RATA: Rapid Atomic Type Analysis by Abstract Interpretation – Application to `JavaScript` Optimization

Francesco Logozzo and Herman Venter

Microsoft Research, Redmond, WA (USA)
{logozzo,hermanv}@microsoft.com

Abstract. We introduce RATA, a static analysis based on abstract interpretation for the rapid inference of atomic types in `JavaScript` programs. RATA enables aggressive type specialization optimizations in dynamic languages. RATA is a combination of an interval analysis (to determine the range of variables), a kind analysis (to determine if a variable may assume fractional values, or `NaN`), and a variation analysis (to relate the values of variables). The combination of those three analyses allows our compiler to specialize `Float64` variables (the only numerical type in `JavaScript`) to `Int32` variables, providing large performance improvements (up to 7.7×) in some of our benchmarks.

1 Introduction

`JavaScript` is probably the most widespread programming platform in the world. `JavaScript` is an object-oriented, dynamically typed language with closures and higher-order functions. `JavaScript` runtimes can be found in every WEB browser (*e.g.*, Internet Explorer,Firefox, Safari and so on) and in popular software such as Adobe Acrobat and Adobe Flash. Large and complex WEB applications such as Microsoft Office WEB Apps or Google Mail, rely on `JavaScript` to run inside every browser on the planet.

A fast `JavaScript` implementation is crucial to provide a good user experience for rich WEB applications and hence enabling their success. Because of its dynamic nature, a `JavaScript` program cannot statically be compiled to efficient machine code. A fully interpreted solution for `JavaScript` runtime is generally acknowledged to be too slow for the new generation of web applications. Modern implementations rely on Just-in-time (JIT) techniques: When a function `f` is invoked at runtime, `f` is compiled to a function `f′` in machine code, and it is then executed. The performance gain of executing `f′` pays off the extra time spent in the compilation of `f`. The quality of the code that the JIT generates for `f′` depends on the amount of dynamic and static information that is available to it at the moment of the invocation of `f`. For instance, if the JIT knows that a certain variable is of an atomic type then it generates specialized machine instructions (*e.g.*, `incr` for an `Int32`) instead of relying on expensive boxing/unboxing operations.

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 66–83, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Motivating Example. Let us consider the `nestedLoops` function in Fig. 1. Without any knowledge of the concrete types of `i` and `j`, the JIT should generate a value wrapper containing: (i) a tag with the dynamic type of the value, and (ii) the value. Value wrappers are disastrous for performance. For instance, the execution of `nestedLoops` takes 310ms on our laptop. [1] In fact, the dynamic execution of the statement `i++` involves: (i) an "unbox" operation to fetch the old value of `i` and check that it is a numerical type; (ii) incrementing `i`; (iii) a "box" operation to update the wrapper with the new value. The JIT can specialize the function if it knows that `i` and `j` are numerical values. In `JavaScript`, the only numerical type is a 64 bits floating point (`Float64`) which follows the IEEE754 standard [16,19]. In our case, a simple type inference can determine that `i` and `j` are `Float64`: they are initialized to zero and only incremented by one. The execution time then goes down to 180ms.

The JIT may do a better job if it knows that `i` and `j` are `Int32`: floating point comparisons are quite inefficient and they usually requires twice or more instructions to perform than integer comparisons on a x86 architecture. A simple type inference does not help, as it cannot infer that `i` and `j` are bounded by 10000. In fact, it is safe to specialize a numerical variable `x` with type `Int32` when one can prove that for all possible executions:

(i) `x` never assumes values outside of the range $[-2^{31}, 2^{31}-1]$; *and*
(ii) `x` is never assigned a fractional value (*e.g.*, 0.5).

Contribution. We introduce RATA, Rapid Atomic Type Analysis, a new static analysis based on abstract interpretation, to *quickly* and *precisely* infer the numerical types of variables. RATA is based on a combination of an interval analysis (to determine the range of variables), a kind analysis (to determine if a variable may assume fractional values, or `NaN`) and a variation analysis (to relate the values of variables). In our example, the first analysis discovers that $\texttt{i} \in [0, 10000], \texttt{j} \in [0, 10000]$ and the second that $\texttt{i}, \texttt{j} \in \mathbb{Z}$. Using this information, the JIT can further specialize the code so that `i` and `j` are allocated in integer registers, and as a matter of fact the execution time (inclusive of the analysis time) drops to 31ms!

The function `bitsinbyte` in Fig. 1 (extracted from the SunSpider benchmarks [31]) illustrates the need for the variation analysis. The interval analysis determines that $\texttt{m} \in [1, 256], \texttt{c} \in [0, +\infty]$. The kind analysis determines that $\texttt{m}, \texttt{c} \in \mathbb{Z}$. If we infer that $\texttt{c} \leq \texttt{m}$ then we can conclude that `c` is an `Int32`. In general, we can solve this problem using a relational, or weakly relational abstract domain, such as Polyhedra [12], Subpolyhedra [23], Octagons [26], or Pentagons [24]. However, all those abstract domains have a cost which is quadratic (Pentagons), cubic (Octagons), polynomial (Subpolyhedra) or exponential (Polyhedra) and hence we rejected their use, as non-linear costs are simply not tolerable at runtime. Our variation analysis infers that: (i) `m` and `c` differ by one

[1] The data we report is based on the experience with our own implementation of a `JavaScript` interpreter for `.Net`. More details will be given in Sect. 6.

```
function nestedLoops()
{
  var i, j;
  for(i = 0; i < 10000; i++)
    for(j = 0; j < i; j++) {
      // do nothing...
    }
}
```

```
function bitsinbyte(b) {
  var m = 1, c = 0;
  while(m<0x100) {
    if(b & m) c++;
    m <<= 1;
  }
  return c;
}
```

Fig. 1. Two small `JavaScript` functions showing the impact of type specialization on performance. With no type information the execution of `nestedLoops` takes 310ms, when `i` and `j` are treated as `Float64` it takes 180ms and when they are treated as `Int32` it only takes 31ms. To infer `i`, `j` to be `Int32`, one needs a more powerful analysis than a simple type inference. In `bitsinbyte` one needs to discover that `c` is bounded by `m` in order to determine thas is an `Int32`.

at the entry of the loop; (ii) `c` is incremented by 0 or 1 at each loop iteration; and (iii) `m`, the guard variable of the loop, monotonically increases at each loop iteration (even if non-linearly). As a consequence, $\mathtt{m} \leq 256$ implies that $\mathtt{c} \leq 256$, which combined with the interval and kind information allows the analysis to conclude that `c` is a `Int32`.

The precision of RATA is in between Intervals [10] and Octagons. It is more precise than Intervals, as it can express kind information and relative variable growth. It is less precise than Octagons, for whereas the Octagon abastract domain can exactly represent relations such as $\mathtt{c} \leq \mathtt{m} \wedge \mathtt{m} \leq 256$ our analysis considers the weaker property $\exists \mathtt{x} \in \mathtt{Vars}.\mathtt{c} \neq \mathtt{x} \wedge \mathtt{c} \leq \mathtt{x} \wedge \mathtt{x} \leq 256$. It is worth remarking that RATA is designed to be very fast, to be invoked by the JIT at runtime, and to be used for program optimization.

2 The JavaScript$^{=}$ Language

We illustrate our analysis using a small untyped imperative language, `JavaScript`$^{=}$, defined in Fig. 2, which models the subset of `JavaScript` we consider in our analysis. A program is a sequence of function declarations and a statement (the global statement). For simplicity we assume functions to have only one parameter. Local (global) variables are declared with the `var` (`global`) keyword. The `JavaScript` language does not provide immediate syntax to differentiate globals from locals, which can be easily determined by the parser. The difference is relevant for the soundness of our analysis, so we make the distinction explicit in the syntax of `JavaScript`$^{=}$. Variable assignment, function invocation, statement concatenation, loop, conditional are as usual. The statement $\mathtt{Ignored}_{\mathtt{C}}$ abstracts the language statements which do not affect locals such as object creation, closures and so on. The statement $\mathtt{Havoc}_{\mathtt{C}}$ models any statement that we do not consider in the analysis, and that may have some effect of locals, *e.g.* `throw` and `eval`. To ease the presentation we admit only strict inequalities and equalities for guards. Expressions can be constants or variables, and they are

$$
\begin{aligned}
\texttt{Prog} &::= \texttt{F C} \\
\texttt{F} &::= \texttt{function f(x) \{C\} F} \mid \epsilon \\
\texttt{C} &::= \texttt{var x;} \mid \texttt{global x;} \mid \texttt{x = e;} \mid \texttt{x = f(e);} \mid \texttt{C C} \mid \texttt{while(b) \{C\};} \\
&\quad \mid \texttt{if(b) \{C \}else \{C \};} \mid \texttt{Havoc}_\texttt{C}\texttt{;} \mid \texttt{Ignored}_\texttt{C} \\
\texttt{b} &::= \texttt{e} < \texttt{e} \mid \texttt{e} \leq \texttt{e} \mid \texttt{e} == \texttt{e} \\
\texttt{e} &::= k \mid \texttt{x} \mid \texttt{e + e} \mid \texttt{e op}_{num}\texttt{e} \mid \texttt{e op}_{int}\texttt{e} \mid \texttt{Ignored}_\texttt{e} \\
k &::= \texttt{NumericalConstant} \mid \texttt{StringConstant} \mid \texttt{Ignored}_\texttt{k} \\
\texttt{op}_{int} &::= << \mid >> \mid \& \mid \hat{} \qquad \texttt{op}_{num} ::= / \mid * \mid \% \mid -
\end{aligned}
$$

Fig. 2. The syntax of the `JavaScript`$^=$ language

combined with binary operators. We distinguish three kinds of binary operators: (i) sum, +, which can be either the usual IEEE754 addition when its operands are numerical values or string concatenation otherwise; (ii) numerical operations which return a numerical value (or `NaN` if the operation is undefined, *e.g.* 0/0); (iii) int operations, which always return a `Int32` value. The expression $\texttt{Ignored}_\texttt{e}$ abstracts the expressions that we do not consider here such as Boolean operators and casting. A constant can either be an IEEE754 64-bits numerical constant, a string literal or some constant we do not deal with (*e.g.*, Boolean constants).

It is worth mentioning that even if in the definition of `JavaScript`$^=$ we ignore some language constructs, our implementation takes care of them *e.g.* by syntax rewriting ("x+ = 2" → "x = x + 2").

3 Background

3.1 IEEE754 Standard

The IEEE754 standard defines, among other things, the arithmetic format for floating point computations. When using 64-bits (`Float64`), the standard format allows numbers as large as $\pm 1.7976931348623157 \cdot 10^{308}$ and as small as $\pm 5 \cdot 10^{-324}$ to be represented. All the integers between -2^{53} and 2^{53} are exactly represented. Outside of this interval, one may lose precision in the trailing digits. Unlike machine integers: (i) `Float64` numbers do not overflow, and (ii) two special values represent infinities: $\pm\infty$. For instance, $1/0 = +\infty = +\infty + 10$. The `Float64` format also specifies a special value `NaN` (Not-a-Number) as the result of invalid operations, *e.g.*, ∞/∞. A peculiar property of `NaN` is that `NaN` $\neq$ `NaN`.

One can specialize a `Float64` variable `x` to a `Int32` without changing the semantics of the program if one can prove that `x` will never assume: (i) a fractional value, a `NaN` or an infinity; and (ii) a value outside of the range $[-2^{31}, 2^{31} - 1]$. The goal of RATA is to enable such specialization.

3.2 Abstract Interpretation

Abstract interpretation [10,11] is a general theory of semantic approximations. Its more interesting application is to define and prove soundness of program analyses. From the abstract interpretation perspective, a static analysis is a program

semantics that is coarse enough to be computable and precise enough to capture the properties of interests. The concrete semantics of a program is defined over a complete lattice $\langle \mathsf{C}, \sqsubseteq \rangle$. The abstract semantics is defined as a fixpoint over a complete lattice $\langle \mathsf{A}, \bar{\sqsubseteq} \rangle$, which is related to C by a Galois connection, *i.e.*, a pair of monotonic functions $\langle \alpha, \gamma \rangle$ such that $\forall \mathsf{c} \in \mathsf{C}.\ \mathsf{c} \sqsubseteq \gamma \circ \alpha(\mathsf{c})$ and $\forall \bar{\mathsf{a}} \in \mathsf{A}.\ \alpha \circ \gamma(\bar{\mathsf{a}}) \bar{\sqsubseteq} \bar{\mathsf{a}}$. We write $\langle \mathsf{C}, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathsf{A}, \bar{\sqsubseteq} \rangle$ to denote that. An abstract transfer function $\bar{\tau}$ is a sound approximation of a concrete τ if $\forall \bar{\mathsf{a}} \in \mathsf{A}.\ \alpha \circ \tau \circ \gamma(\bar{\mathsf{a}}) \bar{\sqsubseteq} \bar{\tau}(\bar{\mathsf{a}})$. In general, the abstract domain A may contain strictly increasing infinite (or very long) chain. Hence, to ensure the convergence of fixpoint iterations one should use a widening operator, which extrapolates the limit of the sequence. Precision lost by the widening can be recovered using a narrowing operator.

4 Numerical Abstract Domains

The Rapid Atomic Type Analysis (RATA) is meant to be used in an online context, as an oracle for the JIT that can use the inferred types to generate more specialized code. RATA is a combination of three different static analyses. An interval analysis to determine the range of the variables. A kind analysis to infer if a variable can assume a fractional or a `NaN` value. A variation analysis to infer loose relationships about program variables, and hence refine the ranges and the kinds. The analysis should be very fast, to avoid causing untoward pauses in normal program execution. We rejected the use of expressive yet expensive numerical abstract domains. For instance, Octagons have a cubic complexity (in the number of program variables), Polyhedra are exponential, and Subpolyhedra lay in between.

4.1 Extended Intervals

The interval abstract domain was introduced by Cousot & Cousot in [10] as example of the application of Abstract Interpretation to program optimization (specifically array bounds check removal). Inspired by this idea, we use it for type specialization. Our extended intervals are a little bit different from the originals, in that we also consider intervals potentially containing `NaN`, intervals abstracting non-numerical values, intervals abstracting floats and intervals bounded *only* by `Int32` values. An interval can either be the empty interval, the interval containing only `NaN`, a `Int32`-bounded interval, an open interval or the unknown interval ($\top_i$):

$$\begin{aligned} \mathsf{Intv} = \bot_i \mid \mathsf{NaN} \mid \mathsf{Normal}(a, b) \mid \mathsf{OpenLeft}(b) \mid \mathsf{OpenRight}(a) \mid \top_i \\ a, b \in \mathtt{Int32} \end{aligned}$$

More formally, the meaning of an interval is given by the concretization function $\gamma_i \in [\mathsf{Intv} \to \mathcal{P}(\mathsf{Val})]$. The set Val is the set of concrete $\mathtt{JavaScript}^{=}$ values. We are interested only in numerical values, so we let $\mathsf{Val} = \mathsf{Ignored}_{\mathsf{Val}} \cup \mathbb{R} \cup \{\pm\infty, \mathtt{NaN}\}$. For simplicity, we let $\mathbb{R}^* = \mathbb{R} \cup \{\pm\infty, \mathtt{NaN}\}$, and we extend the usual axioms over reals so that $\forall r \in \mathbb{R}. -\infty < r < +\infty$ and $\forall r \in \mathbb{R}^*.\ r \neq \mathtt{NaN}$. The concretization function and the induced order $\sqsubseteq_i$ are in Fig. 3.

$$\begin{aligned}
\gamma_i(\bot_i) &= \emptyset \\
\gamma_i(\mathsf{NaN}) &= \{\mathtt{NaN}\} \\
\gamma_i(\mathsf{Normal}(a,b)) &= \{r \mid r \in \mathbb{R}, a \le r \le b\} \cup \{\mathtt{NaN}\} \\
\gamma_i(\mathsf{OpenLeft}(b)) &= \{r \mid r \in \mathbb{R}, r \le b\} \cup \{\mathtt{NaN}\} \\
\gamma_i(\mathsf{OpenRight}(a)) &= \{r \mid r \in \mathbb{R}, a \le r\} \cup \{\mathtt{NaN}\} \\
\gamma_i(\top_i) &= \mathsf{Val}
\end{aligned}$$

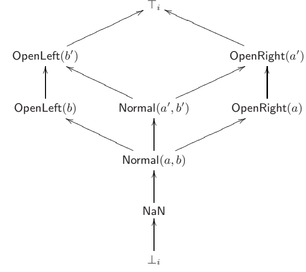

Fig. 3. The concretization γ_i and the order $\sqsubseteq_i$ on the extended intervals. We assume that $a' \le a \le b \le b'$.

Example 1. $\gamma_i(\mathsf{OpenRight}(10)) = \{10 \ldots 11 \cdots + \infty\} \cup \{\mathtt{NaN}\}$.

The abstraction function $\alpha_i \in [\mathcal{P}(\mathsf{Val}) \to \mathsf{Intv}]$ is defined as

$$\alpha_i(R) = \bigsqcup_i \{\dot{\alpha}_i(r) \mid r \in R\} \text{ where}$$
$$\dot{\alpha}_i(r) = \begin{cases} \mathsf{NaN} & r \text{ is } \mathtt{NaN} \\ \mathsf{OpenRight}(2^{31}-1) & 2^{31}-1 < r \le +\infty \\ \mathsf{OpenLeft}(-2^{31}) & -\infty \le r < -2^{31} \\ \mathsf{Normal}(\mathsf{floor}(r), \mathsf{ceiling}(r)) & -2^{31} \le r \le 2^{31}-1 \\ \top_i & \text{otherwise} \end{cases}$$

($\mathsf{floor}(r) = \max\{x \in \mathbb{Z} \mid x \le r\}$ and $\mathsf{ceiling}(r) = \min\{x \in \mathbb{Z} \mid r \le x\}$).

Example 2. $\alpha_i(\{10.3, +\infty, \mathtt{NaN}\}) = \mathsf{OpenRight}(10)$, $\alpha_i(\{3.14\}) = \mathsf{Normal}(3, 4)$.

Theorem 1. $\langle \mathcal{P}(\mathsf{Val}), \subseteq \rangle \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} \langle \mathsf{Intv}, \sqsubseteq_i \rangle$.

It is worth noting that for Th. 1 to hold we need to map $\pm\infty$ to the smallest abstract element containing $\pm\infty$.

The abstract domain Intv is precise enough to capture that the value of a variable is always within the `Int32` range, but it cannot capture the fact that a variable never assumes fractional values, crucial for soundness : *e.g.*, 1/2 is 0.5 with `Float64` semantics and 0 with `Int32` semantics.

Example 3. For the function `bitsinbyte`, the analysis with Intv infers that `m` : $\mathsf{Normal}(0, 512)$, `c` : $\mathsf{OpenRight}(0)$, `b` : $\top_i$.

4.2 Kinds

The elements of the Kind abstract domain are either the empty kind, a 32-bits integer, a 64-bit floating point number or an unknown kind of value:

$$\mathsf{Kind} = \bot_k \mid \mathtt{Int32} \mid \mathtt{Float64} \mid \top_k.$$

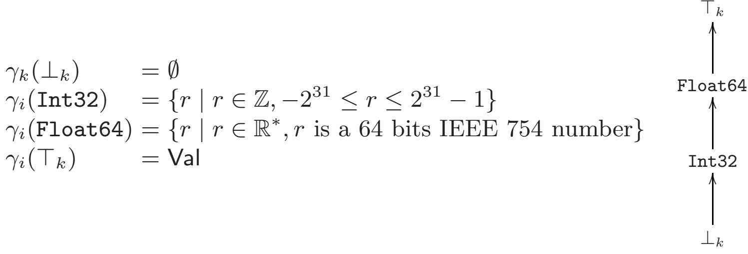

Fig. 4. The concretization γ_k and the order $\sqsubseteq_k$ of the Kinds

The meaning function $\gamma_k \in [\mathsf{Kind} \rightarrow \mathcal{P}(\mathsf{Val})]$ and the induced order $\sqsubseteq_k$ are in Fig. 4. The abstraction function $\alpha_k \in [\mathcal{P}(\mathsf{Val}) \rightarrow \mathsf{Kind}]$ is

$$\alpha_k(R) = \bigsqcup_k \{\dot{\alpha_k}(r) \mid r \in R\} \text{ where } \dot{\alpha_k}(r) = \begin{cases} \texttt{Int32} & r \text{ is a } \texttt{Int32} \\ \texttt{Float64} & r \text{ is a } \texttt{Float64} \\ \top_k & r \text{ otherwise} \end{cases}$$

Example 4. $\alpha_k(\{10.3, +\infty, \texttt{NaN}\}) = \alpha_k(\{3.14\}) = \texttt{Float64}$.

Theorem 2. $\langle \mathcal{P}(\mathsf{Val}), \subseteq \rangle \underset{\alpha_k}{\overset{\gamma_k}{\leftrightarrows}} \langle \mathsf{Kind}, \sqsubseteq_k \rangle$.

The abstract domain of Kind in isolation is of almost no use (maybe except for trivial, loop free programs). In the `nestedloops` example, knowing that `i` is initialized to a `Int32`, it is compared to a `Int32`, and only incremented by one it is not enough to deduce that `i` is an `Int32`. In fact if the loop guard were instead `i`$\leq 2^{31}$, then after the last iteration of the loop `i` $= 2^{31} + 1$ which is a fine `Float64` value, but not an `Int32`.

4.3 K-Intervals

The combination of extended intervals and kinds allow the derivation of very powerful yet rapid analyses. We call the reduced product of Kind and Intv a k-interval. The elements of the abstract domain are pairs in $\mathsf{Intv} \times \mathsf{Kind}$, and the concretization $\gamma_{ki} \in [\mathsf{Intv} \times \mathsf{Kind} \rightarrow \mathcal{P}(\mathsf{Val})]$ is $\gamma_{ki}(\langle i, k \rangle) = \gamma_i(i) \cap \gamma_k(k)$. The abstraction $\alpha_{ki} \in [\mathcal{P}(\mathsf{Val}) \rightarrow \mathsf{Intv} \times \mathsf{Kind}]$ is the simple pairwise abstraction: $\alpha_{ki}(R) = \langle \alpha_i(R), \alpha_k(R) \rangle$. The order $\sqsubseteq_{ki}$ is the pairwise extension of the order on the basic domains. We write `x` $: t$ to denote that the variable `x` has a k-interval t.

Theorem 3. $\langle \mathcal{P}(\mathsf{Val}), \subseteq \rangle \underset{\alpha_{ki}}{\overset{\gamma_{ki}}{\leftrightarrows}} \langle \mathsf{Intv} \times \mathsf{Kind}, \sqsubseteq_{ki} \rangle$.

K-Intervals are more expressive than the single domains and can represent addition information, crucial to type specialization:

```
function loop() {
  var x;
  x = 0;
  while(x < 10000) {
    x = x + 1;
  }
}
```

```
function loopToN(n) {
  var x;
  x = 0;
  while(x < n) {
    x = x + 1;
  }
}

loopToN(99999);
loopToN(1234);
```

Fig. 5. In order to infer that `x : Int32`, in the first example RATA uses a widening with a threshold, and in the second a narrowing as re-execution. The function `loopToN` is analyzed at the first invocation, it is specialized for `Int32`, and the specialization is re-used at the second invocation.

Example 5. The k-interval $t = \langle \mathsf{OpenRight}(10), \mathtt{Int32} \rangle$ represents the set of `Int32` larger than or equal to 10:

$$\gamma_{ki}(t) = \{10 \ldots 11 \cdots + \infty, \mathtt{NaN}\} \cap \{r \mid r \in \mathbb{Z}, -2^{31} \leq r \leq 2^{31} - 1\} \\ = \{10, 11 \ldots 2^{31} - 1\}.$$

It is worth noting that $t' = \langle \mathsf{Normal}(10, 2^{31} - 1), \mathtt{Int32} \rangle$ is such that $\gamma_{ki}(t') = \gamma_{ki}(t)$, so that t and t' are two abstract elements with the same concretization. To keep a low analysis overhead, we do *not* impose a canonical form for abstract elements.

Roughly, if the analysis determines that $\mathtt{x} : t$ then there exists a variable `y`, which is known to be an `Int32`, such that $\mathtt{x} \leq \mathtt{y}$. This information is weaker than that one gets for instance with Octagons, which automatically discovers the particular `y` and $v \geq 0$ such that $\mathtt{x} \leq \mathtt{y} - v$.

5 Rapid Atomic Type Analysis

The Rapid Atomic Type Analysis is defined by structural induction on the program syntax. It has two main phases: (i) numerical invariant inference with $\mathsf{Intv} \times \mathsf{Kind}$; and (ii) type refinement via variation analysis.

5.1 Numerical Analysis

The numerical invariant analysis $\mathbb{N}[\![\cdot]\!]$ infers, for each program point an abstract state $\sigma \in \Sigma = [\mathtt{Vars} \rightarrow \mathsf{Intv} \times \mathsf{Kind}]$, that is a map from variables to k-intervals.

Invocation of the Analysis. When the JIT encounters a function call $\mathtt{f}(v)$, where v is a value of dynamic type t, it first searches the cache to see if it has already specialized `f` for the type t. If this is not the case, it invokes RATA to infer the atomic numerical types for `f`'s locals, to be used for type specialization.

$$
\begin{array}{ll}
\mathsf{eval}(k,\sigma) & = \begin{cases} \langle \mathsf{Normal}(k,k), \mathtt{Int32}\rangle & k \text{ is } \mathtt{Int32} \\ \langle \top_i, \mathtt{Float64}\rangle & k \text{ is } \mathtt{Float64} \\ \langle \top_i, \top_k\rangle & \text{otherwise} \end{cases} \\
\mathsf{eval}(\mathtt{x},\sigma) & = \sigma(\mathtt{x}) \\
\mathsf{eval}(\mathsf{e}_1 + \mathsf{e}_2,\sigma) & = \mathsf{let}\ v_1 = \mathsf{eval}(\mathsf{e}_1,\sigma), v_2 = \mathsf{eval}(\mathsf{e}_2,\sigma)\ \mathsf{in} \\
& \quad \mathsf{if}\ v_1 == \langle \top_i, \top_k\rangle \vee v_2 == \langle \top_i, \top_k\rangle \mathsf{then}\ \langle \top_i, \top_k\rangle \\
& \quad \mathsf{else}\ (v_1 \bar{+} v_2) \sqcap_{ki} \langle \top_i, \mathtt{Float64}\rangle \\
\mathsf{eval}(\mathsf{e}_1\ \mathsf{op}_{num}\ \mathsf{e}_2,\sigma) & = (\mathsf{eval}(\mathsf{e}_1,\sigma) \bar{\mathsf{op}}_{num}\ \mathsf{eval}(\mathsf{e}_2,\sigma)) \sqcap_{ki} \langle \top_i, \mathtt{Float64}\rangle \\
\mathsf{eval}(\mathsf{e}_1\ \mathsf{op}_{int}\ \mathsf{e}_2,\sigma) & = (\mathsf{eval}(\mathsf{e}_1,\sigma) \bar{\mathsf{op}}_{int}\ \mathsf{eval}(\mathsf{e}_2,\sigma)) \sqcap_{ki} \langle \top_i, \mathtt{Int32}\rangle \\
\mathsf{eval}(\mathtt{f}(\mathsf{e})) & = \top_i
\end{array}
$$

Fig. 6. The abstract evaluation of expressions. The abstract operators $\bar{+}$, $\bar{\mathsf{op}}_{num}$, $\bar{\mathsf{op}}_{int}$ are the abstract counterparts of concrete the concrete operators.

Initial State. At the entry point of f, the global values are set to $\langle \top_i, \top_k\rangle$ (any value), the local values are set to $\langle \bot_i, \bot_k\rangle$ (uninitialized), and the actual value v of the parameter x is generalized to $\langle \top_i, \alpha_k(\{v\})\rangle$. We generalize the actual value of the parameter so that the result of the analyses can be re-used.

Example 6. The initial abstract state for the analysis of loopToN(99999) in Fig. 5 is $\sigma_0 = [\mathtt{n} \mapsto \langle \top_i, \mathtt{Int32}\rangle, \mathtt{x} \mapsto \langle \bot_i, \bot_k\rangle]$. The specialization of loopToN can be cached and reused for loopToN(1234) as σ_0 is an over-approximation of $[\mathtt{n} \mapsto \langle \mathsf{Normal}(1234, 1234), \mathtt{Int32}\rangle, \mathtt{x} \mapsto \langle \bot_i, \bot_k\rangle]$.

Variables. RATA is a modular analysis, run on a per-method basis. In the general case, at the moment of the invocation of RATA, we have not seen all the assignments to globals, so that the only sound assumption for globals is the open-world assumption, *i.e.*, they can assume any value.

Assignment. An assignment $\mathtt{x} = \mathsf{e}$ in a pre-state σ_0, updates the entry for x with $\mathsf{eval}(\mathsf{e}, \sigma_0)$ if x is a local variable (or a parameter) or it does nothing otherwise. The evaluation function $\mathsf{eval} \in [\mathsf{e} \times \Sigma \to \mathsf{Intv} \times \mathsf{Kind}]$ is in Fig. 6. The k-interval for a constant is assigned according to its type. The "+" operator is polymorphic in JavaScript: it can either be string concatenation or numerical addition. As a consequence, if no information on the operands is available, nothing can be inferred on the result. Otherwise, we know that it is at least a Float64. The result of a op_{num} (op_{int}) is at least a Float64 (an Int32). The return value of a function call is ignored: to statically determine which function is invoked requires a quite complex global program analysis, out of the scope of this paper.

Test. A precise handling of tests enables the refinement of the abstract states, and hence a more precise analysis. A *too* precise analysis of tests (*e.g.*, using forward/backwards iterations [9]) may cause slowdowns unacceptable for an online analysis. In our implementation we only consider comparisons between a variable and an expression, or between two variables. For equalities, we have that:

$$
\begin{array}{l}
\mathbb{N}[\![\mathtt{x} == \mathsf{e}]\!](\sigma_0) = \sigma_0[\mathtt{x} \mapsto \sigma_0(\mathtt{x}) \sqcap_{ki} \mathsf{eval}(\mathsf{e}, \sigma_0)], \text{ and} \\
\mathbb{N}[\![\mathtt{x} == \mathtt{y}]\!](\sigma_0) = \sigma_0[\mathtt{x}, \mathtt{y} \mapsto \sigma_0(\mathtt{x}) \sqcap_{ki} \sigma_0(\mathtt{y})].
\end{array}
$$

For an inequality $\mathtt{x} < \mathtt{e}$, the upper bound of $\mathtt{x}$ is refined by the upper bound of $\mathsf{eval}(\mathsf{e}, \sigma_0)$ ($\mathsf{upp}(\langle i, t\rangle)$ is the open k-interval bounded by the upper bound of i, it can be $+\infty$):

$$\mathbb{N}[\![\mathtt{x} < \mathsf{e}]\!](\sigma_0) = \sigma_0[\mathtt{x} \mapsto \sigma_0(\mathtt{x}) \sqcap_{ki} \mathsf{upp}(\mathsf{eval}(\mathsf{e}, \sigma_0))].$$

Similarly for an inequality $\mathtt{x} < \mathtt{y}$, the upper bound for $\mathtt{x}$ can be refined by the upper bound of $\mathtt{y}$, and the lower bound of $\mathtt{y}$ can be refined by the lower bound of $\mathtt{x}$:

$$\mathbb{N}[\![\mathtt{x} < \mathtt{y}]\!](\sigma_0) = \sigma_0[\mathtt{x} \mapsto \sigma_0(\mathtt{x}) \sqcap_{ki} \mathsf{upp}(\mathsf{eval}(\mathtt{y}, \sigma_0)), \mathtt{y} \mapsto \sigma_0(\mathtt{y}) \sqcap_{ki} \mathsf{low}(\mathsf{eval}(\mathtt{x}, \sigma_0))].$$

Example 7. Let us assume that $\sigma_0 = [\mathtt{x} \mapsto \langle \mathsf{OpenRight}(10), \top_i\rangle]$. Then:

$$\mathbb{N}[\![\mathtt{x} < \mathtt{1000}]\!](\sigma_0) = [\mathtt{x} \mapsto \langle \mathsf{Normal}(10, 1000), \top_i\rangle].$$

Note that it would be unsound to assume that $\mathtt{x}$ is an `Int32` or that $\mathtt{x}$: $\mathsf{Normal}(10, 999)$.

Sequence. The analysis of a sequence of statements is the composition of the analyses: $\mathbb{N}[\![\mathtt{C_1}\ \mathtt{C_2}]\!](\sigma_0) = \mathbb{N}[\![\mathtt{C_2}]\!](\mathbb{N}[\![\mathtt{C_1}]\!](\sigma_0))$.

Conditional. For a conditional the analysis first refines the pre-state with the guards, and then joins the results (the function `Not` negates the Boolean expression `b`):

$$\mathbb{N}[\![\mathtt{if(b)}\ \{\mathtt{C_1}\}\mathtt{else}\ \{\mathtt{C_2}\};]\!](\sigma_0) = \mathbb{N}[\![\mathtt{C_1}]\!](\mathbb{N}[\![\mathtt{b}]\!](\sigma_0)) \sqcup_{ki} \mathbb{N}[\![\mathtt{C_2}]\!](\mathbb{N}[\![\mathtt{Not(b)}]\!](\sigma_0)).$$

Loop. A loop invariant for $\mathtt{while(b)}\ \{\mathtt{C}\};$ is a fixpoint of the functional $F \in [\Sigma \to \Sigma]$:

$$F(X) = \sigma_0 \dot{\sqcup}_{ki} \mathbb{N}[\![\mathtt{C}]\!](\mathbb{N}[\![\mathtt{b}]\!](X)),$$

where σ_0 is the abstract state at the entry point of the loop and $\dot{\sqcup}_{ki}$ is the pointwise extension of $\sqcup_{ki}$. An invariant can be computed with the usual fixpoint iteration techniques. The abstract domain $\mathsf{Intv} \times \mathsf{Kind}$ does not contain infinite ascending chains, but it contains very very long chains (up to $2^{32} + 3$ elements). We need a widening operator to speed up the convergence of the iterations to a post-fixpoint. A widening with thresholds [6,22], and the re-execution from a post-fixpoint (a form of narrowing [22]) guarantee a good precision yet providing good performance. We illustrate those two techniques with examples.

Example 8. The iterations with the classical widening for the `loop` function of Fig. 5 produce the following sequence of abstract values for `x`:

$$\langle \mathsf{Normal}(0,0), \mathtt{Int32}\rangle \sqsubseteq_{ki} \langle \mathsf{Normal}(0,1), \mathtt{Int32}\rangle \sqsubseteq_{ki} \langle \mathsf{OpenRight}(0), \mathtt{Float64}\rangle,$$

as the upper bound for `x` is extrapolated to $+\infty$. The threshold (or staged) widening tries to extrapolate the upper bound to constants appearing in guards, producing the sequence:

$$\langle \mathsf{Normal}(0,0), \mathtt{Int32}\rangle \sqsubseteq_{ki} \langle \mathsf{Normal}(0,1), \mathtt{Int32}\rangle \sqsubseteq_{ki} \langle \mathsf{Normal}(0,10000), \mathtt{Int32}\rangle.$$

In general, during the analysis we collect all the constants that appear in the tests, and we use them as steps for widening with a threshold.

Example 9. The type of x in function loopToN of Fig. 5 depends on the input parameter. When it is invoked with an Int32 value, then we would like RATA to discover that x is an Int32. Widening with thresholds is of no help here (there are no constants in guards) so the iterations stabilize at $I = \langle \mathsf{OpenRight}(0), \mathtt{Float64} \rangle$. A re-execution of the loop with initial state I will refine the abstract state to $\langle \mathsf{OpenRight}(0), \mathtt{Int32} \rangle$.

Re-execution is justified by Tarski's fixpoint theorem [29], which states that in a partial order $\mathsf{lfp}(F) = \sqcap \{I \mid F(I) \sqsubseteq I\}$. So, if I is a post-fixpoint for F, then $F(I)$ is still above the least fixpoint $\mathsf{lfp}(F)$, and hence it is a sound approximation of the loop invariant. During re-execution, we refine the abstract semantics of the tests appearing in *loops* which involve inequalities where one of the operands is an Int32. For instance in the loopToN example:

$$\mathbb{N}[\![\mathtt{x} < \mathtt{n}]\!]([\mathtt{x} \mapsto \langle \mathsf{OpenRight}(0), \mathtt{Float64} \rangle]) = [\mathtt{x} \mapsto \langle \mathsf{Normal}(0, 2^{31} - 2), \mathtt{Int32} \rangle].$$

In Ex. 7 we pointed out that in general it is not sound to assume $\mathtt{x} : \mathtt{Int32}$ after a test $\mathtt{x} < \mathtt{y}$ when $\mathtt{y} : \mathtt{Int32}$. However during re-execution this is sound as there are essentially three cases why $\mathtt{x} : \mathtt{Float64}$ in I: (i) x was a Float64 at the loop entry; (ii) x may be assigned a fractional value (or NaN or an infinite) in the loop body; or (iii) the analysis of the loop could not figure out that $\mathtt{x} : \mathtt{Int32}$. In the first two cases, $F(I)$ will imply that $\mathtt{x} : \mathtt{Float64}$ (because of the definition of F). In the third case one may hope to recover some of the lost precision. In our running example:

$$\begin{aligned} F(I) &= [\mathtt{x} \mapsto \langle \mathsf{Normal}(0,0), \mathtt{Int32} \rangle] \dot{\sqcup}_{ki} [\mathtt{x} \mapsto \langle \mathsf{Normal}(1, 2^{31} - 1), \mathtt{Int32} \rangle] \\ &= [\mathtt{x} \mapsto \langle \mathsf{Normal}(0, 2^{31} - 1), \mathtt{Int32} \rangle] \mathrel{\dot{\sqsubseteq}_{ki}} I. \end{aligned}$$

(Recall that $\gamma_{ki}(\langle \mathsf{Normal}(0, 2^{31} - 1), \mathtt{Int32} \rangle) = \gamma_{ki}(\langle \mathsf{OpenRight}(0), \mathtt{Int32} \rangle))$.

Ignored Statements and Havoc. Ignored statements have no effect on the local state, so the analysis treats them as the identity : $\mathbb{N}[\![\mathtt{Ignored}_\mathtt{C}]\!](\sigma_0) = \sigma_0$. Havoc statements may have some side-effect on local variables. We abstract them by $\mathbb{N}[\![\mathtt{Havoc}_\mathtt{C}]\!](\sigma_0) = \langle \top_i, \top_k \rangle$.

5.2 Variation Analysis

The numerical analysis alone cannot determine that $\mathtt{c} : \mathtt{Int32}$ in bitsinbyte (Fig. 1). It discovers the loop invariant $\sigma_L = [\mathtt{m} \mapsto \langle \mathsf{Normal}(1, 512), \mathtt{Int32} \rangle, \mathtt{c} \mapsto \langle \mathsf{OpenRight}(0), \mathtt{Float64} \rangle]$ (we omit b). The invariant σ_L can be refined by the variation analysis. At the loop entry, c and m differ by one. At each iteration c is either incremented by one or it remains the same, whereas m is multiplied by 2, thus m grows faster than c. However, m bounded implies that c should be bounded too, thus we can safely refine σ_L to $\sigma_L[\mathtt{c} \mapsto \langle \mathsf{Normal}(0, 512), \mathtt{Int32} \rangle]$.

We run the variation analysis $\mathbb{V}[\![\cdot]\!]$ on a *per-loop* basis. The goal of the analysis is to compute, for each loop and each variable an interval over-approximating the increment of a variable in a *single* loop iteration. The variation analysis is similar in many aspects to the numerical analysis above, with the major difference that the initialization and the assignments are re-interpreted. An abstract state is a map from local variables to intervals. At the loop entry point, all the local variables are set to the interval $[0, 0]$ [2] (no increment). For assignments, we compute variable increments. We consider simple forms of increments and decrements, and we abstract away all the other expressions. So, we let $\mathbb{V}[\![\mathtt{x} = \mathtt{x} \pm k]\!](\sigma_0) = \sigma_0[\mathtt{x} \mapsto \pm[k, k]]$, and $\mathbb{V}[\![\mathtt{x} = \mathtt{e}]\!](\sigma_0) = \sigma_0[\mathtt{x} \mapsto [-\infty, +\infty]]$.

Once we have computed ν, the increment ranges for the variables in the loop, we use this information to refine the numerical loop invariant σ_L to σ'_L according to refinement rules that looks like:

$$\forall \mathtt{x}, \mathtt{y}.\ \mathtt{x} \neq \mathtt{y} \wedge \sigma_0 \models \mathtt{x} < \mathtt{y} \wedge \mathtt{y} \text{ is upper-bounded by } b \wedge \nu \models \mathtt{x} < \mathtt{y}$$
$$\Longrightarrow \sigma'_L(\mathtt{x}) = \sigma_L(\mathtt{x}) \dot{\sqcap}_{ki} \langle \mathsf{OpenLeft}(b), \top_k \rangle,$$

(the intuitive meaning of $\sigma_0 \models \mathtt{x} < \mathtt{y}$ is that in the k-interval σ_0, $\mathtt{x} < \mathtt{y}$ and the meaning of $\nu \models \mathtt{x} < \mathtt{y}$ is that according to ν, `x` grows slower than `y`). The rule above essentially states that if `y` is an upper bound for `x` at the entry of the loop, and `y` is bounded by b during all the executions of the loop, and `x` does not grow more than `y` in the loop, then b should be an upper bound for `x` too. We omit all the other (tedious) refinement rules, which consider the combination of the other cases (*e.g.*, $\sigma_0 \models \mathtt{x} \leq \mathtt{y}$, lower bounds, decrements and so on).

5.3 Atomic Types

The atomic types T for a function are obtained by joining together the post-states of all the statements in the function body. The reason for that is that we want to assign a *unique* atomic type at each local variable. One may wonder why we designed a flow-sensitive analysis if we were interested in a flow-insensitive property (the type of a local variable through all the function's body). Actually, in an early stage of this project we tried to avoid the joining phase by designing a flow-insensitive analysis. For instance, the abstract semantics of the sequence was $\mathbb{N}[\![\mathtt{C}_1\ \mathtt{C}_2]\!](\sigma_0) = \text{let } \sigma = \mathbb{N}[\![\mathtt{C}_1]\!](\sigma_0) \text{in } \mathbb{N}[\![\mathtt{C}_2]\!](\sigma) \dot{\sqcup}_{ki} \sigma$. We immediately realized that a flow-insensitive analysis was too imprecise for handling loops, and in particular it voided the advantages of the re-execution step and the variation analysis which we found crucial for precision. Therefore, we rejected the flow-insensitive analysis for a flow-sensitive followed by a join-all step.

6 Experiments

We have implemented RATA in our `JavaScript` engine for `.Net`. The engine itself is written in C#. It parses the `JavaScript` source, it compiles the main

[2] We use the notation $[a, b]$ to avoid confusion between the range intervals of the previous sections and the increment intervals. In the implementation we share the code, though.

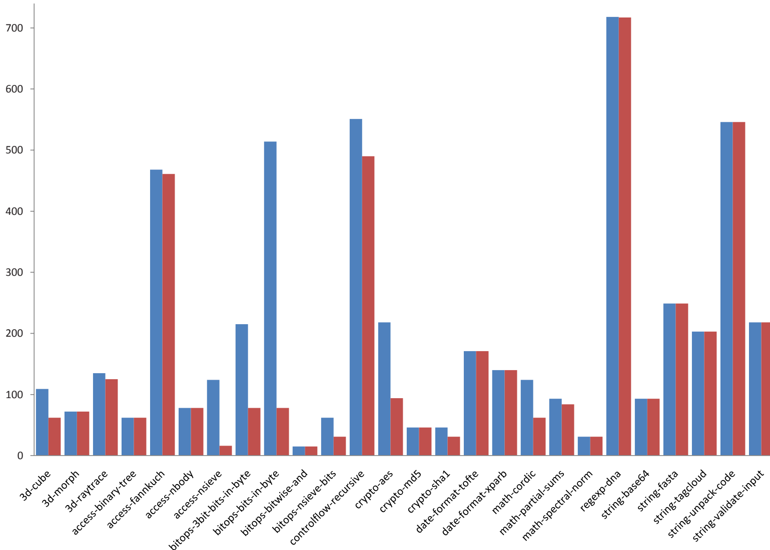

Fig. 7. The results of the optimizations enabled by a text-book type inference algorithm (blue/light bars) and RATA (red/dark bars). Times are expressed in milliseconds. On numerical intensive benchmarks RATA enables up to a 7.7× speed-up.

(global) function and it generates proxies for function invocations. When the execution encounters a function proxy, the `JavaScript` engine resolves it, and it checks if it has a specialized version in the cache which matches the actual parameters. If this is the case, then it executes the cached version. Otherwise: (i) it runs the RATA to infer the atomic types for the locals of the variables; (ii) it compiles the function in memory, performing atomic type specialization; (iii) executes the specialized function, and caches it for future needs. It is worth noting that the specialization is polymorphic: If the same function is invoked at two points of time with two actual parameters of different types, then it is analyzed and specialized twice.

We report the experience of applying RATA on the SunSpider `JavaScript` benchmarks [31]. The SunSpider benchmarks measure `JavaScript` performance for problems that presents difficulties to `JavaScript` implementations. They are designed to be balanced and to stress different areas of the language. They are commonly used to compare the `JavaScript` performance of different browsers, or different versions of the same browser. We run the experiments on a 2.1GHz Centrino Duo Laptop, 4Gbyte, under Windows 7 and `.Net` v3.5. We compared a text-book type inference algorithm [2] with RATA. The type inference algorithm determines which locals are definitely doubles, and for some expressions it can also infer that a local is an `Int32`.

The results of our experiments are in Fig. 7. Measuring the performances of managed programs is quite complex, as their runtime behavior depends on too many variables [18]. In general, when the execution time is too low, it is

```
global x;
x = 0;
while(x < 4) {
  foo(x);
  x = x + 1;
}
```

```
function zeroarray(arr) {
  var x; x = 0;
  while(x < arr.length) {
    arr[x] = 0; x = x + 1; }
}

global a; a = new Array(10);
zeroarray(a)
```

Fig. 8. Two code snippets in which it would be unsound to infer that `x` : `Int32`. In the first case, `x` is declared in the global scope and its value can be changed by `foo`. In the second case, `x` depends on the property `arr.length` which in general is a `UInt32`. Furthermore, `JavaScript` allows the user-redefinition of `Array`, so that we need a global analysis to determine that `a` is an array.

impossible to distinguish the *effective* time spent in computation from the external noise (*e.g.*, the garbage collector, the thread scheduler, network traffic, background services and so on). We run each `JavaScript` program in the SunSpider suite 80 times choosing the best execution time. The execution times of Fig. 7 do not include the compilation and the type inference/RATA time. The reason for that is that we observed the analysis time to be of the same order of magnitude of the experiment noise (few tenths of milliseconds). We also observed that the runtime costs of the type inference and RATA were comparable. We modified some tests so to have them run longer, reducing the external noise, and hence obtaining more meaningful measurements.

The results of Fig. 7 show that in 12 tests RATA enables the JIT to generate more optimized code, and hence to obtain significant performance improvements.

Most of the benchmarks in the `3d` family benefit from `Int32` type inference. The tests themselves manipulate many doubles (and arrays of doubles), but RATA manages to discover that 20 locals in `3d-cube` and 11 locals in `3d-raytrace` are `Int32` which convey respectively a $1.75\times$ and $1.1\times$ speed-up over the double-only version. We inspected the results of the analysis, and we found that in the first test RATA found all the `Int32` variables one may expect, and in the second test it missed three. The reason for that was in an imprecision of handling the `return` statement. Finally, the locals on `3d-morph` depends on some global values, so nothing can be inferred about them.

The best performance improvements are in the `bitops` family benchmarks. RATA discovers that all the local variables are `Int32` in the test `bitops-bits-in-byte`, which provides a $6.6\times$ speed-up with respect to the same test when all the locals are inferred to be doubles. Similar results are observed in the `bitops-3bit-bits-in-byte` ($2.8\times$) and the `bitops-nsieve-bits` ($2\times$) tests, where RATA is again precise enough to infer all the `Int32` locals. The test `bitops-bitwise-and` contains only globals, so there is no hope to statically optimize it.

Example 10. The test `bitops-bitwise-and` contains a main loop that looks like the first code snippet of Ex. 8. In general, it is unsound to infer that `x` : `Int32` as

`foo` may change the value of `x`. Functions are analyzed top-down: first the JIT runs RATA on the global statement, and then, at the first concrete occurrence, it invokes RATA on `foo`. As a consequence when inferring the type of `x`, RATA assumes the worst case for `foo`. Determining `x : Int32` requires a bottom-up purity analysis or an effect analysis [4], which are out-of-the scope of the paper, and in general too expensive to be performed online.

In the `access-nsieve` benchmark, RATA local inference enables a significant speedup (7.7×) over the `Float64`-specialized version. In particular, the inner function contains two nested loops and a counter variable. Fixpoint computation with re-execution and variation analysis are cardinal to infer that all the locals involved are indeed `Int32`. The other two benchmarks of the `access` family benchmarks perform computations which either depend on globals or on very short loops.

The `controlflow-recursive` benchmark stresses `JavaScript` implementations with standard recursive-function benckmarks such as `fibonacci` or `ackerman`. RATA infers that the variables inside those functions are `Int32` and thus achieves a slight performance improvement (1.12×).

The cryptographic benchmarks benefit by an aggressive type specialization. RATA infers all the `Int32` locals for the `crypto-aes` and the `crypto-sha1` benchmarks, enabling a 2.3× and 1.5× speedup. The `crypto-md5` benchmark contains many functions taking an array as parameter, and iterating over its elements. The next example shows that it would be unsound to infer those locals to be `Int32`.

Example 11. Let us consider the `zeroarray` function of Fig. 8. In `JavaScript`, the `length` property of `Array` is a `UInt32`, *i.e.*, it can assume values as large as $2^{32} - 1$. As a consequence, even if we know that `arr` is an array, we cannot conclude `x` : `Int32`. In general, to infer that `x` : `Int32`, we should refine RATA to track that `arr` is an array *and* that `arr.length` $< 2^{31} - 1$. The `JavaScript` languages allows the redefinition of `Array`, so we need a global analysis to guarantee that the value of `a` is actually an array.

The execution time of date and string manipulating benchmarks is heavily dominated by the interaction with the object model, and by other non-numerical computations so that RATA is of no help here.

Math benchmarks manipulate double values, but the inference of some `Int32` locals enable up to a 2× speedup in `math-cordic`, a slight improvement in `math-partialsums`. For atomic type inference, the test `math-spectral-norm` looks like `crypto-md5`, and as a consequence nothing can be statically inferred.

To sum up, RATA is precise enough to infer all (but 3) of the local variables which are `Int32` in the SunSpider benchmarks. One may wonder if broadening the analysis to also consider `Int64`, `UInt32` and so on may provide further performance gains. According to the previous experience of the second author with JScript.NET, those cases are so rare, and they complicate so much the implementation and the JIT code generation, that it seems not worthwhile to try.

7 Related Work

Just-in-time compilation is known at least from 1960. In his LISP paper [25], McCarthy sketches the dynamic compilation of functions into machine code, a process fast enough that the compiler's output does not need to be saved. Deutsch and Schiffman introduced in [14] lazy JIT compilation for Smalltalk, where functions were compiled at the first usage, and cached for further usage. The Self programming language influenced the `JavaScript` design. The first Self compiler used a data-flow analysis ("Class analysis") to compute an over-approximation of the set of possible classes that variables might hold instances of and hence to optimize dynamic dispatching [8]. Further versions of the Self compiler introduced more aggressive type analyses [30], but they did not consider the specialization of atomic types as here [1].

The implementation of popular dynamic languages as Python try to optimize the generated code by performing some kind of online static analysis. The JIT compiler of the PyPy system [28] uses "flexswitches" to perform type specialization [13]. Flexswitches are essentially a form of online partial evaluation [21]. Psyco [27] is another implementation of Python which tries to guess `Int32` variables at *runtime*. The tracing JIT generalizes the ideas of Psyco and PyPy. A tracing JIT essentially identifies frequently executed loop traces at runtime, and it dynamically generates specialized machine code [17]. RATA is complementary to a tracing JIT. In his master thesis, Cannon presented a localized atomic type inference algorithm for Python [7]. His analysis is based on the Cartesian product algorithm, and it is less precise than ours. As a consequence, it is not a surprise that his experimental results are less satisfactory than ours. In [3], Anderson *et al.* introduced an algorithm for type inference of `JavaScript` to derive the types of objects. It is unclear if their algorithm is fast enough to be used in dynamic compilation. They did not consider the inference of `Int32` variables which require reasoning on the *values* of variables. In this sense, our work is then complementary to theirs. In [20], Jensen *et al.* presented an abstract interpretation based static analysis to check the absence of common errors in `JavaScript` programs. Their analysis is more oriented to program verification than optimization. However, for numerical values their abstract domain is less precise than ours and so they are not likely to discover all the numerical properties that RATA can discover.

Abstract Interpretation is mainly applied to program verification (*e.g.*, [6,15]) and *offline* program optimization (*e.g.*, [10,5]). To the best of our knowledge this is the first work which applies full-powered Abstract Interpretation techniques (*e.g.*, infinite lattices, widenings and narrowings) to *online* program optimization. We believe that this is a promising line of work.

8 Conclusions

We have presented RATA, a new static analysis, based on abstract interpretation, for the rapid inference of atomic types in dynamic languages. The analysis is

a combination of three analyses: a range analysis, a kind analysis and a variation analysis. We formalized the underlying abstract domains and we related them to the concrete values via Galois connections. We described the analysis, and we reported the results of the atomic type specialization on the SunSpider `JavaScript` benchmarks (the industrial standard for comparing `JavaScript` implementations). We observed that: (i) RATA is precise enough to infer all the `Int32` locals that one may hope to infer statically; and (ii) the `Int32`-specialization produces remarkable performance improvements in most tests (up to a 7.7× speed-up for numerical intensive ones).

For the future, we plan to extend RATA to whole program analysis, and in particular to apply it to the wider goal of program verification.

References

1. Agesen, O., Hölzle, U.: Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In: OOPSLA 1995. ACM Press, New York (1995)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
3. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for javascript. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 428–452. Springer, Heidelberg (2005)
4. Barnett, M., Fähndrich, M., Garbervetsky, D., Logozzo, F.: Annotations for (more) precise points-to analysis. In: IWACO 2007 (2007)
5. Blanchet, B.: Escape Analysis: Correctness proof, implementation and experimental results. In: POPL 1998 (1998)
6. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI 2003. ACM Press, New York (2003)
7. Cannon, B.: Localized type inference of atomic types in Python. Master's thesis, California Polytechnic State University (2005)
8. Chambers, C., Ungar, D.: Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In: PLDI 1989. ACM Press, New York (1989)
9. Cousot, P.: The calculational design of a generic abstract interpreter. In: Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)
10. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977 (1977)
11. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL 1979 (1979)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978 (1978)
13. Cuni, A., Ancona, D., Rigo, A.: Faster than C#: Efficient implementation of dynamic languages on.NET. In: ICOOOLPS 2009. ACM Press, New York (2009)
14. Deutsch, L.P., Schiffman, A.M.: Efficient implementation of the smalltalk-80 system. In: POPL1980. ACM Press, New York (1980)
15. Ferrara, P., Logozzo, F., Fähndrich, M.A.: Safer unsafe code in.Net. In: OOPSLA 2008 (2008)

16. Flanagan, D.: JavaScript, the definitive guide. O'Reilly, Sebastopol (2009)
17. Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E., Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: PLDI 2009 (2009)
18. Georges, A., Eeckhout, L., Buytaert, D.: Java performance evaluation through rigorous replay compilation. In: OOPSLA 2008 (2008)
19. IEEE. IEEE standard for floating-point arithmetic. Technical report, IEEE (2008)
20. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for javascript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009)
21. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice-Hall, Englewood Cliffs (1993)
22. Laviron, V., Logozzo, F.: Refining abstract interpretation-based static analyses with hints. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 343–358. Springer, Heidelberg (2009)
23. Laviron, V., Logozzo, F.: Subpolyhedra: a (more) scalable approach to infer linear inequalities. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 229–244. Springer, Heidelberg (2009)
24. Logozzo, F., Fähndrich, M.A.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In: SAC 2008 (2008)
25. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part i. Commun. ACM 3(4), 184–195 (1960)
26. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, p. 155. Springer, Heidelberg (2001)
27. Rigo, A.: Representation-based just-in-time specialization and the psyco prototype for Python. In: PEPM 2004. ACM Press, New York (2004)
28. Rigo, A., Pedroni, S.: PyPy's approach to virtual machine construction. In: OOPSLA Companion 2006. ACM Press, New York (2006)
29. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–309 (1955)
30. Ungar, D., Smith, R.B., Chambers, C., Hölzle, U.: Object, message, and performance: How they coexist in self. IEEE Computer 25(10), 53–64 (1992)
31. WebKit. SunSpider JavaScript benchmarks, `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`