastor gives all the objects same type JSObject without further analyzing different object types. We try to get the same benefit with inline caching optimizations. We believe this is a good design tradeoff between cost and benefit. Complete type inference for objects is a NP-complete problem, and at least $O(n^3)$ complexity is required for a reasonable analysis [16] . Simon Holm et al [3] show that the algorithm complexity of a context-sensitive object type analysis is much bigger than that of primitive types' analysis. The reason is that, any difference in the object properties or the scope chains makes different object types, and they can be dynamically generated. As a contrast, there are only a few definite primitive types in JavaScript.

4.2 Flow-sensitive type analysis

TypeCastor performs function-level type analysis by propagating the expression-level type information along the CFG. The function-level analysis achieves both bigger coverage and higher accuracy. The analysis algorithm in TypeCastor is intra-procedural and flow-sensitive. It has two passes. A forward pass propagates the expressive-level inferred types to the entire function, along the define-use chains of the variables; and a backward pass propagates the predicted types along the use-define relations.

In the analysis, type sets will used because a variable may have multiple inferred or predicted types in a function. Although a variable can have a single inferred type at a certain program location, it may have different inferred types at different locations. Meanwhile, a variable can have more than one possible type at one program location, either because the type can dynamically change at runtime, or because the type cannot be identified statically by the algorithm. So type set is an appropriate data structure for the analysis.

In the analysis, every variable will be associated with two kinds of type sets. One kind is *local type set* and the other is *global type set*. The local type sets are used for computation optimizations. A local type set of a variable *v* has all the possible types of *v* at a specific program location. For a single variable *v* at statement *s*, there are two local type sets: the local inference set (*T(S)[v]*) and the local prediction set (*PT(S)[v]*). The global type sets are used for storage optimization. A global type set of a variable *v* contains all the possible types of *v* at different locations of a function. Every variable *v* in the function has two global type sets: the global inference type set (*GT[v]*) and the global prediction type set (*GPT[v]*). Equations (1) to (7) denote how TypeCastor's local type analysis works in a basic block. In the analysis, local type sets are calculated according to equations (1) to (5), and global type sets are calculated by equations (6) and (7).

$$T(S) = in\_T(S) - kill\_T(S) + cg\_T(S) \tag{1}$$

$$PT(S) = in\_PT(S) - ck\_PT(S) + cg\_PT(S) \tag{2}$$

$$cg\_T(S) = \bigcup_{v \in V(S)} \begin{cases} \Omega & in\_T(S)[v] = \emptyset \wedge gen\_T(S)[v] = \emptyset \\ gen\_T(S)[v] & otherwise \end{cases} \tag{3}$$

$$cg\_PT(S) = \bigcup_{v \in V(S) \wedge \Omega \in T(S)[v]} gen\_PT(S)[v] \tag{4}$$

$$ck\_PT(S) = \bigcup_{v \in V(S)} \begin{cases} in\_PT(S)[v] & \Omega \notin T(S)[v] \\ kill\_PT(S)[v] & \Omega \in T(S)[v] \end{cases} \tag{5}$$

$$GT(V) = GT(V) \cup gen\_T(S) \tag{6}$$

$$GPT(V) = GPT(V) \cup cg\_PT(S) \tag{7}$$

In the equations, the type sets of all variables at statement *S* constitute two type sets collections: the local inference type sets collection (*T(S)*) and the local prediction type sets collection (*PT(S)*), so that the type sets can be propagated inner the basic block. *in_T(S)* and *in_PT(S)* are the type sets propagated from the previous statements (For type prediction, the previous statements are the statements before current one in reversed execution order). For the type set analysis inner a statement *S*, *gen_T(S)[v]* and *gen_PT(S)[v]* are used to represent the type sets which are generated by the expression-level type inference rules for the variable *v*; *kill_T(S)* and *kill_PT(S)[v]* are the types sets of variables(or variable *v*) inferred and predicted before *S* that are also inferred and predicted in *S*; *V(S)* is the variable set used in the statement and *V* represents all variables in the function. In the equations, Ø represents the empty type set. For the variables which are not used in the statement, their generated and killed type sets will all be Ø.

Equations (3), (4) and (5) denote how the inferred type sets (*cg_T(S)*), predicted type sets (*cg_PT(S)*) and killed predicted type sets (*ck_PT(S)*) are calculated for each statement *S*. Different with the type inference, type prediction can only be applied to the variables that have no inferred type set before it is used. At the same time, different execution paths may lead to different variable type sets in the flow sensitive analysis. These problems bring about the requirement to trace the un-inferable variable type in the analysis. In the equations, Ω is a new type defined to trace the un-inferable variable type. It means that a variable's type which can be any JavaScript primitive type. In type inference pass, if a variable's type cannot be inferred along an incoming path, it will have a type Ω in its inferred type set. In the type prediction pass, type prediction will only be applied to the variable with Ω in its inferred type set.

$$out\_T(BB) = T(S) \quad S \text{ is the last statement in } BB \qquad (8)$$

$$in\_T(BB) = \bigcup_{p \in pred(BB)} out\_T(P) \qquad (9)$$

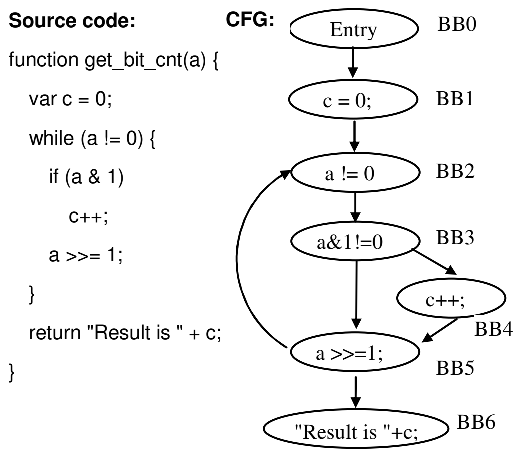
$$out\_PT(BB) = PT(S) \quad S \text{ is the first statement of } BB \qquad (10)$$

$$in\_PT(BB) = \bigcup_{p \in success(BB)} out\_PT(P) \qquad (11)$$

Based on the local type analysis result, global flow sensitive type analysis can be applied to propagate the type information and improve the accuracy of the analysis. To facilitate the analysis, every basic block has live-in/live-out type sets for every variable, including the live-in/live-out inference sets (*in_T(v), out_T(v)*) and the live-in/live-out prediction sets (*in_PT(v), out_PT(v)*). Different with traditional data flow analysis, in the type flow analysis, the live-out type sets of a statement will change according to the change of live-in type sets. So, the type sets of last statement in local type analysis will be live-out type set of the basic block.

The flow equations from (8) to (11) denote the transfer functions used in global type flow analysis. The analysis scans code along the CFG and does the local type analysis and propagates the generated type sets block by block according to these equations. During the CFG scanning, the analysis algorithm checks if the traversal should proceed with current control flow path (or branch) at the entry of every basic block. When a new live-in sets is calculated for a basic block, the algorithm checks if the new live-in set is the same as the block's original live-in set (i.e., *in_T(BB)* in the forward pass, or *in_PT(BB)* in the backward pass). If the live-in sets are different, meaning there are some changes in the variables identified types, the algorithm should continue to analyze the current block. Otherwise, the traversal skips it, and proceeds with the remaining branches. When there is no other branch, this pass terminates.

Table 1 and 2 show how the global type analysis algorithm works on the example in Figure 2. Table 1 presents the process of type inference. Table 2 presents the process of type prediction. To simplify the processes, in our example, each basic block contains single statement, and the live-in/live-out type sets of a basic block equal to the corresponding type sets of the statement in it. From the processes, we can find that: 1) The type of *c* variable in *BB6* will not be predicted due to its inferred type; 2) The type of *a* variable in *BB2* and *BB3* will be predicted because there is a path from entry in which the type of the variable cannot be inferred.


Source code:

```
function get_bit_cnt(a) {
  var c = 0;
  while (a != 0) {
    if (a & 1)
      c++;
    a >>= 1;
  }
  return "Result is " + c;
}
```

Figure 2. Example of global type analysis

Table 1. Type inference process

Basic block	cg_T	Kill_T	In_T	Out_T
0	{{Ω}, Ø}	{Ø , Ø}	{Ø, Ø}	{{Ω}, Ø }
1	{Ø , {Num}}	{Ø , Ø}	{{Ω}, Ø}	{{Ω}, {Num}}
2	{{Ω} , Ø}	{Ø , Ø}	{{Ω} , {Num}}	{{Ω}, {Num}}
3	{{Ω} , Ø}	{Ø , Ø}	{{Ω} , {Num}}	{{Ω}, {Num}}
4	{Ø , {Num}}	{Ø , {Num}}	{{Ω} , {Num}}	{{Ω} , {Num}}
5	{{Num}, Ø}	{{Ω} , Ø}	{{Ω} , {Num}}	{{Num}, {Num}}
2	{{Ω} , Ø}	{Ø, Ø}	{{Num}, {Num}}	{{Num,{Ω}}, {Num}}
3	{{Ω} , Ø}	{Ø, Ø}	{{Num,{Ω}}, {Num}}	{{Num,Ω}}, {Num}}
4	{Ø , {Num}}	{Ø, {Num}}	{{Num,{Ω}}, {Num}}	{{Num,{Ω}}, {Num}}
5	{{Num}, Ø}	{{Num,{Ω}}, Ø}	{{Num,{Ω}}, {Num}}	{{Num}, {Num}}
6	{Ø, Ω}	{Ø, Ø}	{{Num}, {Num}}	{{Num}, {Num}}

Table 2. Type prediction process

Basic block	cg_PT	cg_PT	In_PT	Out_PT
6	{ Ø, Ø}	{ Ø, Ø }	{ Ø, Ø }	{ Ø, Ø }
5	{ Ø, Ø }	{ Ø, Ø }	{ Ø, Ø }	{ Ø, Ø }
4	{ Ø, Ø }	{ Ø, Ø }	{ Ø, Ø }	{ Ø, Ø }
3	{{Num}, Ø }	{ Ø, Ø }	{ Ø, Ø }	{{Num}, Ø }
2	{{Num}, Ø }	{{Num}, Ø }	{{Num}, Ø}	{{Num}, Ø }
5	{ Ø, Ø }	{{Num}, Ø }	{{Num}, Ø }	{ Ø, Ø }
1	{ Ø, Ø }	{ Ø, Ø }	{{Num}, Ø }	{{Num}, Ø }
0	{ Ø, Ø }	{ Ø, Ø }	{{Num}, Ø }	{{Num}, Ø }

Since there are only a handful of primitive types, the number of all different sets under union operation is a small constant. The termination condition of the algorithm checks for the live-in type sets difference. This means the algorithm always converges in *O(nm)* time, where *n* is the number of the basic blocks in the CFG and *m* is the number of variables.

Compared to the type analysis methods developed in other JavaScript type checking tools or JavaScript engines, the TypeCastor algorithm is simpler because it does not distinguish different object types. But it is effective enough in our evaluation. The result shows that statically inferred and correctly predicted type instances can cover more than 99% of all the primitive type instances in the SunSpider benchmark.

4.3 Type analysis based optimizations

Based on the result of type analysis, TypeCastor applied two optimizations to reduce the overhead introduced by dynamic typing. They are static type specialization and scalar representation.

4.3.1 Static type specialization

Type specialization is commonly used in current JavaScript engines to optimize the execution [17] . In other known engines, type specialization is mostly based on runtime type profiling, while in TypeCastor it is based on the static type analysis result. If the type analysis is accurate enough, TypeCastor can avoid the profiling overhead without losing any benefits of the dynamic type specialization. Nonetheless, the two approaches are complementary, i.e., the static type specialization does not exclude the application of dynamic type specialization.

In TypeCastor, type specialization is used for both inferred and predicted types. Since the inferred types are exact at certain program locations, TypeCastor directly generates code for the inferred types. When a variable's inferred types are different at different program locations, the code for type conversion is added before the second access instance. For the predicted types, TypeCastor generates two paths for the same piece of code: One is the fast path specialized with the predicted type and the other is the slow path in case the prediction is wrong.

Source code:

```
funcc add_prefix(x) {
   return "pre"+x;
}
```

Predicted type of x:

```
PT(x)= {String}
```

Generated code:

```
if(Typeof(x)==String){
   //fast path
   s = x.stringValue();
   return "pre" + s;
}else{ //slow path
   sx=ConvertToString(x);
   s = sx.stringValue();
   return "pre" + s;
}
```

Figure 3. Example of type specialization

Figure 3 shows an example of type specialization in pseudo-code. In the example, 'x' is predicted to have string type. The generated code firstly compares the type of 'x' with string type. If the check returns true, the code takes the fast path and loads the string directly. Otherwise it goes to the slow path, where a string conversion is needed before the string loading.

4.3.2 Scalar representation

Since the type of a variable is dynamic at runtime, most JavaScript engines [4] [5] [6] use heap data structure to represent the variable. This is simple for the implementation, but has performance impact due to the memory management and heap access overhead. To improve it, Chrome V8 multiplexes the object reference representation for integer values, and TraceMonkey uses unboxing technique when appropriate.

Different from the others, TypeCastor tries to use scalars to represent JavaScript variables directly. It works in this way:

For a local variable whose type can be statically inferred at certain program location, a scalar is used to represent the variable at that location. Note that different locations may access different scalars of the same variable because it may have different inferred types at different locations.

For a local variable whose types are predicted, TypeCastor generates scalars for it in the same way as for type-inferred variables except that, the allocated space for the variable should be able to hold a type tag and an object reference so that the variable can be represented by a heap data structure in case the prediction is wrong.

For a global variable accessed only in leaf functions, TypeCastor applies the unboxing technique as well as other engines do.

The scalar representation in TypeCastor reduces the amount of memory dereferences dramatically, because the scalars can be largely allocated in the registers by the compiler.

5. Inline caching to reduce property lookup overhead

In JavaScript, an object is actually a mapping from its property names to its property values. New object is created through object inheritance (or called "object cloning" in some literatures). The properties of an object can be dynamically added or removed. Due to the dynamism, object property access is usually slow in JavaScript, since the runtime physical position of a property is hard to determine in the compilation time. Inline caching is commonly used to speed up the property access. Assuming that the object type is not changed frequently, the property related info can be cached in a data buffer or be embedded in code at the access spot (with self-modifying technique) to speed up the property access [4][5][6][10].

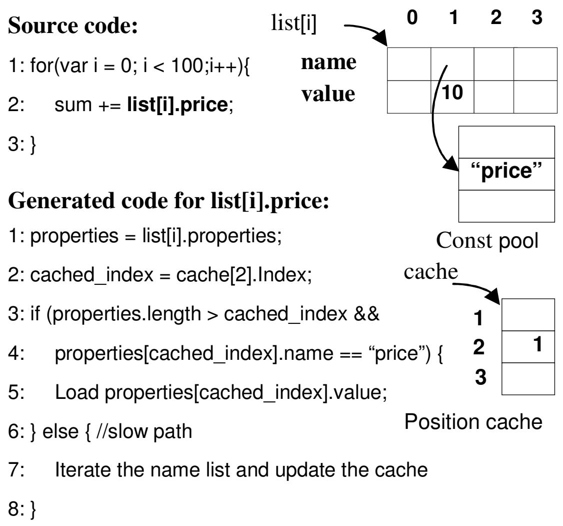
Guarding code is needed to ensure the correctness when the cached info is no longer valid. In current known inline caching solutions, the guarding code is normally to check if the object type is changed. Essentially all these solutions treat the property access problem as a type resolution problem.

Different from the known solutions, TypeCastor develops two new solutions, position inline caching and prototype inline caching. Position inline caching treats the property access problem as a value locating problem. It can improve the property access even if the object type is changed at runtime. Prototype inline caching predicts the property position in the prototype object when the property is not overridden by the object.

5.1 Position inline caching

In TypeCastor, an open-addressing hash-table is used to represent an object. The property name is the hash key. When a property is to be accessed with its name, a hashing function is used to compute the hash-table entry number from the name string, and then access the property value in that entry. The entry number is the data to be cached in the mechanism, which is the position of the property in the object hash-table. Once the position is known, the value in that position (hash-table entry) can be accessed directly.

Position inline caching works as follows. The cache is arranged as an array that is indexed with the source code line number. The array element stores the position of the object property accessed at that line number. At runtime when the code accesses a property with its name string, it firstly loads the property's position value from the cache. Then it loads the property's name string from that entry of the object, and checks if it is the accessed property. If the check returns true, it means a cache hit and the property value is loaded from the same entry. Otherwise, it is a cache miss and a slow path is taken to lookup the property in traditional way. Note that we use immutable strings in TypeCastor to speed up the string comparison.



Source code:

```
1: for(var i = 0; i < 100;i++){
2:     sum += list[i].price;
3: }
```

Generated code for list[i].price:

```
1: properties = list[i].properties;
2: cached_index = cache[2].Index;
3: if (properties.length > cached_index &&
4:     properties[cached_index].name == "price") {
5:     Load properties[cached_index].value;
6: } else { //slow path
7:     Iterate the name list and update the cache
8: }
```

Figure 4. Illustration of position inline caching

Figure 4 illustrates position inline caching with an example. At line number 2 of the source code, there is a property access list[i].price. In the object layout table, the property "price" is stored in entry number 1, which is cached in the position cache buffer[2]. TypeCastor generates inline caching code to access list[i].price.

Another example can exhibit the difference between position inline caching and other traditional mechanism. (Shown as Code A in Appendix). The source code is an algorithm sorting an object list. The objects initially are of same type with a property "value". Then two alternating properties "is_even" and "is_odd" in the objects are initialized to mark different objects. Traditionally the objects with "is_even" or "is_odd" are treated as two different object types. When the code accesses the property "value" in a loop, traditional object inline caching mechanism does not work at all (i.e., the cache always misses) due to the constant change of the object types (with alternating "is_even" or "is_odd" properties.) But TypeCastor works in this case, because the positions of the property "value" are the same across different objects.

We use two versions of the sorting algorithm to show the difference of Chrome V8's hidden class and TypeCastor's position inline caching. Version 1 and version 2 are the same as described above except that version 2 does not have the properties of "is_odd" and "is_even". We measure the performance slowdown when running version 1 compared to version 2. We find that, when introducing properties "is_odd" and "is_even", the program slows down more than 70% with hidden class technique, while the slowdown is less than 1% with position inline caching.

5.2 Prototype inline caching

In JavaScript language, if a prototype property of an object is not overridden, the runtime engine needs to traverse the object inheritance tree to find the target property. This searching process is time consuming. TypeCastor optimizes it by caching the prototype object reference and the property position in the prototype. This is called "prototype inline caching".

Prototype inline caching works in this way: When the position inline caching described in previous subsection fails to hit in the cache for a target property access, the code checks if the following conditions are satisfied:

- The target property in the prototype is not overridden by current object;
- The cached prototype is the prototype of current object;
- The cached position has the target property.

If all the checks return true, the property value can be retrieved directly from the cached prototype property without traversing the inheritance tree.

Prototype inline caching only works when the accessed property is not overridden by the object. This is not a limitation but actually a neat feature. The reason is, if the property is in the object, position inline caching will hit in cache; otherwise, prototype inline caching effects. These two caching mechanisms are very well complementary. The position inline caching is like the first-level cache in microprocessor architecture, and the prototype inline caching is like the second-level cache. In our evaluation, they indeed behave like the microprocessor cache hierarchy.

6. Memory optimizations for special objects property access

Besides the type analysis based optimizations and inline caching optimizations, TypeCastor develops additional techniques to further reduce the dynamic typing overhead. One is related to array access, where traditional array layout is used to represent JavaScript array. The other optimization is about the scope-based data localization.

6.1 Dense array optimization

Dense array refers to the array instance whose element accesses mostly fall into a small range of array indices. In JavaScript language, an array is like a common object except that the properties can be accessed with numerical index. To optimize dense array access, a linear array (with adjacent elements in heap) is used to represent the dense part of the array, and the elements out of the range of the linear array are stored in a hash-table, as usual. The linear array is a C-style reference array, whose elements are object references pointing to the element objects. In this way, most of the array accesses can be satisfied by the linear array indexing without looking up the object hash-table; hence the access time can be reduced.

Different from other JavaScript engines, TypeCastor represents one linear array with a few Java-style scalar arrays. One of the scalar arrays records the types of the array elements (called type array), and each of the rest scalar arrays (called value arrays) only stores the array elements of one type. When a dense array is initialized, two scalar arrays are allocated: One is the type array, and the other is the value array of certain initial type (e.g., array of numbers). The initial type is predicted by the type analysis. Essentially, TypeCastor predicts that all the elements in the linear array have the same type. Since the prediction can be wrong, at runtime if a new type is detected for an element, a new value array will be allocated to store the value of that new type. In the worst case, totally four scalar arrays can be used, i.e., type array, value arrays of number, string and object reference. The assumption is, in most cases, only one value array is needed. Our experiments confirm this assumption with SunSpider benchmark. That is, in most cases, the first two scalar arrays (the type array and value array) are enough throughout the program execution. Using scalar array instead of object reference array can eliminate a memory indirect reference when accessing array element, since the element value can be directly retrieved from the value array.

In case the initial length of the linear array is not enough, TypeCastor can dynamically adjust the length of the linear array by maintaining an array utilization ratio, i.e., the ratio between the number of occupied entries and the linear array length. Every time when the code needs to access a new element that is out of the range of the linear array, the utilization ratio is evaluated. If it is not big enough (< 50% in our setting), the new element is stored in the hash-table as usual. Otherwise, the linear array is expanded to double length to accommodate more elements. In our implementation, the initial array length is 32.

6.2 Scope-based data localization

JavaScript has a special kind of objects called "scope" objects. A scope object is created every time a function is called or a "with" statement is executed. It is eliminated when the function returns or the "with" statement finishes. Every variable of the function is treated as a property of the scope object of that function.

TypeCastor tries to allocate the data structures associated with the variables on runtime stack whenever possible. This has three benefits:

- Reduce the memory management overhead of those data structure;
- Optimize the data accesses in runtime stack;
- Eliminate the unnecessary scope objects.

TypeCastor does scope-based data localization as follows. A scope-tree based algorithm is implemented to determine if a variable is scope local or function local. The algorithm scans the CFG in each scope in a bottom-up way along the scope-tree. If there is no "closure" or "eval" function call in current or child scopes, all the variables in current scope object are scope-local. If the scope local variables are not used in the code of any child scopes, the variables are function-local. Both scope-local and function-local variables can be localized, i.e., allocated on the runtime stack.

7. Experimental evaluations

SunSpider JavaScript benchmark is used to evaluate the techniques we develop in TypeCastor. It is an industry standard benchmark suite provided by Webkit. The benchmark is focused on the tasks in 3D rendering, bit-bashing, cryptographic encoding, code decompression, math kernels, and string processing1.

Apache Harmony [18] is used as the second phase runtime engine of TypeCastor to run the bytecode IR files generated from the static compilation phase. The platform we use is Intel Core2 Quad with 4 GB RAM, running Microsoft Windows XP SP3. All the "average" data below are arithmetic mean.

7.1 Speedups of the optimization techniques

We conduct more experiments to understand the effects of TypeCastor optimization techniques.

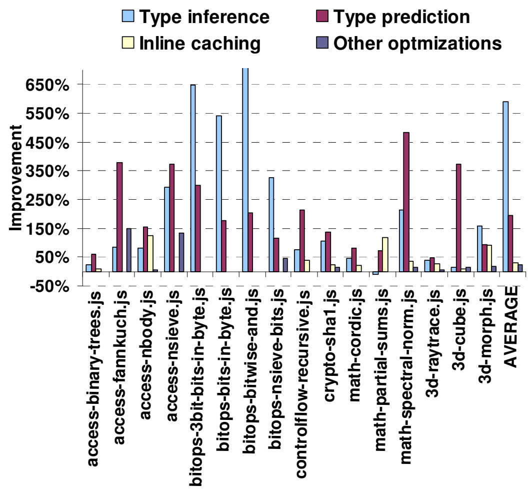


Figure 5. Speedups breakdown for optimizations

Figure 5 shows the breakdown of the performance gains from different optimizations: type inference, type prediction, inline caching and the rest. The inline caching includes both the position and prototype inline caching mechanisms.

From the data we find that the type prediction optimization is the biggest contributor in ten of the 17 benchmarks. This means type prediction does a very good job for the variables whose types are not statically inferable. Based on this observation, it is fair to expect a more complex inference algorithm to infer more types. But if we consider that the extra runtime overhead of a predicted type compared to an inferred type is only a compare-and-branch operation, it might be acceptable to trade the highly complex type inference for a simple but highly accurate type prediction.

From the data we also find that type inference optimization is the biggest contributor in five benchmarks. A closer look at those benchmarks exposes that four of them are integer benchmarks with mainly bit operations. This means type inference works extremely well for bit operations. Overall, 15 of the 16 benchmarks get their major speedups from type analysis based optimizations. In the total performance gain, 98% of it is obtained from type analysis based optimizations. This probably

[1] We exclude the string processing applications in our evaluation, because TypeCastor does not have regular expression support at the moment.

suggests that, static analysis can play an important role in dynamic languages.

Note that inline caching is the biggest speedup contributor for one benchmark. It contributes 31% speedup in average to the benchmark suite, which accounts for more than 1% of the overall performance gain. The rest memory optimizations contribute another 1% to the total performance gain.

7.2 Coverage of type analysis

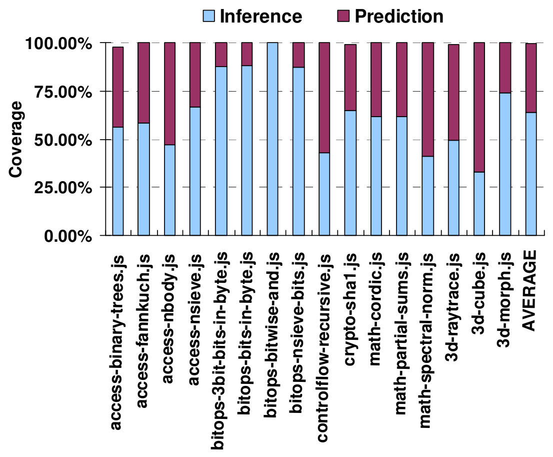


Figure 6. Coverage of type analysis

In order to have a better understanding on the type dynamism, we collect all the real type access instances at runtime, and measure how many of the type instances can be inferred or predicted by TypeCastor. Figure 6 shows the coverage of the successfully inferred and correctly predicted type instances in all the type access instances. We find that type inference covers 63%, and type prediction correctly predicts 36%. Together, type analysis in TypeCastor successfully identified 99% of all runtime type access instances. This means that dynamic typing is not that dynamic as people expect. At the same time, the data suggests that the type analysis algorithm of TypeCastor is highly effective, and the type inference and prediction are very well complementary.

It is worth noting that all the types of bitops-bitwise-and benchmark can be inferred. Examining the source code reveals that there is no property access in the benchmark. This explains why TypeCastor can achieve dramatically better performance than v8 (explained later), which does not have similar type inference.

7.3 Hit ratio of inline caching

In order to understand the object dynamism, we collect the cache hit ratio of the inline caching optimizations in TypeCastor, i.e., how accurate the object property lookups can be predicted. The scalar benchmarks that do not have lots of object property accesses are excluded since they are not representative for the evaluation.

Figure 7 shows the hit ratio data for all the property lookup instances. We find that position inline caching achieves excellent hit ratio (95.2%) in average. Naturally it contributes most in speeding up the property accesses.

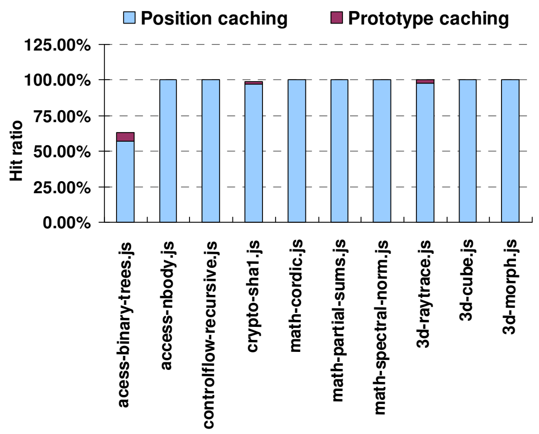


Figure 7. Hit ratios of inline caching

Together with prototype inline caching, close to 100% cache hit ratio (99.9%) is achieved in average excluding benchmark access-binary-trees.js. The reason for the relatively low hit ratio (63%) in access-binary-trees.js is that, many of its property accesses are write operations that produce new properties. These operations cannot hit in cache because the properties are not existent yet at the writing time. It worth noting that the behavior of inline caching in TypeCastor is quite similar to that of cache hierarchy in microprocessor architecture: Most of the accesses hit in first level cache, and almost all the accesses can be satisfied by the two-level cache. Hardware acceleration for JavaScript property access might be able to learn from the traditional cache design.

7.4 Overall performance results

Although we develop the TypeCastor techniques to demystify the dynamic typing behavior, it is still interesting to look at its pure performance with the benchmark. The data are not supposed to be interpreted as a quality judgment of the JavaScript engines. We only want to show how further we can go with the techniques in controlling the dynamism.

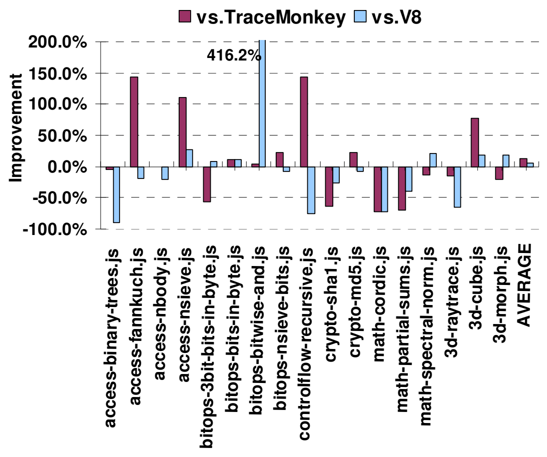


Figure 8. Comparison to V8 and TraceMonkey

We run the benchmark with three engines: TypeCastor, TraceMonkey and Chrome V8. Figure 8 shows the improvements achieved by TypeCastor against TraceMonkey and V8. We can find that TypeCastor performs slightly better than V8 and TraceMonkey, with 5.6% and 12.7% average speedups respectively. TypeCastor has best performance for four of the 16 benchmarks.

It is interesting that V8 obviously does not perform well with bitops-bitwise-and application compared to TypeCastor and TraceMonkey. If without this application, V8 should have much better average performance than TypeCastor. As we have already described, TypeCastor can infer all the types of bitops-bitwise-and. We guess TraceMonkey can identify all the types as well by type tracing, while V8 cannot due to the lack of similar type analysis or type tracing.

8. Discussion and future work

This paper describes the techniques we develop in TypeCastor to demystify the dynamism of JavaScript language. Since our focus is not the ultimate performance, we do not fine tune the optimizations, and we do not develop more optimizations.

In our evaluation, we find that most of the type access instances can be statically analyzed in the compilation time. We demonstrate that type analysis can play a critical role for JavaScript performance improvement. For object property access, we propose new inline caching mechanisms that are more capable than the traditional ones. When the position and prototype inline caching mechanisms work together, most of the object property lookups can be satisfied by the cache. As a side effect, when we apply the techniques to TypeCastor, it achieves best performance among the known engines when evaluated with SunSpider benchmark.

Although the results look promising, there are limitations in our work.

Firstly it is not a complete JavaScript engine. For example, TypeCastor does not support DOM, hence cannot work with a real browser. It is unknown how our analysis techniques perform when applied to real web workloads.

Secondly we do not know if the well-recognized SunSpider benchmark is really representative for JavaScript's dynamic typing behavior. For example, we find the inline caching techniques we develop already satisfy most of the property lookups, which means the object type analysis is not critical for SunSpider's performance. But it does not necessarily mean that the object type analysis is not important for other web workloads.

Finally, we do not know if our techniques can be easily integrated to other known JavaScript engines such as V8 or TraceMonkey. This is one of the major areas we are looking at recently.

For next step, we plan to do following things. We will evaluate our techniques with more workloads. At the same time, we plan to introduce more traditional complier analysis and optimizations into the engine, assuming the dynamism is reduced largely by our techniques. The other area to look at is how to apply our technique into existing engines. It is also interesting is to see the tradeoffs between the static compilation and the runtime engine execution.

9. References

[1] Peter Thiemann. Towards a type system for analyzing javascript programs. In ESOP, pages 408–422, 2005.

[2] Christopher Anderson, et al. Towards type inference for JavaScript. In Proc. 19th European Conference on Object-Oriented Programming, ECOOP '05, volume 3586 of LNCS. Springer-Verlag, July 2005.

[3] Simon Holm Jensen, et al. Type Analysis for JavaScript. SAS, volume 5673 of Lecture Notes in Computer Science, page 238-255. Springer, 2009

[4] Mozilla Rhino. JavaScript for Java. http://www.mozilla.org/rhino/

[5] Mads Sig Ager. V8 Internals. Google IO 2009, http://dl.google.com/io/2009/pres/W_1230_V8 BuildingaHighPerformanceJavaScriptEngine.pdf

[6] Andreas Gal, et al. Trace-based Just-in-Time Type Specialization for Dynamic Languages. In Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation, Dublin, Ireland, 2009.

[7] Phillip Heidegger and Peter Thiemann. Recency types for dynamically-typed object-based languages. In Proc. International Workshops on Foundations of Object-Oriented Languages, FOOL '09, January 2009

[8] Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In Kwangkeun Yi, editor, SAS, number 4134, pages 221–239. Springer, 2006.

[9] Dongseok Jang and Kwang-Moo Choe. Points-to analysis for JavaScript. In Proc.24th Annual ACM Symposium on Applied Computing, SAC '09, Programming Language Track, March 2009

[10] Surfin' Safari. Announcing SquirrelFish Extreme. http://webkit.org/blog/214/ introducing-squirrelfish-extreme/

[11] M. Berndl, et al. Context Threading: a Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters. In 2005 International Symposium on Code Generation and Optimization, p. 15–26, March 2005.

[12] Mozilla.org. SpiderMonkey (JavaScript-C) Engine. http://www.mozilla.org/js /spidermonkey/

[13] Manuel Fähndrich, Alexander Aiken. Program Analysis Using Mixed Term and Set Constraints. Proceedings of the 4th International Symposium on Static Analysis, p.114-126, September 08-10, 1997

[14] Ecma International Organization. Standard ECMA-262 ECMAScript: A general purpose, cross-platform programming language. http://www.ecmainternational.org/publications/standards/ Ecma-262.htm

[15] WebKit Organization. SunSpider JavaScript Benchmark. http://www2.webkit. org/perf/ sunspider-0.9/sunspider.html

[16] M. Bugliesi and S. M. Pericas-Geertsen. Type inference for variant object types. Information and Computation, v.177 n.1, p.2-27, 25 August 2002

[17] M. Chang, et al. Efficient Just-In-Time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical Report ICS-TR-07-10, Donald Bren School of Information and Computer Science, University of California, Irvine, 2007.

[18] The Apache Software Foundation. Apache Harmony. http://harmony.apache.org.

[19] Mozilla Corporation. SpiderMonkey Internals. https://developer.mozilla.org/En/SpiderMonkey/Internals/Property_cache

[20] Chambers, Ungar, Lee .An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. Lisp and Symbolic Computation, 4, 3, 1991

Appendix

Table 3. Expression based set constraints for JavaScript.

The first column "Expression" has the operation definitions. The second column "Result typing rule" gives the constraints for result operand, and the third column (with two sub-columns) "Operand typing rule" gives the constraints for source operand(s).

Notations:

1. e,e1,e2,… : expressions.
2. p,p1,p2,…: property names.
3. a1,a2,…: formal argument names.
4. v: variable names.
5. T(e): the type set of the expression.
6. U: Set of all possible types.
7. JSObject: JavaScript object.

Expression	Result typing rule	Operand typing rule	
c (constant)	T(c)	c	U
v (variable)	T(v)	v	U
new e(e1,e2,…)	{JSObject}	e	{JSObject}
		e1,e2,…	U
this	{JSObject}		
{p1:e1,p2:e2,pn:en}	{JSObject}	e1, e2…	U
function (a1,a2,…){s}	{JSObject}		
e.p	U	e	{JSObject}
e[e1]	U	e	{JSObject}
		e1	{Number}
e(e1,e2,…)	U	e	{JSObject}
		e1,e2,…	U
v++/v--/++v/--v	{Number} T(v)={Number}	v	{Number}
+e/-e/~e/ e++/e--/++e/--e	{Number}	e	{Number}
!e	{Boolean}	e	{Boolean}
delete e.p	{Boolean}	e	{JSObject}
delete e[e1]	{Boolean}	e	{JSObject}
		e1	{Number}
{e1 op e2 | op∈ {-, *, /, %, <<, >>, >>>, |, &, ^}}	{Number}	e1	{Number}
		e2	{Number}
{e1 op e2 | op∈ {<, <=, >, >= }}	{Boolean}	e1	{String} if T(e2) = {String} {Number} otherwise
		e2	{String} if T(e1) = {String} {Number} otherwise
{e1 op e2 op∈ {==, ===, !==, != }}	{Boolean}	e1	T(e2) if T(e2)!= U {Number} otherwise
		e2	T(e1) if T(e1)!= U {Number} otherwise
e1 in e2	{Boolean}	e1	{String}
		e2	{JSObject}
e1 instanceof e2	{Boolean}	e1	{JSObject}
		e2	{JSObject}

e1 + e2	{String} if T(e1) = {String} or T(e2) = {String}; {Number} if $\{String\} \not\subset (T(e1) \cup T(e2))$ and $\{JSObject\} \not\subset (T(e1) \cup T(e2))$; {Number, String} otherwise	e1	{String} if T(e2)= {String}; {Number} otherwise
		e2	{String} if T(e1) = {String}; {Number} otherwise
e1 && e2	$(T(e1) - \{JSObject\}) \cup T(e2)$	e1	{Boolean}
		e2	{Boolean}
e1 || e2	$(T(e1) - \{Null, Undefined\}) \cup T(e2)$	e1	{Boolean}
		e2	{Boolean}
e1 ?e2 : e3	T(e2) if T(e1)={JSObject}; T(e3) if $T(e1) \subseteq \{Undefined, Null\}$; $T(e2) \cup T(e3)$ otherwise	e1	{Boolean}
		e2	{Number}
		e3	{Number}
e1,e2	T(e2)	e1	U
		e2	U
v=e1	T(e1) T(v) = T(e1)	e1	U
e.p=e1	T(e1)	e	{JSObject}
		e1	U
e[e1] = e2	T(e2)	e	{JSObject}
		e1	{Number}
		e2	U

Code A: Revised bubble sorting algorithm

```
var ELEMENT_CNT = 4000;
var TOTAL_LOOP_CNT = 100;
var list;
for(var i = 0;i < TOTAL_LOOP_CNT;i++){//LOOP1
  list = new Array(ELEMENT_CNT);
  var n = list.length;
  for(var i = 0; i < ELEMENT_CNT; i++) {//LOOP2
    list[i].value = Math.random();
    if(rand_integer() % 2 == 0) {
      list[i].is_even = true;
    } else {
      list[i].is_odd = true;
    }
  }
  for(var i = 0;i < n;i++) {//LOOP3
    for(var j = n - 2; j >= i;j--) {//LOOP4
      var x = list[j];
      var y = list[j + 1];
      if(x.value > y.value) {
         list[j] = y;
         list[j + 1] = x;
}}}}
```