An Overview of the ECMAScript 4[1] Type System

Incomplete Draft, 21 November 2007

Address comments to cormac@ucsc.edu
(Note that FIXMEs are Trac ticket candidates or more generally language aspects that need to be clarified.)

Introduction

ECMAScript 4 (ES4) includes a *gradual type system* that supports a range of typing disciplines, including dynamically-typed code (as in ES3), statically-typed code (much like in Java or C#), and various combinations of statically- and dynamically-typed code, with convenient interoperation between the two.

In ES4, type annotations are supported, but not required. A variable with no type annotation implicitly is assigned type "*", meaning that it can hold any kind of value (much like variables is ES3).

ES4 provides two language modes. In "standard" mode, all type annotations are (conceptually, at least) enforced via dynamic checks. In "strict" mode, a program is first checked by a static type checker or *verifier* before being executed; any program that is ill-typed will be rejected by the verifier. If the program is not rejected, then it is executed as in "standard" mode. That is, there is no difference in run-time behavior between "strict" and "standard" modes.

Although type annotations are conceptually enforced via dynamic checks, this of course does not preclude compile-time analyses that attempts to optimize away many or all of these checks. These analyses could work in both "strict" and "standard" modes, and may resemble various forms of type inference, etc.

The ES4 Type Language

This section really depends on the Syntax Whitepaper.

A *reifiable type* is a type that is not "*", a like type, a wrap type, or a union type.

Allocated Types of Values

Every value in ES4 has an associated *allocated type*, which is a type that is associated with that value when the value is first allocated or created. An allocated type is always a reifiable type. The allocated type of a value is invariant; for example, updating the fields of an object cannot change the allocated type of that object.

- An object allocated via the syntax "{ ... } : T" has allocated type T.
- A function "function(x1:T1,...,xn:Tn) : S {...}" has allocated type "function(T1,...,Tn):S".
- etc

Current Types of Values

The *current type* of a value is determined by looking, not just at the allocated type of that value, but also at the contents of the value.

- If a structural object has fields x1,...,xn bound to values v1,...,vn, where each vi has current type Ti, then the current type of the structural object is "{ x1:T1,...,xn:Tn }".
- If an array contains values v1,...,vn, where each vi has current type Ti, then the type of the array is "[T1, ..., Tn]".

---

[1] Based on the 2007-10-23 Proposed ECMAScript 4th Edition Language Overview, the (unpublished) 2007-09-20 Library Specification draft, and the ES4 wiki; see http://www.ecmascript.org/. This paper uses the term "ES4" to denote the Proposed ECMAScript 4 language. Details given in this paper are subject to future adjustments as ES4 evolves.

- An all other cases (integers, functions, class instances etc) the current type of a value is simply its allocated type.

The Compatibility Relation on Types

The *compatibility* relation is a binary relation on types. We write S ~: T to indicate that the type S is compatible with type T. If S ~: T, then a value of type S can generally be assigned to a variable of type T. Note that because of gradual typing, some run-time checks may be involved in the presence of "*" or "like" types.

Compatibility is somewhat similar to the more traditional *subtype* relation, except that it is not transitively-closed. In particular, we have that "int ~: *" and "* ~: like bool", but the relation "int ~: like bool" does not hold.

- The type "*" is the maximal type under the compatibility relation: "T ~: *" for any type T.
- Variables of type "*" are essentially untyped, and can be implicitly cast to other types via the use of "like" types. In particular, "* ~: like T".
- Compatibility is reflexive: "T ~: T" for any type T.
- If a class C extends class D, then "C ~: D". More generally, given "class C.<x1,...,xn> extends D.<T1,...,Tm>", we have that "C.<S1,...,Sn> ~: D.<T1[x1:=S1,...,xn:=Sn],...,Tm[x1:=S1.,,,.xn:=Sn]>".
- For arrays, we have "[S] ~: [T]" provided that
  - Si ~: Ti for 1 <= i <= m and
  - Ti ~: like Si for 1 <= i <= m

Note that, unlike in Java, array compatibility is not invariant. That is, if a class C extends class D, then even though we have "C ~: D", the relation "[C] ~: [D]" does not hold.
The array type "[*]" functions as a maximal array type, in that any array type "[T]" is compatible with "[*]". This maximal array type plays a similar role to "Object[]" in Java, and requires a run-time check on writes to array elements.
- This notion of array compatibility extends to tuple types and structural object types in a similar fashion.
-

Run-time Conversions

Verification in "Strict" Mode

Union types are used to give names to sets of existing types, creating after-the-fact interfaces.