Experimental New Directions for JavaScript

Andreas Rossberg, V8/Google
Motivation

Broad need for (more) scalable JavaScript

- Usability, esp. maintainability
- Performance, esp. predictability

ES6 opens up new opportunities

Types desperately needed (but tricky)
An Experiment

Embrace Harmony

Shun bad legacy

Grow types

In a VM!
Two Parts (Working titles)

“SaneScript” – a cleaner subset of JavaScript
• Focus on removing features
• Transition path to...

“SoundScript” – a gradual type system for JavaScript
• Based on TypeScript, but sound & effective
• Does not depend on, but benefits from, SaneScript

Both fully interoperate with good old JavaScript
Plan

• Implement in V8, prototype in Traceur
• Test in the field, iterate
• Need feedback! Collaboration welcome
• Ideally, develop into ES proposals eventually
“SaneScript”

In an insane world, it was the sanest choice.
— Sarah Connor
Motivation

Guide programmers on well-lit path

• **Safer** semantics

• **Predictable** performance

• Aim for the 95% use cases
“Sane” Mode

- Opt-in: “use sanity” (TBD)
- Implies strict mode
- Freely interoperable with “insane” code
- Can still be run as “insane” code (with caveats)
Subsetting the Language

- **Static** restrictions (early errors)
- **Dynamic** restrictions (exceptions)
- **Per-object** restrictions ("sane objects")
Subsetting Compatibility

- Sane code not hitting any of the restrictions would have **same meaning** outside the mode.
- That is, “correct” sane code can run **unchanged** on VMs not recognising the opt-in.
Sane Scoping

- No more `var`
- No undeclared variables
- No use before declaration (**static dead zone**), except mutually recursive `function` declarations

`let` is the new `var`. Proper scoping FTW.
Sane Objects

- Accessing missing properties throws (on both reads & writes)
- Objects created in sane mode are non-extensible
- No freezing of non-configurable properties

If you want maps, you take maps.
Sane Classes

• Class objects and their prototypes are **frozen**

• Instances are created **sealed**

• Methods require proper instances

Fast and safe method & field access FTW.
Sane Arrays

• No holes, no accessors, no reconfiguration

• Length always in sync

• No out-of-bounds access, except extension at the end

Fast arrays FTW. Maps are the new sparse arrays.
Sane Functions

• No arguments object

• Calling with too few arguments throws

Optional and rest arguments FTW.
Sane Coercions

- Nothing implicit besides ToBoolean (almost?)
- == and != require compatible typeof

No more WAT, no more WTF.
Plan

• Implement in Q1/2
“SoundScript”

That’s sound advice at any time.

— Jean-Luc Picard
Motivation

• Everybody keeps inventing type systems for JS
  • Both user-facing and internal
• We strongly support standardisation!
• But inside a VM new requirements arise
• ...and new opportunities!
• Needs investigation
Design Goals

• Based on TypeScript (familiarity, reuse)
• Gradual (interop with untyped code)
• Sound (reliability, non-local optimisations)
• Precise (aggressive optimisations)
• Effective (feasible inside VM)
• Modular (lazy compilation, caching)
Sound Gradual Typing

- When it claims $E: T$, then, in fact, $E: T$
- But type `any` means “dynamically typed”
- Type `any` induces **runtime type checks** if necessary
- Protects invariants of statically typed code
- Disallow higher-order casts that’d require wrapping (expensive; observable in JavaScript!)
Runtime Type Checking

- Objects and functions carry (more) *runtime type information*
- Operations at type *any* may need to check
  - get, set, call, ...
- Should not be a common case
- Much cheaper when done by VM!
interface T extends U {
  a : number,
  m(x : string) : number,
  (x : boolean) : T,
  new(x : string) : U
}

(x : T) => U := {(x : T) : U}
Functions & Methods

• Can annotate type of this:
  function(this : T, x : U) {}  

• Function types are **contravariant** (soundness!)

• **Method** types are different, **covariant** in this
  (tied to concise method syntax)

• Method extraction only allowed when this : any
Nominal Classes

- Class C introduces **nominal instance type** C
- ...and **nominal class type** typeof C
- Both are **subtypes** of respective structural types
- “Interfaces” remain structural
Why Nominal?

- Sound private state
- Sound binary methods
- Sound first-class classes
- More efficient code
- More efficient compilation (it’s runtime, too!)
Nominal Typing, Example

class D extends C {
    public a : T
    constructor(x : T) {}
    m(x : T) : U {}
    static s(x : T) : U
}

• D < C

• D < {a:T, m(x:T):U, constructor: typeof D, ...C’s...}

• typeof D < {new(x:T):D, s(x:T):U, ...C’s...}
Subtyping

- Nominal type are subtypes of structural
- Vice versa also allowed (semi-structural types)
- No (depth) subtyping on mutable properties
- But on immutable properties
  - various requests for immutable data
- Invariant generics (at least for now)
• **Sound** (for realz)

• Runtime **type passing** (i.e., unerased)

• But no first-class instantiation (that is, \( f<T> \) is not a value)

• **Rationale:** would change operational semantics
Going More Gradual

- Choice between T or any not gradual enough
- Enter `any<T>` — restricts uses as if T, but provides no more guarantees than any
- Essentially, TypeScript’s interpretation of T
- Mainly for typing intrinsics, programmers shouldn’t need it often
Type Inference

- Bidirectional type checking
- No inference across function boundaries
- Don’t break lazy compilation!
Lazy Compilation

- Keep supporting function granularity jit
- Mayhaps require “deferred early errors”
- Consider eager type-checking later (cost?)
Numerous Challenges

- Would like “non-nullable” as default, feasible?
- Would like a proper integer type, how?
- How much immutability can we require in typed code?
- Full ES6: symbols, how avoid dependent types?
- Full ES6: first-class classes, how deal with generativity?
- Control-flow dependent typing, how much?
- Reflection, what API?
- Syntax, what to do about incompatibilities?
- Performance, how keep cost of type checking low?
- Blame tracking, do we need any in the absence of wrapping?
- Object.observe breaks all optimisation ideas
- ...
Plan

- Implement in Q2-4 (?)
Types in VM: Challenges

- Type system must respect *open world* assumption
  - additional definitions can be added at any time
- Type checking must be *efficient* enough
  - preference for nominal typing
- Must not break *lazy compilation* of functions
  - precludes non-local type inference
  - necessitates “deferred early error” semantics
Types in VM: Opportunities

- More optimisations!
  - aggressive ones require soundness
- Affordable runtime type checks
  - easier debugging
  - enabler for soundness
- Predictable performance
  - Reduced warm-up time
  - No opt/deopt cycles of death
- Ahead-of-time compilation/optimisation
Summary

- Both new **challenges** and new **opportunities** putting types into a VM
- Standardising an unsound type system would be a big lost opportunity
- This is an **experiment**
- All public, we would like your **feedback**!
Encore
Optional Types

- All types are "non-nullable" by default
  - preferably exclude both `null` and `undefined`, but the latter might be very hard to reconcile with existing APIs
- Type `?T` as short-hand for `T | undefined | null`
First-Class Classes

- Requires proper **class types**: `class C extends T {...}`
- Essentially, **F-bounded existential type**
- **Generative**: functions returning a class create a new class (i.e., existential type) with each call
- **Implicitly opened** when bound to a variable
- Classes as parameters behave dually (universal type)
- **do-expressions will introduce “avoidance problem”**