Ecma/TC39/2008/058

1 Values

1 The evaluation of a program, described in section ..., entails among its effects the calculation and manipulation of values.

FIXME Draft 1 of the specification does not include a description of evaluation.

2 An *ECMAScript value* is either undefined, null, or an object. Every ECMAScript value has an associated ECMAScript type, called the value's *allocated type*. The allocated type is fixed when the value is allocated in memory, and cannot change over the lifetime of the value.

Semantics

3
```
datatype VALUE = ObjectValue of OBJECT
               | UndefinedValue
               | NullValue
```

COMPATIBILITY NOTE In the 3rd edition of the language, several individual *types* were defined. The three types formerly called primitive (number, string, boolean) are now represented as object values. The term *type* has a different meaning in the 4th edition.

1.1 Object Values

1 An *object value* is any ECMAScript value that is not the null value or the undefined value.

2 An object value consists of a mutable property binding map, an immutable fixture map, an immutable object identifier, an immutable tag, and an immutable prototype reference.

Semantics

3
```
and OBJECT =
    Object of { propertyMap: PROPERTY_MAP,
                fixtureMap: FIXTURE_MAP,
                ident: OBJECT_IDENTIFIER,
                tag: TAG,
                proto: VALUE }
```

1.1.1 Property Maps

1 A *property map* associates at most one property with any name. If an object's property map associates a property `P` with a name `N`, then the object is said to have a *binding* for `N`. Alternatively, the property `P` is said to be *bound to* the name `N`, in the object.

2 Bindings can be added, removed, or replaced within a property map. The semantic type of a property map is unspecified.

3 A property map stores the order in which properties are added to the map. A property's position in this order is unchanged when the property is replaced. This order is used by property enumeration (see the chapter on Statements).

FIXME "Replacement" is not an adequate abstraction here; we wish to have an "update" operation. Replacement on the language level occurs when a property is deleted by the delete operator (or an equivalent mechanism) and a new property with the same name is inserted; under this kind of replacement, the property's position may change. With a "replacement" on the semantic level, that is to say an update, the property's position never changes.

Semantics

4
```
and PROPERTY_MAP = …
```

1.1.1.1 Properties

1 A *property* consists of a type, a state, and a set of attributes. The type of a property is also called the property's *storage type*, to differentiate it from the allocated type of any value that the property may contain.

COMPATIBILITY NOTE In earlier editions of the language, some characteristics of an object were modeled as *internal properties* with distinct names such as `[[Class]]` or `[[Value]]`. These characteristics of objects are described differently in the 4th edition, using a combination of supporting semantic and ECMAScript standard library functionality.

Semantics

2
```
and PROPERTY = { ty: TYPE,
                 state: PROPERTY_STATE,
                 attrs: ATTRS }
```

FIXME The term *storage type* is not ideal because it also applies to the return value constraint on a function object; there is no "storage" in that context. It's possible that *annotated type* would be a better term.

1.1.1.2 Property States

1 The *state* of a property encodes either a value associated with the property, or else a pair of functions that describe a "virtual" value.

2 If a property is in the *value* state, reading the property returns the value and writing the property updates the value.

3 If a property is in the *virtual value* state, reading the property executes the associated "getter" function, and writing the property executes the associated "setter" function.

Semantics

4
```
and PROPERTY_STATE = ValueProperty of VALUE
                   | VirtualProperty of
                     { getter: CLOSURE option,
                       setter: CLOSURE option }
```

FIXME It is probably not necessary for the getter and setter to be "option", the missing part of the pair is always generated by the language implementation.

1.1.1.3 Property Attributes

1 The *attributes* of a property govern its behavior in various operations. There are 4 attributes on every property:

Attribute	Description
`writable`	An attribute that can be one of three values. When the value is `Writable`, the property can be written to an arbitrary number of times. When the value is `WriteOnce`, the property can be written to once, after which the attribute assumes the value `ReadOnly`. When the value is `ReadOnly`, attempts to write to the property after initialization will fail.
`enumerable`	A boolean attribute. If `true`, then the property is to be enumerated by **for-in** and **for-each-in** enumeration. If `false`, the property is ignored by such enumeration.
`removable`	A boolean attribute. If `true`, then the property can be removed using the **delete** operator. If `false`, the **delete** operator fails.
`fixed`	A boolean attribute. If `true`, then the property was defined as a fixture in the object's fixture map and dominates most non-fixed properties during name resolution. If `false`, then the property is a dynamic addition to the object and is usually consulted *after* fixed properties during name resolution.

2

3 The `fixed` attribute is mutually exclusive with the `removable` attribute.

4 If a property is not `Writable` it is also not `removable`.

5 If a property is `fixed` it is not `enumerable`.

Semantics

6
```
datatype WRITABILITY = ReadOnly | WriteOnce | Writable

type ATTRS = { removable: BOOLEAN,
               enumerable: BOOLEAN,
               fixed: BOOLEAN,
               writable: WRITABILITY }
```

1.1.1.4 Names

1 A *name* consists of a namespace and an identifier.

2 A name may identify a property binding within a property map.

Semantics

3
```
type NAME = { ns: NAMESPACE, id: IDENTIFIER }
```

1.1.1.4.1 Identifiers

1 An *identifier* is a string.

Semantics

2
```
type IDENTIFIER = STRING
```

1.1.1.4.2 Namespaces

1 A *namespace* is a semantic value that can be either transparent or opaque.

2 A *transparent namespace* consists of a character string that identifies the namespace.

3 An *opaque namespace* consists of a unique namespace identifier of unspecified representation.

Semantics

4
```
datatype NAMESPACE =
         TransparentNamespace of STRING
       | OpaqueNamespace of OPAQUE_NAMESPACE_IDENTIFIER

type OPAQUE_NAMESPACE_IDENTIFIER = …
```

Special namespaces

5 Two important namespaces are used throughout the following sections.

6 The *public namespace* is the transparent namespace whose identifying string is the empty string, "".

7 The *4th Edition namespace* is the transparent namespace whose identifying string is the string **__ES4__**.

8 All global property names defined subsequently in this specification are written in one of two forms:

- Qualified, using the ECMAScript qualitied name expression notation **namespace::identifier**
- Unqualified, using the notation of a bare **identifier**, implicitly qualified by the 4th Edition namespace

9 The public namespace is bound to the name **public**.

10 More information on special namespaces is given in section ...Names, Special namespaces.

1.1.2 Object Prototype

1 The *prototype* of an object is a means of dynamically delegating behavior from one object to another. In various conditions, the result of an unsuccessful property access on an object is defined in terms of subsequent property accesses on the object's prototype.

2 The value of the prototype can be the null value or an object value.

1.1.3 Object Identifier

1 The *identifier* of an object uniquely identifies the object. The semantic type of an object identifier is unspecified, and its value cannot be directly observed by ECMAScript code. Equality of objects is partially defined in terms of equality of the objects' identifiers, so all identifiers must be comparable with one another for equality.

Semantics

2
```
and OBJECT_IDENTIFIER = …
```

1.1.4 Object Tag

1 The *tag* of an object encodes both the object's allocated type, and any underlying semantic value associated with the object.

2 The `RecordTag` tag is present on an object of with the structural type `RecordType` as its allocated type.

3 The `ArrayTag` tag is present on an object of with the structural type `ArrayType` as its allocated type.

4 The `PrimitiveTag` tag is present on objects that are instances of a small number of classes, described in the following section.

5 The `InstanceTag` tag is present on any object that is an instance of a class but does *not* have an `RecordTag`, `ArrayTag` or `PrimitiveTag` tag.

6 The `NoTag` tag is present only on un-named objects that implement scopes.

Semantics

7
```
and TAG =
    RecordTag of FIELD_TYPE list
  | ArrayTag of (TYPE list * TYPE option)
  | PrimitiveTag of PRIMITIVE
  | InstanceTag of CLASS
  | NoTag
```

1.1.4.1 Primitive Tag

1 In addition to an allocated type, some objects have an extra semantic value stored in their tag. Such objects are called *primitive objects* and have a *primitive tag* containing the semantic value. The extra semantic value is only directly accessible in semantic code.

2 ECMAScript code can determine if an object is primitive through a correspondence between primitive tags and a set of 9 specific ECMAScript types. The correspondence is a bijection: any instance of these types has the corresponding primitive tag, and any object with a primitive tag is an value of the corresponding type.

3 The allocated type of a primitive object may be a subtype of the corresponding type listed here. In particular, function objects may have more specific subtypes of the class **public::Function**, and class and interface objects are both subtypes of the interface **helper::Type**. In such cases, the allocated type of the object is described by the semantic value held by the primitive tag.

4 The correspondence between primitive tags and object types is the following:

- The primitive tag `BooleanPrimitive` corresponds to the class **boolean**.
- The primitive tag `DoublePrimitive` corresponds to the class **double**.
- The primitive tag `DecimalPrimitive` corresponds to the class **decimal**.
- The primitive tag `StringPrimitive` corresponds to the class **string**.
- The primitive tag `NamespacePrimitive` corresponds to the class **Namespace**.
- The primitive tag `FunctionPrimitive` corresponds to the class **public::Function**.
- The primitive tag `TypePrimitive` corresponds to the class **helper::Type**.
- The primitive tag `GeneratorPrimitive` corresponds to the class **helper::GeneratorImpl**.
- The primitive tag `ArgumentsPrimitive` corresponds to the class **helper::Arguments**.

Semantics

5
```
and PRIMITIVE =
    BooleanPrimitive of BOOLEAN
  | DoublePrimitive of IEEE_754_BINARY_64_BIT
  | DecimalPrimitive of IEEE_754R_DECIMAL_128_BIT
  | StringPrimitive of STRING
  | NamespacePrimitive of NAMESPACE
  | FunctionPrimitive of CLOSURE
  | TypePrimitive of TYPE
  | ArgumentsPrimitive of SCOPE
  | GeneratorPrimitive of GENERATOR
```

1.1.5 Fixture Map

1 A *fixture map* is a structure that describes, but does not contain, a set of fixed property bindings. These descriptions of fixed properties are called *fixtures*. The properties described by an object's fixture map are *lazily instantiated* as fixed property bindings on the object. Any attempt to access a property binding described by a fixture in an object's fixture map, but not present in the object's property map, causes the property to be added to the property map.

2 All instances of a class share a single fixture map.

1.2 Undefined Value

1 The *undefined value* is a unique constant denoted by the semantic value `UndefinedValue` and stored in the global constant property **public::undefined** in ECMAScript.

2 The *undefined type* is the allocated type of the undefined value. The undefined value is the only value with the undefined type as its allocated type. The undefined type is denoted by the semantic value `UndefinedType`, which is denoted in ECMAScript type-expression contexts by the identifier **undefined**.

COMPATIBILITY NOTE Inside of type-expression contexts, the token **undefined** is reserved and has a fixed meaning. Outside of type-expression contexts the token is interpreted as in earlier editions.

1.3 Null Value

1 The *null value* is a unique constant denoted by the semantic value `NullValue` and by the null literal **null** in ECMAScript.

2 The *null type* is the allocated type of the null value. The null value is the only value with the null type as its allocated type. The null type is denoted by the semantic value `NullType` and denoted in ECMAScript type-expression contexts by the null literal **null**.

NOTE While the null and undefined values have similar meanings, they have different conventions of use. The null value is intended to indicate a missing object value, while the undefined value is intended to indicate a missing property on an existing object value or an uninitialized property or variable. These intended uses are conventions, and are not enforced by the language semantics.

1.4 Semantic Values

1 Many aspects of the language depend on the semantic values associated with primitive objects. The following sections describe the semantic values and the correspondences that exist between particular semantic values and the ECMAScript values they are held by.

1.4.1 Special Constructors

1 While much of the behavior of primitive objects is defined *inside* the ECMAScript language (in the section ...library), the means of *constructing* primitive objects and associating semantic values with them is (at least partially) defined *outside* the ECMAScript language, in semantic code and specification prose.

2 Therefore the construction of any primitive object is described by a *special constructor* defined in semantic code and specification prose, rather than a *standard constructor* that would otherwise be defined in standard library code. The specifications of any such special constructors are given in the following sections, accompanying the specifications of the semantic values.

1.4.2 Boolean Values

1 A *boolean value* is one of two semantic values called `true` and `false`. These correspond to the ES4 boolean literal values **true** and **false**, which denote the two sole instances of the class **boolean**. Such objects are called *boolean objects*.

2 No instances of the class **boolean** can be constructed aside from the two values **true** and **false**: the **boolean** constructor is a special constructor that always evaluates to one of the two boolean objects.

1.4.3 Double Values

FIXME I cut this section down significantly from ES3, since the corresponding section 8.5 in the old standard mostly consisted of a very weird sort of selective paraphrasing of bits of 754 itself: restatements of algorithms that are perfectly well described in 754, or of facts such as the definition of the denormalized numbers that *never even get used* in the subsequent spec. I assume anyone reading this section and caring about 754 doubles actually has the 754 spec and can read it. Spelling out the whole 754 spec title in this section likewise seems redundant, since that's the point of the normative references section at the beginning of the document.

FIXME Waldemar objects to that paring down, pointing out that the purpose of the selective paraphrasing was to include a specific subset of IEEE 754 arithmetic into ES3. For example, signalling NaNs are not part of that subset, and there are (supposedly) competing round-to-nearest algorithms, of which one needed to be selected. So it's possible that the real fix here is to be explicit as to why a subset of IEEE 754 arithmetic is described in the ECMAScript Specification.

1 A *double value* is a double precision, 64-bit format binary floating point value, as specified in the IEEE 754 standard.

2 A double value can be held in the primitive tag of an instance of the class **double**. Instances of **double** are called *double objects*.

3 Two special double values are held in special double objects: one "Not-a-Number" (NaN) value, stored in the global constant **public::NaN**, and one "infinite" value, stored in the global constant **public::Infinity**.

FIXME There are also NaN and Infinity properties (as well as others) on the `Number` object; those are all double values. ES4 will have NaN and Infinity properties on the `decimal` object, and probably on the `double` object for the sake of consistency.

4 ECMAScript provides no way of distinguishing any of the different IEEE 754 NaN values from one another. All NaN values are considered unequal to themselves, and to every other value.

5 In this specification, the phrase "the number value of *x*" where *x* represents an exact nonzero real mathematical quantity means a number chosen according to the IEEE 754 rounding mode "rounds to nearest".

FIXME That does not take into account decimal.

6 Some ECMAScript operators deal only with integers in the range -2^{31} through $2^{31}-1$, inclusive, or in the range 0 through $2^{32}-1$ inclusive. These operators accept any double or decimal value but first convert each such value to one of 2^{32} integer values. See descriptions of the `ToInt32` and `ToUint32` operators in sections ...

1.4.4 Decimal Values

1 A *decimal value* is a 128-bit format decimal floating point value, as specified in the IEEE 754r standard.

2 A decimal value can be held in the primitive tag of an instance of the class **decimal**. Such objects are called *decimal objects*.

3 Some ECMAScript operators convert double values to decimal values when either operand to the operator is a decimal value. This conversion can be lossy.

FIXME More information will appear here.

1.4.5 String Values

1 A *string value* is a finite ordered sequence of zero or more unsigned integer values ("elements"). The elements of a string must be either 16 or 32 bits wide. An implementation of ECMAScript may provide elements of either size, but all strings in a single implementation must consist of elements of the same size.

2 String values are generally used to represent textual data, in which case each element in the string is treated as a code point value (see section ...).

3 A string value can be held in the primitive tag of an instance of the class **string**. Such objects are called *string objects*.

4 Each element of a string is regarded as occupying a position within the sequence. These positions are indexed with nonnegative integers. The first element (if any) is at position 0, the next element (if any) is at position 1, and so on. The length of a string is the number of elements (16 or 32-bit values) within it. The empty string has length zero and therefore contains no elements.

5 All operations on string (except as otherwise stated) treat them as sequences of undifferentiated 16 or 32-bit unsigned integers. In particular, operations on strings do not ensure the resulting string is in normalised form, they do not ensure language-sensitive results, and they do not alter their behavior when dealing with 16 or 32-bit values outside the legal range of UTF-16 or UTF-32 code points, respectively.

NOTE The rationale behind these decisions was to keep the implementation of strings as simple and high-performing as possible. The intent is that textual data coming into the execution environment from outside (e.g., user input, text read from a file or received over the network, etc.) be converted to Unicode Normalised Form C before the running program sees it. Usually this would occur at the same time incoming text is converted from its original character encoding to Unicode (and would impose no additional overhead). Since it is recommended that ECMAScript source code be in Normalised Form C, string literals are guaranteed to be normalised (if source text is guaranteed to be normalised), as long as they do not contain any Unicode escape sequences.

FIXME The previous paragraphs regarding string values are adapted from ES3, but personally I think they are very awkward-reading, and would like to rewrite them a bit.

6 String literals evaluate to string objects.

7 The equality of string objects -- in both the **==** and **===** sense -- is defined as the equality of the underlying string values. This in turn is established by the identities of the string elements, considered pairwise and in sequence. Inequalities and relational operations of strings are similarly defined in terms of sequence comparisons on string elements. No other forms of textual equality or collation are defined.

1.4.6 Namespace Values

1 Namespaces are defined in section ...namespaces.

2 A namespace can be held in the primitive tag of an instance of the class **Namespace**. Such objects are called *namespace objects*.

3 A namespace is defined as a fixture in a global or class static scope by a **namespace** definition.

1.4.7 Type Values

1 A *type value* is a description of a set of values. Types are described in chapter ...types.

2 A type value can be held in the tag of an object, in a primitive tag `TypePrimitive`. An object of such a primitive type tag is called a *type object*.

3 Two sorts of type values are of particular significance: class values and interface values.

1.4.7.1 Class Values

1 A *class value* consists of a name and a set of namespaces, fixture maps, types and flags governing the behavior of various objects.

2 A class value can be held in a `ClassType` value, which can be held in the tag of a type object. An object carrying a primitive type tag of class type is called a *class object*.

3 A class value is defined as a fixture in the global scope by a **class** definition.

4 Each *class definition* corresponds to zero or more class values, and thus zero or more class objects. If a class definition is not type-parametric, it corresponds to exactly one class object, and that class object is called *the value of* the class definition.

5 A class value holds class fixtures and instance fixtures. If **C** is a class object, then the *class fixture map* of the associated class value describes the fixed properties found on the class object **C**, and the *instance fixture map* describes the fixed properties found on objects that are instances of the class value held in **C**.

6 If a class definition is type-parametric, each unique application of a set of type arguments produces a new class object with its own property map and unique copy of the class fixture map, specialized to the type arguments provided.

7 Class values can be *instantiated* to produce new objects. Instantiation is described in section....

Semantics

8
```
and CLASS =
    Class of
      { name: NAME,
        privateNS: NAMESPACE,
        protectedNS: NAMESPACE,
        parentProtectedNSs: NAMESPACE list,
        typeParams: IDENTIFIER list,
        nonnullable: BOOLEAN,
        dynamic: BOOLEAN,
        extends: TYPE option,
        implements: TYPE list,
        classFixtureMap: FIXTURE_MAP,
        instanceFixtureMap: FIXTURE_MAP,
        instanceInits: HEAD,
        constructor: CTOR option }
```

1.4.7.2 Instance Types and Class Types

1 A class corresponds to a pair of types: an instance class type and a static class type.

2 The *instance type* of a class value `C` is `InstanceType C`, denoted in a type expression by the name of `C` itself, and is the allocated type of any instance of `C`. The tag of any instance of `C` is `InstanceTag C`.

3 The *class type* of a class value `C` is the allocated type of the class object *holding* `C`. The tag of such an object is `PrimitiveTag (TypePrimitive (ClassType C))`. The allocated type of such an object is `ClassType C`, which is defined as a subtype of the `InstanceType` **helper::ClassTypeImpl**. The class fixtures in the class `C` are defined as instance fixtures on the class object holding `C`.

1.4.7.3 Interface Values

1 An *interface value* consists of a name and a set of fixtures and types.

2 An interface value can be held in an `InterfaceType` value, which can be held in the tag of a type object. An object carrying a primitive type tag of interface type is called an *interface object*.

3 An interface value is defined as a fixture in the global scope by an **interface** definition.

4 Each *interface definition* corresponds to zero or more interface objects. If an interface definition is not type-parametric, it corresponds to exactly one interface object, and that interface object is called *the value of* the interface definition.

5 An interface value contains declarations of *instance fixtures*, but no definitions.

6 Interfaces are *implemented* by classes, and any class implementing an interface must define, for each instance fixture declared in the interface, an instance fixture with the same name and type of the instance fixture.

7 An interface value `I` also defines a type `InterfaceType I`. If a class `C` implements interface `I`, the type `ClassType C` is a subtype of `InterfaceType I`.

Semantics

8
```
and INTERFACE =
    Interface of
      { name: NAME,
        typeParams: IDENTIFIER list,
        nonnullable: BOOLEAN,
        extends: TYPE list,
        instanceFixtureMap: FIXTURE_MAP }
```

1.4.8 Closure values

1 A *closure value* consists of a captured scope chain, an optional captured `this` object, and a function value.

2 A closure value can be held in the primitive tag of an instance of the class **public::Function**. Such objects are called *function objects*.

3 A closure value is defined as a fixture in a scope using a **function** definition.

4 Each *function definition* corresponds to zero of more function objects.

5 A *function expression* may also evaluate to a function object.

6 A *function value* contains a set of parameter fixtures, a type, and a block of ECMAScript code.

7 Closure values can be *invoked* to evaluate the ECMAScript code stored in the block of the closure's associated function value. Invocation is described in section....

FIXME Function definitions can be type-parametric; needs to be described.

Semantics

8
```
withtype CLOSURE =
         { func: FUNC,
           this: OBJECT option,
           env: SCOPE }

and FUNC =
    Func of
    { name: FUNC_NAME,
      fsig: FUNC_SIG,
      native: BOOLEAN,
      generator: BOOLEAN,
      block: BLOCK option, (* NONE => abstract *)
      param: HEAD,
      defaults: EXPRESSION list,
      ty: TYPE,
      loc: LOC option }
```

1.4.9 Generator Values

FIXME fill in

2 Reading and Writing Properties

1 This chapter describes the algorithms for *property access*: testing objects for the presence of a property, reading from and writing to a property, and removing a property. Property access is always by the name of the property. A property name is represented either as an instance of the pre-defined class **Name**, or as a **string** (which represents a name in the **public** namespace).

SPEC NOTE This chapter complements the chapter on names, scopes, and name resolution. At this time, there is some overlap between the two chapters.

2 Property accesses are subject to run-time checks, and property access fails (an exception is thrown) if a check does not pass. The exact exception depends on the particular check.

NOTE For example, a property created by **let** or **const** or a property whose type is a non-nullable type without a default value must be written (initialized) before it is read; properties created by **const** cannot be written more than once; and properties that have type annotations can be updated with a new value

only if the allocated type of the new value is a compatible subtype of the storage type of the property. A **ReferenceError** is thrown in the first two instances, and a **TypeError** is thrown in the last.

3 A property may be virtual, that is to say, the reading and writing of the property may be implemented by *getter* and *setter* methods on the object, and an expression that is syntactically a reference to the property is in fact an invocation of these methods. Virtual dynamic properties may be implemented by *catch-all* methods.

FIXME We need a definition of "method", this is the first use.

2.1 Catch-All Methods

1 This section contains a normative overview of the catch-all facility. A more precise, also normative, description is given in later sections of this chapter, as part of the general description of property access.

SPEC NOTE Any conflicts between the two descriptions are obviously bugs.

2 Objects may contain fixtures in the **meta** namespace: **meta::get**, **meta::set**, **meta::has**, and **meta::delete**. These properties always name methods. Jointly they are known as *catch-all methods*.

NOTE The requirement that the **meta** properties always name methods must be checked by the language implementation. The namespace **meta** is reserved and known to the implementation and may only be used in specific circumstances. See section (...).

3 If a catch-all method is defined on the object then it is invoked when a dynamic property is accessed: **meta::has** is invoked to determine if the object has the property; **meta::get** is invoked to read the property's value; **meta::set** is invoked to update or create the property; and **meta::delete** is invoked to delete the property. A catch-all method is invoked even if the dynamic property that is being accessed already exists on the object.

4 A catch-all method operates on the receiver object of the method call, not on the receiver's prototype objects.

5 If a catch-all method returns normally then the value it returns (if any) becomes the result of the property access, possibly after being converted to a canonical type.

6 If a catch-all method throws an exception, and the exception thrown is an instance of the pre-defined class **DefaultBehaviorClass**, then the default behavior for the catch-all is triggered.

7 **DefaultBehaviorClass** is a singleton class; its only instance is is stored in the global constant **DefaultBehavior**.

NOTE The mechanism is analogous to the one defined for iterators, where an instance of the singleton **iterator::StopIterationClass** is stored in the global property **iterator::StopIteration**.

8 The **meta::get** method is invoked on one argument, a property name. The value returned is the property value. The default behavior for **meta::get** is to retrieve the value from a dynamic property in the object's property map.

9 The **meta::set** method is invoked on two arguments, a property name and a value. Any value returned is ignored. The default behavior for **meta::set** is to update or attempt to create a dynamic property in the object's property map.

10 The **meta::has** method is invoked on one argument, a property name. Any value returned by the method is converted to **boolean**. The default behavior for **meta::has** is to search for a dynamic property in the object's property map.

11 The **meta::delete** method is invoked on one argument, a property name. Any value returned by the method is converted to **boolean**. The default behavior for **meta::delete** is to attempt to delete a dynamic property from the object's property map.

2.2 Checking for the Presence of a Property

1 The `HasOwnProperty` protocol is invoked to check whether an object *obj* contains a property named by *name*.

SPEC NOTE In terms of the 3rd Edition Specification, the `HasOwnProperty` protocol implements the test for whether an object "has a property", as used in the implementations of `[[Get]]`, `[[Put]]`, `[[HasProperty]]`, and other internal subroutines.

2 An object is said to contain a property if the property is in the object's property map or if the **meta::has** catchall claims the property to be present.

Semantics

3
```
and hasOwnProperty (regs : REGS)
                   (obj  : OBJECT)
                   (n    : NAME)
    : bool =
    let
        val Object { propertyMap, ... } = obj
    in
        if Fixture.hasFixture (getFixtureMap regs obj) (PropName n)
        then true
        else
        if hasFixedProp propertyMap n then
```

```
            true
        else
            if hasFixedProp propertyMap meta_has then
                let
                    val v = evalNamedMethodCall regs obj meta_has [newName regs n]
                in
                    toBoolean v
                end
                handle ThrowException e =>
                        let
                            val ty = typeOfVal regs e
                            val defaultBehaviorClassTy =
                                instanceType regs helper_DefaultBehaviorClass []
                        in
                            if ty <* defaultBehaviorClassTy then
                                hasProp propertyMap n
                            else
                                throwExn e
                        end
            else
                hasProp propertyMap n
    end
```

NOTE The `regs` parameter represents the virtual machine state. The operator `<*` tests subtype compatibility.

2.3 Reading a property value

1 The `GetPropertyValue` protocol is invoked to read the value of a property named by *name* from an object *obj*. The flag *isStrict* is true if the ES4 code that caused GetProperty to be invoked was compiled in strict mode.

2 Specifically, there will be an AST node for the property reference whose `strict` flag is set because it represents a source code phrase that was recognized in a region of code that was covered by a strict mode pragma.

SPEC NOTE There may be several types of AST nodes carrying strict flags and invoking `GetPropertyValue`, depending on how the AST is eventually structured.

FIXME Strict mode is not implemented in this code.

Semantics

3
```
and getPropertyValue (regs:REGS)
                     (obj:OBJECT)
                     (name:NAME)
    : VALUE =
    getPropertyValueOrVirtual regs obj name true

and getPropertyValueOrVirtual (regs:REGS)
                              (obj:OBJECT)
                              (name:NAME)
                              (doVirtual:bool)
    : VALUE =
    let
        val Object { propertyMap, tag, ... } = obj
    in
        case findProp propertyMap name of
            SOME {state=(ValueProperty v), ...}
            => v

          | SOME {state=(VirtualProperty { getter, ... }), ...}
            => if doVirtual
               then
                   case getter of
                       SOME g => invokeFuncClosure (withThis regs obj) g NONE []
                     | _ => UndefinedValue
               else
                   UndefinedValue

          | NONE =>
            case Fixture.findFixture (getFixtureMap regs obj) (PropName name) of
                SOME fixture
                =>
                   (reifyFixture regs obj name fixture;
                   getPropertyValueOrVirtual regs obj name doVirtual)

              | NONE =>
                case (isNumericName name, tag) of
                    (true, ArrayTag (_, SOME defaultType))
                    => let
                            val defaultVal = defaultValueForType regs defaultType
```

```
                        in
                            case defaultVal of
                                NONE => throwExn (newTypeErr …)
                              | SOME dv
                                => (setPropertyValueOrVirtual regs obj name dv false;
                                    dv)
                        end
                  | _
                    => if doVirtual andalso
                          Fixture.hasFixture (getFixtureMap regs obj) (PropName meta_get)
                       then
                           evalNamedMethodCall regs obj meta_get [newString regs (#id
name)]
                           handle ThrowException e =>
                                  let
                                      val ty = typeOfVal regs e
                                      val defaultBehaviorClassTy =
                                          instanceType regs helper_DefaultBehaviorClass []
                                  in
                                      if ty <* defaultBehaviorClassTy then
                                          getPropertyValueOrVirtual regs obj name false
                                      else
                                          throwExn e
                                  end
                       else
                           if isDynamic regs obj
                           then UndefinedValue
                           else throwExn (newRefErr …)
    end
```

NOTE A bound method has identity, so if **m** is a method on the class of some object **o** then the expression **o.m** always evaluates to the same object value (in terms of **===**).

2.4 Writing a property value

1 The `SetProperty` protocol is invoked to write a value *value* to a property named by *name* on an object *obj*. The object may or may not have a property of that name when SetProperty is invoked, and SetProperty may attempt to create the property. The flag *isStrict* is true if the ES4 code that caused SetProperty to be invoked was compiled in strict mode.

2 Specifically, there will be an AST node for the property update whose `strict` flag is set because it represents a source code phrase that was recognized in a region of code that was covered by a strict mode pragma.

FIXME Strict mode is not implemented in this code.

Semantics

3
```
and setPropertyValue (regs:REGS)
                     (base:OBJECT)
                     (name:NAME)
                     (v:VALUE)
    : unit =
    setPropertyValueOrVirtual regs base name v true

and setPropertyValueOrVirtual (regs:REGS)
                              (obj:OBJECT)
                              (name:NAME)
                              (v:VALUE)
                              (doVirtual:bool)
    : unit =
    let
        val Object { propertyMap, tag, ... } = obj
    in
        case findProp propertyMap name of
            SOME existingProp =>
            let
                val { state, attrs, ty, ... } = existingProp
                val { removable, enumerable, fixed, writable } = attrs

                fun writeExisting _ = writeProperty regs propertyMap name v ty
                                                    removable enumerable fixed
                                                    (case writable of
                                                         ReadOnly => ReadOnly
                                                       | WriteOnce => ReadOnly
                                                       | Writable => Writable)
            in
                case state of
                    ValueProperty _
                    => writeExisting ()
```

```
                    | VirtualProperty { setter, ... }
                      =>
                      if doVirtual
                      then
                          case setter of
                              NONE => ()
                            | SOME s => (invokeFuncClosure (withThis regs obj) s
                                                           NONE [v]; ())
                      else
                          if writable = ReadOnly
                          then throwExn (newTypeErr …)
                          else writeExisting ()

              end
            | NONE =>
              case Fixture.findFixture (getFixtureMap regs obj) (PropName name) of
                  SOME (ValFixture {ty, writable})
                  => writeProperty regs propertyMap name v ty
                                   false false true
                                   (if writable
                                    then Writable
                                    else ReadOnly)

                | SOME f
                    (reifyFixture regs obj name f;
                     setPropertyValueOrVirtual regs obj name v doVirtual)

                | NONE
                  =>
                  case (isNumericName name, tag) of
                      (true, ArrayTag (_, SOME defaultType))
                      => writeProperty regs propertyMap name v defaultType true true
                                       false Writable

                    | _
                      =>
                      if
                          doVirtual andalso
                          Fixture.hasFixture (getFixtureMap regs obj)
                                             (PropName meta_set)
                      then
                          ((evalNamedMethodCall regs obj meta_set
                                                [newString regs (#id name), v]; ())
                          handle ThrowException e =>
                                 let
                                     val ty = typeOfVal regs e
                                     val defaultBehaviorClassTy =
                                         instanceType regs helper_DefaultBehaviorClass []
                                 in
                                     if ty <* defaultBehaviorClassTy then
                                         setPropertyValueOrVirtual regs obj name v false
                                     else
                                         throwExn e
                                 end
                          else
                              if isDynamic regs obj
                              then writeProperty regs propertyMap name v AnyType true true
                                                 false Writable
                              else throwExn (newTypeErr …)
    end

and writeProperty (regs:REGS)
                  (propertyMap:PROPERTY_MAP)
                  (name:NAME)
                  (v:VALUE)
                  (ty:TYPE)
                  (removable:BOOLEAN)
                  (enumerable:BOOLEAN)
                  (fixed:BOOLEAN)
                  (writable:WRITABILITY)
    : unit =
    let
        val newProp = { state = ValueProperty (checkAndConvert regs v ty),
                        ty = ty,
                        attrs = { removable = removable,
                                  enumerable = enumerable,
                                  fixed = fixed,
                                  writable = writable } }
    in
```

```
        if hasProp propertyMap name
        then updateProp propertyMap name newProp
        else addProp propertyMap name newProp
    end
```

FIXME We must take into account the `[[CanPut]]` functionality from ES3!

2.5 Deleting a property

1 The `DeleteProperty` protocol is invoked to remove a property named by *name* from an object *obj*. The object may or may not have a property of that name when DeleteProperty is invoked. The flag *isStrict* is true if the ES4 code that caused DeleteProperty to be invoked was compiled in strict mode.

2 Specifically, there will be an AST node for the property deletion whose `strict` flag is set because it represents a source code phrase that was recognized in a region of code that was covered by a strict mode pragma.

FIXME Strict mode is not implemented in this code.

Semantics

3
```
and deletePropertyValue (regs:REGS)
                        (base:OBJECT)
                        (name:NAME)
    : VALUE =
    deletePropertyValueOrVirtual regs base name true

and deletePropertyValueOrVirtual (regs:REGS)
                                 (obj:OBJECT)
                                 (name:NAME)
                                 (doVirtual:bool)
    : VALUE =
    let
        val Object { propertyMap, tag, ... } = obj
        val existingProp = findProp propertyMap name
    in
        case existingProp of
            SOME { attrs = { fixed = true, ...}, ...}

            => newBoolean regs false

          | _
            => if
                doVirtual andalso
                Fixture.hasFixture (getFixtureMap regs obj) (PropName meta_delete)
            then
                 ((evalNamedMethodCall regs obj meta_delete
                                                [newString regs (#id name)])
                  handle ThrowException e =>
                         let
                             val ty = typeOfVal regs e
                             val defaultBehaviorClassTy =
                                 instanceType regs helper_DefaultBehaviorClass []
                         in
                             if ty <* defaultBehaviorClassTy then
                                 deletePropertyValueOrVirtual regs obj name false
                             else
                                 throwExn e
                         end
            else
                case existingProp of
                    SOME { attrs = { removable = true, ... }, ... }
                    => (delProp propertyMap name;
                        newBoolean regs true)

                  | _
                    => newBoolean regs false
    end
```

3 Types

FIXME Double-check that the specification and implementation of the subtype relation are consistent.

1 ECMAScript includes a gradual type system that supports optional type annotations on properties (e.g., on variables and fields). These type annotations are currently enforced dynamically, during evaluation.

2 Every value has an *allocated type*. The allocated type is the type given to a value when it is created and which defines its fixed structure.

3 Every property and fixture has a *storage type*. The storage type of a property or fixture is given by its declaration and constrains the set of values that can be stored in the property. The storage type of a property or fixture is also called it's *type constraint*.

4 The declarations of properties can carry type *annotations*, which define the storage type of the property. Annotation is denoted by following the annotated property name with a colon and a type expression. Annotations are not required: any property lacking an annotation is implicitly given the storage type `*`, meaning that the property can hold a value of any allocated type.

5 If a property holds a value, then that value must have an allocated type that is a *compatible subtype* of property's storage type. The compatible subtype relation is an extension of the traditional subtype relation that supports interoperation between typed and untyped code. The definition of the compatible subtype relation is included below.

6 For a given type **`T`**, a set of values is said to *populate* **`T`** if the values all have allocated types that are compatible subtypes of **`T`**. Some types are specified by specifying the values that populate them.

3.1 The Type Language

1 ES4 includes the following types:

3.1.1 The any type

1 The *any type* is the type populated by every possible value. In other words, every other type is a compatible subtype of the any type.

2 The any type is denoted in a type expression as `*`.

3 No value has the any type as its allocated type. The any type is meaningful only as the storage type of a property.

3.1.2 The null type

1 The *null type* is the type populated only by the semantic value `NullValue`.

2 The null type is denoted in a type expression as **`null`**.

3.1.3 The undefined type

1 The *undefined type* the type populated only by the semantic value `UndefinedValue`.

2 The undefined type is denoted in type expressions as **`undefined`**.

3.1.4 Nominal types

1 A *nominal type* is either a class type, an instance type or an interface type.

2 A *class type* and an *instance type* are both defined by a class definition.

3 An *interface type* is a type defined by an interface definition.

4 Nominal types are arranged in an explicit subtype relation through the use of **`extends`** and **`implements`** clauses in class and interface definitions.

5 An instance or interface type is denoted in type expressions by the name of the class or interface that defined the type, respectively.

6 An instance or interface type **`C`** (or **`C.<`**T_1**`, .., `**T_n**`>`) can be declared as a *non-null* type via any of the following declarations:

```
class C! ..
class C.<X1, .., Xn>! ..
interface C! ..
interface C.<X1, .., Xn>! ..
```

7 An instance or interface type is *nullable* if it is not a non-null type.

3.1.5 Record types

1 A *record type* is a subtype of the **public::Object** instance type that has additional type constraints on some specific set of named properties.

2 Record types are arranged implicitly into a subtype relation through structural comparison of their property constraints.

3 A record type is denoted in a type expression by listing the names of the specified properties in a comma separated list, with optional type annotations, enclosed in curly braces.

4 An example is **{x: Number, y: String}**, which denotes a record type with two properties **x** and **y**, the first constrained to type **Number** and the second to type **String**. The type **{ }** denotes the empty record type.

3.1.6 Array types

1 An *array type* is a subtype of the **public::Array** type that has type constraints on some prefix of the set of all possible unsigned-integer-indexed properties. An array type may be either *fixed-length* or *variable-length.*

2 Array types are arranged implicitly into a subtype relation through structural comparison of their property constraints.

3.1.6.1 Fixed-length array types

1 A *fixed-length array type* describes an explicit set of initial integer-indexed property constraints that must be satisfied by properties found at those indices.

2 A fixed-length array type is denoted in a type expression by listing the types of the specified properties in a comma-separated list enclosed in square brackets.

3 For example, the type **[Number, String]** describes fixed-length arrays of length at least 2, where the entry at index 0 has type **Number** and the entry at index 1 has type **String**.

4 The type **[ ]** describes fixed-length arrays of length at least 0, that is, it describes all fixed-length arrays.

FIXME Do we need to discuss holes here?

3.1.6.2 Variable-length array types

1 A *variable-length array type* describes an explicit set of initial integer-indexed property constraints and then a *final constraint* that is implied for any further integer-indexed properties (including zero further properties).

2 A variable length array type is denoted, initially, the same way a fixed-length array is, but concludes its type list with symbol **...** and a trailing type expression.

3 For example, the type **[Number, ... String]** describes arrays of length at least 1, where the entry at index 0 has type **Number**, and any remaining entries have type **String**. The type **[... Number]** describes arrays of zero or more elements, all of which must be of type **Number**.

3.1.7 Union types

1 A *union type* is a storage type that is populated by all values that populate all of the types that make up the union.

2 A union type is denoted in a type expresison by listing the types of the union members, separated by the vertical-bar character, enclosed in parentheses.

3 For example, the type **(Number | String)** denotes a type that is populated by both **Number** and **String** values. A property annotated with this type can therefore hold either instances of the **Number** type *or* instances of the **String** type.

4 No value has a union type as its allocated type. Union types are only meaningful as the storage types of properties.

3.1.8 Function types

1 A *function type* is a subtype of the **public::Function** type that describes additional type constraints on any function populating it.

2 A function type describes the number and type of required parameters, any optional parameters, any trailing "rest" parameter that accumulates excess arguments, and the return value.

3 Function types are denoted with the keyword **function**, followed by a parenthesis-enclosed, comma-separated list of parameter types -- optionally including default and rest symbols -- and an optional colon and trailing return type.

4 An example of a function type is:

```
function (Number, String) : String
```

5 This function type is populated by any function that is declared as taking a **Number** value and a **String** value as parameters, and returning a **String** value.

6 The return type of a function type can be omitted, in which case the return type is implicity the any type.

7 If a function should not return a value, the function return type can be annotated as **void**, which is a special notation for indicating the absence of a return type; there is no separate "void type" that can be denoted elsewhere.

8 A function type may include a type constraint for the **this** binding. Such a constraint must be listed as the first parameter in the function type parameter list, and must be denoted with the keyword **this** and a colon. For example, the function type

```
function(this : Number, String) : String
```

denotes a type of functions that require a **Number** value as their implicit **this** parameter, as well as taking a **String** argument and returning a **String**. The type constraint for the **this** binding defaults to the any type ***** if omitted.

9 A function type may denote the presence of default value assignments for some suffix of its parameter types by annotating the types of such parameters with trailing **=** symbols. For example, the function type

```
function(Number, String=) : String
```

denotes a type of function that takes a mandatory **Number** argument and an optional second **String** argument, and returns a **String**.

10 A function type may denote the presence of a trailing "rest-argument" with the symbol **...** in the final position of the function parameter list. This final parameter, if present, indicates that there is no maximum number of arguments to the function: additional arguments beyond the parameter list are collected into an array object and passed to the function. For example, the function type

```
function(String, ... ) : String
```

denotes a type of function that takes a **String** and any number of additional arguments (of any type), returning a **String**. Rest arguments cannot have type constraints.

11 Function types can optionally include a parameter name preceding each argument type, and separated from that type by a colon. These parameter names are for documentation purposes only. For example, the type of a `substring` function might be specified as:

```
function(str : String, start : double, end : double ) : String
```

3.1.9 Nullable types

1 A *nullable type* is an abbreviation for a union between some type and the null type.

2 A nullable type is denoted **?T** for some type **T**.

3 For example, the nullable type **?String** is an abbreviation for the union type **(String | null)**.

4 Nullable types are purely a syntactic convenience, and are not given further special treatment.

3.1.10 Non-null types

1 A *non-null type* is a type that excludes the **null** value from the population of a nullable instance or interface type.

2 A non-null type is denoted **!T** for some instance or interface type **T**.

3 For example, the non-null type **!String** is populated by instances of **public::String** but *excludes* null values.

3.1.11 Parametric types

1 A *parametric type* is a user-defined *type constructor* -- not a proper type -- associated with some type definition such as an instance type, interface type or type abbreviation. A parametric type takes some number of types as arguments and produces a new type as its result.

2 Parametric types are denoted by appending a type-parameter list to the name of a class, interface, or type at the site of its definition. A type parameter list consists of a single period, a less-than (or "left angle bracket") character, a comma-separated list of identifiers, and a greater-than (or "right angle-bracket") character.

3 For example, the class definition

```
class Vector.<X> { .. }
```

defines a class **Vector** that is parameterized over a single type variable **X**. This class definition itherefore also serves as a parametric instance type that can be used in type applications to form proper types.

3.1.12 Type applications

1 A *type application* is a combination of a parametric type with a set of *type arguments* that serve to *instantiate* the parametric type into a proper type that can be populated by values.

2 A type application is denoted by appending a type-argument list to the name of a parametric type. A type argument list consists of a single period, a less-than character, a comma-separated list of type expressions, and a greater-than character.

3 For example, the type application **Vector.<Number>** denotes an instance type that can be used as the allocated type of new objects.

3.1.13 Type names

1 A *type name* is a symbolic reference to an instance type, an interface type, a type abbreviation, or a type variable bound by a parameter in a parametric type.

2 A type name is denoted in a type expression by the same syntax as a name expression.

3 Type names are resolved during *type resolution*, described in Section 3.3 below.

3.2 Semantics of the Type Language

Semantics

1
```
and TYPE =
    AnyType
  | NullType
  | UndefinedType
  | RecordType of (NAME_EXPRESSION * TYPE) list
  | ArrayType  of (TYPE list * TYPE option)
  | UnionType of TYPE list
  | FunctionType of FUNCTION_TYPE
  | NonNullType of TYPE
  | AppType of (TYPE * TYPE list)
  | TypeName of (NAME_EXPRESSION * NONCE option)
  | ClassType of CLASS
  | InstanceType of CLASS
  | InterfaceType of INTERFACE

and FUNCTION_TYPE =
    { typeParams : IDENTIFIER list,
      thisType   : TYPE,
      params  : TYPE list,
      minArgs : int,
      hasRest : BOOLEAN,
      result  : TYPE option    (* NONE indicates return type is void *)
    }

type NONCE = int
```

To help avoid name collisions, each type variable bound in a type parameter list is assigned a unique integer, or *nonce*. Any reference to that type variable is then resolved into a `TypeName` that includes that nonce.

3.3 Type Resolution

1 At run-time, when a type **T** is encountered in the source program, that type is immediately *resolved*. This type resolution process proceeds as follows:

2 In the scope of a type definition

```
type X = S
```

any reference to a type variable **X** in **T** is replaced by the type **S**.

3 In the scope of a parametric type definition

type X.<y_1, .., y_n> = S

a type application **X.<S_1, .., S_n>** in **T** is replaced by the type **S[y_1:=S_1, .., y_n:=S_n]**.

4 In the scope of a class definition that associates a non-nullable instance type name **C** with a class definition D, type resolution replaces any TypeName that refers to **C** with InstanceType D. For references to a nullable instance type, the same replacement is made, but the result (or the enclosing AppType node, if there is one) is unioned with the **null** type.

5 Similarly, in the scope of an interface definition that associates an interface name **I** with an interface definition D, type resolution replaces any TypeName that refers to **I** with InterfaceType D (again, unioned with the **null** type, if **I** is a nullable interface).

3.3.1 Implementation of Type Resolution

1 The following function resolveTypeNames performs type resolution on a particular type ty in the context of an environment env.

2 This function relies on the auxiliary function Fixture.resolveNameExpr (described in section ...) to resolve each type name. The function Fixture.resolveNameExpr finds the corresponding fixture, and returns a triple containing (1) the environment that fixture was defined in; (2) the fully-resolved name for the given name expression, and (3) the corresponding fixture.

3 If the resulting fixture is for a non-parametric type definition, the body of that type definition is resolved in its environment, and then replaces the original type name.

4 If the resulting fixture is for a class or interface definition, the type name is replaced by an instance type or an interface type (unioned with the type **null** if the instance or interface type is declared as nullable).

5 A type application that refers to a type-parametric type definition is replaced by the body of that type definition, after the replacement of each formal parameter name with the corresponding resolved type argument.

6 A type application of a type name that refers to a type-parametric instance type or interface type is replaced by a type application that directly includes that instance or interface type (unioned with the type **null** if the nominal type is declared as nullable).

7 If none of the above cases apply, then resolveTypeNames uses the helper function mapType to perform type name resolution on each sub-term of the given type.

8 The function error reports error messages, and the module LogErr contains functions for converting various data structures into corresponding Strings.

FIXME Is the "LogErr." prefix too verbose on calls to error?

Semantics

9
```
fun resolveTypeNames (env : FIXTURE_MAPS)
                     (ty  : TYPE)
    : TYPE =
    let fun maybeUnionWithNull nonnullable ty =
            if nonnullable then
                ty
            else
                UnionType [ty, NullType]
        fun checkArgs typeArgs typeParams =
            if length typeArgs = length typeParams then
                ()
            else
                error ["Incorrect no of arguments to parametric typedefn"]
    in
    case ty of

        TypeName (nameExpr, _) =>
        let in
            case (Fixture.resolveNameExpr env nameExpr) of

                (envOfDefn, _,  TypeFixture ([], typeBody)) =>
                resolveTypeNames envOfDefn typeBody

              | (_, _, ClassFixture (c as Class {nonnullable, typeParams=[], ...})   ) =>
                maybeUnionWithNull nonnullable (InstanceType c)

              | (_, _,
                 InterfaceFixture (i as Interface {nonnullable, typeParams=[], ...})) =>
                maybeUnionWithNull nonnullable (InterfaceType i)
```

```
                | (_, n, _) => error ["name ", LogErr.name  n, " in type expression ",
                                    LogErr.ty ty, " is not a proper type"]
            end

          | AppType (TypeName (nameExpr, _), typeArgs) =>
            let in
                case Fixture.resolveNameExpr env nameExpr of
                    (envOfDefn, _,  TypeFixture (typeParams, typeBody)) =>
                    let in
                        checkArgs typeArgs typeParams;
                        resolveTypeNames envOfDefn
                                         (substTypes typeParams
                                                     (map (resolveTypeNames env)
                                                          typeArgs)
                                                     typeBody)
                    end

                  | (_, _, ClassFixture (c as Class {nonnullable, typeParams, ...})) =>
                    let in
                        checkArgs typeArgs typeParams;
                        maybeUnionWithNull nonnullable (AppType (InstanceType c, typeArgs))
                    end

                  | (_, _,
                     InterfaceFixture (i as Interface {nonnullable, typeParams, ...})) =>
                    let in
                        checkArgs typeArgs typeParams;
                        maybeUnionWithNull nonnullable (AppType (InterfaceType i, typeArgs))
                    end

                  | _ => mapType (resolveTypeNames env) ty
            end

          | _ => mapType (resolveTypeNames env) ty
        end

    fun mapType (f : TYPE -> TYPE)
                (ty: TYPE)
        : TYPE =
        case ty of
            RecordType fields =>
            RecordType (map (fn (name, ty) => (name, f ty)) fields)
          | UnionType types =>
            UnionType (map f types)
          | ArrayType (types, restType) =>
            ArrayType (map f types, Option.map f restType)
          | FunctionType { typeParams, params, result, thisType, hasRest, minArgs } =>
            FunctionType { typeParams = typeParams,
                           params = map f params,
                           result = Option.map f result,
                           thisType = f thisType,
                           hasRest = hasRest,
                           minArgs = minArgs }
          | NonNullType ty =>
            NonNullType (f ty)
          | AppType ( base, args ) =>
            AppType ( f base, map f args )
          | _ => ty
```

3.3.2 Resolved Types

1 A *resolved type* is one that is the result of the preceding type resolution process.

2 Resolved types do not include:

- type names that refer to instance or interface types (`InstanceType` and `InterfaceType` are used instead)
- type names that refer to type abbreviations (which are inlined)

3 Resolved types may include type names that refer to type parameters; these references include a nonce.

3.4 The Subtype and Type Equivalence Relations

3.4.1 The Subtype Relation

1 The *subtype relation* is a binary relation on types. It is defined by the collection of subtype rules described below and in the following subsections.

2 Subtyping is reflexive, so every type is a subtype of itself.

3 Subtyping is transitive, so if **S** is a subtype of **T** and **T** is in turn a subtype of **U**, then **S** is also a subtype of **U**.

3.4.2 Implementation of the Subtype Relation

1 The subtype relation is defined by the following function `subType`. This function takes an additional argument called `extra`, which is later used to extend the subtype relation with additional rules (for example, to define the compatible-subtyping relation below).

2 Reflexivity is included explicitly in the code below, whereas transitivity is a consequence of the remainder of the algorithm. This function dispatches to additional subtype functions described in the following subsections.

Semantics

3
```
fun subType (extra : TYPE -> TYPE -> bool)
            (type1 : TYPE)
            (type2 : TYPE)
    : bool =
    (type1 = type2)                      orelse
    (subTypeRecord   extra type1 type2) orelse
    (subTypeArray    extra type1 type2) orelse
    (subTypeUnion    extra type1 type2) orelse
    (subTypeFunction extra type1 type2) orelse
    (subTypeNominal  extra type1 type2) orelse
    (subTypeStructuralNominal extra type1 type2) orelse
    (extra type1 type2)
```

3.4.3 The Type Equivalence Relation

1 The type *equivalence*/ relation is also a binary relation on types. Two types are equivalent if and only if they are both subtypes of each other.

3.4.3.1 Implementation of the Type Equivalence Relation

The function `equivType` below checks type equivalence in a straightforward manner by checking subtyping in both directions. Like `subType`, `equivType` also takes an `extra` parameter.

IMPLEMENTATION NOTE The following implementation is straightforward and sufficies for a specification, but its worst-case time complexity is exponential in the height of a type, and so this naive approach would be inadequate in an implementation.

Semantics

1
```
and equivType (extra : TYPE -> TYPE -> bool)
              (type1 : TYPE)
              (type2 : TYPE)
    : bool =
      (subType extra type1 type2) andalso
      (subType (fn type1 => fn type2 => extra type2 type1)
               type2 type1)
```

3.4.4 Subtyping Record Types

1 A record type **{N_1:S_1, .., N_n:S_n}** (where each distinct **N_i** is a name and each **S_i** is a type) is a subtype of **{N_1:T_1, .., N_m:T_m}** if m ≤ n and **S_i** is equivalent to **T_i** for all i in 1..m.

2 The ordering of the **Name:Type** bindings in a record type is irrelevant, and so re-arranging these bindings yields an equivalent type. In particular, this re-arranging may be necessary in order to make the above rule applicable. The function `nameExpressionEqual` checks if two field names are equal.

Semantics

3
```
and subTypeRecord extra type1 type2 =
    case (type1, type2) of

        (RecordType fields1, RecordType fields2) =>
        List.all (fn ( name2, type2 ) =>
                     List.exists (fn ( name1, type1 ) =>
                                      nameExpressionEqual name2 name1 andalso
                                      equivType extra type2 type1)
                                 fields1)
                 fields2
```

```
      | _ => false

fun nameExpressionEqual (name1 : NAME_EXPRESSION)
                        (name2 : NAME_EXPRESSION)
    : bool
  = …
```

3.4.5 Subtyping Array Types

1 A fixed-length array type **[S1, .., Sn, S]** is a subtype of **[S1, .., Sn]**. The supertype demands one fewer element in the array than the subtype does. For example, **[Number, String, Boolean]** is a subtype of **[Number, String]**.

2 A fixed-length array type **[S1, .., Sn]** is a subtype of **[T1, .., Tn]** if each **Si** is equivalent to **Ti** for i in 1..n.

3 A variable-length array type **[S1, .., Sn, S, ... S]** is a subtype of **[S1, .., Sn, ... S]**. The supertype demands one fewer element in the array than the subtype does. For example, **[Number, String, Boolean, ... Function]** is a subtype of **[Number, ... Function]**, via transitivity.

NOTE Since **...** denotes concrete syntax, we use the *meta-syntax* **S1, .., Sn** to denote a sequence of zero-or-more comma-separated types.

4 A variable-length array type **[S1, .., Sn, ... S]** is a subtype of **[T1, .., Tn, ... T]** if **S** is equivalent to **T** and if each **Si** is equivalent to **Ti** for i in 1..n.

5 Via transitivity, the above rules may be applied multiple times, in various combinations. The following code combines all of these rules into a single deterministic algorithm for array subtyping.

Semantics

6
```
and subTypeArray extra type1 type2 =
    case (type1, type2) of

        (ArrayType (types1, rest1),
         ArrayType (types2, rest2))
        =>
        let
            val min = Int.min( length types1, length types2 )
        in
            ListPair.all (fn (type1, type2) => equivType extra type1 type2)
                         (List.take(types1, min),
                          List.take(types2, min))
            andalso
            (case (rest1, rest2) of
                 (NONE,    NONE   ) => length types1 >= length types2
               | (NONE,    SOME _ ) => false
               | (SOME _,  NONE   ) => false
               | (SOME t1, SOME t2) =>
                 length types1 >= length types2 andalso
                 equivType extra t1 t2 andalso
                 List.all (fn types1 => equivType extra type1 t2)
                          (List.drop(types1, length types2)))
        end

      | _ => false
```

3.4.6 Subtyping Union Types

1 A union type **(S1 | .. | Sn)** is a subtype of a type **T** if **Si** is a subtype of **T** for all i in 1..n.

2 A type **S** is a subtype of **(T1 | .. | Tn)** if there exists some i in 1..n such that **S** is a subtype of **Ti**.

Semantics

3
```
and subTypeUnion extra type1 type2 =
    case (type1, type2) of

        (UnionType types1, type2)
        => List.all    (fn type1 => subType extra type1 type2) types1

      | (type1, UnionType types2)
        => List.exists (fn type2 => subType extra type1 type2) types2
```

```
    | _ => false
```

3.4.7 Subtyping Function Types

1 A function type **function(S$_1$, .., S$_n$) : U** is a subtype of **function(T$_1$, .., T$_n$) : R** if **U** is a subtype of **R** and **S$_i$** is equivalent to **T$_i$** for all i in 1..m.

NOTE Function subtyping is invariant in the argument position, and covariant in the result type.

2 This rule generalizes to **this** arguments, default arguments, and rest arguments according to the following rule, where the number of default arguments (indicated via the **=** symbol) in each function type may be zero, and where **[...]** indicates an optional rest argument. A function type

function(this:S$_1$, S$_2$, .., S$_n$, S$_{n+1}$=, .., S$_m$=, [...]) : U

is a subtype of

function(this:T$_1$, T$_2$, .., T$_p$, T$_{p+1}$=, .., T$_q$=, [...]) : R

if **U** is a subtype of **R** and n ≤ p and **S$_i$** is equivalent to **T$_i$** for all i in 1..min(q,m). In addition:

- If neither function type has a rest argument, then we require that q ≤ m.
- If only the first function type has a rest argument, then no additional conditions are needed.
- If only the second function type has a rest argument, then subtyping does not hold.
- If both function types have a rest argument, then **S$_i$** must be equivalent to the any type ***** for all i in (q+1)..m.

3 For type-parametric functions, alpha-renaming of the type variable preserves the meaning of types. Moreover,

function.<X$_1$,..,X$_n$> (argtypes1) : R1

is a subtype of

function.<X$_1$,..,X$_n$> (argtypes2) : R2

if and only if

function(argtypes1) : R1

is a subtype of

function(argtypes2) : R2

Hence, to check subtyping between type-parametric functions, we alpha-rename the type variables to be identical in both types, and then proceed to check subtyping on the non-type-parametric versions of the two function types.

4 The types in a subtype relation may contain free type variables, which are assumed to denote the same unknown type in both arguments to the subtype relation. For example, within the scope of a binding for a type variable **X**, the type **[X, ... X]** is a subtype of the type **[... X]**.

Semantics

5
```
and subTypeFunction extra type1 type2 =
    case (type1, type2) of

        (FunctionType
             { typeParams = typeParams1,  params     = params1,
               result     = result1,      thisType   = thisType1,
               hasRest    = hasRest1,     minArgs    = minArgs1 },
         FunctionType
             { typeParams = typeParams2,  params     = params2,
               result     = result2,      thisType   = thisType2,
               hasRest    = hasRest2,     minArgs    = minArgs2 })
        =>
        (* set up a substitution to alpha-rename typeParams to be identical *)
        let
            val subst = rename typeParams1 typeParams2
            val min = Int.min( length params1, length params2 )
        in
            length typeParams1 = length typeParams2
          andalso
            (case (result1, result2) of
```

```
                (SOME type1, SOME type2) => subType extra type1 (subst type2)
              | (NONE,       NONE)       => true)
        andalso
          equivType extra thisType1 (subst thisType2)
        andalso
          minArgs1 <= minArgs2
        andalso
          ListPair.all (fn (type1, type2) => equivType extra type1 (subst type2))
                       (List.take(params1, min),
                        List.take(params2, min))
        andalso
          (case (hasRest1, hasRest2) of
                (false, false) => length params2 <= length params1
              | (true,  false) => true
              | (false, true ) => false
              | (true,  true ) =>
                    List.all (fn t => equivType extra t AnyType)
                             (List.drop(params1, min)))
    end

  | _ => false
```

6 The following function `rename` performs the capture-free substitution of references to any of the identifiers in `typeParams1` with references to the corresponding identifier in `typeParams2` in the type `ty`.

Semantics

7
```
fun rename (typeParams1 : IDENTIFIER list)
           (typeParams2 : IDENTIFIER list)
           (ty : TYPE)
  :  TYPE
= …
```

3.4.8 Subtyping Nominal Types

1 Given a class definition

class C extends D implements I_1, .., I_n { ... }

the instance type **C** is a subtype of instance type **D**, and instance type **C** is also a subtype of interface type $\mathbf{I_j}$ for j in 1..n.

2 Given an interface definition

interface K extends I_1, .., I_n { ... }

the type **K** is a subtype of $\mathbf{I_j}$ for j in 1..m.

3 These rules generalize to applications of type-parametric instance and interface types via appropriate renaming of bound variables. For example, given a type-parametric interface type defined by

class C.<x_1, .., x_n> extends D.<T_1, .., T_m> { ... }

we have that **C.<S_1, .., S_n>** is a subtype of

D.<T_1[x_1:=S_1,..,x_n:=S_n], .., T_m[x_1:=S_1,..,x_n:=S_n]>

4 Also, **C.<T_1, .., T_n>** is a subtype of **C.<S_1, .., S_n>** if each type $\mathbf{T_i}$ is equivalent to the corresponding type $\mathbf{S_i}$ for i in 1..n.

NOTE The notation **T[x_1:=S_1,..,x_n:=S_n]** denotes the type **T** with each occurrence of the type variable $\mathbf{x_i}$ replaced (in a capture-free manner) by the corresponding type $\mathbf{S_i}$.

NOTE The above rules also apply if **C** is declared as a non-nullable instance type.

NOTE There is a distinction between the type name **C** and the instance type to which it refers, in that the type name **C** includes the type **null** if **C** is a nullable type, whereas the instance type **C** describes only class instances.

Semantics

5
```
and subTypeNominal extra type1 type2 =
    case (type1, type2) of

        ( InstanceType (Class { typeParams = [], extends, implements, ...}), _ )
        => (case extends of
```

```
                NONE => false
              | SOME extends => subType extra extends type2)
          orelse
          List.exists
              (fn iface => subType extra iface type2)
              implements

      | ( AppType
            (InstanceType (Class { typeParams, extends, implements, ...}),
             typeArgs),
        _ )
        => (case extends of
                NONE => false
              | SOME extends => subType extra
                                        (substTypes typeParams typeArgs extends)
                                        type2)
          orelse
          List.exists
              (fn iface => subType extra
                                   (substTypes typeParams typeArgs iface)
                                   type2)
              implements

      | ( InterfaceType (Interface { typeParams = [], extends, ...}), _ )
        => List.exists
              (fn iface => subType extra iface type2)
              extends

      | ( AppType
            (InterfaceType (Interface { typeParams, extends, ...}),
             typeArgs),
        _ )
        => List.exists
              (fn iface => subType extra
                                   (substTypes typeParams typeArgs iface)
                                   type2)
              extends

      | ( AppType (typeConstructor1, typeArgs1),
          AppType (typeConstructor2, typeArgs2) )
        =>
        typeConstructor1 = typeConstructor2 andalso
        length typeArgs1 = length typeArgs2 andalso
        ListPair.all
            (fn (type1, type2) => equivType extra type1 type2)
            (typeArgs1, typeArgs2)

      | _ => false
```

The following function `substTypes` performs the capture-free replacement of all occurrences of `typeParams` by `typeArgs` within the type `ty`.

```
fun substTypes (typeParams : IDENTIFIER list)
               (typeArgs   : TYPE list)
               (ty         : TYPE)
    : TYPE
  = …
```

3.4.9 Relating Structural and Nominal Types

1 A record type **{N_1:S_1, .., N_n:S_n}** is a subtype of the instance type **public::Object**.

2 An array type **[S_1, .., S_n]** is a subtype of the instance type **public::Array**, which is a subtype of the instance type **public::Object**.

3 Any function type is a subtype of the instance type **public::Function**, which is a subtype of the instance type **public::Object**.

Semantics

4
```
and subTypeStructuralNominal extra type1 type2 =
    case (type1, type2) of

        (RecordType _,  InstanceType (Class { name, ... }))
        => nameEq name Name.public_Object

      | (ArrayType _, InstanceType (Class { name, ... }))
```

```
      => nameEq name Name.public_Array orelse
         nameEq name Name.public_Object

    | (FunctionType _, InstanceType (Class { name, ... }))
      => nameEq name Name.public_Function orelse
         nameEq name Name.public_Object

    | _ => false
```

3.5 Compatible Types

1 The *compatibility* relation is a binary relation on type values. A type **S** is compatible with a type **T** if **T** can be obtained from **S** by replacing certain portions of **S** by the any type *****.

2 For example, the record type **{x : double}** is compatible with both **{x : *}** and with *****, but the type **{x : *}** is not compatible with **{x : double}**.

3 Also, **T.<Number>** is compatible with **T.<*>**.

4 This compatibility relation is reflexive and transitive, but not symmetric.

3.6 Compatible-Subtyping

1 The *compatible-subtype* relation is a binary relation on types. 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**.

2 For example, the record type **{x : double, y : boolean}** is a compatible-subtype of the types **{x : *, y : *}**, **{x : double}**, **{x : *}**, and *****.

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

4 The compatible-subtyping relation is implemented by calling the previously-defined `subType` predicate and passing in an `extra` parameter that implements the compatibility relation, that every type is compatible with *****.

Semantics

5
```
fun compatibleSubtype (type1 : TYPE) (type2 : TYPE) : bool =
    subType
        (fn type1 => fn type2 => type2 = AnyType)
        type1 type2
```

3.7 Type Invariants at Run Time

1 A type is *allocatable* if it is not the any type or a union type.

2 Every value in ES has an associated *allocated type*, which is a type that is associated with the value when the value is first allocated or created. An allocated type is always an allocatable 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.

3 If a property of storage type **T** hold a value **v** of type **S**, then **S** is a compatible-subtype of **T**.

4 Names

1 Names in ECMAScript are defined in section ...names.

2 Names are used to identify properties within property maps and fixtures within fixture maps.

3 A name is calculated from a name expression found in ECMAScript source code.

4.1 Name Expressions

1 A *name expresison* is either qualified or unqualified.

2 A *qualified name expression* consists of a namespace expression and an identifier. The former is either a literal namespace value (resulting from using a string as a namespace qualifier) or else a further name expression identifying a namespace fixture in the lexical environment. Examples of a qualified names are **intrinsic::subtring** or **"org.w3.dom"::DOMNode**

3 An *unqualified name expression* consists of an identifier and a list of sets of open namespaces, determined by context. An example of an unqualified name is **encodeURI**.

4 An unqualified name is subject to *name resolution*, and must resolve to a unique qualified name. The algorithm for name resolution is presented in subsequent sections, and differs depending on the context the unqualified name occurs within.

5 Names that are used in contexts denoting types and namespaces must be resolved statically at definition time. Names that denote other properties may be resolved repeatedly at evaluation time.

NOTE In a qualified name such as **intrinsic::substring** the leftmost identifier, **intrinsic**, is itself unqualified and subject to definition-time resolution.

Semantics

6
```
and NAME_EXPRESSION =
    QualifiedName of { namespace: NAMESPACE_EXPRESSION,
                       identifier: IDENTIFIER }
  | UnqualifiedName of { identifier: IDENTIFIER,
                         openNamespaces: OPEN_NAMESPACES }

and NAMESPACE_EXPRESSION =
    Namespace of NAMESPACE
  | NamespaceName of NAME_EXPRESSION
```

4.1.1 Open namespaces list

1 The *open namespaces list* of an unqualified name expression is a list of sets of namespaces open at the point of the program where the name expression occurs, and is ordered by priority, with sets of namespaces earlier in the list taking priority over sets later in the list. The list reflects the nesting of lexical scopes, with the namespaces opened in the "innermost" lexical scope held in the first set of namespaces in the list, and subsequent sets holding namespaces opened in enclosing lexical scopes.

Semantics

2
```
type NAMESPACE_SET = …

type OPEN_NAMESPACES = NAMESPACE_SET list
```

4.1.2 Special namespaces

1 Several namespaces are assigned special meaning, and are generated by an ECMAScript implementation in specific contexts.

2 These namespaces are bound to predefined names and implicitly opened in their associated scopes. In the following sections, when a namespace is said to be *implicitly opened* in a given lexical scope, the specified meaning is that a new namespace set is added to the front of the open namespaces list for the duration of the scope containing the implicitly opened namespaces.

4.1.2.1 Public and 4th Edition namespaces

1 The public and 4th Edition namespaces are defined in section Standard Namespaces of Values, and are defined identically in all programs and lexical scopes.

2 The 4th Edition namespace is bound to the global property name **""::__ES4__** (that is, the name formed by qualifying the identifier **__ES4__** with the public namespace) and can therefore be seen by code loaded in either 3rd Edition or 4th Edition mode.

3 The public namespace is bound to the global property name **__ES4__::public** (that is, the name formed by qualifying the identifier **public** with the 4th Edition namespace).

4 When a program is loaded in 3rd Edition or 4th Edition mode, the public namespace is implicitly opened.

5 When a program is loaded in 4th Edition mode, after the public namespace is implicitly opened, the 4th edition namespace is implicitly opened. The 4th Edition namespace is therefore opened at a higher priority than the public namespace.

NOTE The **public** namespace is distinguished in several ways. The names of properties added dynamically to objects are qualified by **public** by default, so all properties created by 3rd Edition code running on a 4th Edition implementation are **public**, and **public** is sometimes called "the compatibility namespace" for that reason. The default namespace qualifier that is applied to declarations in every scope is **public**, so absent other qualification every property on every object and every lexically bound name is in the **public** namespace.

4.1.2.2 Internal namespaces

1 Each program (compilation unit) has a new implementation-generated opaque namespace implicitly defined as its *internal namespace* at the start of the definition phase.

2 The internal namespace for a program is bound to the name **internal** in the global fixture map for the duration of definition and evaluation. The binding to **internal** is removed after definition and evaluation of a program, and is re-bound to new internal namespaces for any subsequent programs loaded.

3 When a program is loaded in 4th Edition mode, after the 4th Edition namespace is implicitly opened, the program's internal namespace is implicitly opened. The internal namespace is therefore opened at a higher priority than the 4th Edition namespace.

NOTE An internal namespace can be used to qualify definitions that are not intended to be visible to any other program.

4.1.2.3 Private and protected namespaces

1 Each class definition has two new implementation-generated opaque namespaces implicitly defined as its *private namespace* and *protected namespace*.

2 The private and protected namespaces for a class are bound to the names **private** and **protected**, respectively, within the lexical scope of the class definition they are associated with.

3 The private and protected namespaces for a class are implicitly opened within the lexical scope of the class.

4 The protected namespace for a class `C` is also implicitly opened within the lexical scope of *every class that extends* `C`.

4.2 Reference Expressions

1 A *reference expression* provides context for resolving a name expression to a name, and identifying a particular fixture or property to which the name refers. A reference expression is either a lexical reference, an object name reference, or an object index reference.

2 A *lexical reference* is a reference expression that resolves to a name within a lexical scope, and therefore a property or fixture stored in a scope object. Some lexical references are required to be resolved to fixtures statically during program definition, while others may be resolved dynamically during program evaluation. Examples of lexical references are **encodeURI** or **public::Function**.

3 An *object name reference* is formed by conjoining an object expression and a name expression with a period ("."). A name expression in an object name reference resolves to the name of a fixture or property on the provided object, or a fixture or property on the object's prototype chain. Some object references may be resolved to fixtures statically, but the specified behavior of object references is as if they are always resolved dynamically during program evaluation. Examples of object references are **s.length** or **s.intrinsic::substring**, where **s** is the name of an object.

4 An *object index reference* is similar to an object name reference, in that it combines an expression for a name with an object and resolves the calculated name against the provided object. An object index reference differs from an object name reference by the fact that there is no proper name expression inside it: rather an object expression is conjoined with a general ECMAScript expression, enclosed within square brackets, and determining the name to resolve may require arbitrary evaluation of the bracketed expression. An example of an object index reference is **s[f()]**, where **s** is the name of an object, and the name to be resolved against **s** is *calculated dynamically* by evaluating the function expression **f()**. Object index expressions can therefore never be resolved statically.

Semantics

5
```
datatype EXPRESSION =
         LexicalReference of { name: NAME_EXPRESSION }
       | ObjectNameReference of { object: EXPRESSION,
                                  name: NAME_EXPRESSION }
       | ObjectIndexReference of { object: EXPRESSION,
                                   index: EXPRESSION }
       …
```

NOTE An `ObjectIndexReference` is evaluated by evaluating its `index` operand to a `Name` object and then treating that value the same as a resolved qualified name. Index operands that do not evaluate to `Name` objects are converted to `string`, and a `Name` object is formed from the string and the **public** namespace.

4.3 Lexical scopes

1 Lexical scopes are defined in section ...scopes.

2 Defining and binding forms introduce names into a lexical scope. These names are then visible to lexical references that occur within the scope of the binding. The scope of a binding is primarily determined by the textual boundaries of the scope (ECMAScript is primarily *lexically scoped*) and depends also on the defining or binding form that introduced the binding.

NOTE For example, the scope of a **var** binding inside a block statement is the entire body of the function or program containing the block, whereas the scope of a **let** binding inside a block statement is that block statement.

3 Scopes nest textually, and a name that is bound in one scope may be *shadowed* in an inner scope by a binding of the same name in the inner scope; name expressions in the inner scope will not be able to access the outer binding.

4 In this specification, the nesting of scopes is modelled as a list of fixture maps in the definition phase and a list of objects during evaluation. The former list is called the *static scope chain* or the *static environment*. The latter list is called the

dynamic scope chain or the *dynamic environment*. Both lists are generically referred to as *scope chains*, with the distinction between the static and dynamic environments indicated where not otherwise clear from context.

5 Each scope holds a fixture map of the named fixtures defined in that scope.

6 The fixture maps in the static environment are arranged into a simple list.

Semantics

7
```
and FIXTURE_MAPS = ((FIXTURE_NAME * FIXTURE) list) list
```

8 The fixture maps in the dynamic environment are arranged into accompanying objects, each with a corresponding property map in which values may be stored as properties.

Semantics

9
```
and SCOPE =
    Scope of { object: OBJECT,
               parent: SCOPE option,
               temps: TEMPS,
               kind: SCOPE_KIND }

and SCOPE_KIND =
    WithScope
  | GlobalScope
  | InstanceScope of CLASS
  | ClassScope
  | ActivationScope
  | BlockScope
  | TypeArgScope
  | EvalScope
```

10 At each point in the program, both during definition and evaluation, exactly one scope chain is in effect. This scope chain is called *the scope chain* or *the environment* containing an expression, statement or definition.

11 Some objects that appear on evaluation-time scope chains are dynamically extensible. For example, class objects appear on the scope chain of class and instance methods, and properties can be added to and removed from class objects; however, these properties are not visible to lexical references within the class.

4.3.1 Prototype chain

1 Every object has a distinguished value called its *prototype* (see section Object prototype in Values).

2 If the prototype value of an object is another object, then the prototype value is called the object's *prototype object*, and the connection between the initial object and its prototype object is called its *prototype link*.

3 The *prototype chain* is the list of objects formed by following prototype links from an object. The prototype chain of an object begins with the object itself, and ends with the first object having a null prototype value.

4 When a name is to be resolved against an object, if resolution initially fails because the object does not contain a property matching the name, then resolution continues along the object's prototype chain.

4.4 Name Resolution

4.4.1 Overview

1 The purpose of name resolution is to take an unresolved name and a list of objects and return an unambiguous name (consisting of a namespace value and an identifier) and an object that contains a property with that name. The objects are searched in order, and the first object to contain a property with the name is selected.

2 There are two complications. The first appears with the need for disambiguation. When an unqualified name is resolved the resolution is performed in the context of the namespaces that were open at the point of reference. Thus the search of any one object may find multiple bindings that match the name, up to one binding per open namespace. Instead of making this an error, the name resolver disambiguates by trying to select the most desirable of those namespaces. Selection is performed by filtering the applicable namespaces until we are left with one. (If we have more than one then the name is deemed ambiguous.)

3 We first select those namespaces among the matching namespaces that are in use by the least specific class of the object that contains the name. For example, if **`C`** is a subclass of **`B`** and **`B`** is a subclass of **`A`**, and our name **`n`** matched **`ns1::n`**, **`ns2::n`**, and **`ns3::n`**, and **`ns1::n`** and **`ns2::n`** were defined in **`B`** and **`ns3::n`** was defined in **`C`**, then we'd be left with just **`ns1`** and **`ns2`**.

4 (The motivation for using the order in which names are introduced in the class hierarchy is to guarantee that the meaning of valid references to object properties doesn't change. In other words, if **o.x** is ever valid, then it shall always refer to the same property **x** as long as the type of **o** doesn't change.)

5 We then filter by namespace priority. The open namespaces are organized in a prioritized list of namespace sets. If one of the matching names has a namespace that is from a set with a higher priority than all the other matching names, then that's the namespace we want. So if the referencing context of **n** opened **ns2** in a scope nested inside the one that opened **ns1**, then we are left with just **ns2** -- and a single binding, **ns2::n**.

6 (The motivation for disambiguation by the scope in which a namespace is opened, is simple: it allows more programs to run. Furthermore, since the priority of namespaces during disambiguation is under the control of the programmer, the programmer can rely on disambiguation to control which names are found.)

7 The second complication is that some names are required to be resolved successfully at definition time -- names that denote namespaces and types. (We require definition-time resolution in order to make names and types constant, which generally simplifies the language and makes programs more easily comprehensible.) The consequence is that namespace and type references are illegal inside scopes introduced by **with** or scopes that may be extended by the **eval** operator, because those scopes make definition time resolution impossible -- their contents are unknown. Such programs result in a syntax error being signalled. (It is possible to ease that restriction in various ways but we have not done so.)

8 However, we also require that type and namespace names that are resolved at definition time must resolve to the same bindings that they would resolve to if they were to be resolved at evaluation time. (We require that because it simplifies the user's model of the language: equal names in the same scope have the same meaning, provided they resolve at all.) The consequence is that the language must provide protection against ambiguities that can be introduced at a later time. If a name is resolved at definition time to a global binding then compilation units loaded later may introduce new global bindings that will make the resolved binding ambiguous. For example, consider the following program.

```
namespace NS1
namespace NS2
NS1 type T
use namespace NS1, namespace NS2

   ... var x: T
```

9 The reference to **T** in the type annotation is resolved uniquely at definition time to **NS1::T**. Then another compilation unit is loaded:

```
NS2 type T = ...
```

10 Since the global environment is "flat"--code in earlier compilation units can see bindings introduced by later compilation units--the reference to **T** from the first program is now ambiguous.

11 ES4 protects against this eventuality by *reserving* global names that are resolved at definition time. When **T** is resolved in the first program and found to be in **NS1**, the name **NS2::T** is reserved: it is made off-limits to later programs. As a consequence, the second program above would not be loaded, because the introduction of **NS2::T** would be an error.

12 Names are reserved in namespaces at the same or higher priority level as the namespace that the name was resolved to, so in the example above neither **public::T** nor **internal::T** would become reserved, as those namespaces are at lower priority levels than **NS1** and **NS2**.

NOTE Top-level "use namespace" pragmas are given a higher priority level than names originating "outside" the compilation unit, as is the case for **public** and **internal**.

4.4.2 Definition-Time Resolution of Namespace and Type Expressions

1 The definition time scope chain is modelled as a list of fixture maps, defined elsewhere. A fixture map maps names to fixture bindings that result from defining and binding forms (**var**, **function**, **type**, **class**, **interface**, **namespace**, and others). Fixture maps have no dynamic properties.

2 Definition time resolution resolves name expressions that denote namespaces and types, and performs reservation of global names if necessary.

3 The following algorithm resolves a name expression to a specific name and fixture in the list of fixture maps.

Semantics

4
```
and resolveNameExpr (fixtureMaps : Ast.FIXTURE_MAPS)
                    (ne   : Ast.NAME_EXPRESSION)
    : (Ast.FIXTURE_MAPS * Ast.NAME * Ast.FIXTURE) =

    case ne of
        Ast.QualifiedName { namespace, identifier }
        => resolveQualifiedName fixtureMaps identifier namespace
```

```
      | Ast.UnqualifiedName { identifier, openNamespaces, ... }
        => case (resolveUnqualifiedName fixtureMaps identifier openNamespaces) of

            NONE
            => error ["unresolved name ", LogErr.nameExpr ne]

          | SOME ([], _)
            => error ["unresolved name ", LogErr.nameExpr ne]

          | SOME ([fixtureMap], name)
            => ( reserveNames name openNamespaces ;
                 ([fixtureMap], name, getFixture fixtureMap (Ast.PropName name)) )

          | SOME (fixtureMaps, name)
            => (fixtureMaps, name, getFixture (hd fixtureMaps) (Ast.PropName name))
```

4.4.2.1 Qualified Name Expressions

1 A qualified name expression is resolved by resolving the namespace part and then returning the tail of the list of fixture maps such that the first fixture map on the tail contains a binding for the name.

NOTE The name can't be ambiguous because there is only one namespace.

Semantics

2
```
fun resolveQualifiedName (fixtureMaps          : Ast.FIXTURE_MAPS)
                         (identifier    : IDENTIFIER)
                         (namespaceExpr : Ast.NAMESPACE_EXPRESSION)
    : (Ast.FIXTURE_MAPS * NAME * Ast.FIXTURE) =
    let
        val ns = resolveNamespaceExpr fixtureMaps namespaceExpr
        val name = { ns = ns, id = identifier }
        fun search (r::rs) = if hasFixture r (Ast.PropName name) then
                                 (r::rs)
                             else
                                 search rs
          | search [] = []
    in
        case (search fixtureMaps) of
            []
            => error ["qualified name not present in fixtureMaps: ", LogErr.name name]

          | fixtureMaps'
            => (fixtureMaps', name, getFixture (hd fixtureMaps') (Ast.PropName name))
    end
```

4.4.2.2 Unqualified Name Expressions

1 An unqualified name expression is resolved according to the full algorithm outlined above. It returns the tail of the list of fixture maps such that the first fixture map on the tail contains an unambiguous binding for the name.

Semantics

2
```
and resolveUnqualifiedName (fixtureMaps           : Ast.FIXTURE_MAPS)
                           (identifier     : IDENTIFIER)
                           (openNamespaces : OPEN_NAMESPACES)
    : (Ast.FIXTURE_MAPS * NAME) option =
    let
        val namespaces = List.concat (openNamespaces)
        val matches = fixtureMapListSearch (fixtureMaps, namespaces, identifier)
    in
        case matches of
            NONE
            => NONE

          | SOME (fixtureMaps, [namespace])
            => SOME (fixtureMaps, {ns=namespace, id=identifier})

          | SOME (fixtureMaps, namespaces)
            => case selectNamespaces (identifier,
                                      namespaces,
                                      [],
                                      openNamespaces) of

                 [namespace]
                 => SOME (fixtureMaps, {ns=namespace, id=identifier})

               | ns::nss
```

```
                    => error ["ambiguous reference: ", Ustring.toAscii identifier]
    end

fun fixtureMapListSearch ([], _, _) = NONE

  | fixtureMapListSearch (fixtureMaps       : Ast.FIXTURE_MAPS,
                          namespaces : NAMESPACE_SET,
                          identifier : IDENTIFIER)
    : (Ast.FIXTURE_MAPS * NAMESPACE_SET) option =
    case fixtureMapSearch (hd fixtureMaps, namespaces, identifier) of
        NONE
        => fixtureMapListSearch (tl fixtureMaps, namespaces, identifier)

      | SOME (_, m)
        => SOME (fixtureMaps, m)

fun fixtureMapSearch (fixtureMap        : Ast.FIXTURE_MAP,
                      namespaces : NAMESPACE_SET,
                      identifier : IDENTIFIER)
    : (Ast.FIXTURE_MAP * NAMESPACE_SET) option =
    case List.filter (fn ns =>
                          hasFixture fixtureMap (Ast.PropName {ns=ns, id=identifier}))
                     namespaces of
        [] => NONE
      | m  => SOME (fixtureMap, m)
```

4.4.2.3 Reserving Names

1 Statically resolved names must keep their meaning at runtime and therefore cannot be shadowed or be made ambiguous by the later introduction of names. Therefore we reserve the set of names that would cause such conflicts at runtime.

2 Given a name and a list of sets of open namespaces, the following algorithm computes a set of names consisting of the identifier and each of the open namespaces with an equal or higher priority than the given namespace.

FIXME Obviously we need more prose here. Also we want to be sure to note that reservation only happens in the global object.

Semantics

3
```
and reserveNames (name)
                 (openNamespaces)
    = …
```

4.4.3 Evaluation-time Resolution of Lexical References

1 The evaluation time scope chain is modelled as a list of arbitrary objects. A scope object maps names to properties (both fixtures and dynamic properties). Apart from scope objects introduced by the **with** statement, the evaluation time scope chain mirrors the definition time scope chain.

2 The following algorithm resolves a name expression to an object and the name of a property on that object.

Semantics

3
```
and resolveLexicalReference (regs            : REGS)
                            (nameExpression  : NAME_EXPRESSION)
                            (errorIfNotFound : bool)
    : (OBJECT * NAME) =
    let
        val {scope, ...} = regs
    in
        case nameExpression of

            QualifiedName {identifier, namespace}
            => resolveQualifiedLexicalReference regs identifier namespace

          | UnqualifiedName { identifier, openNamespaces, ... }
            => resolveUnqualifiedLexicalReference regs identifier openNamespaces
    end
```

4.4.3.1 Qualified Lexical References

1 To resolve a qualified lexical reference we evaluate its namespace expression (it must yield a namespace value) and then look up the name comprised of the namespace value and the qualified reference's identifier. If a binding is not found then we return the global object, otherwise the object that contained the binding for the name.

Semantics

2

```
and resolveQualifiedLexicalReference (regs          : REGS)
                                     (identifier    : IDENTIFIER)
                                     (namespaceExpr : NAMESPACE_EXPRESSION)
    : (OBJECT * NAME) =
    let
        val {scope, global, ...} = regs
        val namespace = evalNamespaceExpr regs namespaceExpr
        val result = searchScopeChain (regs, SOME scope, identifier, [namespace])
    in
        case result of
            NONE
            => (global, {ns=publicNS, id=identifier})

          | SOME (object, namespaces)
            => (object, {ns=namespace, id=identifier})
    end
```

4.4.3.2 Unqualified Lexical References

1 To resolve an unqualified lexical reference we make use of the full algorithm outlined above, finding the first object that maches the unqualified name in all open namespaces and then disambiguating the set of resulting namespaces.

Semantics

2
```
and resolveUnqualifiedLexicalReference (regs           : REGS)
                                       (identifier     : IDENTIFIER)
                                       (openNamespaces : OPEN_NAMESPACES)
    : (OBJECT * NAME) =
    let
        val {scope, global, ...} = regs
        val namespaces = List.concat openNamespaces
        val result = searchScopeChain (regs, SOME scope, identifier, namespaces)
    in
        case result of
            NONE
            => (global, {ns=publicNS, id=identifier})

          | SOME (object, namespaces)
            => let
                   val classFixtureMaps = [getFixtureMap regs object]
                   val result = Fixture.selectNamespaces (identifier,
                                                          namespaces,
                                                          classFixtureMaps,
                                                          openNamespaces)
               in
                   case result of
                       [namespace]
                       => (object, {ns=namespace, id=identifier})

                     | _
                       => error regs ["ambiguous reference"]
               end
    end
```

4.4.3.3 Resolve on a Scope Chain

1 To find an object matching an identifier and a set of namespaces in a scope chain. [CHANGE] remove second lookup pass

Semantics

2
```
and searchScopeChain (regs, NONE, _, _) = NONE

  | searchScopeChain (regs       : REGS,
                      SOME scope : SCOPE option,
                      identifier : IDENTIFIER,
                      namespaces : NAMESPACE_SET)
    : (OBJECT * NAMESPACE_SET) option =
    let
        val matches = searchScope (regs, scope, namespaces, identifier)
        val Scope { parent, ... } = scope
    in
        case matches of
            NONE
            => searchScopeChain (regs, parent, identifier, namespaces)

          | _
            => matches
    end
```

```
fun searchScope (regs       : REGS,
                 scope      : SCOPE,
                 namespaces : NAMESPACE_SET,
                 identifier : IDENTIFIER)
    : (OBJECT * NAMESPACE_SET) option =
    let
        val (object, kind) = getScopeObjectAndKind (scope)
    in
        case kind of
            (WithScope | EvalScope | GlobalScope)  (* FIXME EvalScope is unused *)
            => searchObject (regs, SOME object, NONE, identifier, namespaces, false)

          | (InstanceScope class)
            => searchObject (regs, SOME object, SOME class, identifier, namespaces, true)

          | _
            => searchObject (regs, SOME object, NONE, identifier, namespaces, true)
    end
```

4.4.4 Evaluation-Time Resolution of Object References

1 Object references are resolved along the prototype chain of the object. Both fixed and dynamic properties are searched in each object, in a single pass over the prototype chain.

2 `ObjectIndexReference` expressions represent computed lookup. The index expression is computed; if it evaluates to a **Name** object then it is used as is, otherwise the value is converted to **string** and qualified with the **public** namespace.

FIXME The following algorithm does not yet handle **Name** objects.

Semantics

3
```
and resolveObjectReference (regs:REGS)
                           (ObjectNameReference { object, name, ... }: EXPRESSION)
    : (OBJECT option * (OBJECT * NAME)) =
    let
        val obj = evalObjectExpr regs object
    in
        case name of
            UnqualifiedName { identifier, openNamespaces, ... }
            => (SOME obj, resolveUnqualifiedObjectReference regs obj identifier
                                                            openNamespaces)

          | QualifiedName { namespace, identifier }
            => (SOME obj, resolveQualifiedObjectReference regs obj identifier
                                                          namespace)
    end

  | resolveObjectReference regs
                           (ObjectIndexReference {object, index, ...}) =
    let
        val obj = evalObjectExpr regs object
        val idx = evalExpr regs index
        val identifier = toUstring regs idx
        (* FIXME if its an Name, then don't convert *)
        val namespace = Namespace publicNS
    in
        (SOME obj, resolveQualifiedObjectReference regs obj identifier namespace)
    end
```

4.4.4.1 Qualified Object References

1 Here we describe how an identifier and a namespace expression is resolved to a name of a binding on a specfic object.

2 To resolve a qualified object reference we evaluate its namespace expression (it must yield a namespace value) and then simply return the object value and the evaluated name.

Semantics

3
```
and resolveQualifiedObjectReference (regs: REGS)
                                    (object: OBJECT)
                                    (identifier: IDENTIFIER)
                                    (namespaceExpr: NAMESPACE_EXPRESSION)
    : (OBJECT * NAME) =
    let
        val namespaces = [evalNamespaceExpr regs namespaceExpr]
        val openNamespaces = []
    in
```

```
        resolveOnObject regs object identifier namespaces openNamespaces
    end
```

4.4.4.2 Unqualified Object References

1 To resolve an unqualified object reference we make use of the full algorithm outlined above, finding the first object that maches the unqualified name in all open namespaces and then disambiguating the set of resulting namespaces.

Semantics

2
```
and resolveUnqualifiedObjectReference (regs: REGS)
                                      (object: OBJECT)
                                      (identifier: IDENTIFIER)
                                      (openNamespaces: OPEN_NAMESPACES)
    : (OBJECT * NAME) =
    let
        val namespaces = List.concat openNamespaces
    in
        resolveOnObject regs object identifier namespaces openNamespaces
    end
```

4.4.4.3 Resolve Name on an Object

Semantics

1
```
and resolveOnObject (regs:REGS)
                    (object:OBJECT)
                    (identifier:IDENTIFIER)
                    (namespaces:NAMESPACE_SET)
                    (openNamespaces: OPEN_NAMESPACES)
    : (OBJECT * NAME) =
    let
        val result = searchObject (regs, SOME object, NONE, identifier,
                                   namespaces, false)
    in
        case result of
            NONE => (object, {ns=publicNS, id=identifier})
          | SOME (object, namespaces) =>
            selectNamespacesByInstanceFixtureMaps regs object identifier
                                                  namespaces openNamespaces
    end
```

4.4.5 Common Name Resolution Algorithms

1 The following algorithms are common to the preceding resolver algorithms.

4.4.5.1 Single Object Search

1 Given an object, an identifier and a set of namespaces, this algorithm searches for a matching property name in the object and the object's prototype chain.

Semantics

2
```
fun searchObject (_, NONE, _, _, _, _) = NONE

  | searchObject (regs        : REGS,
                  SOME object : OBJECT option,
                  class       : Ast.CLASS option,
                  identifier  : IDENTIFIER,
                  namespaces  : NAMESPACE_SET,
                  fixedOnly   : bool)
    : (OBJECT * NAMESPACE_SET) option =
    let
        val matches = getBindingNamespaces (regs,
                                            object,
                                            class,
                                            identifier,
                                            namespaces,
                                            fixedOnly)
    in
        case matches of
            []
            => if fixedOnly then
                   NONE
               else
                   searchObject (regs,
                                 getPrototypeObject (object),
```

```
                                    NONE,
                                    identifier,
                                    namespaces,
                                    fixedOnly)

          | _
            => SOME (object, matches)
    end
```

4.4.5.2 Disambiguation by Filtering

1 Given an identifier, a list of namespaces, a list of classes, a list of open namespaces, the following algorithm coordinates the filtering of the set of namespaces: according to the order that the namespaces appear in bindings in the given classes first, and in the priority given by the list of open namespaces second.

Semantics

2
```
fun selectNamespaces (identifier     : IDENTIFIER,
                      namespaces     : NAMESPACE_SET,
                      instanceFixtureMaps   : Ast.FIXTURE_MAPS,
                      openNamespaces : OPEN_NAMESPACES)
    : NAMESPACE_SET =
    let
        val openNamespaceSet = List.concat (openNamespaces)
    in
        case namespaces of

            _ :: []
            => namespaces

          | _ =>
            let
                val matches' =
                    selectNamespacesByClass (instanceFixtureMaps,
                                             openNamespaceSet,
                                             identifier)
            in
                case matches' of
                    []
                    => raise (LogErr.NameError "internal error")

                  | [_]
                    => matches'

                  | _ =>
                    let
                        val matches'' =
                            selectNamespacesByOpenNamespaces (openNamespaces,
                                                              namespaces)
                    in
                        case matches'' of

                            []
                            => raise (LogErr.NameError "internal error")

                          | _
                            => matches''
                    end
            end
    end
```

4.4.5.2.3 Class Base Namespace Filtering

1 Given a list of classes, an identifier and a set of namespaces, the following algorithm selects the namespaces used on the most generic class of that list. This step is necessary to avoid object integrity issues that arise when a derived class introduces a binding with the same identifier and a different namespace in the open namespaces.

2 **Informal description**: Search a class for any instance fixture name bindings that are named by the provided identifier and any of the namespaces in the provided set. Collect the set of matching namespaces used in all such bindings. If the set of matching namespaces is nonempty, return it. Otherwise repeat the process on the next instance fixture map. If all the classes in the list are searched and no matching namespaces are found, return the empty set.

Semantics

3
```
fun selectNamespacesByClass ([], namespaces, _) = namespaces

  | selectNamespacesByClass (instanceFixtureMaps : Ast.FIXTURE_MAPS,
                             namespaces   : NAMESPACE_SET,
```

```
                            identifier   : IDENTIFIER)
    : NAMESPACE list =
    let
        val fixtureMap = hd instanceFixtureMaps
        val bindingNamespaces =
            getInstanceBindingNamespaces (fixtureMap, identifier, namespaces)
        val matches =
            intersectNamespaces (bindingNamespaces, namespaces)
    in
        case matches of

            []
            => selectNamespacesByClass (tl instanceFixtureMaps,
                                        namespaces,
                                        identifier)

          | _
            => matches
    end
```

4.4.5.2.4 Open Namespace Based Namespace Filtering

1 Given a list of sets of open namespaces (ordered from most recently opened to least recently opened) and a set of matching namespaces, this algorithm returns a subset of the matching set that occurs entirely within a single open namespace set.

2 **Informal description**: intersect the head of the provided open namespace list with the provided set of namespaces. If that intersection is nonempty, return it. Otherwise repeat the process with the tail of the open namespace list. If the end of the list of open namespace sets is reached without producing a nonempty intersection, return an empty set.

Semantics

3
```
fun selectNamespacesByOpenNamespaces ([], _) = []

  | selectNamespacesByOpenNamespaces (namespacesList : NAMESPACE_SET list,
                                      namespaces     : NAMESPACE_SET)
    : NAMESPACE list =
    let
        val matches = intersectNamespaces (hd namespacesList, namespaces)
    in
        case matches of

            []
            => selectNamespacesByOpenNamespaces (tl namespacesList, namespaces)

          | _
            => matches
    end
```