An Overview of the ECMAScript 4 Type System

Incomplete Draft, 8 January 2007

Address comments to cormac@ucsc.edu

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.

The ES4 Type Language

This section really depends on the Syntax Whitepaper.

In the following grammer, where *P* is a pattern we use the meta-syntax "P,.." to denote zero-or-more comma-separated occurrences of syntax matching *P*. We use this two-dot token ".." as part of the meta-syntax since the three-dot token "..." is used in the concrete syntax.

A type may be any of the following:

The type "null".

The dynamic type "*".

A *name*, possibly with a namespace qualifier, which may refer to a generic type parameter or a nominal type.

An instantiated nominal type "name.<Type,Type,..>", where *name* refers to a class or interface type.

A non-null type "Type!".

A nullable type "Type?"

A function type "function(Type,.., Type=,..):Type", with zero-or-more normal arguments and zero-or-more optional arguments.

A function type "function(Type,.., Type=,.., ...Type):Type", with zero-or-more normal arguments, zero-or-more optional arguments, and a rest argument. The final ":Type" is optional, and defaults to ":*" if omitted.

A generic function type "function.<X,..>(Type,.., Type=,..):Type" or "function.<X,..>(Type,.., Type=,.., ...Type):Type", where "X" is a binding occurrence of a simple name.

A structural object type "{name:Type,..}".

A union type "(Type,..)".

An array type "[Type]".

A tuple type "[Type,Type..]" with at least two element types.

A *like type* "like Type".

A *wrap type* "wrap Type".

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

In the context of a class definition "class C.<X> { ... }", the class name "C" is considered to be a type, but it is somewhat unusual in that we cannot declare a variable of type "C". We refer to "C" as a *higher-order type*, in that it requires additional type parameters to become a normal or *first-order type* of the form "C.<T>".

Subtyping

The subtyping relation is a binary relation on types. If S is a subtype of T, then a value of type S can generally be assigned to a variable or slot of type T. The types in a subtype relation may contain free type variables, which are assumed to denote the same but unknown type. (See the rule for generic functions below.) Two types are *equivalent* if they are both subtypes of each other. The subtyping relation is defined by the following rules:

Subtyping is reflexive and transitive.

Given a class definition "class C extends D { ... }", we have that C is a subtype of D.

More generally, given a class definition "class C.<x1,..,xn> extends D.<T1,..,Tm>", we have that "C.<S1,..,Sn>" is a subtype of "D.<T1[x1:=S1,..,xn:=Sn],..,Tm[x1:=S1.,,,.xn:=Sn]>". We use the notation "T[x:=S]" to denote the replacement of all free occurrences of the type variable "x" within "T" by the type "S", and we extend this notation to denote the simultaneous substitution of multiple type variables via the meta-syntax "T1[x1:=S1,..,xn:=Sn]".

For arrays, we have "[S]" is a subtype of "[T]" provided that S is equivalent to T. (Note that, unlike in Java, array subtyping is invariant, not covariant.)

For tuples, we have that "[S1,..,Sn]" is a subtype of "[T1,..,Tm]" if n>=m and for all i in 1..m, Si is equivalent to Ti.

A tuple "[S1,..,Sn]" is a subtype of the array type "[T]" if each S_i is equivalent to T.

The non-nullable type "T?" is equivalent to the union type "(T,null)".

For structural object types, "{l1:S1,..,ln:Sn}" is a subtype of "{l1:T1,..,lm:Tm}" if n>=m and for all i in 1..m, Si is equivalent to Ti. The ordering of "label:Type" bindings in a structural object type is irrelevant, and can be re-arranged without changing the meaning of that type.

For union types, S is a subtype of "(T1,..,Tn)" if there exists i in 1..n such that S is a subtype of Ti. Also, "(S1,..,Sn)" is a subtype of T if for all i in 1..n, Si is a subtype of T.

For function types, "function(S1,..,Sn):U" is a subtype of "function(T1,..,Tm):R' " if U is a subtype of R and n<=p and for all i in 1..m, Si is equivalent to Ti. (Note that function subtyping is *invariant* in the argument position, instead of contravariant.)

This rule generalizes to default arguments as follows: "function(S1,..,Sn,S_{n+1}=,..,Sm=):U" is a subtype of "function(T1,..,Tp,T_{p+1}=,..,Tq=):R " if U is a subtype of R and n<=p and q<=m and for all i in 1..q, Si is equivalent to Ti.

For rest arguments in the subtype only,"function(S1,..,Sn,S_{n+1}=,..,Sm=, ...A):U" is a subtype of "function(T1,..,Tp,T_{p+1}=,..,Tq=):R " if U is a subtype of R and n<=p and for all i in 1..min(q,m), Si is equivalent to Ti and A is equivalent to an array type "[B]" and for all i in m+1..q, Ti is equivalent to "B".

For rest arguments in both types,"function(S1,..,Sn,S_{n+1}=,..,Sm=, ...A):U" is a subtype of "function(T1,..,Tp,T_{p+1}=,..,Tq=, ...C):R " if U is a subtype of R and n<=p and for all i in 1..min(q,m), Si is equivalent to Ti and A is equivalent to an array type "[B]" and for all i in m+1..q, Ti is equivalent to "B" and A is equivalent to C.

For generic functions, alpha-renaming of the generic type variable preserves subtyping and equivalence. Moreover, "function.<X1,..,Xn>(argtypes):R" is a subtype of "function.<X1,..,Xn>(argtypes'):R' " if "function(argtypes):R" is a subtype of "function(argtypes'):R' ". That is, to check subtyping between generic functions, we rename the generic type variables to be identical in both types, and then proceed to ignore them.

A type S is always a subtype of “like S”, since “like S” describes more values. Also, the “like” type constructor is covariant, and so “like S” is a subtype of “like T” if S is a subtype of T.

A type “wrap S” is a subtype of S since it describes certain kinds of S-values (those that are wrapped). The “wrap” type constructor is covariant, and so “wrap S” is a subtype of “wrap T” if S is a subtype of T.

Compatible Types

The compatibility relation is a binary relation on types. Two types S and T are *compatible* if T can be obtained from S by replacing certain portions of S by the dynamic type "*". Thus, for example, "{x:int}" is compatible with both "{x:*}" and with "*", but "{x:*}" is not compatible with "{x:int}".

The compatibility relation is reflexive and transitive, but not symmetric.

Compatible Subtyping

A type S is a *compatible-subtype* of a type T if there exists some type U such that S is a subtype of U and U compatible with T.

For example, "{x:int, y:bool}" is a compatible-subtype of "{x:*, y:*}", "{x:int}", "{x:*}", and "*".

The compatible-subtyping relation is reflexive and transitive, but not symmetric.

Evaluation

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".

In general, if a slot of type T holds a value v of type S, then S is a compatible-subtype of T. (There are some exceptions where T is a like-type.)

Note: Perhaps the evaluation whitepaper will cover the allocated types of values, the convert operation, is-checks, and wrap operations (terminology from the ValleyScript paper).

Consistent Types

The consistency relation is a binary relation on types. Two types S and T are *consistent* if there exists some type U such that both S and T can be converted into U by replacing occurrences of the dynamic type "*" by more specific types. For example, "{x:int, y:*}" is consistent with both "{x:*, y:bool}".

The consistency relation is reflexive and symmetric, but not transitive.

Matching Types

The *matching* relation is used in strict mode (see below). This *matching* relation is a binary relation on types. It is largely defined by the following rule:

A type S matches type T if there exists some type U such that S is a subtype of U and U is consistent with T.

In addition, the matching relation deals with various kinds of implicit conversions. (TODO: fill in these details.)

Strict Mode

In strict mode, whenever the result of an expression of type S is assigned to a slot of type T, the verifier checks that the type S *matche*s type T.

(TODO: fill in these details.)