Fast and Precise Hybrid Type Inference for JavaScript

Authors: Brian Hackett, Shu-yu Guo

Reading 04.07.2023 Korostinskiy Roman

Paper Meta Data

Conference: PLDI (Programming Language Design and Implementation)

Categories and Subject Descriptors: Compilers, optimization

Keywords: type inference, hybrid, just-in-time compilation

Year: 2012

Number of Authors: 2

Citations: 8

Pages: 13

References: 27

What is the study about?

- JavaScript performance is often bound by its dynamically typed nature.
- The authors speed up the Mozilla Firefox JavaScript JIT (just in time) compiler.
- To do this, the authors use static type inference.
- Using complete type inference for JavaScript is too sophisticated and slow.
- In order to optimize it, the authors use a hybrid approach of static type inference and dynamic type checking.

What is the study about?

Motivating example

```
1   function Box(v) {
2     this.p = v;
3   }
4
5   function use(a) {
6     var res = 0;
7     for (var i = 0; i < 1000; i++) {
8       var v = a[i].p;
9       res = res + v;
10    }
11    return res;
12  }
13
14  function main() {
15    var a = [];
16    for (var i = 0; i < 1000; i++)
17      a[i] = new Box(10);
18    use(a);
19  }
```

Static Type inference

1. [17]: `Box(10)` **=>** `Box.p` - integer
2. [17]: `a[i] = new Box(10)` **=>** `a` - may contain `Box`
3. [18]: `use(a)` **&&** #2 **=>** [8]: `a[i]` - may be `Box`
4. [8]: `var v = a[I].p;` **&&** #1 **=>** `v` - may be integer
5. [6]: `res = 0` **&&** #4 **=>** [9]: `res + v` - may be integer

What is the study about?

Motivating example

```
1   function Box(v) {
2     this.p = v;
3   }
4
5   function use(a) {
6     var res = 0;
7     for (var i = 0; i < 1000; i++) {
8       var v = a[i].p;
9       res = res + v;
10    }
11    return res;
12  }
13
14  function main() {
15    var a = [];
16    for (var i = 0; i < 1000; i++)
17      a[i] = new Box(10);
18    use(a);
19  }
```

Static Type inference

1. [17]: `Box(10)` **=>** `Box.p` - integer
2. [17]: `a[i] = new Box(10)` **=>** `a` - may contain `Box`
3. [18]: `use(a)` **&&** #2 **=>** [8]: `a[i]` - may be `Box`
4. [8]: `var v = a[I].p;` **&&** #1 **=>** `v` - may be integer
5. [6]: `res = 0` **&&** #4 **=>** [9]: `res + v` - may be integer

But

- The read of `a[i]` may access a hole in the array
- Similarly, the read of `a[i].p` may be accessing a missing property
- The addition `res + v` may overflow.

What is the study about?

Motivating example

```
1   function Box(v) {
2     this.p = v;
3   }
4
5   function use(a) {
6     var res = 0;
7     for (var i = 0; i < 1000; i++) {
8       var v = a[i].p;
9       res = res + v;
10    }
11    return res;
12  }
13
14  function main() {
15    var a = [];
16    for (var i = 0; i < 1000; i++)
17      a[i] = new Box(10);
18    use(a);
19  }
```

Static Type inference

1. [17]: `Box(10)` **=>** `Box.p` - integer
2. [17]: `a[i] = new Box(10)` **=>** `a` - may contain `Box`
3. [18]: `use(a)` **&&** #2 **=>** [8]: `a[i]` - may be `Box` ***<- Unsound***
4. [8]: `var v = a[I].p;` **&&** #1 **=>** `v` - may be integer
5. [6]: `res = 0` **&&** #4 **=>** [9]: `res + v` - may be integer

But

- The read of `a[i]` may access a hole in the array
- Similarly, the read of `a[i].p` may be accessing a missing property
- The addition `res + v` may overflow.

Solution - Semantic triggers (dynamically checked)

- If `a[i]` acesses - hole **=>** inferred type possibly undefined.
- If `res + v` overflows **=>** inferred type possibly a double.

What is the study about?

Motivating example

```
1   function Box(v) {
2     this.p = v;
3   }
4
5   function use(a) {
6     var res = 0;
7     for (var i = 0; i < 1000; i++) {
8       var v = a[i].p;
9       res = res + v;
10    }
11    return res;
12  }
13
14  function main() {
15    var a = [];
16    for (var i = 0; i < 1000; i++)
17      a[i] = new Box(10);
18    use(a);
19  }
```

Static Type inference

1. [17]: `Box(10)` **=>** `Box.p` - integer
2. [17]: `a[i] = new Box(10)` **=>** `a` - may contain `Box`
3. [18]: `use(a)` **&&** #2 **=>** [8]: `a[i]` - may be `Box` ***<- Unsound***
4. [8]: `var v = a[I].p;` **&&** #1 **=>** `v` - may be integer
5. [6]: `res = 0` **&&** #4 **=>** [9]: `res + v` - may be integer

But 2

- [8]: `a[I].p` may be a string
- Then expression res + v will be compiled 4 times for all combinations of string and integers. This is inefficient in JIT compilation

Solution - Type barriers (dynamically checked)

- [8]: `a[I].p` is a type barier. This expression will be dynamically checked

Table of Content

Feedback

- **Problem statement** is clear thanks to the motivating example. Also there is working tool that.
- **Innovation:** before this work, there were some hybrid approaches (are mentioned in Related Works)
- **Contributions:** Main technical contribution is a hybrid inference algorithm. Practical contributions include both an implementation of algorithm and evaluations.
- **Proof of statements:** The efficiency of the algorithm was compared with a usual JIT compiler, and not with the same hybrid approach.
- **Readability:** Good readability thanks to examples.