Ecma/TC39/2008/042

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 = Undefined
               | Null
               | Object of OBJ
```

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 Undefined

1 There is exactly one undefined value, denoted by the semantic value `Undefined` and stored in the global constant property **public::undefined** in ECMAScript.

NOTE The namespace **public** is predefined and is used for all global names that were also defined by the 3rd Edition Specification.

2 The allocated type of the undefined value is called the undefined type. 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 the 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.

1.2 Null

1 There is exactly one null value, denoted by the semantic value `Null` and by the null literal **null** in ECMAScript.

2 The allocated type of the null value is called the null type. 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. These indeded uses are conventions, and are not enforced by the language semantics.

1.3 Object

1 All values except the null and undefined values are object values.

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

Semantics

3
```
and OBJ =
    Obj of { props: PROPERTY_BINDINGS,
             proto: VALUE,
             ident: OBJ_IDENTIFIER,
             tag: TAG
           }
```

1.3.1 Property Binding Map

1 A property binding map associates at most one property with any name. If an object's property binding 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 binding map. The semantic type of a property binding map is unspecified.

3 A property binding 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).

Semantics

4
```
and PROPERTY_BINDINGS = …
```

1.3.1.1 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 }
```

1.3.1.2 Property States

1 The *state* of a property encodes either a value associated with the property, or else one of a small number of intermediate non-value conditions that a property can assume during evaluation.

2 When a property is created, it is *allocated* in a state that derives from its storage type.

3 A property with storage type `*` is allocated in the value state, with the undefined value.

4 A property with a storage type that has the null type as a subtype is allocated in the value state, with the null value.

5 Any other property is allocated in the uninitialized state, and must be initialized during the initialization phase of object construction, before the object's first constructor begins evaluation.

6 If a property is in value state, then the allocated type of the value held in the property is a compatible subtype of the storage type of the property.

7 Additional property states are defined for encoding non-value properties, such as types, type variables, and virtual properties (defined by getter and setter functions).

Semantics

8
```
and PROPERTY_STATE = UninitProp
                   | ValProp of VALUE
                   | TypeProp
                   | TypeVarProp
                   | VirtualValProp of
                     { getter: FUN_CLOSURE option,
                       setter: FUN_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.

1.3.1.3 Property Attributes

1 A property can have zero or more attributes from the following set:

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. Otherwise 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 part of the object's fixed structure 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.

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: bool,
               enumerable: bool,
               fixed: bool,
               writable: WRITABILITY }
```

1.3.2 Object Prototype

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

1.3.3 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 OBJ_IDENTIFIER = …
```

1.3.4 Object Tag

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

2 The `ObjectTag` and `ArrayTag` tags are present on objects of structural types `ObjectType` and `ArrayType`, respectively.

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

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

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

Semantics

6
```
and TAG =
    ObjectTag of FIELD_TYPE list
  | ArrayTag of (TYPE list * TYPE option)
  | PrimitiveTag of PRIMITIVE
  | InstanceTag of CLASS
  | NoTag
```

1.3.4.1 Primitive Tag

1 Some objects have additional an semantic value associated with them. Such objects are called *primitive objects* and have a *primitive tag*. The semantic value is held in the tag, and 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 10 specific ECMAScript class 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 instance of the corresponding class.

3 The allocated type of a primitive object may be more specific than the corresponding class type. In particular, function objects may have more specific subtypes of the class **public::Function**. In such cases, the allocated type of the object is present in the semantic value held by the primitive tag.

4 The correspondence between primitive tags and classes is the following:

- The primitive tag `Boolean` corresponds to the class **__ES4__::boolean**.
- The primitive tag `Double` corresponds to the class **__ES4__::double**.
- The primitive tag `Decimal` corresponds to the class **__ES4__::decimal**.
- The primitive tag `String` corresponds to the class **__ES4__::string**.
- The primitive tag `Namespace` corresponds to the class **__ES4__::Namespace**.
- The primitive tag `Class` corresponds to the class **__ES4__::Class**.
- The primitive tag `Interface` corresponds to the class **__ES4__::Interface**.
- The primitive tag `Function` corresponds to the class **public::Function**.
- The primitive tag `Type` corresponds to the class **__ES4__::Type**.
- The primitive tag `Generator` corresponds to the class **helper::GeneratorImpl**.

Semantics

5
```
and PRIMITIVE =
    BooleanPrimitive of bool
  | DoublePrimitive of Real64.real
  | DecimalPrimitive of Decimal.DEC
  | StringPrimitive of Ustring.STRING
  | NamespacePrimitive of NAMESPACE
  | ClassPrimitive of CLASS
  | InterfacePrimitive of INTERFACE
  | FunctionPrimitive of FUN_CLOSURE
  | TypePrimitive of TYPE
  | GeneratorPrimitive of GEN
```

NOTE The type `Real64.real` represents IEEE 64 bit binary floating point values. The type `Decimal.DEC` represents IEEE 128 bit decimal floating point numbers. The type `Ustring.STRING` represents Unicode strings. The type `bool` represents boolean values.

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.

2 Therefore the construction of any primitive object is described by a *special constructor* defined in semantic code, rather than a *standard constructor* that would otherwise be defined in standard library code. The behavior of each special constructor is described in the following sections.

1.4.2 Boolean Values

1 A *boolean value* is either 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 **__ES4__::boolean**. Such objects are called *boolean objects*.

NOTE The namespace **__ES4__** is predefined. It is used to tag global names that have been introduced in the 4th Edition.

2 No instances of the class **__ES4__::boolean** can be constructed aside from the two values **true** and **false**: the **__ES4__::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.

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 **__ES4__::double**. Such objects 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 in 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 draft IEEE 754r standard.

2 A decimal value can be held in the primitive tag of an instance of the class **__ES4__::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 32 bit unsigned integer values ("elements"). 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 ...). ES3 required code points to be 16 bit unsigned integer values; ES4 will likely allow code points to be either 16 bits or 32 bits.

FIXME This section must accomodate implementations that wish to stick with 16-bit code points, as ES3 requires.

2 A string value can be held in the primitive tag of an instance of the class **__ES4__::string**. Such objects are called *string objects*.

3 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 (32-bit values) within it. The empty string has length zero and therefore contains no elements.

4 All operations on string (except as otherwise stated) treat them as sequences of undifferentiated 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 32-bit values outside the legal range of UTF-32 code points.

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.

5 String literals evaluate to string objects.

6 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 and discussed in section ...names. Their notable features are recounted here.

FIXME We should probably define them here and reference this section from the Names chapter.

2 A *namespace value* is either transparent or opaque. A transparent namespace has an associated identifying string value. An opaque namespace has an associated unique identifier of unspecified representation.

3 A namespace value can be held in the primitive tag of an instance of the class **__ES4__::Namespace**. Such objects are called *namespace objects*.

4 A namespace value can be defined as a fixture in a global or class static scope using a **namespace** definition.

5 Any two transparent namespaces with equal identifying strings are equal. Any two opaque namespaces with equal identifiers are equal.

Semantics

6
```
type OPAQUE_NAMESPACE_IDENTIFIER = …

datatype NAMESPACE =
         TransparentNamespace of Ustring.STRING
       | OpaqueNamespace of OPAQUE_NAMESPACE_IDENTIFIER
```

1.4.7 Class Values

1 A *class value* consists of a name and a set of namespaces, fixtures and types.

2 A class value can be held in the primitive tag of an instance of the class **__ES4__::Class**. Such objects are called *class objects*.

3 A class value can be defined as a fixture in the global scope using 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 fixtures of the associated class value describe fixed properties found on the class object **C**. In this way, the class fixtures effectively describe an implicit anonymous subtype of **__ES4__::Class** that the class object **C** is an instance of.

FIXME __ES4__::Class is obsolete, probably. It still exists but we have proper metaobjects for this sort of thing.

FIXME This tying-knots stuff at the top of the type hierarchy is always a little subtle and hard to word. Suggestions welcome.

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

Semantics

7
```
and CLASS =
    Class of
      { name: NAME,
        privateNS: NAMESPACE,
        protectedNS: NAMESPACE,
        parentProtectedNSs: NAMESPACE list,
        typeParams: IDENTIFIER list,
        nonnullable: bool,
        dynamic: bool,
        extends: TYPE option,
        implements: TYPE list,
        classRib: RIB,
        instanceRib: RIB,
        instanceInits: HEAD,
        constructor: CTOR option,
        classType: TYPE }
```

NOTE A `RIB` datum is a map from property names to fixtures: types, names, and fixture properties.

1.4.7.1 Instance Fixtures

1 In addition to class fixtures, a class value holds *instance fixtures*. These describe the fixed properties found on *instances of* the class.

2 An object is an *instance* of a class value `C` if the object's tag is `InstanceTag` and the class type in the tag is `ClassType C`.

3 If an object **X** is an instance of a class value `C`, then for every *instance fixture* `F` in `C`, a property `P` exists on **X** satisfying the following conditions:

- `P` is not `removable`.
- `P` is not `enumerable`.
- `P` is `fixed`.
- If `F` is declared as **const** then the `writable` attribute of `P` is initially `WriteOnce`. Otherwise the attribute is initially `Writable`.
- The type of `F` is the type of `P`.

1.4.7.2 Class Types and Class-Instance Types

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

2 The *instance class type* of a class value `C` is `ClassType 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 *static class type* of a class value `C` is the allocated type of the class object *holding* `C`. In such an object, the static class type is stored in a field within `C`, and is an anonymous subtype of the `ClassType` of **__ES4__::Class**. The tag of such an object is `PrimitiveTag (Class C)`.

1.4.8 Interface Values

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

2 An interface closure value can be held in the primitive tag of an instance of the class **__ES4__::Interface**. Such objects are called *interface objects*.

FIXME __ES4__::Interface is obsolete, probably. It still exists but we have proper metaobjects for this sort of thing.

3 An interface value can be defined as a fixture in the global scope using 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: bool,
        extends: TYPE list,
        instanceRib: RIB }
```

1.4.9 Function Closures

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

2 A function closure value can be held in the primitive tag of an instance of the class **__ES4__::Function**. Such objects are called *function objects*.

3 A function closure value can be 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 set of parameter fixtures and a block of ECMAScript code.

7 Function 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 FUN_CLOSURE =
         { func: FUNC,
           this: OBJ option,
           env: SCOPE }
```

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

1.4.10 Type Values

FIXME fill in

1.4.11 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.

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 fixture properties 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*.

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 a property's value; `meta::set` is invoked to update or create a property; and `meta::delete` is invoked to delete a 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 object that contains the method, not on that object's prototype objects.

5 If a catch-all method returns normally then the value it returns (if any) becomes the result of the operation, 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 `StopIterationClass` is stored in the global property `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 prototype map or if the `meta::has` catchall claims the property to be present.

Semantics

3
```
and hasOwnProperty (regs : REGS)
                   (obj  : OBJ)
                   (n    : NAME)
    : bool =
    let
        val Obj { props, ... } = obj
    in
        if hasFixedProp props n then
            true
        else if hasFixedProp props 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 ES4_DefaultBehaviorClass []
                   in
                       if ty <* defaultBehaviorClassTy then
                           hasProp props n
                       else
                           throwExn e
                   end
        else
            hasProp props n
    end
```

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

2.3 Reading a property value

1 The GetProperty 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 GetProperty, depending on how the AST is eventually structured.

3 The GetProperty protocol queries the object for the presence of the property using the HasOwnProperty protocol, moving up the object's prototype chain if the object does not contain the property. Once an object on the prototype chain is found that contains the object, the internal `getPropertyHelper` function is invoked to extract the property value. If no property is found, then a default value may be returned, or, in strict mode, an exception may be thrown.

FIXME The GetProperty protocol below overlaps with the `searchObject` algorithm described in Names; the two algorithms must be reconciled. That will happen when the protocol here is described in terms of SML (because then they will use the same code).

Semantics

4
```
fun GetProperty(obj, name, isStrict)
  for every object in obj, obj's prototype, ...
    if HasOwnProperty(obj, name)
      return getPropertyHelper(obj, name, isStrict)
    end
  end

  if obj allows dynamic property creation
    if obj has a structural array type with a "rest" type constraint, denote it T
      if T has a default value
        return that default value
      else
        throw a ReferenceError
            "Cannot read uninitialized property with non-nullable type"
      end
    end
    return undefined
  end

  if isStrict
    throw a ReferenceError
        "Trying to read undefined property from non-dynamic object"
  end

  return undefined
end
```

5 The internal `getPropertyHelper` function reads the property named by *name* from an object *obj*, implementing strict mode checking if *isStrict* is true. An error is signalled in strict mode if the object's `meta::has` catch-all returned true for *name* and the property cannot be read.

Semantics

6
```
fun getPropertyHelper(obj, name, isStrict)
  if (name is a property in the property map of obj, denote it obj.name &&
      the fixed attribute of obj.name is true)
    if obj.name is a method
      return a closure
          where obj is bound as this to the method extracted from obj.name
    end
    if obj.name is a getter/setter pair, denote the getter obj.name.[[Getter]]
      return obj.name.[[Getter]](name)
    end
```

```
    return the value of obj.name
  end

  if (meta::get is a property in the property map of obj, denote it obj.meta::get
      && the fixed attribute of obj.meta::get is true)
    try
      return obj.meta::get(name)
    catch DefaultBehavior
      ; fall through to the next case
    end
  end

  if name is a property in the property map of obj
    if obj.name is a getter/setter pair, denote the getter obj.name.[[Getter]]
      return obj.name.[[Getter]](name)
    end
    return obj.name
  end

  if isStrict
    throw a ReferenceError, "Property not found"
  else
    return undefined
  end
end
```

FIXME We need to specify whether the bound method is cached or not, ie, whether, given that o.m is a method, (o.m === o.m).

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 This protocol must be specified as SML code.

Semantics

3
```
fun SetProperty(obj, name, value, isStrict)
  if (name is a property in the property map of obj, denote it obj.name &&
      the fixed attribute of obj.name is true)
    return setPropertyHelper(obj, name, value, isStrict)
  end

  if (meta::set is a property in the property map of obj, denote it obj.meta::set
      && the fixed attribute of meta::set is true)
    try
      invoke obj.meta::set(name, value)
      return
    catch DefaultBehavior
      ; fall through to the next case
    end
  end

  if name is a property in the property map of obj
    return setPropertyHelper(obj, name, value, isStrict)
  end

  if the dynamic attribute on obj is true
    if isStrict && obj is a global object
      throw a ReferenceError
          "illegal to create props on global obj in strict mode"
    end
```

```
    if (obj has an array type with a "rest" type constraint, denote it T &&
        the type of value is not a compatible subtype of T)
      throw a TypeError
          "allocated type of value is not compatible with \
          \storage type of property"
    end

    place a new object into the property map of obj
      property name = name
      property value = value
      property flags = (writable=true,
                        enumerable=true,
                        removable=true,
                        fixed=false)
    return
  end

  if isStrict
    throw a ReferenceError, "cannot create a property on a non-dynamic object"
  end
}
```

4 The internal `setPropertyHelper` function is invoked when *name* is known to name a property in *obj*.

FIXME Here I simply assume that methods are not writeable, but this may or may not be the right abstraction we want to use; the RI has two cases here anyway, and that's the code that will eventually be here.

Semantics

5
```
fun setPropertyHelper(obj, name, value, isStrict)
  if obj.name is a getter/setter pair, denote the setter obj.name.[[Setter]]
    return obj.name.[[Setter]](name)
  end

  if the writeable attribute of obj.name is false
    if isStrict
      throw a ReferenceError, "Attempting to update read-only value"
    end
    return
  end

  if (obj.name has a type annotation, denote it T &&
      the type of value is not a compatible subtype of T)
    throw a TypeError, "Attempting to store value of incompatible type"
  end

  store value in obj.name

  if the writeOnce attribute of obj.name is true
    set the writeOnce attribute of obj.name to false
    set the writeable attribute of obj.name to false
  end
end
```

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 This protocol must be specified as SML code.

Semantics

```
3  fun DeleteProperty(obj, name, isStrict)
     if (name is a property in the property map of obj, denote it obj.name &&
         the fixed attribute of obj.name is true)
       if isStrict
         throw a ReferenceError, "can't delete fixture properties"
       end
       return false
     end

     if (meta::delete is a property in the property map of obj,
                denote it obj.meta::delete
         && the fixed attribute of meta::delete is true)
       try
         return obj.meta::delete(name)
       catch DefaultBehavior
         ; fall through to the next case
       end
     end

     if name is a property in the property map of obj, denote it obj.name
       if the removable attribute of obj.name is true
         remove obj.name from the property map of obj
         return true
       end

       if isStrict
         throw a ReferenceError, "can't delete non-removable property"
       end
     end

     return false
   end
```

3 Types

FIXME Cross-check with normative grammer on terminology, etc

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.

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 has a *storage type*. The storage type of a property is given by its declaration and constrains the set of values that can be stored in the property. The storage type of a property is also called the property'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 only meaningful as the storage type of a property.

3.1.2 The null type

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

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 value stored in the global constant **`public::undefined`**.

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 or an interface type.

2 A *class type* is a type 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 A nominal type is denoted in type expressions by the name of the class or interface that defined the type.

3.1.5 Record types

1 A *record type* is a subtype of the **`public::Object`** class 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 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-lenght 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.

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 notation for defining return types of function types only; 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 preceeding each argument types, 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

FIXME Cormac recently reformulated the non-null operator such that it does *not* model deletion-of-null-from-union but rather persists, as a normal-form of a type term, wrapping a class or interface type, and modifies the contained class or interface type to reject null as a subtype. It works this way too, but there may be left over text that describes it the older way.

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

2 A non-null type is denoted **!T** for some class 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 fixed definition such as a class, interface or type definition. 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 therefore also serves as a parametric 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 class 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 a class, an interface, a type definition, 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 A type name that refers to a class resolves to a class type. A type name that refers to an interface resolves to an interface type.

3.2 Semantics of the Type Language

Semantics

1
```
datatype 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
       | InterfaceType of INTERFACE

and FUNCTION_TYPE =
    { typeParams : IDENTIFIER list,
      thisType   : TYPE,
      params  : TYPE list,
      minArgs : int,
      hasRest : bool,
      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 The Subtype and Type Equivalence Relations

3.3.1 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**.

3.3.2 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)   (* reflexivity *) orelse
    (subTypeRecord   extra type1 type2) orelse
    (subTypeArray    extra type1 type2) orelse
    (subTypeUnion    extra type1 type2) orelse
    (subTypeFunction extra type1 type2) orelse
    (subTypeNonNull  extra type1 type2) orelse
    (subTypeNullable extra type1 type2) orelse
    (subTypeNominal  extra type1 type2) orelse
    (subTypeHierarchy extra type1 type2) orelse
    (subTypeStructuralNominal extra type1 type2) orelse
    (extra type1 type2)
```

3.3.3 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.

3.3.3.1 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)
```

3.3.4 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 ( name1, type1 ) =>
                    List.exists (fn ( name2, type2 ) =>
                                    nameExpressionEqual name1 name2 andalso
```

```
                                    equivType extra type1 type2)
                                fields2)
                fields1

      | _ => false

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

3.3.5 Subtyping Array Types

1 A fixed-length array type **[S$_1$, .., S$_n$, S]** is a subtype of **[S$_1$, .., S$_n$]**. The supertype demands one fewer element in the array than the subtype does. For example, **[Number, String, String]** is a subtype of **[Number, String]**.

2 A fixed-length array type **[S$_1$, .., S$_n$]** is a subtype of **[T$_1$, .., T$_n$]** if each **S$_i$** is equivalent to **T$_i$** for i in 1..n.

3 A variable-length array type **[S$_1$, .., S$_n$, S, ... S]** is a subtype of **[S$_1$, .., S$_n$, ... S]**. The supertype demands one fewer element in the array than the subtype does. For example, **[Number, String, String, ... String]** is a subtype of **[Number, ... String]**, via transitivity.

NOTE Since **...** denotes concrete syntax, we use the *meta-syntax* **S$_1$, .., S$_n$** to denote a sequence of zero-or-more comma-separated types.

4 A variable-length array type **[S$_1$, .., S$_n$, ... S]** is a subtype of **[T$_1$, .., T$_n$, ... T]** if **S** is equivalent to **T** and if each **S$_i$** is equivalent to **T$_i$** 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
```

3.3.6 Subtyping Union Types

1 A union type **(S$_1$ | .. | S$_n$)** is a subtype of a type **T** if **S$_i$** is a subtype of **T** for all i in 1..n.

2 A type **S** is a subtype of **(T$_1$ | .. | T$_n$)** if there exists some i in 1..n such that **S** is a subtype of **T$_i$**.

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
```

3.3.7 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 generic 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 generic functions, we alpha-rename the type variables to be identical in both types, and then proceed to check subtyping on the non-generic 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
  = …
```

3.3.8 Subtyping Non-Null Types

1 A non-null type **!S** is a subtype of type **T** if **S** is a subtype of the union type **(T | null)**.

2 A type **S** is a subtype of a non-null type **!T** if **S** is a subtype of **T** and the type **null** is *not* a subtype of **S**.

Semantics

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

        (NonNullType type1, type2) =>
        subType extra type1 (UnionType [type2, NullType])

      | (type1, NonNullType type2) =>
        subType extra type1 type2 andalso
        not (subType extra NullType type1)

      | _ => false
```

3.3.9 Subtyping Nullable Nominal Types

1 A nominal 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>! ..
```

2 A nominal type is *nullable* if it is not a non-null type.

3 The type **null** is a subtype of any nullable nominal type.

Semantics

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

        (NullType,
         ClassType (Class { nonnullable = false, ... }))
        => true

      | (NullType,
         AppType (ClassType (Class { nonnullable = false, ... }), typeArgs))
        => true

      | (NullType,
         InterfaceType (Interface { nonnullable = false, ... }))
        => true

      | (NullType,
         AppType (InterfaceType (Interface { nonnullable = false, ...}), typeArgs))
        => true

      | _ => false
```

3.3.10 Subtyping Nominal Types

1 Given a class definition

```
class C extends D implements I1, .., In { ... }
```

the type **C** is a subtype of **D**, and **C** is also a subtype of **I_j** for j in 1..n.

2 Given an interface definition

```
interface K extends I1, .., In { ... }
```

the type **K** is a subtype of **Ij** for j in 1..m.

3 These rules generalize to applications of generic classes and interfaces via appropriate renaming of bound variables. For example, given a generic class definition

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 **T_i** is equivalent to the corresponding type **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 **x_i** replaced (in a capture-free manner) by the corresponding type **S_i**.

Semantics

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

        ( 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

and subTypeHierarchy extra type1 type2 =
    case (type1, type2) of

        ( ClassType (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
              (ClassType (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

       | _ => 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
  = …
```

3.3.11 Relating Structural and Nominal Types

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

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

3 Any function type is a subtype of the class type **public::Function**.

Semantics

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

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

      | (ArrayType _, ClassType (Class { name, ... }))
        => nameEq name Name.public_Array orelse
           nameEq name Name.public_Object

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

      | _ => false
```

3.4 Type Normalization

1 At run-time, when a type **T** is encountered in the source program, that type is immediately *normalized*. Type normalization consists of two phases: *type resolution* followed by *type canonicalization*.

3.4.1 Type Resolution

1 Type resolution on a type **T** 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 generic 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 class name **C** with a class definition D, type resolution replaces any TypeName that refers to **C** with ClassType D.

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.

6 The type resolution code is not presented here.

3.4.1.1 Implementation of Type Resolution

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

2 This function relies on the auxiliary function Fixture.resolveNameExpr (described in section ...) to resolve each type name. The function Fixture.resolveNameExpr returns the corresponding fixture (as the third component of the result triple), plus the environment that fixture was defined in, and the fully-resolved name for the given name expression.

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 a class type or an interface type.

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 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.

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

Semantics

8
```
fun resolveTypeNames (env : RIBS)
                     (ty  : TYPE)
    : TYPE =
    case ty of

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

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

              | (_, _, ClassFixture c    ) => ClassType c
              | (_, _, InterfaceFixture i) => 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
```

```
                    if length typeArgs = length typeParams then
                        ()
                    else
                        error ["Incorrect no of arguments to parametric typedefn"];
                    resolveTypeNames envOfDefn
                                    (substTypes typeParams
                                                (map (resolveTypeNames env)
                                                     typeArgs)
                                                typeBody)
                end

              | _ => mapType (resolveTypeNames env) ty
        end

      | _ => mapType (resolveTypeNames env) ty

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
```

3.4.2 Type Canonicalization

1 Each type **T** is considered equivalent (under the equivalence relation defined above) to some collection of types. The process of *type canonicalization* converts a type in the program source code into a canonical or representative element of its equivalence class. In particular, if two types $\mathbf{T_1}$ and $\mathbf{T_2}$ are equivalent, then canonicalization will convert them both into an identical normalized type.

2 This canonicalization process is necessary to efficiently support type-parametric classes.

3 The type canonicalization code is not presented here.

3.4.3 Normalized Types

1 A *normalized type* is one that is the result of the preceeding normalization process.

2 Normalized types do not include:

- type names that refer to nominal types (`ClassType` and `InterfaceType` are used instead)
- type names that refer to type definitions (which are inlined)

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

3.5 Compatible Types

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

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

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 : int, y : bool}** is a compatible-subtype of the types **{x : *, y : *}**, **{x : int}**, **{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 reasons about compatibility, in 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 *reifiable* 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 a reifiable type. The allocated type of a value is invariant; for example, updating the fields of an object cannot change the allocated type of that object.

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 constants that are comprised of a namespace value and an identifier.

2 Names denote types, namespaces, and locations (properties bound in objects and scopes). The denotation of a name depends on the context of the name's use: When a name is used in a type annotation context it denotes a type; when it is used in a qualifier context it denotes a namespace; and in all other contexts it denotes a location.

3 *Unqualified names* are expressed as simple identifiers, for example `encodeURI`. *Qualified names* are expressed as pairs of namespace expressions and simple identifiers, for example `intrinsic::substring` or `"org.w3.dom"::DOMNode`.

4 Unqualified names are subject to *name resolution*: every unqualified name must resolve to a unique qualified name. Names that denote types and namespaces are resolved at definition time, while names that denote locations are 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.

5 Name resolution makes use of the *open namespaces* that implicitly qualify any unqualified name. In every compilation unit the open namespaces starts out being comprised of the **public** and **internal** namespaces. The program can open additional namespaces by means of the `use namespace` pragma.

6 Name resolution is performed differently depending on whether the name is a *lexical reference* (for example, the variable reference `encodeURI`) or a *property reference* on an object (for example, the reference `s.substring`) .

7 In the case of a lexical reference a name is resolved as a reference to a name *bound* in the *scope* of the reference; each entry in the chain formed by active scope objects binds names to which the reference may resolve, with resolutions in scopes closer to the point of reference (in "inner scopes") preferred over those in scopes further away (in "outer scopes").

8 In the case of a property reference a name is resolved as a reference to a property on a specific object; each entry in the chain formed by the object and its prototype objects in order provides named properties to which the reference may resolve, with resolutions in objects closer to the original object preferred over those further out in the prototype chain.

9 A reference may be found to be ambiguous. The resolution algorithm incorporates several forms of disambiguation, described later, but some references are inherently ambiguous. Such references cause errors to be signalled at definition or evaluation time.

10 Names that denote types, namespaces, and locations are resolved by the same algorithm. Suppose an unqualified name that denotes a type or namespace is resolved to a particular type or namespace definition in a particular scope. Then the same unqualified name denoting a location will be resolved unambiguously to an immutable location that holds a value that represents the type or namespace, if resolution takes place in the same scope as for the first name. A *reservation mechanism* ensures that names that are resolved at definition time cannot become ambiguous at evaluation time by the introduction of new bindings.

4.1 Name Values

1 A name is a constant value comprised of a namespace value and an identifier.

Semantics

2
```
type NAME = { ns: NAMESPACE, id: IDENTIFIER }
```

3 An identifier is a character string.

Semantics

4
```
type IDENTIFIER = Ustring.STRING
```

5 A namespace value is an immutable object. A namespace is *transparent* or *opaque*. A transparent namespace contains a character string that identifies the namespace; two transparent namespaces are equal if and only if their contained strings are equal. An opaque namespace contains an unforgeable system-generated value that identifies the namespace; two opaque namespaces are equal if and only if their contained identifier values are the same object.

Semantics

6
```
datatype NAMESPACE =
         TransparentNamespace of Ustring.STRING
       | OpaqueNamespace of OPAQUE_NAMESPACE_IDENTIFIER

type OPAQUE_NAMESPACE_IDENTIFIER = …

fun compareNamespaces (n1: NAMESPACE, n2: NAMESPACE) : bool =
    case (n1, n2) of
        (Ast.TransparentNamespace s1, Ast.TransparentNamespace s2) => s1 = s2
      | (Ast.OpaqueNamespace i1, Ast.OpaqueNamespace i2) => i1 = i2
      | _ => false
```

4.2 The namespace public

1 The namespace known as **public** is the transparent namespace whose identifying string is the empty string.

Semantics

2
```
val publicNS = Ast.TransparentNamespace Ustring.empty
```

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.

4.3 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 an object's *prototype object*, and the connection between the initial object and its prototype object is called a *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 or undefined 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 Scopes and visibility

1 Defining and binding forms introduce names into a program. These names can be referenced by name expressions that occur within the *scope* of the binding. The scope of a binding is primarily determined textually (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.

2 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.

3 In this Specification, the nesting of scopes is modelled as a list of ribs in the definition phase and a list of objects during evaluation. Both environments are generically called the *scope chain*. Which kind of list is being referred to will be clear from the context in which the term is used.

4 Each scope holds a table of named bindings in that scope. Ribs hold a table of fixture bindings and objects hold a table of property bindings. Every time a new scope is entered the scope chain is extended with a new rib or object, and at every point in the program one particular scope chain is in effect.

FIXME What's a fixture? Is it defined somewhere?

FIXME Exhibit the definition-time and evaluation-time structures for scope chains here.

5 Some objects that appear on evaluation-time scope chains are dynamically extensible, in effect providing a form of dynamic scope. 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; these properties then become visible and invisible to the methods.

6 In order for ECMAScript to have a lexically scoped flavor, bindings that are textually visible (*lexical bindings*) are usually preferred over dynamically added bindings (*dynamic bindings*) during name resolution. See the section "Name Resolution" below.

4.5 Name Expressions

1 There are two kinds of name expressions, the unqualified name (such as `encodeURI`) and the namespace-qualified name (such as `intrinsic::subtring` or `"org.w3.dom"::DOMNode`). Name resolution transforms name expressions into name values.

Semantics

2
```
and NAME_EXPRESSION =
    QualifiedName of { namespace: NAMESPACE_EXPRESSION,
                       identifier: IDENTIFIER }
  | UnqualifiedName of { identifier: IDENTIFIER,
                         openNamespaces: OPEN_NAMESPACES }
```

3 A qualified name expression is comprised 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 a name expression denoting a namespace binding.

Semantics

4
```
and NAMESPACE_EXPRESSION =
    Namespace of NAMESPACE
  | NamespaceName of NAME_EXPRESSION
```

5 An unqualified name expression is comprised of the namespaces that are open at the point where the name expression occurs in the source text and and identifier. The open namespaces will be used to resolve the name: an

unqualified name matches any binding or property that has the same identifier and a namespace value from among the open namespaces.

6 The open namespaces are represented as a list of sets of namespace values. Each set contains namespace values that are given the same priority during name resolution. The list holds sets in priority order.

NOTE A new set is added to the list every time a new lexical scope is entered, and the innermost (highest priority) set is extended by the `use namespace` pragma. An unqualified name expression retains a reference to the open namespaces data structure as it appears at the point where the expression occurs.

The two lowest priority sets are singleton sets holding the **public** and **internal** namespaces, respectively. The namespace **internal** is specific to each compilation unit.

Semantics

7
```
type NAMESPACE_SET = NAMESPACE list

type OPEN_NAMESPACES = NAMESPACE_SET list
```

8 The first element on an `OPEN_NAMESPACES` list is the highest priority element.

4.6 Reference Expressions

1 Name expressions are incorporated into *reference expressions* that provide context to the name expressions. Name expressions that reference names bound in a scope, such as `encodeURI` and `intrinsic::substring`, are contained in `LexicalReference` nodes.

2 Name expressions that reference properties on objects, such as `s.intrinsic::substring`, are contained in `ObjectNameReference` nodes, which contain both the object expression (`s`, in the example) and the name (`intrinsic::substring`).

3 Finally, the node `ObjectIndexReference` represents names that are computed at evaluatin time, such as `s[e]`.

Semantics

4
```
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.

4.7 Name Resolution

4.7.1 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 three complications. First, the search is performed differently for object chains (an object and its prototypes) and scope chains. An object chain is searched in a single pass and each object's fixed and dynamic properties are considered when the object is searched. A scope chain, on the other hand, is searched in two passes, with the first pass considering mainly fixed properties and the second pass considering also dynamic properties. (The search ends as soon as an object matching the name is found, so the second pass may never be run.) Thus fixed properties in outer scopes shadow dynamic properties in inner scopes. However, for reasons of compatibility with ES3, the first pass searches both dynamic and fixed properties in scopes that are introduced by the `with` statement or in scopes that have been extended by the `eval` operator evaluating a `function` or `var` directive.

3 (The motivation for the preference for fixed bindings in scopes is to retain the lexically scoped flavor of ECMAScript. Some of the objects on the scope chain--class objects and instance objects--are dynamically

extensible, and allowing dynamic properties to shadow static properties would make programs harder to understand, and it would make them slower, as it would be hard to perform early binding.)

4 The second complication 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.)

5 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**.

6 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**.

7 (The motivation for disambiguation is simple: disambiguation allows more programs to run. Furthermore, since the priority or namespaces during disambiguation is under the control of the programmer, the programmer can rely on disambiguation to control which names that are found.)

8 The third complication is that some names are required to be resolved successfully at definition time -- names that denote namespaces and types. (We require that in order to make names and types constant, which generally simplifies the language and makes programs more easily comprehensible.) The consequence of that 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.)

9 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 of that 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
```

10 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
```

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

12 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.

13 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`.

4.7.2 Definition-Time Resolution of Namespace and Type Expressions

1 The definition time scope chain is modelled as a list of `RIB` data structures, defined elsewhere. A rib maps names to fixture bindings that result from defining and binding forms (`var`, `function`, `type`, `class`, `interface`, `namespace`, and others). Ribs 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 ribs.

Semantics

4
```
and resolveNameExpr (ribs : Ast.RIBS)
                    (ne   : Ast.NAME_EXPRESSION)
    : (Ast.RIBS * Ast.NAME * Ast.FIXTURE) =

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

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

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

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

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

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

4.7.2.1 Qualified Name Expressions

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

Semantics

2
```
fun resolveQualifiedName (ribs          : Ast.RIBS)
                         (identifier    : IDENTIFIER)
                         (namespaceExpr : Ast.NAMESPACE_EXPRESSION)
    : (Ast.RIBS * NAME * Ast.FIXTURE) =
    let
        val ns = resolveNamespaceExpr ribs 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 ribs) of
            []
            => error ["qualified name not present in ribs: ", LogErr.name name]

          | ribs'
```

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

4.7.2.2 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 ribs such that the first rib on the tail contains a binding for the name.

Semantics

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

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

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

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

               | ns::nss
                 => error ["ambiguous reference: ", Ustring.toAscii identifier]
    end

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

  | ribListSearch (ribs       : Ast.RIBS,
                   namespaces : NAMESPACE_SET,
                   identifier : IDENTIFIER)
    : (Ast.RIBS * NAMESPACE_SET) option =
    case ribSearch (hd ribs, namespaces, identifier) of
        NONE
        => ribListSearch (tl ribs, namespaces, identifier)

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

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

4.7.2.3 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.

Semantics

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

4.7.3 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)
    : (OBJ * NAME) =
    let
        val {scope, ...} = regs
    in
        case nameExpression of

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

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

4.7.3.1 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.

Semantics

2
```
and resolveQualifiedLexicalReference (regs          : REGS)
                                     (identifier    : IDENTIFIER)
                                     (namespaceExpr : NAMESPACE_EXPRESSION)
    : (OBJ * NAME) =
    let
        val {scope, global, ...} = regs
        val namespace = evalNamespaceExpr regs namespaceExpr
        val result = searchScopeChain (SOME scope, identifier, [namespace])
    in
        case result of
            NONE
            => (global, {ns=publicNS, id=identifier})

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

4.7.3.2 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)
    : (OBJ * NAME) =
    let
        val {scope, global, ...} = regs
        val namespaces = List.concat openNamespaces
        val result = searchScopeChain (SOME scope, identifier, namespaces)
    in
        case result of
            NONE
            => (global, {ns=publicNS, id=identifier})

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

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

4.7.3.3 Searching a Scope Chain

1 To find an object matching an identifier and a set of namespaces in a scope chain, first make a pass over the scope chain looking only at fixed properties (except where the scope object is introduced by `with` or is subject to modification by the `eval` operator), and if none are found, make a second pass looking also for dynamic properties.

Semantics

2
```
fun searchScopeChain (scope      : SCOPE option,
                      identifier : IDENTIFIER,
                      namespaces : NAMESPACE_SET)
    : (OBJECT * NAMESPACE_SET) option =
    let
        val result = searchScopeChainOnce(scope, identifier, namespaces, true)
    in
        case result of
            NONE
            => searchScopeChainOnce(scope, identifier, namespaces, false)

          | SOME _
            => result
    end

and searchScopeChainOnce (NONE, _, _, _) = NONE

  | searchScopeChainOnce (SOME scope : SCOPE option,
                          identifier : IDENTIFIER,
                          namespaces : NAMESPACE_SET,
```

```
                          fixedOnly  : bool)
    : (OBJECT * NAMESPACE_SET) option =
    let
        val matches = searchScope (scope, namespaces, identifier, fixedOnly)
        val Scope { parent, ... } = scope
    in
        case matches of
            NONE
            => searchScopeChainOnce (parent, identifier, namespaces, fixedOnly)

          | _
            => matches
    end

fun searchScope (scope      : SCOPE,
                 namespaces : NAMESPACE_SET,
                 identifier : IDENTIFIER,
                 fixedOnly  : bool)
    : (OBJECT * NAMESPACE_SET) option =
    let
        val (object, kind) = getScopeObjectAndKind (scope)
    in
        case (kind, fixedOnly) of
            (WithScope, true)
            => searchObject (SOME object, identifier, namespaces, false)

          | (WithScope, false)
            => NONE

          | (_,_)
            => searchObject (SOME object, identifier, namespaces, fixedOnly)
    end
```

4.7.4 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)
    : (OBJ option * (OBJ * NAME)) =
    let
        val obj = evalObjectExpr regs object
    in
        case name of
            UnqualifiedName { identifier, openNamespaces, ... }
            => (SOME obj,
                resolveUnqualifiedObjectReference regs
                                                  obj
                                                  identifier
                                                  openNamespaces)

          | QualifiedName { namespace, identifier }
            => 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
        resolveQualifiedObjectReference regs obj identifier namespace
    end
```

4.7.4.1 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: OBJ)
                                    (identifier: IDENTIFIER)
                                    (namespaceExpr: NAMESPACE_EXPRESSION)
    : (OBJ option * (OBJ * NAME)) =
    let
        val namespace = evalNamespaceExpr regs namespaceExpr
    in
        (SOME object, (object, {ns=namespace, id=identifier}))
    end
```

4.7.4.2 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: OBJ)
                                      (identifier: IDENTIFIER)
                                      (openNamespaces: OPEN_NAMESPACES)
    : (OBJ * NAME) =
    let
        val namespaces = List.concat openNamespaces
        val result = searchObject (SOME object, identifier, namespaces, false)
    in
        case result of
            NONE
            => (object, {ns=publicNS, id=identifier})

          | SOME (object, namespaces)
            => let
                   val instanceRibs = instanceRibsOf (object)
                   val result = Fixture.selectNamespaces (identifier,
                                                          namespaces,
                                                          instanceRibs,
                                                          openNamespaces)
               in
                   case result of
                       []
                       => internalError ["empty namespace set"]
```

```
                    | namespace :: []
                     => (object, {ns=namespace, id=identifier})

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

4.7.5 Common Algorithms

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

4.7.5.1 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 (SOME object : OBJECT option,
                  identifier  : IDENTIFIER,
                  namespaces  : NAMESPACE_SET,
                  fixedOnly   : bool)
    : (OBJECT * NAMESPACE_SET) option =
    let
        val matches = getBindingNamespaces (object,
                                            identifier,
                                            namespaces,
                                            fixedOnly)
    in
        case matches of
            []
            => if fixedOnly then
                   NONE
               else
                   searchObject (getPrototypeObject (object),
                                 identifier,
                                 namespaces,
                                 fixedOnly)

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

4.7.5.2 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,
                      instanceRibs   : Ast.RIBS,
                      openNamespaces : OPEN_NAMESPACES)
    : NAMESPACE_SET =
    let
        val openNamespaceSet = List.concat (openNamespaces)
    in
        case namespaces of

            _ :: []
            => namespaces
```

```
          | _ =>
            let
                val matches' =
                    selectNamespacesByClass (instanceRibs,
                                             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
```

4.7.5.2.1 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 rib. 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 (instanceRibs : Ast.RIBS,
                             namespaces   : NAMESPACE_SET,
                             identifier   : IDENTIFIER)
    : NAMESPACE list =
    let
        val rib = hd instanceRibs
        val bindingNamespaces =
            getInstanceBindingNamespaces (rib, identifier, namespaces)
        val matches =
            intersectNamespaces (bindingNamespaces, namespaces)
    in
        case matches of

            []
            => selectNamespacesByClass (tl instanceRibs,
                                        namespaces,
                                        identifier)
```

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

4.7.5.2.2 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
```