POPL - 2015

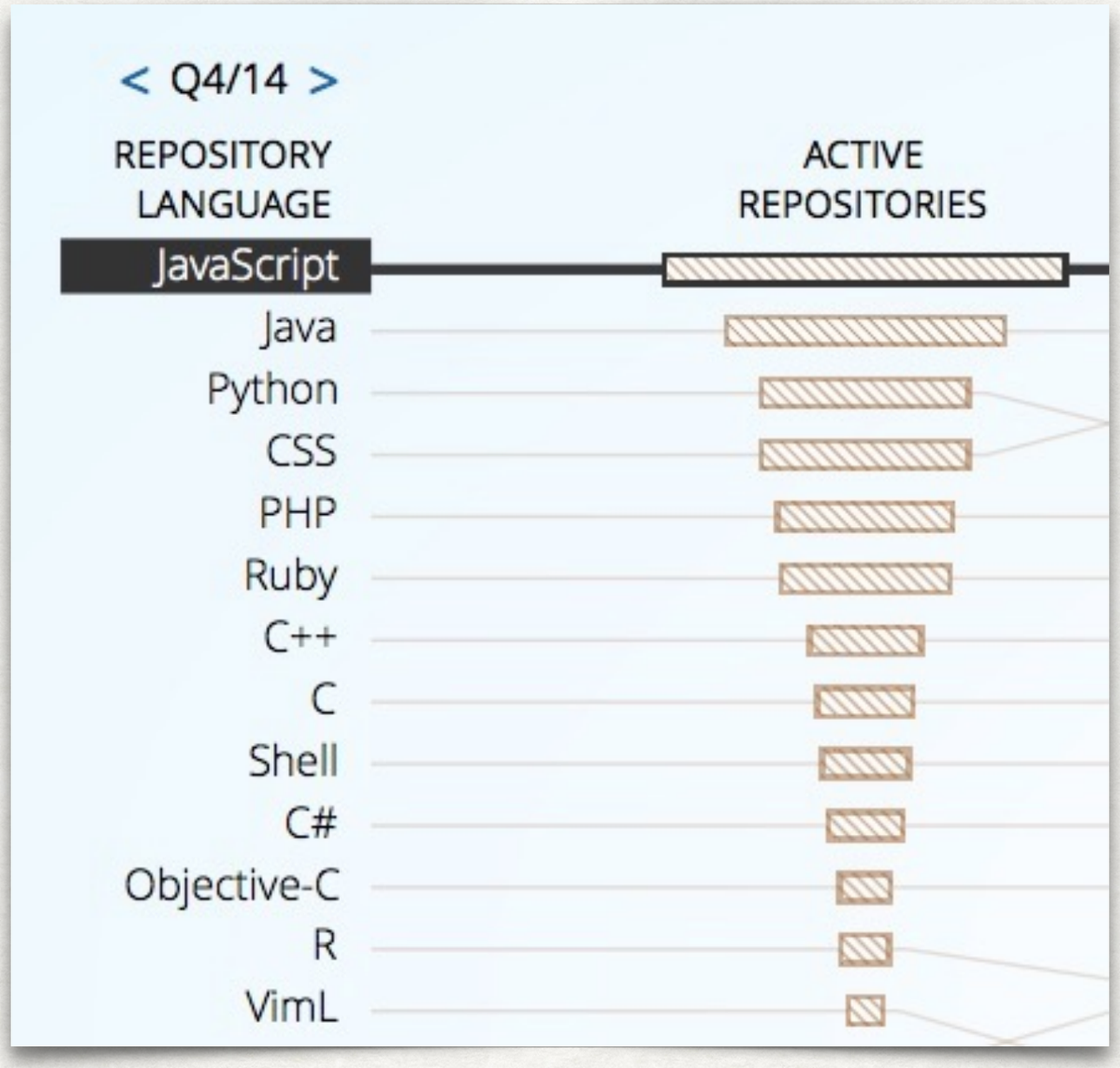
SAFE AND EFFICIENT GRADUAL TYPING FOR TYPESCRIPT

CONTENT

- **Introduction**
- A new style of efficient, RTTI based gradual typing
- Performance Evaluation and Experiments

JAVASCRIPT

GITHUB TRENDS (IMAGE FROM GITHUT.INFO)



WHAT IS TYPESCRIPT?

WHY DO WE NEED IT WHEN WE HAVE JAVASCRIPT?

- Javascript was designed for casual scripting and now it is used to develop large applications.
- Does not have static types, interfaces and classes
- We definitely need types. Tell your story.
- Proposals
  1. Clojure
  2. Dart
  3. Flow (by Facebook)
  4. Typescript (by Microsoft)

TYPESCRIPT = JAVASCRIPT (+ TYPE ANNOTATIONS)

EVERY VALID JAVASCRIPT CODE IS VALID TYPESCRIPT CODE

- Typescript is essentially Javascript with optional type annotations
- Adds a Object Oriented Gradual Type System
- All types are removed at compilation
- Syntactically close to source
- Typescript and Javascript run with same efficiency.

```
1 function Greeter(greeting: string) {
2     this.greeting = greeting;
3 }
```

```
function Greeter(greeting) {
    this.greeting = greeting;
}
```

ADVANTAGES OF TYPESCRIPT

OVER VANILLA JS

- Convenient notation for documenting code.
- Some static type checking.
- Code completion.
- Refactoring.
- But …. they do not prevent runtime errors.
- Typescript's typing system is intentionally unsound.

```
private parseColorCode (c:string) { if (typeof c !== "string") return −1; . . . }
```

NEED SOUND GRADUAL TYPE SYSTEM

ENTER SAFE TYPESCRIPT

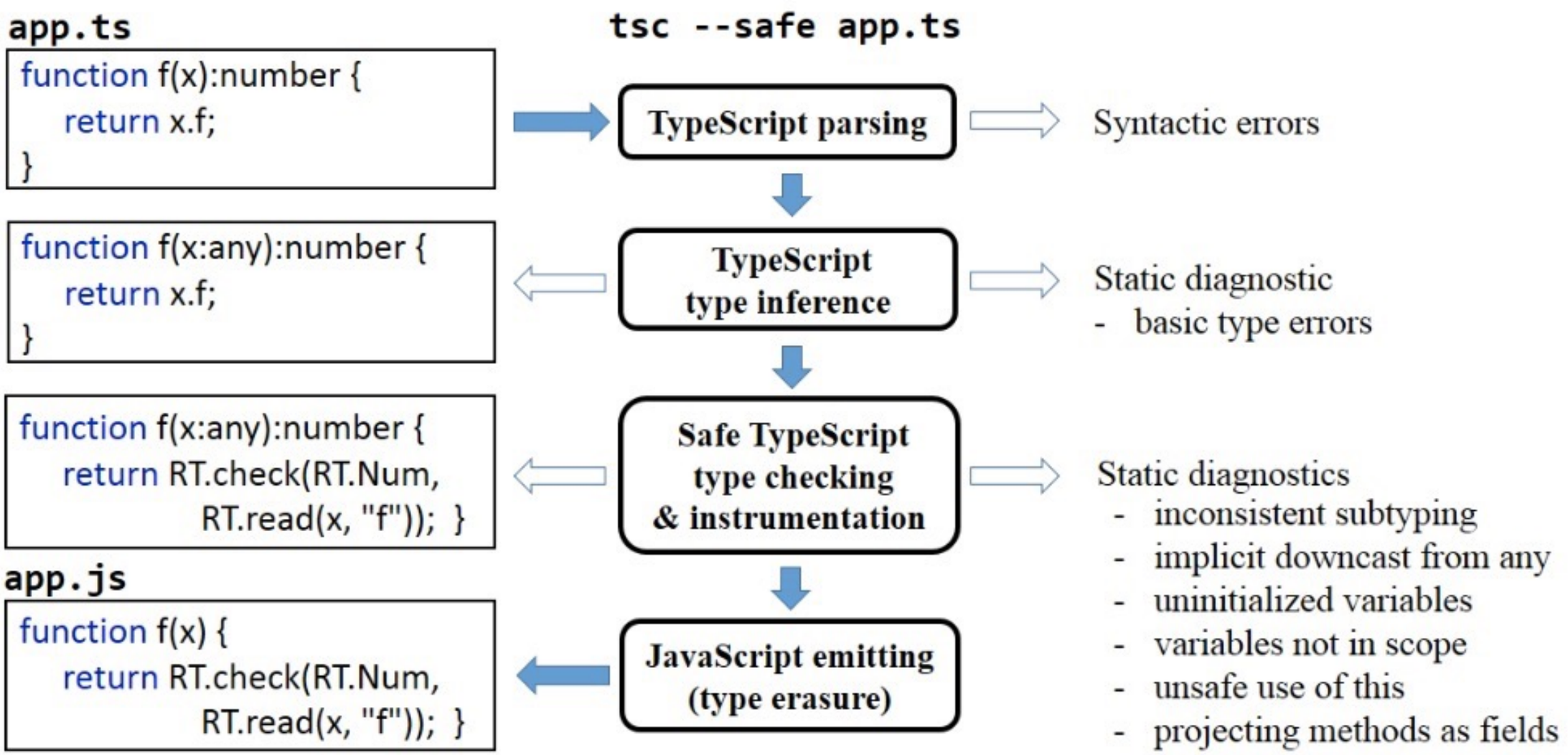


Figure 1: Architecture of Safe TypeScript

CONTENT

- **A new style of efficient, RTTI based gradual typing**
- Performance Evaluation and Experiments

NEW STYLE OF EFFICIENT SYSTEM OF GRADUAL TYPING USING RTTI

RTTI = RUNTIME TYPE INFORMATION

NOMINAL CLASSES AND STRUCTURAL INTERFACES

PART - 1

```
interface Point { x:number; y:number }
class MovablePoint implements Point {
   constructor(public x:number, public y:number) {}
   public move(dx:number, dy:number) { this.x += dx; this.y += dy; }
}
function mustBeTrue(x:MovablePoint) {

   return !x || x instanceof MovablePoint;
 }
```

t_p ={x:number; y:number}

t_o ={x:number; y:number; move(dx:number, dy:number): void}

NOMINAL CLASSES AND STRUCTURAL INTERFACES

PART - 2

```
interface Point { x:number; y:number }
class MovablePoint implements Point {
   constructor(public x:number, public y:number) {}
   public move(dx:number, dy:number) { this.x += dx; this.y += dy; }
}
function mustBeTrue(x:MovablePoint) {

    return !x || x instanceof MovablePoint;
  }
```

t_p ={x:number; y:number}

t_o ={x:number; y:number; move(dx:number, dy:number): void}

{x:0, y:0, move(dx:number, dy:number){}}

- Treat types nominally, but view them structurally. MovablePoint is a subtype of Tp and To, but not vice versa

- Interfaces remain structural. Point = tp

SEE THE BUG?

DEFINITELY NEED RUNTIME TYPE CHECKING

```
1   interface Point { x:number; y:number }
2   interface Circle { center:Point; radius:number }
3   function copy(p:Point, q:Point) { q.x=p.x; q.y=p.y; }
4   function f(q:any) {
5       var c = q.center;
6       copy(c, {x:0, y:0});
7       q.center = {x:"bad"};  }
8   function g(circ:Circle) : number {
9       f(circ);
10      return circ.center.x;  }
```

COMPILING

- Statically typed code should suffer no runtime penalty.
- Insert checks only between static and dynamically typed code.

```
1  interface Point { x:number; y:number }
2  interface Circle { center:Point; radius:number }
3  function copy(p:Point, q:Point) { q.x=p.x; q.y=p.y; }
4  function f(q:any) {
5      var c = q.center;
6      copy(c, {x:0, y:0});
7      q.center = {x:"bad"};  }
8  function g(circ:Circle) : number {
9      f(circ);
10     return circ.center.x;  }
```

```
1  RT.reg("Point",{"x":RT.num,"y":RT.num});
2  RT.reg("Circle",{"center":RT.mkRTTI("Point"), "radius":RT.num});
3  function copy(p, q) { q.x=p.x; q.y=p.y; }
4  function f(q) {
5     var c = RT.readField(q,"center");
6     copy(RT.checkAndTag(c, RT.mkRTTI("Point")),{x:0,y:0});
7     RT.writeField(q, "center", {x:"bad"});  }
8  function g(circ) {
9     f(RT.shallowTag(circ, RT.mkRTTI("Circle")));
10    return circ.center.x;  }
```

Figure 2: Sample source TypeScript program (left) and JavaScript emitted by the Safe TypeScript compiler (right).

TRANSPILED TYPESCRIPT CODE

CATCHES BUG AT LINE 7, NOT AT 10

```
1   RT.reg("Point",{"x":RT.num,"y":RT.num});
2   RT.reg("Circle",{"center":RT.mkRTTI("Point"), "radius":RT.num});
3   function copy(p, q) { q.x=p.x; q.y=p.y; }
4   function f(q) {
5       var c = RT.readField(q,"center");
6       copy(RT.checkAndTag(c, RT.mkRTTI("Point")),{x:0,y:0});
7       RT.writeField(q, "center", {x:"bad"}); }
8   function g(circ) {
9       f(RT.shallowTag(circ, RT.mkRTTI("Circle")));
10      return circ.center.x; }
```

REGISTERING USER DEFINED TYPES WITH THE RUNTIME

1 OF 4

```
1    interface Point { x:number; y:number }
2    interface Circle { center:Point; radius:number }
```

```
RT.reg("Point",{"x":RT.num,"y":RT.num});
RT.reg("Circle",{"center":RT.mkRTTI("Point"), "radius":RT.num});
```

TAGGING OBJECTS WITH RTTI TO LOCK INVARIANTS

PART 2 OF 4

- When g passes ‘circ’ to ‘f’, which uses an imprecise type ‘any’
- ‘circ’ must be treated as a Circle even in dynamically typed code
- ‘circ’ is instrumented using RT.shallowTag()

```
function g(circ:Circle) : number {
      f(circ);
      return circ.center.x;  }
```

```
function g(circ) {
     f(RT.shallowTag(circ, RT.mkRTTI("Circle")));
     return circ.center.x;  }
```

TAGGING OBJECTS

PART 2 OF 4 CONTD

- RTTI is maintained in an additional field
- It may evolve at runtime. Guarantee that it never becomes less precise.
- At each call to ShallowTag(c, t), ensure that c has type t.
- For performance, shallowTag does not descend recursively. Single tag at the outermost object suffices.

```
function shallowTag(c, t) {
  if (c!==undefined) { c.rtti = combine(c.rtti, t); }
  return c; }
```

PROPAGATING INVARIANTS IN DYNAMICALLY TYPED CODE

PART 3 OF 4

- Reading a field requires tagging the value
- Benefit gained by not descending into the structure in shallowTag() is offset by the cost of propagating RTTI.
- Empirically good tradeoff.

```
var c = q.center;
```

```
var c = RT.readField(q,"center");
```

```
function readField(o,f) {
  if (f==="rtti") die("reserved name");
  return shallowTag(o[f], fieldType(o.rtti, f)); }
```

ESTABLISHING INVARIANTS BY UPDATING RTTI

PART 4 OF 4

- When passing ‘c’ to ‘copy’, we need to check that ‘c’ is a ‘Point’
- checkAndTag() descends into the structure of ‘c’ and
- checks that ‘c’ is structurally a ‘Point’ and tags it’s RTTI.

- f() is called from g(), this check succeeds

```
function checkAndTag(v, t) {
  if (v === undefined) return v;
  if (isPrimitive(t)) {
    if (t === typeof v) return v;
    else die("Expected a " + t);
  } else if (isObject(t)) {
    for (var f in fields(t)) {
```

```
      checkAndTag(v[f.name], f.type);
    }; return shallowTag(v, t);
  } . . . }
```

ESTABLISHING INVARIANTS BY UPDATING RTTI

PART 4 OF 4 CONTD

- The call writeField(o, f, v) ensures that the value v being written into field f of object o is consistent with the typing invariants of that field.
- This call fails in our example. {x:"bad"} cannot be typed as "Point"

```
q.center = {x:"bad"};
```

```
RT.writeField(q, "center", {x:"bad"});
```

```
function writeField(o, f, v) {
  if (f==="rtti") die("reserved name");
  return (o[f]=checkAndTag(v,fieldType(o.rtti,f))); }
```

CONTENT

- A new style of efficient, RTTI based gradual typing
- **Performance Evaluation and Experiments**

PERFORMANCE, EVALUATION AND EXPERIMENTS

PERFORMANCE EVALUATION AND EXPERIMENTS

FOR SAFE TYPESCRIPT

- Differential subtyping is 1.4 - 3x faster than the models that added RTTI when object was generated.
- Migrated the Google's Octane benchmark suite to Safe TypeScript.
    1. Straight forward migration : just add "any" everywhere
    2. Discover a semantic bug
    3. Adding types restores performance parity to JavaScript.
- Bootstrapped the Safe TypeScript compiler. Detected static and runtime errors.
- Cost of dynamic type safety is a performance slowdown of 15%.

REFERENCES

- Aseem Rastogi, et. al “Safe and Efficient Gradual Typing for TypeScript”
- githut.info (image on slide 3)
- http://www.typescriptlang.org/play/ (image on slide 5)

FEEDBACK?

THAT'S ALL FOLKS