Dart Programming Language Specification

Draft Version 0.01

The Dart Team

October 10th, 2011

Contents

1 Notes

This is a work in progress. Expect the contents to evolve over time. Please mail comments to gbracha@google.com.

1.1 Licensing

2 Notation

We distinguish between normative and non-normative text. Normative text defines the rules of Dart. It is given in this font. At this time, non-normative text includes:

Rationale Discussion of the motivation for language design decisions appears in italics. *Distinguishing normative from non-normative helps clarify what part of the text is binding and what part is merely expository.*

Commentary Comments such as "The careful reader will have noticed that the name Dart has four characters" serve to illustrate or clarify the specification, but are redundant with the normative text. The difference between commentary and rationale can be subtle. *Commentary is more general than rationale, and may include illustrative examples or clarifications.*

Open questions (**in this font**). Open questions are points that are unsettled in the mind of the author(s) of the specification; expect them (the questions, not the authors; precision is important in a specification) to be eliminated in the final specification. **Should the text at the end of the previous bullet be rationale or commentary?**

Reserved words and built-in identifiers (10.29) appear in **bold**.
Examples would be, **switch** or **class**.

Grammar productions are given in a common variant of EBNF. The left hand side of a production ends with a :. On the right hand side, alternation is represented by vertical bars, and sequencing by spacing. Optional elements of a production are suffixed by a question mark like so: anElephant?. Appending a star to an element of a production means it may be repeated zero or more times. Appending a plus sign to a production means it occurs one or more times. Parentheses are used for grouping. Negation (the not combinator of PEGs) is represented by prefixing an element of a production with a tilde.

An example would be:

```
AProduction:
  AnAlternative |
 AnotherAlternative |
 OneThing After Another |
 ZeroOrMoreThings* |
 AnOptionalThing? |
 (Some Grouped Things) |
 A_LEXICAL_THING
;
```

Both syntactic and lexical productions are represented this way. Lexical productions are distinguished by their names. The names of lexical productions consist exclusively of upper case characters and underscores. As always, within grammatical productions, whitespace and comments between elements of the production are implicitly ignored unless stated otherwise.

Productions are embedded, as much as possible, in the discussion of the constructs they represent.

A list $x_1, \ldots, x_n$ denotes any list of n elements of the form $x_i, 1 \leq i \leq n$. Note that n may be zero, in which case the list is empty. We use such lists extensively throughout this specification.

The notation $[x_1, \ldots, x_n/y_1, \ldots, y_n]E$ denotes a copy of E in which all occurrences of $x_i, 1 \leq i \leq n$ have been replaced with y_i.

The specifications of operators often involve statements such as $x\ op\ y$ is equivalent to the method invocation $x.op(y)$. Such specifications should be understood as a shorthand for:

- $x\ op\ y$ is equivalent to the method invocation $x.op'(y)$, assuming the class of x actually declared a non-operator method named op' defining the same function as the operator op.

This circumlocution is required because x.op(y), where op is an operator, is not legal syntax. However, it is painfully verbose, and we prefer to state this rule once here, and use a concise and clear notation across the specification.

When the specification refers to the order given in the program, it means the order of the program source code text, scanning left-to-right and top-to-bottom.

3 Overview

Dart is a class-based, single-inheritance, pure object-oriented programming language. Dart is optionally typed (13) and supports reified generics and interfaces.

Dart programs may be statically checked. The static checker will report some violations of the type rules, but such violations do not abort compilation or preclude execution.

Dart programs may be executed in one of two modes: production mode or checked mode. In production mode, static type annotations (13.1) have

absolutely no effect on execution. In checked mode, assignments are dynamically checked, and certain violations of the type system raise exceptions at run time.

The coexistence between optional typing and reification is based on the following:

1. Reified type information reflects the types of objects at runtime and may always be queried by dynamic typechecking constructs (the analogs of instanceOf, casts, typecase etc. in other languages). Reified type information includes class and interface declarations, the class of an object, and type arguments to constructors.

2. Static type annotations determine the types of variables and function declarations (including methods and constructors).

3. Production mode respects optional typing. Static type annotations do not affect runtime behavior.

4. Checked mode utilizes static type annotations and dynamic type information aggressively yet selectively to provide early error detection during development.

Dart programs are organized in a modular fashion into units called *libraries* (12). Libraries are units of encapsulation and may be mutually recursive.

However they are not first class. To get multiple copies of a library running simultaneously, one needs to spawn an isolate.

3.1 Scoping

Dart is lexically scoped and uses a single namespace for variables, functions and types. It is a compile-time error if there is more than one entity with the same name declared in the same scope. Names in inner scopes may hide names in enclosing scopes, however, it is a static warning if a declaration introduces a name that is available in a lexically enclosing scope.

Names may be introduced into a scope by declarations within the scope or by other mechanisms such as imports or inheritance.

3.2 Privacy

Dart supports two levels of privacy: *public* and *private*. A declaration is *private* iff it begins with an underscore (the _ character) otherwise it is *public.* A declaration m is *accessible to library* L if m is declared in L or if m is public.

This means private declarations may only be accessed within the library in which they are declared.

Privacy is, at this point, a static notion tied to a particular piece of code (a library). It is designed to support software engineering concerns rather than security concerns. Untrusted code should always run in an another isolate. It is possible that libraries will become first class objects and privacy will be a dynamic notion tied to a library instance.

Privacy is indicated by the name of a declaration - hence privacy and naming are not orthogonal. This has the advantage that both humans and machines can

recognize access to private declarations at the point of use without knowledge of the context from which the declaration is derived.

3.3 Concurrency

Dart code is always single threaded. There is no shared-state concurrency in Dart. Concurrency is supported via actor-like entities called *isolates*.

An isolate is a unit of concurrency. It has its own memory and its own thread of control. Isolates communicate by message passing (10.14.4). No state is ever shared between isolates. Isolates are created by spawning (10.11).

4 Errors and Warnings

This specification distinguishes between several kinds of errors.

Compile-time errors are errors that preclude execution. A compile-time error must be reported by a Dart compiler before the erroneous code is executed.

A Dart implementation has considerable freedom as to when compilation takes place. Modern programming language implementations often interleave compilation and execution, so that compilation of a method may be delayed, e.g., until it is first invoked. Consequently, compile-time errors in a method m may be reported as late as the time of m's first invocation.

As a web language, Dart is often loaded directly from source, with no intermediate binary representation. In the interests of rapid loading, Dart implementations may choose to avoid full parsing of method bodies, for example. This can be done by tokenizing the input and checking for balanced curly braces on method body entry. In such an implementation, even syntax errors will be detected only when the method needs to be executed, at which time it will be compiled (JITed).

Static warnings are those errors reported by the static checker. They have no effect on execution. Many, but not all, static warnings relate to types, in which case they are known as *static type warnings.*

Dynamic type errors are type errors reported in checked mode.

Run-time errors are exceptions raised during execution. Whenever we say that an exception *ex* is *raised* or *thrown*, we mean that a throw statement (11.14) of the form: **throw** *ex*; was implicitly executed. When we say that *a C is thrown*, where C is a class, we mean that an instance of class $C$is thrown.

5 Variables

Variables are storage locations in memory.

variableDeclaration:
  declaredIdentifier (', ' identifier)*
;

initializedVariableDeclaration:
declaredIdentifier ('=' expression)? (', ' initializedIdentifier)*
;

initializedIdentifierList:
initializedIdentifier (', ' initializedIdentifier)*
;

initializedIdentifier:
identifier ('=' expression)?
;

declaredIdentifier:
finalVarOrType identifier
;

finalVarOrType:
final type? |
var |
type
;

A variable that has not been initialized has the initial value **null** (10.2). A *final variable* is a variable whose declaration includes the modifier **final**. A final variable can only be assigned once, when it is initialized, or a compile-time error occurs.

A *static variable* is a variable that is not associated with a particular instance, but rather with an entire library or class.

A variable that is both static and final must be initialized to a compile-time constant (10.1) or a compile-time error occurs.

Why tie together two orthogonal concepts like ***static*** *and* ***final*** *by requiring the use of constants? Because we do not want a language where expensive initialization computations are defined, causing long application startup times. This is especially crucial for Dart, which is designed for coding client applications.*

One time initializations using constants should incur negligible cost at run time.

If a variable declaration does not explicitly specify a type, the type of the declared variable(s) is **Dynamic**, the unknown type (13.6).

A top-level variable is implicitly static. It is a compile-time error to preface a top level variable declaration with the built-in identifier (10.29) **static**.

A top level variable marked **final** must be initialized to a compile-time constant or a compile-time error occurs.

6 Functions

Functions abstract over executable actions.

functionSignature:
  returnType? identifier formalParameterList
;
functionPrefix:
  returnType? identifier
;

functionBody:
  '=>' expression ';' |
 block
;

block:
  '{' statements '}'
;

Functions include function declarations (6.1), methods (7.1, 7.6), getters (7.2), setters (7.3), constructors (7.5) and function literals (10.8).

All functions have of a signature and a body. The signature describes the formal parameters of the function, and possibly its name and return type. The body is a block statement (11.1) containing the statements (11) executed by the function. A function body of the form of the form $=> e$ is equivalent to a body of the form $\{$**return** $e;\}$.

If the last statement of a function is not a return statement, the statement **return null**; is implicitly appended to the function body.

Because Dart is optionally typed, we cannot guarantee that a function that does not return a value will not be used in the context of an expression. Therefore, every function must return a value. See further discussion in section 11.10.

6.1 Function Declarations

A *function declaration* is a function that is not a method, getter, setter or function literal. Function declarations include *library functions*, which are function declarations at the top level of a library, and *local functions*, which are functions declarations declared inside other functions.

A function declaration of the form $T_0\ id(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}])\{s\}$ is equivalent to a variable declaration of the form **final** $F\ id = (T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}])\{s\}$, where F is the function type alias (13.3.1) **typedef** $T_0\ F(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]$. Likewise, a function declaration of the form $id(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}$

x_{n+k}])$\{s\}$ is equivalent to a variable declaration of the form **final** F id = (T_1 $a_1, \ldots, T_n$ $a_n, [T_{n+1}$ $x_{n+1}, \ldots, T_{n+k}$ x_{n+k}])$\{s\}$, where F is the function type alias **typedef** $F(T_1$ $a_1, \ldots, T_n$ $a_n, [T_{n+1}$ $x_{n+1}, \ldots, T_{n+k}$ $x_{n+k}]$.

We need to cover library getters as well.

Some obvious conclusions:

A function declaration of the form $id(T_1$ $a_1, \ldots, T_n$ $a_n, [T_{n+1}$ $x_{n+1}, \ldots, T_{n+k}$ $x_{n+k}]) => e$ is equivalent to a variable declaration of the form **final var** id = $((T_1$ $a_1, \ldots, T_n$ $a_n, [T_{n+1}$ $x_{n+1}, \ldots, T_{n+k}$ $x_{n+k}]) => e$.

A function literal of the form $(T_1$ $a_1, \ldots, T_n$ $a_n, [T_{n+1}$ $x_{n+1}, \ldots, T_{n+k}$ $x_{n+k}]) => e$ is equivalent to a function literal of the form $(T_1$ $a_1, \ldots, T_n$ $a_n, [T_{n+1}$ $x_{n+1}, \ldots, T_{n+k}$ $x_{n+k}]){ **return** e;}.

It is a compile-time error to preface a function declaration with the built-in identifier **static**.

6.2 Formal Parameters

Every function declaration includes a *formal parameter list*, which consists of a list of required parameters, followed by any optional parameters. Optional parameters consist of a set of named parameters.

formalParameterList:
  '(' ')' |
 '(' normalFormalParameters (', namedFormalParameters)? ')' |
 (namedFormalParameters)
;

normalFormalParameters:
  normalFormalParameter (', ' normalFormalParameter)*
;

namedFormalParameters:
  '[' defaultFormalParameter (', ' defaultFormalParameter)* ']'
;

6.2.1 Positional Formals

A *positional formal parameter* is a simple variable declaration (5).

normalFormalParameter:
  functionSignature |
 fieldFormalParameter |
 simpleFormalParameter
;

simpleFormalParameter:
declaredIdentifier |
identifier
;

fieldFormalParameter:
finalVarOrType? **this** '.' identifier
;

6.2.2 Named Optional Formals

Optional parameters may be specified and provided with default values.

defaultFormalParameter:
normalFormalParameter ('=' expression)?
;

It is a compile-time error if the default value of a named parameter is not a compile-time constant (10.1). It is a compile-time error if the name of a named optional parameter begins with an _ character.

The need for this restriction is a direct consequence of the fact that naming and privacy are not orthogonal. If we allowed named parameters to begin with an underscore, they would be considered private and inaccessible to callers from outside the library where it was defined. If a method outside the library overrode a method with a private optional name, it would not be a subtype of the original method. The static checker would of course flag such situations, but the consequence would be that adding a private named formal would break clients outside the library in a way they could not easily correct.

6.3 Type of a Function

If a function does not declare a return type explicitly, its return type is **Dynamic**. Let F be a function with required formal parameters $T_1\ p_1 \ldots, T_n\ p_n$, return type T_0 and named optional parameters $T_{n+1}\ p_{n+1}, \ldots, T_{n+k}\ p_{n+k}$. Then the type of F is $(T_1, \ldots, T_n, [T_{n+1}\ p_{n+1}, \ldots, T_{n+k}\ p_{n+k}]) \rightarrow T_0$.

7 Classes

A *class* defines the form and behavior of a set of objects which are its *instances*.

classDefinition:
class identifier typeParameters? superclass? interfaces?
'{' classMemberDefinition* '}'

;

classMemberDefinition:
declaration ';' |
methodSignature functionBody
;

methodSignature:
factoryConstructorSignature |
static functionSignature |
getterSignature |
setterSignature |
operatorSignature |
functionSignature initializers? |
namedConstructorSignature initializers?
;

declaration:
constantConstructorSignature (redirection | initializers)? |
constructorSignature (redirection | initializers)? |
functionSignature redirection |
namedConstructorSignature redirection |
abstract getterSignature |
abstract setterSignature |
abstract operatorSignature |
abstract functionSignature |
static final type? staticFinalDeclarationList |
static? initializedVariableDeclaration
;

staticFinalDeclarationList:
:
staticFinalDeclaration (', ' staticFinalDeclaration)*
;

staticFinalDeclaration:
identifier '=' expression
;

A class has constructors, instance members and static members. The instance members of a class are its instance methods, getters, setters and instance variables. The static members of a class are its static methods, getters, setters and static variables.

Every class has a single superclass except class Object which has no superclass. A class may implement a number of interfaces by declaring them in its implements clause (7.9).

An *abstract class* is either a class that is explicitly declared with the **abstract** modifier, or a class that declares at least one abstract method (7.1.1).

The abstract modifier for classes is not yet implemented

The *interface of class* C is an implicit interface that declares instance members that correspond to the instance members declared by C, and whose direct superinterfaces are the direct superinterfaces of C (7.9). When a class name appears as a type or interface, that name denotes the interface of the class.

It is a compile-time error if a class declares two members of the same name, except that a getter and a setter may be declared with the same name provided both are instance members or both are static members.

What about a final instance variable and a setter? This case is illegal as well. If the setter is setting the variable, the variable should not be final.

7.1 Instance Methods

Instance methods are functions (6) whose declarations are immediately contained within a class declaration and that are not declared **static**. The instance methods of a class C are those instance methods declared by C and the instance methods inherited by C from its superclass.

It is a static warning if an instance method m_1 overrides (7.8.1) an instance member m_2 and the type of m_1 is not a subtype of the type of m_2.

7.1.1 Abstract Methods

An *abstract method* declares an instance method without providing an implementation. The declaration of an abstract method is prefixed by the built-in identifier (10.29) **abstract**. It is a compile-time error if any default values are specified in the signature of an abstract method. **This could all be enforced syntactically.**

The abstract modifier for classes is not implemented. It is intended to be used in scenarios where an abstract class A inherits from another abstract class B. In such a situation, it may be that A itself does not declare any abstract methods. In the absence of an abstract modifier on the class, the class would be interpreted as a concrete class. However, we want different behavior for concrete classes and abstract classes. If A is intended to be abstract, we want the static checker to warn about any attempt to instantiate A, and we do not want the checker to complain about unimplemented methods in A. In contrast, if A is intended to be concrete, the checker should warn about all unimplemented methods, but allow clients to instantiate it freely.

Invoking an abstract method always results in a run-time error. This may be NoSuchMethodError or a subclass, such as AbstractMethodError.

These errors are ordinary objects and are therefore catchable.

Unless explicitly stated otherwise, all ordinary rules that apply to methods apply to abstract methods.

7.1.2 Operators

Operators are instance methods with special names.

operatorSignature:
  returnType? **operator** operator formalParameterList
;

operator:
  unaryOperator |
 binaryOperator |
 '[]' |
 '[]=' |
 negate,
;

unaryOperator:
  negateOperator
;

binaryOperator:
  multiplicativeOperator |
 additiveOperator |
 shiftOperator |
 relationalOperator |
 equalityOperator |
 bitwiseOperator |

;

prefixOperator:
  additiveOperator |
 negateOperator
;

negateOperator:
  '!' |
 '~'
;

An operator declaration is identified using the built-in identifier (10.29) operator.

The following names are allowed for user-defined operators: ==, <, >, <=, >=, -, +, *, ~/, %, |, ^, &, <<, >>, >>>, []=, [], ~, **negate**.

The built-in identifier **negate** is used to denote unary minus.

It is a compile-time error if the number of formal parameters of the user-declared operator []= is not 2. It is a compile-time error if the number of formal parameters of a user-declared operator with one of the names: ==, <, >, <=, >=, -, +, *, ?, ~/, %, |, ^, &, <<, >>, >>>, [] is not 1. It is a compile-time error if the arity of a user-declared operator with one of the names: ~, **negate** is not 0.

7.2 Getters

Getters are functions (6) that are used to retrieve the values of object properties.

getterSignature:
  static? returnType? **get** identifier formalParameterList
;

If no return type is specified, the return type of the getter is **Dynamic**.

A getter definition that is prefixed with the **static** modifier defines a static getter. Otherwise, it defines an instance getter. The name of the getter is given by the identifier in the definition.

It is a compile-time error if a getter's formal parameter list is not empty.

It is a compile-time error if a class has both a getter and a method with the same name. This restriction holds regardless of whether the getter is defined explicitly or implicitly, or whether the getter or the method are inherited or not.

It is a static warning if a getter m_1 overrides (7.8.1) a getter or method m_2 and the type of m_1 is not a subtype of the type of m_2.

7.3 Setters

Setters are functions (6) that are used to set the values of object properties.

setterSignature:
  static? returnType? **set** identifier formalParameterList
;

If no return type is specified, the return type of the setter is **Dynamic**.

A setter definition that is prefixed with the **static** modifier defines a static setter. Otherwise, it defines an instance setter. The name of the setter is given by the identifier in the definition.

It is a compile-time error if a setter's formal parameter list does not consists of exactly one required formal parameter *p. We could enforce this via the grammar, but we'd have to specify the evaluation rules in that case.*

It is a compile-time error if a class has both a setter and a method with the same name. This restriction holds regardless of whether the setter is defined explicitly or implicitly, or whether the setter or the method are inherited or not.

It is a static warning if a setter declares a return type other than **void**. It is a static warning if a setter m_1 overrides (7.8.1) a setter or method m_2 and the type of m_1 is not a subtype of the type of m_2. It is a static warning if a class has a setter with argument type T and a getter with return type S, and T may not be assigned to S.

7.4 Instance Variables

Instance variables are variables whose declarations are immediately contained within a class declaration and that are not declared **static**. The instance variables of a class C are those instance variables declared by C and the instance variables inherited by C from its superclass.

If an instance variable declaration has the form $T\ v = e$; or the form **var** $v = e$; then the expression e must be a compile-time constant (10.1).

*In Dart, all uninitialized variables have the value **null**, regardless of type. Numeric variables in particular are therefore best explicitly initialized; such variables will not be initialized to 0 by default. The form above is intended to ease the burden of such initialization.*

An instance variable declaration of one of the forms $T\ v$;, **final** $T\ v$; , $T\ v = e$; or **final** $T\ v = e$; always induces an implicit getter function (7.2) with signature

T **get** $v()$

whose invocation evaluates to the value stored in v.

An instance variable declaration of one of the forms **var** v;, **final** v;, **var** $v = e$; or **final** $v = e$; always induces an implicit getter function with signature

get $v()$

whose invocation evaluates to the value stored in v.

Getters are introduced for all instance and static variables, regardless of whether they are final or not.

A non-final instance variable declaration of the form $T\ v$; or the form $T\ v = e$; always induces an implicit setter function (7.3) with signature

void set $v(T\ x)$

whose execution sets the value of v to the incoming argument x.

A non-final instance variable declaration of the form **var** v; or the form **var** $v = e$; always induces an implicit setter function with signature

set $v(x)$

whose execution sets the value of v to the incoming argument x.

7.5 Constructors

A *constructor* is a special member that is used in instance creation expressions (10.10) to produce objects. Constructors may be generative or they may be factories.

A *constructor name* always begins with the name of its immediately enclosing class or interface, and may optionally be followed by a dot and an identifier. The name of a non-factory constructor must be a constructor name.

Interfaces can have constructor declarations (but not bodies). See the discussion of factories.

If no constructor is specified for a class C, it implicitly has a default constructor C() : **super**() {}.

7.5.1 Generative Constructors

A *generative constructor* consists of a constructor name, a constructor parameter list, an initializer list and an optional body.

constructorSignature:
  identifier formalParameterList |
 namedConstructorSignature
;

namedConstructorSignature:
  identifier '.' identifier formalParameterList
;

A *constructor parameter list* is a parenthesized, comma-separated list of formal constructor parameters. A *formal constructor parameter* is either a formal parameter (6.2) or an initializing formal. An *initializing formal* has the form **this**.id, where id is the name of an instance variable of the immediately enclosing class.

A *fresh instance* is an instance whose identity is distinct from any previously allocated instance of the class. A generative constructor always allocates a fresh instance of its immediately enclosing class.

The above holds if the constructor is actually run, as it is by **new**. If a constructor c is referenced by **const**, c may not be run; instead, a canonical object may be looked up. See the section on instance creation (10.10).

If a generative constructor c is not a redirecting constructor and no body is provided, then c implicitly has an empty body {}.

Redirecting Constructors A generative constructor may be *redirecting*, in which case its only action is to invoke another generative constructor. A redirecting constructor has no body; instead, it has a redirect clause that specifies which constructor the invocation is redirected to, and with what arguments.

redirection:
  ':' **this** ('.' identifier)? arguments
;

Initializer Lists An initializer list begins with a colon, and consists of a comma-separated list of individual *initializers*. There are two kinds of initializers.

- A *superinitializer* specifies a *superconstructor* - that is, a specific constructor of the superclass. Execution of the superinitializer causes the initializer list of the superconstructor to be executed.
- An *instance variable initializer* assigns a value to an individual instance variable.

initializers:
  ':' superCallOrFieldInitializer (', ' superCallOrFieldInitializer)*
;

superCallOrFieldInitializer:
  super arguments |
  super '.' identifier arguments |
  fieldInitializer
;

fieldInitializer:
  (**this** '.')? identifier '=' conditionalExpression
;

Let k be a generative constructor. Then k may include at most one superinitializer in its initializer list or a compile-time error occurs. If no superinitializer is provided, an implicit superinitializer of the form **super**() is added at the end of k's initializer list, unless the enclosing class is class Object. It is a compile-time error if more than one initializer corresponding to a given instance variable appears in k's initializer list. It is a compile-time error if k's initializer list contains an initializer for a variable that is initialized by means of an initializing formal of k.

Each final instance variable f declared in the immediately enclosing class must have an initializer in k's initializer list unless it has already been initialized by one of the following means:

- Initialization at the declaration of f.
- Initialization by means of an initializing formal of k.

or a compile-time error occurs. It is a compile-time error if k's initializer list contains an initializer for a variable that is not an instance variable declared in the immediately surrounding class.

The initializer list may of course contain an initializer for any instance variable declared by the immediately surrounding class, even if it is not final.

It is a compile-time error if a generative constructor of class Object includes a superinitializer.

Execution of a generative constructor proceeds as follows:

First, a fresh instance (7.5.1) i of the immediately enclosing class is allocated. Next, the instance variable declarations of the immediately enclosing class are visited in the order they appear in the program text. For each such declaration d, if d has the form $finalVarOrType\ v = e;$ then the instance variable v of i is bound to the value of e (which is necessarily a compile-time constant). Next, any initializing formals declared in the constructor's parameter list are executed in the order they appear in the program text. Then, the constructor's initializers are executed in the order they appear in the program.

We could observe the order by side effecting external routines called. So we need to specify the order.

After all the initializers have completed, the body of the constructor is executed in a scope where **this** is bound to i. Execution of the body begins with execution of the body of the superconstructor with respect to the bindings determined by the argument list of the superinitializer of k.

*This process ensures that no uninitialized final field is ever seen by code. Note that **this** is not in scope on the right hand side of an initializer (see 10.9) so no instance method can execute during initialization: an instance method cannot be directly invoked, nor can **this** be passed into any other code being invoked in the initializer.*

An initializer of the form $v = e$ is equivalent to an initializer of the form **this**.$v = e$.

Execution of a superinitializer of the form **super**$(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ (respectively **super**.$id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ proceeds as follows:

First, the argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated.

Let C be the class in which the superinitializer appears and let S be the superclass of C. If S is generic, let $U_1, , \ldots, U_m$ be the actual type parameters passed to S in the superclass clause of C.

Then, the initializer list of the constructor S (respectively $S.id$) is executed with respect to the bindings that resulted from the evaluation of the argument list, with **this** bound to the current binding of **this**, and the type parameters (if any) of class S bound to the current bindings of $U_1, , \ldots, U_m$.

It is a compile-time error if class S does not declare a constructor named S (respectively $S.id$)

7.5.2 Factories

A *factory* is a static method prefaced by the built-in identifier (10.29) **factory**.

factoryConstructorSignature:
factory qualified typeVariables? ('.' identifier)? formalParameterList

;

It is a static warning if the name of the method is not either:

- A constructor name.
- The name of a constructor of an interface that is in scope at the point where the factory is declared.

The *return type* of a factory whose signature is of the form **factory** M or the form **factory** $M.id$ is M. The return type of a factory whose signature is of the form **factory** $M < T_1$ **extends** $B_{11}, \ldots, B_{1k_1}, \ldots, T_n$ **extends** $B_{n1}, \ldots, B_{nk_n} >$ or the form **factory** $M < T_1$ **extends** $B_{11}, \ldots, B_{1k_1}, \ldots, T_n$ **extends** $B_{n1}, \ldots, B_{nk_n} > .id$ is $M < T_1, \ldots, T_n >$. It is a compile-time error if M is not the name of the immediately enclosing class or the name of an interface in the enclosing lexical scope. It is a compile-time error if the type M declares p type parameters where $p \neq n$.

In checked mode, it is a dynamic type error if a factory returns an object whose type is not a subtype of its actual (13.8.1) return type.

It seems useless to allow a factory to return null. But it is more uniform to allow it, as the rules currently do.

Factories address classic weaknesses associated with constructors in other languages. Factories can produce instances that are not freshly allocated: they can come from a cache. Likewise, factories can return instances of different classes.

7.5.3 Constant Constructors

A *constant constructor* may be used to create compile-time constant (10.1) objects. A constant constructor is prefixed by the reserved word **const**. It is a compile-time error if a constant constructor has a body.

constantConstructorSignature:
  const qualified formalParameterList
;

All the work of a constant constructor must be handled via its initializers.

7.6 Static Methods

Static methods are functions whose declarations are immediately contained within a class declaration and that are declared **static**. The static methods of a class C are those static methods declared by C.

Inheritance of static methods has little utility in Dart. Static methods cannot be overridden. Any required static function can be obtained from its declaring library, and there is no need to bring it into scope via inheritance. Experience

shows that developers are confused by the idea of inherited methods that are not instance methods.

Of course, the entire notion of static methods is debatable, but it is retained here because so many programmers are familiar with it. Dart static methods may be seen as functions of the enclosing library.

It is a compile-time error if a class has two static methods with the same name. It is a static warning if a class has a static method with the same name as a static method of one of its superclasses.

This last restriction makes classes more brittle with respect to changes in the class hierarchy. It stems from a general observation that shadowing of names in the same scope is questionable and should elicit a warning. There is no hiding of static methods, or of static variables.

7.7 Static Variables

Static variables are variables whose declarations are immediately contained within a class declaration and that are declared **static**. The static variables of a class C are those static variables declared by C.

A static variable declaration of the form **static** T v; or the form **static** T v = e; always induces an implicit static getter function (7.2) with signature

static T **get** $v()$

whose invocation evaluates to the value stored in v.

A static variable declaration of the form **static var** v; or the form **static var** $v = e$; always induces an implicit static getter function with signature

static get $v()$

whose invocation evaluates to the value stored in v.

A non-final static variable declaration of the form **static** T v; or the form **static** T $v = e$; always induces an implicit static setter function (7.3) with signature

static void set $v(T\ x)$

whose execution sets the value of v to the incoming argument x.

A static variable declaration of the form **static var** v; or the form **static var** $v = e$; always induces an implicit static setter function with signature

static set $v(x)$

whose execution sets the value of v to the incoming argument x.

It is a compile-time error if a class has two static variables with the same name. It is a static warning if a class has a static variable with the same name as a static variable of one of its superclasses.

7.8 Superclasses

The **extends** clause of a class C specifies its superclass. If no **extends** clause is specified, then either:

- C is Object, which has no superclass. OR
- The superclass of C is Object.

It is a compile-time error to specify an **extends** clause for class Object.

superclass:
  extends type
;

It is a compile-time error if the extends clause of a class C includes a type expression that do es not denote a class available in the lexical scope of C.

A class S is *a superclass* of a class C iff either:

- S is the superclass of C, or
- S is a superclass of a class S' and S' is a superclass of C.

It is a compile-time error if a class C is a superclass of itself.

7.8.1 Inheritance and Overriding

A class C *inherits* any instance members of its superclass that are not overridden by members declared in C.

A class may override instance members that would otherwise have been inherited from its superclass.

Let C be a class declared in library L, with superclass S and let C declare an instance member m, and assume S declares an instance member m' with the same name as m. Then m *overrides* m' iff m is accessible (3.2) to L and one of the following holds:

- m is an instance method.
- m is a getter and m' is a getter or a method.
- m is a setter and m' is a setter or a method.

Whether an override is legal or not is described elsewhere in this specification.

For example getters and setters may not legally override methods and vice versa.

It is nevertheless convenient to define the override relation between members in this way, so that we can concisely describe the illegal cases.

Note that instance variables do not participate in the override relation, but the getters and setters they induce do. Also, getters don't override setters and vice versa. Finally, static members never override anything.

7.9 Superinterfaces

A class has a set of direct superinterfaces. This set includes the interface of its superclass and the interfaces specified in the the **implements** clause of the class.

interfaces:
implements typeList
;

It is a compile-time error if the implements clause of an class C includes a type expression that does not denote a class or interface available in the lexical scope of C.

In particular, one cannot inherit from a type variable.

It is a compile-time error if the implements clause of a class includes type **Dynamic**. It is a compile-time error if a type T appears more than once in the implements clause of a class.

One might argue that it is harmless to repeat a type in this way, so why make it an error? The issue is not so much that the situation described in program source is erroneous, but that it is pointless. As such, it is an indication that the programmer may very well have meant to say something else - and that is a mistake that should be called to her or his attention. Nevertheless, we could simply issue a warning; and perhaps we should and will. That said, problems like these are local and easily corrected on the spot, so we feel justified in taking a harder line.

It is a compile-time error if the interface induced by a class C is a superinterface of itself.

It is a static warning if an imported superinterface of a class C declares private members.

This last rule is problematic. As code evolves in one library (L_1) it may add private members to a class or interface I_1 implemented or inherited in another library L_2 breaking L_1. This is a direct result of coupling an interface based type system with library based privacy. We are considering alternative semantics that might help resolve this issue.

A class does not inherit members from its superinterfaces. However, its implicit interface does.

8 Interfaces

An *interface* defines how one may interact with an object. An interface has methods , getters, setters and constructors, and a set of superinterfaces.

interfaceDefinition:
interface identifier typeParameters? superinterfaces? factorySpecification? '{' (interfaceMemberDefinition)* '}'
;

interfaceMemberDefinition:
static final type? initializedIdentifierList ';' |
functionSignature ';' |

constantConstructorSignature ';' |
namedConstructorSignature ';' |
getterSignature ';' |
setterSignature ';' |
operatorSignature ';' |
variableDeclaration ';'
;

It is a compile-time error if any default values are specified in the signature of an interface method, getter, setter or constructor.

8.1 Methods

An interface method declaration specifies a method signature but no body. It is a static warning if an interface method m_1 overrides an interface method m_2 and the type of m_1 is not a subtype of the type of m_2.

8.1.1 Operators

Operators are instance methods with special names. Some, but not all, operators may be defined by user code, as described in section 7.1.2.

8.2 Getters and Setters

An interface may contain getter and/or setter signatures. These are subject to the same compile-time and static checking rules as getters and setters in classes (7.2, 7.3).

8.3 Factories and Constructors

An interface may specify a *factory class*, which is a class that will be used to provide instances when constructors are invoked via the interface.

factorySpecification:
factory identifier typeParameters?
;

An interface can specify the signatures of constructors that are used to provide objects that conform to the interface. It is a compile-time error if an interface declares a constructor without declaring a factory class.

Let I be an interface named N_I with factory class F, and let N_F be the name of F. If F implements I then it is a compile-time error if I and F do not have the same number of type parameters.

A constructor k_I of I *corresponds to a constructor* k_F of its factory class F iff either

- F does not implement I and k_I and k_F have the same name, OR
- F implements I and either
  - k_I is named N_I and k_F is named N_F, OR
  - k_I is named $N_I.id$ and k_F is named $N_F.id$.

It is a compile-time error if an interface I declares a constructor k_I and there is no constructor k_F in the factory class F such that k_I corresponds to k_F.

Let k_I be a constructor declared in an interface I, and let k_F be its corresponding constructor. Then:

- It is a compile-time error if k_I and k_F do not have the same number of required parameters.
- It is a compile-time error if k_I and k_F do not have identically named optional parameters.
- It is a compile-time error if k_I and k_F do not have identical type parameters.
- It is a static type warning if the type of the nth required formal parameter of k_I is not identical to the type of the nth required formal parameter of k_F.
- It is a static type warning if the types of named optional parameters with the same name differ between k_I and k_F.

If F implements I, and F is generic, then the factory clause of I must include a list of type parameters that is identical to the type parameters given in the type declaration of F, or a compile-time error occurs.

It is a compile-time error if any default values are specified in the signature of an interface constructor.

8.4 Superinterfaces

An interface has a set of direct superinterfaces. This set consists of the interfaces specified in the **extends** clause of the interface.

superinterfaces:
  extends typeList
;

An interface J is a superinterface of an interface I iff either J is a direct superinterface of I or J is a superinterface of a direct superinterface of I.

It is a compile-time error if the **extends** clause of an interface I includes a type expression that does not denote a class or interface available in the lexical scope of I. It is a compile-time error if the extends clause of an interface includes

type **Dynamic**. It is a compile-time error if an interface is a superinterface of itself.

It is a static warning if an imported superinterface of an interface I declares private members.

This last rule is problematic - see the discussion above.

8.4.1 Inheritance and Overriding

An interface I inherits any members of its superinterfaces that are not overridden by members declared in I.

However, if the above rule would cause multiple members $m_1, \ldots, m_k$ with the same name n to be inherited (because identically named members existed in several superinterfaces) then at most one member is inherited. If the static types $T_1, \ldots, T_k$ of the members $m_1, \ldots, m_k$ are not identical, then there must be a member m_x such that $T_x <: T_i, 1 \leq x \leq k$ for all $i, 1 \leq i \leq k$, or a static type warning occurs. The member that is inherited is m_x, if it exists; otherwise:

- If all of $m_1, \ldots, m_k$ have the same number r of required parameters and the same set of named parameters s, then I has a method named n, with r required parameters of type **Dynamic**, named parameters s of type **Dynamic** and return type **Dynamic**.
- Otherwise none of the members $m_1, \ldots, m_k$ is inherited.

The only situation where the runtime would be concerned with would be is during reflection, if a mirror attempted to obtain the signature of an interface member.

The current solution is a tad complex, but is robust in the face of type annotation changes. Alternatives: (a) No member is inherited in case of conflict. (b) The first m is selected (based on order of superinterface list) (c) Inherited member chosen at random.

(a) means that the presence of an inherited member of an interface varies depending on type signatures. (b) is sensitive to irrelevant details of the declaration and (c) is liable to give unpredictable results between implementations or even between different compilation sessions.

An interface may override instance members that would otherwise have been inherited from its superinterfaces.

Let I be an interface declared in library L, with superinterface S and let I declare an instance member m, and assume S declares an instance member m' with the same name as m. Then m *overrides* m' iff m is accessible (3.2) to L and one of the following holds:

- m is an instance method.
- m is a getter and m' is a getter or a method.
- m is a setter and m' is a setter or a method.

Whether an override is legal or not is described elsewhere in this specification.

9 Generics

A class (7) or interface (8) G may be *generic*, that is, G may have formal type parameters declared. A generic declaration induces a family of declarations, one for each set of actual type parameters provided in the program.

typeParameter:
  identifier (**extends** type)?
;
typeParameters:
  '<' typeParameter (',' typeParameter)* '>'
;

A type parameter T may be suffixed with an **extends** clause that specifies the *upper bound* for T. If no **extends** clause is present, the upper bound is Object.

The type parameters of a generic declaration G are in scope in the bounds of all of the type parameters of G, in the **extends** and **implements** clauses of G (if these exist) and in the non-static members of G.

Because type parameters are in scope in their bounds, we support F-bounded quantification (if you don't know what that is, don't ask).

Even where type parameters are in scope there are numerous restrictions at this time:

- A type parameter cannot be used to name a constructor in an instance creation expression (10.10).
- A type parameter cannot be used as an identifier expression (10.29).
- A type parameter cannot be used as a superclass or superinterface (7.8, 7.9, 8.4).

The normative versions of these are given in the appropriate sections of this specification. Some of these restrictions may be lifted in the future.

10 Expressions

An *expression* is a fragment of Dart code that can be evaluated at run time to yield a *value*, which is always an object. Every expression has an associated static type (13.1). Every value has an associated dynamic type (13.2).

expression:
  assignableExpression assignmentOperator expression |
 conditionalExpression
;

expressionList:
  expression (‘, ’ expression)*
;

primary:
  thisExpression |
 super assignableSelector |
 functionExpression |
 literal |
 identifier |
 newExpression constantObjectExpression |
 ‘(’ expression ‘)’ |

;

10.1 Constants

A *constant expression* is an expression whose value can never change, and that can be evaluated entirely at compile time.

A constant expression is one of the following:

- A literal number (10.3).
- A literal boolean (10.4).
- A literal string (10.5) that does not involve string interpolation (10.5.1).
- **null** (10.2).
- A reference to a static final variable (5).
- A constant constructor invocation (10.10.2).
- A constant list literal (10.6).
- A constant map literal (10.7).
- An expression of one of the forms $e_1 == e_2$, e_1 != e_2, $e_1 === e_2$ or e_1! == e_2 ,where e_1 and e_2 are constant expressions that evaluate to a numeric, string or boolean value.
- An expression of one of the forms e_1 && e_2 or $e_1||e_2$, where e_1 and e_2 are constant expressions that evaluate to a boolean value.
- An expression of one of the forms ˜e, e_1 ˜/ e_2, e_1 ˆ e_2, e_1 & e_2, $e_1|e_2$, $e_1 >> e_2$ or $e_1 << e_2$, where e_1 and e_2 are constant expressions that evaluate to an integer value.

- An expression of one of the forms $e_1 + e_2$, e_1 - e_2, e_1 * e_2, e_1 / e_2, $e_1 > e_2$, $e_1 < e_2$, $e_1 >= e_2$, $e_1 <= e_2$ or e_1 % e_2, where e_1 and e_2 are constant expressions that evaluate to a numeric value.

literal:
  nullLiteral |
booleanLiteral |
numericLiteral |
stringLiteral |
mapLiteral |
listLiteral
;

10.2 Null

The reserved word **null** denotes the *null object*.

nullLiteral:
  null
;

The null object is the sole instance of the built-in class Null. Attempting to instantiate Null causes a run-time error. It is a compile-time error for a class or interface attempt to extend or implement Null. Invoking a method on **null** yields a NullPointerException unless the method is explicitly implemented by class Null.

The static type of **null** is $\bot$.

The decision to use $\bot$ *instead of* Null *allows* ***null*** *to be be assigned everywhere without complaint by the static checker.*

Here is one way in which one might implement class Null:

```
class Null {
factory Null._() throw "cannot be instantiated";
noSuchMethod(InvocationMirror msg) {
throw new NullPointerException();
}
/* other methods, such as == */
}
```

10.3 Numbers

A *numeric literal* is either a decimal or hexadecimal integer of arbitrary size, or a decimal double.

numericLiteral:
  NUMBER |

```
  HEX_NUMBER
;

NUMBER:
   DIGIT+ ('.' DIGIT*)? EXPONENT? |
  '.' DIGIT+ EXPONENT?  |

EXPONENT:
   ('e' | 'E') ('+' | '-')? DIGIT+
;

HEX_NUMBER:
   '0x' HEX_DIGIT+ |
  '0X' HEX_DIGIT+
;

HEX_DIGIT:
   'a'..'f' |
  'A'..'F' |
  DIGIT
;
```

If a numeric literal begins with the prefix '0x' or '0X', it denotes the hexadecimal integer represented by the part of the literal following '0x' (respectively '0X'). Otherwise, if the numeric literal does not include a decimal point denotes it denotes a decimal integer. Otherwise, the numeric literal denotes a 64 bit double precision floating point number as specified by the IEEE 754 standard.

Integers are not restricted to a fixed range. Dart integers are true integers, not 32 bit or 64 bit or any other fixed range representation. Their size is limited only by the memory available to the implementation.

An *integer literal* is either a hexadecimal integer literals or a decimal integer literal. The static type of an integer literal is int. A *literal double* is a numeric literal that is not an integer literal. The static type of a literal double is double.

10.4 Booleans

The reserved words **true** and **false** denote objects that represent the boolean values true and false respectively. They are the *boolean literals*.

```
booleanLiteral:
   true |
  false
;
```

Both **true** and **false** implement the built-in interface bool. It is a compile-time error for a class or interface to attempt to extend or implement bool.

It follows that the two boolean literals are the only two instances of bool.

The static type of a boolean literal is bool.

10.4.1 Boolean Conversion

Boolean conversion maps any object *o* into a boolean defined as

(bool v){ **if** (**null** == v) { **throw new** AssertionError('null is not a bool')}; return true === v; }(o)

Boolean conversion is used as part of control-flow constructs and boolean expressions. Ideally, one would simply insist that control-flow decisions be based exclusively on booleans. This is straightforward in a statically typed setting. In a dynamically typed language, it requires a dynamic check. Sophisticated virtual machines can minimize the penalty involved. Alas, Dart must be compiled into Javascript. Boolean conversion allows this to be done efficiently.

*At the same time, this formulation differs radically from Javascript, where most numbers and objects are interpreted as **true**. Darts approach prevents usages such* **if** (a-b) ; *because it does not agree with the low level conventions whereby non-null objects or non-zero numbers are treated as **true**. Indeed, there is no way to derive **true** from a non-boolean object via boolean conversion, so this kind of low level hackery is nipped in the bud.*

*Dart also avoids the strange behaviors that can arise due to the interaction of boolean conversion with autoboxing in Javascript. A notorious example is the situation where **false** can be interpreted as **true**. In Javascript, booleans are not objects, and instead are autoboxed into objects where needed. If **false** gets autoboxed into an object, that object can be coerced into **true** (as it is a non-null object).*

10.5 Strings

A *string* is a sequence of valid unicode code points.

stringLiteral:
'@'? MULTI_LINE_STRING |
SINGLE_LINE_STRING
;

A string can be either a single line string or a multiline string.

SINGLE_LINE_STRING:
'"' ' STRING_CONTENT_DQ* ' " ' |
' ' STRING_CONTENT_SQ* '\'' |
'@' " ' ((' ' | NEWLINE))* ' ' ' |
'@' ' " ' ((' " ' | NEWLINE))* ' " '
;

A single line string may not span more than one line of source code. A single line string is delimited by either matching single quotes or matching double quotes.

Hence, abc and abc are both legal strings, as are 'He said "To be or not to be" did he not?' and "He said 'To be or not to be' didn't he". However "This ' is not a valid string, nor is 'this".

MULTI_LINE_STRING:
' """ ' .* ' """ ' |
' ' ' .* ' ' ' '
;

ESCAPE_SEQUENCE:
\ n |
\ r |
\ f |
\ b |
\ t |
\ v |
\ x HEX_DIGIT HEX_DIGIT |
\ u HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT |
\ u{ HexDigitSequence }
;

HEX_DIGIT_SEQUENCE:
HEX_DIGIT HEX_DIGIT? HEX_DIGIT? HEX_DIGIT? HEX_DIGIT? HEX_DIGIT? HEX_DIGIT?
;

Multiline strings are delimited by either matching triples of single quotes or matching triples of double quotes.

Strings support escape sequences for special characters. The escapes are:

- \n for newline, equivalent to \x0A.
- \r for carriage return, equivalent to \x0D.
- \f for form feed, equivalent to \x0C.
- \b for backspace, equivalent to \x08.
- \t for tab, equivalent to \x09.
- \v for vertical tab, equivalent to \x0B
- \x $HexDigit_1$ $HexDigit_2$, equivalent to \u{$HexDigit_1$ $HexDigit_2$}.

- \u $HexDigit_1$ $HexDigit_2$ $HexDigit_3$ $HexDigit_4$, equivalent to \u{$HexDigit_1$ $HexDigit_2$ $HexDigit_3$ $HexDigit_4$}.
- \u{*HexDigitSequence*} is the unicode scalar value represented by the HexDigitSequence. It is a compile-time error if the value of the *HexDigitSequence* is not a valid unicode scalar value.
- $ indicating the beginning of an interpolated expression.
- Otherwise, \$k$ indicates the character k for any k not in $\{n, r, f, b, t, v, x, u\}$.

It is a compile-time error if a string literal to contains a character sequence of the form \x that is not followed by a sequence of two hexadecimal digits. It is a compile-time error if a string literal to contains a character sequence of the form \u that is not followed by either a sequence of four hexadecimal digits, or by curly brace delimited sequence of hexadecimal digits.

However, any string may be prefixed with the character @, indicating that it is a *raw string*, in which case no escapes are recognized.

```
STRING_CONTENT_DQ:
  ~( '\' | '"' | '$' | NEWLINE ) |
 '\' ~( NEWLINE ) |
 STRING_INTERPOLATION
;

STRING_CONTENT_SQ:
  ~( '\' | "'" | '$' | NEWLINE ) |
 '\ ~( NEWLINE ) |
 STRING_INTERPOLATION
;

NEWLINE:
  '\ n' |
 '\ r'
;
```

All string literals implement the built-in interface String. It is a compile-time error for a class or interface to attempt to extend or implement String. The static type of a string literal is String.

10.5.1 String Interpolation

It is possible to embed expressions within string literals, such that the these expressions are evaluated, and the resulting values are converted into strings and concatenated with the enclosing string. This process is known as *string interpolation*.

STRING_INTERPOLATION:
'$' IDENTIFIER_NO_DOLLAR |
'$' '{' Expression '}'
;

The reader will note that the expression inside the interpolation could itself include strings, which could again be interpolated recursively.

An unescaped $ character in a string signifies the beginning of an interpolated expression. The $ sign may be followed by either:

- A single identifier *id* that must not contain the $ character.
- An expression e delimited by curly braces.

The form $id is equivalent to the form ${id}. An interpolated string '$s_1$${e}s_2' is equivalent to 's_1' + e.toString() + 's_2'. Likewise an interpolated string "$s_1$${e}s_2" is equivalent to "s_1" + e.toString() + "s_2", assuming + is the string concatenation operator.

The string interpolation syntax is designed to be familiar and easy to use, if somewhat awkward to parse. The intent is to encourage its use over alternatives such as $s1 + s2$. In a dynamically typed language, the use of the + operator requires dynamic dispatch. In contrast, in the case of string interpolation we can statically determine that the string concatenation operation is required, making the operation more efficient. Even more importantly, it helps the system to determine if other uses of + are numeric, helping the implementation speed up those operations. This is especially crucial for a language that must be efficiently compiled into Javascript.

10.6 Lists

A *list literal* denotes a list, which is an integer indexed collection of objects.

listLiteral:
const? typeArguments? '[' (expressionList ', '?)? ']'
;

A list may contain zero or more objects. The number of elements in a list is its size. A list has an associated set of indices. An empty list has an empty set of indices. A non-empty list has the index set $\{0 \ldots n-1\}$ where n is the size of the list. It is a runtime error to attempt to access a list using an index that is not a member of its set of indices.

If a list literal begins with the reserved word **const**, it is a *constant list literal* and it is computed at compile-time. Otherwise, it is a *run-time list literal* and it is evaluated at run-time.

It is a compile-time error if an element of a constant list literal is not a compile-time constant. It is a compile-time error if the type argument of a constant list literal includes a type variable.

The value of a constant list literal **const** $< E > [e_1 \dots e_n]$ is an object a that implements the built-in interface $List < E >$. The ith element of a is v_{i+1}, where v_i is the value of the compile-time expression e_i. The value of a constant list literal **const** $[e_1 \dots e_n]$ is defined as the value of the constant list literal **const**$<$ **Dynamic** $> [e_1 \dots e_n]$.

Let $list_1 =$ **const** $< V > [e_{11} \dots e_{1n}]$ and $list_2 =$ **const** $< U > [e_{21} \dots e_{2n}]$ be two constant list literals and let the elements of $list_1$ and $list_2$ evaluate to $o_{11} \dots o_{1n}$ and $o_{21} \dots o_{2n}$ respectively. Iff $o_{1i} === o_{2i}$ for $i \in 1..n$ and $V = U$ then $list_1 === list_2$.

In other words, constant list literals are canonicalized.

A run-time list literal $< E > [e_1 \dots e_n]$ is evaluated as follows:

- First, the expressions $e_1 \dots e_n$ are evaluated in order they appear in the program, yielding objects $o_1 \dots o_n$.
- A fresh instance (7.5.1) a that implements the built-in interface $List < E >$ is allocated.
- The ith element of a is set to $o_{i+1}, 0 \leq i \leq n$.
- The result of the evaluation is a.

Note that this document does not specify an order in which the elements are set. This allows for parallel assignments into the list if an implementation so desires. The order can only be observed in checked mode: if element i is not a subtype of the element type of the list, a dynamic type error will occur when $a[i]$ is assigned o_{i-1}.

A runtime list literal $[e_1 \dots e_n]$ is evaluated as $<$ **Dynamic** $> [e_1 \dots e_n]$.

There is no restriction precluding nesting of list literals. It follows from the rules above that $< List < int >> [[1, 2, 3][4, 5, 6]]$ is a list with type parameter $List < int >$, containing two lists with type parameter **Dynamic**.

The static type of a list literal of the form **const**$< E > [e_1 \dots e_n]$ or the form $< E > [e_1 \dots e_n]$ is $List < E >$. The static type a list literal of the form **const** $[e_1 \dots e_n]$ or the form $[e_1 \dots e_n]$ is $List <$ **Dynamic** $>$.

It is tempting to assume that the type of the list literal would be computed based on the types of its elements. However, for mutable lists this may be unwarranted. Even for constant lists, we found this behavior to be problematic. Since compile-time is often actually runtime, the runtime system must be able to perform a complex least upper bound computation to determine a reasonably precise type. It is better to leave this task to a tool in the IDE. It is also much more uniform (and therefore predictable and understandable) to insist that whenever types are unspecified they are assumed to be the unknown type ***Dynamic****.*

10.7 Maps

A *map literal* denotes a map from strings to objects.

mapLiteral:
  const? '{' (mapLiteralEntry (', ' mapLiteralEntry)* ', '?)? '}'
;

mapLiteralEntry:
  identifier ':' expression |
 stringLiteral ':' expression
;

A *map literal* consists of zero or more entries. Each entry has a *key*, which is a string, and a *value*, which is an object. The key of an entry may be specified via an identifier or via a compile-time constant string. If the key is specified via an identifier id, the specification is interpreted as if it was the string 'id'.

The use of identifiers as keys to literal maps is not implemented at the time of this writing.

If a map literal begins with the reserved word **const**, it is a *constant map literal* and it is computed at compile-time. Otherwise, it is a *run-time map literal* and it is evaluated at run-time.

It is a compile-time error if either a key or a value of an entry in a constant map literal is not a compile-time constant. It is a compile-time error if the type argument of a constant map literal includes a type variable.

The value of a constant map literal $\textbf{const} < V > \{k_1 : e_1 \ldots k_n : e_n\}$ is an object m that implements the built-in interface $Map < String, V >$. The entries of m are $u_i : v_i, i \in 1..n$, where u_i is the value of the compile-time expression k_i and v_i is the value of the compile-time expression e_i. The value of a constant map literal $\textbf{const}\ \{k_1 : e_1 \ldots k_n : e_n\}$ is defined as the value of a constant map literal $\textbf{const} < \textbf{Dynamic} > \{k_1 : e_1 \ldots k_n : e_n\}$.

As specified, a typed map literal takes only one type parameter. If we generalize literal maps so they can have keys that are not strings, we would need two parameters. The implementaion currently insists on two type parameters.

Let $map_1 = \textbf{const} < V > \{k_{11} : e_{11} \ldots k_{1n} : e_{1n}\}$ and $map_2 = \textbf{const} < U > \{k_{21} : e_{21} \ldots k_{2n} : e_{2n}\}$ be two constant map literals. Let the keys of map_1 and map_2 evaluate to $s_{11} \ldots s_{1n}$ and $s_{21} \ldots s_{2n}$ respectively, and let the elements of map_1 and map_2 evaluate to $o_{11} \ldots o_{1n}$ and $o_{21} \ldots o_{2n}$ respectively. Iff $o_{1i} === o_{2i}$ and $s_{1i} === s_{2i}$ for $i \in 1..n$, and $V = U$ then $map_1 === map_2$.

In other words, constant map literals are canonicalized.

A runtime map literal $< V > \{k_1 : e_1 \ldots k_n : e_n\}$ is evaluated as follows:

- First, the expressions $e_1 \ldots e_n$ are evaluated in left to right order, yielding objects $o_1 \ldots o_n$.
- A fresh instance (7.5.1) m that implements the built-in interface $Map < String, V >$ is allocated.

- Let u_i be the value of the compile-time constant string specified by k_i. An entry with key u_i and value o_i is added to $m, 0 \leq i \leq n$.
- The result of the evaluation is m.

A runtime map literal $\{k_1 : e_1 \ldots k_n : e_n\}$ is evaluated as $<$ **Dynamic** $>$ $\{k_1 : e_1 \ldots k_n : e_n\}$.

It is a static warning if the values of any two keys in a map literal are equal.

A map literal is ordered: iterating over the keys and/or values of the maps always happens in the order the keys appeared in the source code.

Of course, if a key repeats, the order is defined by first occurrence, but the value is defined by the last.

The static type of a map literal of the form **const**$< V > \{k_1 : e_1 \ldots k_n : e_n\}$ or the form $< V > \{k_1 : e_1 \ldots k_n : e_n\}$ is $Map < String, V >$. The static type a map literal of the form **const**$\{k_1 : e_1 \ldots k_n : e_n\}$ or the form $\{k_1 : e_1 \ldots k_n : e_n\}$ is $Map <$ *String*, **Dynamic** $>$.

10.8 Function Expressions

A *function literal* is an object that encapsulates an executable unit of code.

functionExpression:
(returnType? identifier)? formalParameterList functionExpressionBody
;

functionExpressionBody:
'=>' expression |
block
;

A function literal implements the built-in interface Function.

The static type of a function literal of the form $(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]) => e$ or of the form $id(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]) => e$ is $(T_1 \ldots, T_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]) \rightarrow T_0$, where T_0 is the static type of e. In any case where $T_i, 1 \leq i \leq n + k$, is not specified, it is considered to have been specified as **Dynamic**.

The static type of a function literal of the form $T_0\ id(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}])\{s\}$ or of the form $T_0\ id(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]) => e$ is $(T_1 \ldots, T_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]) \rightarrow T_0$. In any case where $T_i, 1 \leq i \leq n + k$, is not specified, it is considered to have been specified as **Dynamic**.

The static type of a function literal of the form $id(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}])\{s\}$ or of the form $(T_1\ a_1, \ldots, T_n\ a_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}])\{s\}$ is $(T_1 \ldots, T_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]) \rightarrow$ **Dynamic**. In any case where $T_i, 1 \leq i \leq n + k$, is not specified, it is considered to have been specified as **Dynamic**.

10.9 This

The reserved word **this** denotes the target of the current instance member invocation.

thisExpression:
  this
;

The static type of **this** is the interface of the immediately enclosing class.
We do not support self-types at this point.

10.10 Instance Creation

Instance creation expressions invoke constructors to produce instances.

It is a static warning to instantiate an abstract class (7.1.1). It is a static type warning if any of the type arguments to a constructor of a generic type invoked by a new expression or a constant object expression are not subtypes of the bounds of the corresponding formal type parameters.

10.10.1 New

The *new expression* invokes a constructor (7.5).

newExpression:
  new type ('.' identifier)? arguments
;

Let e be a new expression of the form **new** $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ or the form **new** $T(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$. It is a compile-time error if T is not a class or interface optionally followed by type arguments.

If e of the form **new** $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ it is a compile-time error if $T.id$ is not the name of a constructor declared by the type T. If e of the form **new** $T(a_1, \ldots, a_n)$ it is a compile-time error if the type T does not declare a constructor with the same name as the declaration of T.

If T is a parameterized type (13.8) $S < U_1, \ldots, U_m >$, let $R = S$. It is a compile-time error if S is not a generic (9) type with m type parameters. If T is not a parameterized type, let $R = T$. If R is an interface, let C be the factory class (8.3) of R. Otherwise let $C = R$. Furthermore, if e is of the form **new** $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ then let q be the constructor of C that corresponds (8.3) to the constructor $T.id$, otherwise let q be the constructor of C that corresponds to the constructor T. Finally, if C is generic but T is not a parameterized type, then for $i \in 1..m$, let $V_i =$ **Dynamic**, otherwise let $V_i = U_i$.

Evaluation of e proceeds as follows:

First, if q is a generative constructor (7.5.1), then:

Let T_i be the type parameters of C (if any) and let $B_{i1}, \ldots, B_{ik_i}$ be the bounds of $T_i, 1 \leq i \leq m$. It is a dynamic type error if, in checked mode, V_i is not a subtype of $[V_1, \ldots, V_m/T_1, \ldots, T_m]B_{ij}, 1 \leq j \leq k_i, 1 \leq i \leq m$.

A fresh instance (7.5.1), i, of class C is allocated. For each instance variable f of i, if the variable declaration of f has an initializer, then f is bound to that value (which is necessarily a compile-time constant). Otherwise f is bound to **null**.

Next, the argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated. Next, the initializer list of q is executed with respect to the bindings that resulted from the evaluation of the argument list, and with **this** bound to i and the type parameters (if any) of C bound to the actual type arguments $V_1, \ldots, V_m$. Then, the body of q is executed with respect to the bindings that resulted from the evaluation of the argument list. The result of the evaluation of e is i.

Otherwise, if q is a redirecting constructor (7.5.1), then: The argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated. Then, the redirect clause of q is executed with respect to the bindings that resulted from the evaluation of the argument list and the type parameters (if any) of C bound to the actual type arguments $V_1, \ldots, V_m$ resulting in an object i that is necessarily the result of another constructor call. The result of the evaluation of e is i.

Otherwise, q is a factory constructor (7.5.2). Then: Let T_i be the type parameters of q (if any) and let $B_{i1}, \ldots, B_{ik_i}$ be the bounds of $T_i, 1 \leq i \leq m$. It is a dynamic type error if, in checked mode, V_i is not a subtype of $[V_1, \ldots, V_m/T_1, \ldots, T_m]B_{ij}, 1 \leq j \leq k_i, 1 \leq i \leq m$. The argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated. Then, the body of q is executed with respect to the bindings that resulted from the evaluation of the argument list and the type parameters (if any) of q bound to the actual type arguments $V_1, \ldots, V_m$ resulting in an object i. The result of the evaluation of e is i.

The static type of a new expression of either the form **new** $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ or the form **new** $T(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is T. It is a static warning if the static type of $a_i, 1 \leq i \leq n+k$ may not be assigned to the type of the corresponding formal parameter of the constructor $T.id$ (respectively T).

10.10.2 Const

A *constant object expression* invokes a constant constructor (7.5.3).

constObjectExpression:
  const type ('.' identifier)? arguments
;

Let e be a constant object expression of the form **const** $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ or the form **const** $T(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} :$

a_{n+k}). It is a compile-time error if T is not a class or interface optionally followed by type arguments.

If e of the form **const** $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ it is a compile-time error if $T.id$ is not the name of a constructor declared by the type T. If e of the form **const** $T(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ it is a compile-time error if the type T does not declare a constructor with the same name as the declaration of T.

If T is a parameterized type (13.8) $S < U_1, \ldots, U_m >$, let $R = S$. It is a compile-time error if S is not a generic (9) type with m type parameters. If T is not a parameterized type, let $R = T$. If R is an interface, let C be the factory class (8.3) of R. Otherwise let $C = R$. Finally, if C is generic but T is not a parameterized type, then for all $i \in 1..m$, let $V_i =$ **Dynamic**, otherwise let $V_i = U_i$.

Evaluation of e proceeds as follows:

First, if e is of the form **const** $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ then let i be the value of the expression **new** $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$. Otherwise, e must be of the form **const** $T(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$, in which case let i be the result of evaluating **new** $T(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$. Then:

- If during execution of the program, a constant object expression has already evaluated to an instance j of class C with type arguments $V_i, 1 \leq i \leq m$, then:
  - For each instance variable f of i, let v_{if} be the value of the f in i, and let v_{jf} be the value of the field f in j. If $v_{if} === v_{jf}$ for all fields f in i, then the value of e is j, otherwise the value of e is i.
- Otherwise the value of e is i.

In other words, constant objects are canonicalized. In order to determine if an object is actually new, one has to compute it; then it can be compared to any cached instances. If an equivalent object exists in the cache, we throw away the newly created object and use the cached one. Objects are equivalent if they have identical fields and identical type arguments. Since the constructor cannot induce any side effects, the execution of the constructor is unobservable. The constructor need only be executed once per call site, at compile-time.

The static type of a constant object expression of either the form **const** $T.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ or the form **const** $T(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is T. It is a static warning if the static type of $a_i, 1 \leq i \leq n + k$ may not be assigned to the type of the corresponding formal parameter of the constructor $T.id$ (respectively T).

10.11 Spawning an Isolate

Spawning an isolate is accomplished via what is syntactically an ordinary library call. However, such calls have the semantic effect of creating a new isolate with its own memory and thread of control.

10.12 Property Extraction

Property extraction allows for a member of an object to be concisely extracted from the object. If o is an object, and if m is the name of a method member of o, then $o.m$ is defined to be equivalent to $(r_1, \ldots, r_n, [p_1 = d_1, \ldots, p_k = d_k])\{$**return** $o.m(r_1, \ldots, r_n, p_1, \ldots, p_k);\}$ if m has required parameters $r_1, \ldots, r_n$, and named parameters $p_1, \ldots, p_k$ with defaults $d_1, \ldots, d_k$.

Otherwise, if m is the name of a getter (7.2) member of o (declared implicitly or explicitly) then $o.m$ evaluates to the result of invoking the getter.

Observations:

1. One cannot extract a getter or a setter.
2. One can tell whether one implemented a property via a method or via field/getter, which means that one has to plan ahead as to what construct to use, and that choice is reflected in the interface of the class.

10.13 Function Invocation

Function invocation occurs in the following cases: when a function expression (10.8) is invoked (10.13.4), when a method is invoked (10.14) or when a constructor is invoked (either via instance creation (10.10), constructor redirection (7.5.1) or super initialization). The various kinds of function invocation differ as to how the function to be invoked, f, is determined, as well as whether **this** is bound. Once f has been determined, the formal parameters of f are bound to corresponding actual arguments. The body of f is then executed with the aforementioned bindings. Execution of the body terminates when the first of the following occurs:

- An uncaught exception is thrown.
- A return statement (11.10) immediately nested in the body of f is executed.
- The last statement of the body completes execution.

10.13.1 Actual Argument List Evaluation

Function invocation involves evaluation of the list of actual arguments to the function and binding of the results to the functions formal parameters.

arguments:
  '(' argumentList? ')'
;

argumentList:
  namedArgument (', ' namedArgument)* |
  expressionList (', ' namedArgument)*

;

namedArgument:
label expression
;

Evaluation of an actual argument list of the form $(a_1, \ldots, a_m, q_1 : a_{m+1}, \ldots, q_l : a_{m+l})$ proceeds as follows:

The arguments $a_1, \ldots, a_{m+l}$ are evaluated in the order they appear in the program, yielding objects $o_1, \ldots, o_{m+l}$.

Simply stated, an argument list consisting of m positional arguments and l named arguments is evaluated from left to right.

10.13.2 Binding Actuals to Formals

Let f be a function, let $p_1 \ldots p_n$ be the positional parameters of f and let $p_{n+1}, \ldots, p_{n+k}$ be the named parameters declared by f.

An evaluated actual argument list $o_1 \ldots o_{m+l}$ derived from an actual argument list of the form $(a_1, \ldots, a_m, q_1 : a_{m+1}, \ldots, q_l : a_{m+l})$ is bound to the formal parameters of f as follows:

Again, we have an argument list consisting of m positional arguments and l named arguments. We have a function with n required parameters and k named parameters. The number of positional arguments must be at least as large as the number of required parameters. All named arguments must have a corresponding named parameter. You may not provide the same parameter as both a positional and a named argument. If an optional parameter has no corresponding argument, it gets its default value. In checked mode, all arguments must belong to subtypes of the type of their corresponding formal.

If $m < n$, a run-time error occurs. Furthermore, each $q_i, 1 \leq i \leq l$, must be a member of the set $\{p_{m+1}, \ldots, p_{m+k}\}$ or a run time error occurs. Then p_i is bound to $o_i, i \in 1..m$, and q_j is bound to $o_{m+j}, j \in 1..l$. All remaining formal parameters of f are bound to their default values.

All of these remaining parameters are necessarily optional and thus have default values.

It is a dynamic type error if, in checked mode, o_i is not **null** and the actual type (13.8.1) of p_i is not a supertype of the type of $o_i, i \in 1..m$. It is a dynamic type error if, in checked mode, o_{l+j} is not **null** and the actual type (13.8.1) of q_j is not a supertype of the type of $o_{m+j}, j \in 1..l$.

10.13.3 Unqualified Invocation

An unqualified function invocation i has the form $id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$, where id is an identifier.

If there exists a lexically visible declaration named id, let f_{id} be the innermost such declaration. Then:

- If f_{id} is a local function, a library function, a library or static getter or a variable then i is interpreted as a function expression invocation (10.13.4).
- Otherwise, if f_{id} is a static method of the enclosing class C, i is equivalent the static method invocation $C.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

Otherwise, if there is an accessible (3.2) static method named id declared in a superclass S of the immediately enclosing class C then i is equivalent to the static method invocation $S.id(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

Unqualified access to static methods of superclasses is inconsistent with the idea that static methods are not inherited. It is not particularly necessary and may be restricted in future versions.

Otherwise, i is equivalent to the ordinary method invocation **this**.$id((a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

10.13.4 Function Expression Invocation

A function expression invocation i has the form $e_f(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k}])$, where e_f is an expression. If e_f is an identifier id, then id must necessarily denote a local function, a library function, a library or static getter or a variable as described above, or i is not considered a function expression invocation. If e_f is a property access expression, then i is treated as an ordinary method invocation (10.14.1). Otherwise:

Evaluation of a function expression invocation $e_f(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ proceeds as follows:

First, e_f is evaluated to a value v_f of type T_f. If T_f is not of a function type, an ObjectNotAClosure is thrown. Next, the argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated. The body of v_f is then executed with respect to the bindings that resulted from the evaluation of the argument list. The value of i is the value returned after v_f is executed.

It is a static warning if the static type F of e_f may not be assigned to a function type F. If F is not a function type, the static type of i is **Dynamic**. Otherwise:

- the static type of i is the declared return type of F.
- It is a static warning if the static type of $a_i, 1 \leq i \leq n + k$ may not be assigned to the type of the corresponding formal parameter of F.

10.14 Method Invocation

Method invocation can take several forms as specified below.

10.14.1 Ordinary Invocation

An ordinary method invocation i has the form $o.m(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

The result of a lookup of a method m in object o with respect to library L is the result of a lookup of method m in class C with respect to library L, where C is the class of o.

The result of a lookup of method m in class C with respect to library L is: If C declares an instance method named m that is accessible to L, then that method is the result of the lookup. Otherwise, if C has a superclass S, then the result of the lookup is the result of looking up m in S with respect to L. Otherwise, we say that the method lookup has failed.

Evaluation of an ordinary method invocation i of the form $o.m(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ proceeds as follows:

First, the expression o is evaluated to a value v_o. Next, the argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated. Let f be the result of looking up method m in v_o with respect to the current library L. If the method lookup succeeded, the body of f is executed with respect to the bindings that resulted from the evaluation of the argument list, and with **this** bound to v_o. The value of i is the value returned after f is executed.

If the method lookup has failed, then let g be the result of looking up getter m in v_o with respect to L. If the getter lookup succeeded, let v_g be the value of the getter invocation $o.m$. If v_g is a function then it is executed with respect to the bindings of the evaluated argument list. The value of i is the value returned after v_g is executed.

If v_g is not a function then an ObjectNotAClosure is thrown.

Should we just invoke call() on the result and let noSuchMethod take its course? Or do all objects have a call() method that by default throws ObjectNotAClosure?

If the getter lookup has also failed, then a new instance im of the predefined interface InvocationMirror is created by calling its factory constructor with arguments m, **this**, $[e_1, \ldots, e_n]$ and $\{x_{n+1} : e_{n+1}, \ldots, x_{n+k} : e_{n+k}\}$. Then the method noSuchMethod() is looked up in o and invoked with argument im, and the result of this invocation is the result of evaluating i.

Notice that the wording carefully avoids re-evaluating the receiver o and the arguments a_i.

Let T be the static type of o. It is a static type warning if T does not have an accessible (3.2) instance member named m. If $T.m$ exists, it is a static type warning if the type F of $T.m$ is not a function type. If $T.m$ does not exist, or if F may not be assigned to a function type, the static type of i is **Dynamic**; otherwise:

- the static type of i is the declared return type of F.
- Let T_i be the static type of $a_i, i \in 1..n + k$. It is a static warning if F is not a supertype of $(T_1, \ldots, t_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]) \rightarrow \bot$.

10.14.2 Static Invocation

A static method invocation i has the form $C.m(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

It is a compile-time error if C does not denote a class in the current scope. It is a compile-time error if C does not declare a static method or getter m.

Note the requirement that C *declare* the method. This means that static methods declared in superclasses of C cannot be invoked via C.

Evaluation of i proceeds as follows:

First, the argument list$(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated. If the member m declared by C is a getter, then let f be the result of evaluating the getter invocation $C.m$. If v_g is not a function then an ObjectNotAClosure is thrown. Otherwise, let f be the the method m declared in class C.

The body of f is then executed with respect to the bindings that resulted from the evaluation of the argument list. The value of i is the value returned after the body of f is executed.

It is a static type warning if the type F of $C.m$ may not be assigned to a function type. If F is not a function type, the static type of i is **Dynamic**. Otherwise:

- the static type of i is the declared return type of F.
- Let T_i be the static type of $a_i, i \in 1..n+k$. It is a static warning if F is not a supertype of $(T_1, \ldots, t_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]) \rightarrow \bot$.

10.14.3 Super Invocation

A super method invocation has the form

super.$m(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$.

Evaluation of an super method invocation i of the form

super.$m(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$

proceeds as follows:

First, the argument list $(a_1, \ldots, a_n, x_{n+1} : a_{n+1}, \ldots, x_{n+k} : a_{n+k})$ is evaluated. Let f be the result of looking up method m in S with respect to the current library L, the superclass of the class of **this**. If the method lookup succeeded, the body of f is executed with respect to the bindings that resulted from the evaluation of the argument list, and with **this** bound to the current value of **this**. The value of i is the value returned after f is executed.

If the method lookup has failed, then let g be the result of looking up getter m in S with respect to L. If the getter lookup succeeded, let v_g be the value of the getter invocation **super**.m. If v_g is a function then it is called with the evaluated argument list. The value of i is the value returned after v_g is executed.

If v_g is not a function then an ObjectNotAClosure is thrown.

If getter lookup has also failed, then a new instance im of the predefined interface InvocationMirror is created by calling its factory constructor with arguments m, **this**, $[e_1, \ldots, e_n]$ and $\{x_{n+1} : e_{n+1}, \ldots, x_{n+k} : e_{n+k}\}$. Then the method noSuchMethod() is looked up in S and invoked with argument im, and the result of this invocation is the result of evaluating i.

It is a compile-time error if a super method invocation occurs in a top-level function or variable initializer, in class Object, or in a static method or variable initializer.

It is a static type warning if S does not have an accessible (3.2) instance member named m. If $S.m$ exists, it is a static type warning if the type F of $S.m$ may not be assigned to a function type. If $S.m$ does not exist, or if F is not a function type, the static type of i is **Dynamic**; otherwise:

- the static type of i is the declared return type of F.
- Let T_i be the static type of $a_i, i \in 1..n+k$. It is a static warning if F is not a supertype of $(T_1, \ldots, t_n, [T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}]) \rightarrow \bot$.

10.14.4 Sending Messages

Messages are the sole means of communication among isolates. Messages are sent by invoking specific methods in the Dart libraries; there is no specific syntax for sending a message.

In other words, the methods supporting sending messages embody primitives of Dart that are not accessible to ordinary code, much like the methods that spawn isolates.

10.15 Getter Invocation

A getter invocation provides access to the value of a property.

The result of a lookup of a getter (respectively setter) m in object o with respect to library L is the result of looking up getter (respectively setter) m in class C with respect to L, where C is the class of o.

The result of a lookup of a getter (respectively setter) m in class C with respect to library L is: If C declares an instance getter (respectively setter) named m that is accessible to L, then that getter (respectively setter) is the result of the lookup. Otherwise, if C has a superclass S, then the result of the lookup is the result of looking up getter (respectively setter) m in S with respect to L. Otherwise, we say that the lookup has failed.

Evaluation of a getter invocation i of the form $e.m$ proceeds as follows:

First, the expression e is evaluated to an object o. Then, the getter function (7.2) m is looked up in o with respect to the current library, and its body is executed with **this** bound to o. The value of the getter invocation expression is the result returned by the call to the getter function.

If the getter lookup has failed, then a new instance im of the predefined interface InvocationMirror is created by calling its factory constructor with arguments 'get m', **this**, [] and {}. Then the method noSuchMethod() is looked up in o and invoked with argument im, and the result of this invocation is the result of evaluating i.

Let T be the static type of e. It is a static type warning if T does not have a getter named m. The static type of i is the declared return type of $T.m$, if $T.m$ exists; otherwise the static type of i is **Dynamic**.

Evaluation of a getter invocation i of the form $C.m$ proceeds as follows:

The getter function $C.m$ is invoked. The value of i is the result returned by the call to the getter function.

It is a compile-time error if there is no class C in the enclosing lexical scope of i, or if C does not declare, implicitly or explicitly, a getter named m. The static type of i is the declared return type of $C.m$.

Evaluation of a top-level getter invocation i of the form m, where m is an identifier, proceeds as follows:

The getter function m is invoked. The value of i is the result returned by the call to the getter function.

The static type of i is the declared return type of m.

10.16 Assignment

An assignment changes the value associated with a mutable variable or property.

assignmentOperator:
'=' |
compoundAssignmentOperator
;

Evaluation of an assignment of the form $v = e$ proceeds as follows:

If there is no declaration d with name v in the lexical scope enclosing the assignment, then the assignment is equivalent to the assignment **this**.$v = e$. Otherwise, let d be the innermost declaration whose name is v, if it exists.

If d is the declaration of a local or library variable, the expression e is evaluated to an object o. Then, the variable v is bound to o. The value of the assignment expression is o.

Otherwise, if d is the declaration of a static variable in class C, then the assignment is equivalent to the assignment $C.v = e$.

Otherwise, the assignment is equivalent to the assignment **this**.$v = e$.

It is a dynamic type error if, in checked mode, o is not **null** and the interface induced by the class of o is not a subtype of the actual type (13.8.1) of v.

It is a static type warning if the static type of e may not be assigned to the static type of v.

Evaluation of an assignment of the form $C.v = e$ proceeds as follows:

The expression e is evaluated to an object o. Then, the setter $C.v$ is invoked with its formal parameter bound to o. The value of the assignment expression is o.

It is a compile-time error if there is no class C in the enclosing lexical scope of assignment, or if C does not declare, implicitly or explicitly, a setter v. It is a dynamic type error if, in checked mode, o is not **null** and the interface induced by the class of o is not a subtype of the static type of $C.v$.

It is a static type warning if the static type of e may not be assigned to the static type of $C.v$.

Evaluation of an assignment of the form $e_1.v = e_2$ proceeds as follows:

The expression e_1 is evaluated to an object o_1. Then, the expression e_2 is evaluated to an object o_2. Then, the setter v is looked up in o_1 with respect to the current library, and its body is executed with its formal parameter bound to o_2 and **this** bound to o_1.

If the setter lookup has failed, then a new instance im of the predefined interface InvocationMirror is created by calling its factory constructor with arguments 'set v', o_1, $[o_2]$ and {}. Then the method noSuchMethod() is looked up in o_1 and invoked with argument im. The value of the assignment expression is o_2 irrespective of whether setter lookup has failed or succeeded.

It is a dynamic type error if, in checked mode, o_2 is not **null** and the interface induced by the class of o_2 is not a subtype of the actual type of $e_1.v$.

It is a static type warning if the static type of e_2 may not be assigned to the static type of $e_1.v$.

10.16.1 Compound Assignment

A compound assignment of the form v *op* $=e$ is equivalent to $v=v$ *op* e. A compound assignment of the form $C.v$ *op*$=e$ is equivalent to $C.v=C.v$ *op* e. A compound assignment of the form $e_1.v$ *op* $= e_2$ is equivalent to $((x) => x.v = x.v$ *op* $e_2)(e_1)$ where x is a variable that is not used in e_2.

compoundAssignmentOperator:
  '*=' |
  '/=' |
  '~/=' |
  '%=' |
  '+=' |
  '-=' |
  '<<=' |
  '>' '>' '>' '=' ? |
  '<' '<' '=' ? |
  '&=' |
  '^=' |
  '|='
;

10.17 Conditional

A *conditional expression* evaluates one of two expressions based on a boolean condition.

conditionalExpression:
  logicalOrExpression ('?' expression ':' expression)?
;

Evaluation of a conditional expression c of the form $e_1?e_2 : e_3$ proceeds as follows:

First, e_1 is evaluated to an object o_1. In checked mode, it is a dynamic type error if o_1 is not of type bool. Otherwise, o_1 is then subjected to boolean conversion (10.4.1) producing an object r. If r is true, then the value of c is the result of evaluating the expression e_2. Otherwise the value of c is the result of evaluating the expression e_3.

It is a static type warning if the type of e_1 is not bool. The static type of c is the least upper bound (13.8.2) of the static type of e_2 and the static type of e_3.

10.18 Logical Boolean Expressions

The logical boolean expressions combine boolean objects using the boolean conjunction and disjunction operators.

logicalOrExpression:
  logicalAndExpression ('||' logicalAndExpression)*
;

logicalAndExpression:
  bitwiseOrExpression ('&&' bitwiseOrExpression)*
;

A *logical boolean expression* is either a bitwise expression (10.19), or an invocation of a logical boolean operator on an expression e_1 with argument e_2.

Evaluation of a logical boolean expression b of the form $e_1||e_2$ causes the evaluation of e_1; if e_1 evaluates to true, the result of evaluating b is true, otherwise e_2 is evaluated to an object o, which is then subjected to boolean conversion (10.4.1) producing an object r, which is the value of b.

Evaluation of a logical boolean expression b of the form $e_1\&\&e_2$ causes the evaluation of e_1; if e_1 does not evaluate to true, the result of evaluating b is false, otherwise e_2 is evaluated to an object o, which is then subjected to boolean conversion producing an object r, which is the value of b.

The static type of a logical boolean expression is bool.

10.19 Bitwise Expressions

Bitwise expressions invoke the bitwise operators on objects.

bitwiseOrExpression:
  bitwiseXorExpression ('|' bitwiseXorExpression)* |
 super ('|' bitwiseXorExpression)+
;

bitwiseXorExpression:
  bitwiseAndExpression (‘$\hat{}$’ bitwiseAndExpression)* |
 super (‘$\hat{}$’ bitwiseAndExpression)+
;

bitwiseAndExpression:
  equalityExpression (‘&’ equalityExpression)* |
 super (‘&’ equalityExpression)+
;

bitwiseOperator:
  ‘&’ |
 ‘^’ |
 ‘|’
;

A *bitwise expression* is either an equality expression (10.20), or an invocation of a bitwise operator on either **super** or an expression e_1, with argument e_2.

A bitwise expression of the form e_1 *op* e_2 is equivalent to the method invocation $e_1.op(e_2)$. A bitwise expression of the form **super** *op* e_2 is equivalent to the method invocation **super**.op(e_2).

It should be obvious that the static type rules for these expressions are defined by the equivalence above - ergo, by the type rules for method invocation and the signatures of the operators on the type e_1. The same holds in similar situations throughout this specification.

10.20 Equality

Equality expressions test objects for identity or equality.

equalityExpression:
  relationalExpression (equalityOperator relationalExpression)? |
 super equalityOperator relationalExpression
;

equalityOperator:
  ‘==’ |
 ‘!=’ |
 ‘===’ |
 ‘!==’
;

An *equality expression* is either a relational expression (10.21), or an invocation of a equality operator on either **super** or an expression e_1, with argument e_2.

An equality expression of the form $e_1 == e_2$ is equivalent to the method invocation $e_1.==(e_2)$. An equality expression of the form **super** $== e$ is equivalent to the method invocation **super**.$==(e)$.

An equality expression of the form $e_1 \; != e_2$ is equivalent to the expression $!(e_1 == e_2)$. An equality expression of the form **super** $!= e$ is equivalent to the expression $!($**super** $==(e))$.

Evaluation of an equality expression ee of the form $e_1 === e_2$ proceeds as follows:

The expression e_1 is evaluated to an object o_1; then the expression e_2 is evaluated to an object o_2. Next, if o_1 and o_2 are the same object, then ee evaluates to **true**, otherwise ee evaluates to **false**.

An equality expression of the form **super** $=== e$ is equivalent to the expression **this** $=== e$. An equality expression of the form **super** $!== e$ is equivalent to the expression $!($**super** $=== e)$.

An equality expression of the form $e_1 \; !== e_2$ is equivalent to the expression $!(e_1 === e_2)$.

The static type of an equality expression of the form $e_1 == e_2$ is bool.

The static types of other equality expressions follow from the definitions above. The forms $e_1 \; != e_2$, $e_1 \; !== e_2$ and **super** $!= e$ are negations and have static type bool. The expression $e_1 == e_2$ is typed as a method invocation so its static type depends on the operator method declaration.

10.21 Relational Expressions

Relational expressions invoke the relational operators on objects.

relationalExpression:
  shiftExpression (isOperator type | relationalOperator shiftExpression)? |
 super relationalOperator shiftExpression
;

relationalOperator:
  '>' '=' ? |
 '>' |
 '<=' |
 '<'
;

A *relational expression* is either a shift expression (10.22), or an invocation of a relational operator on either **super** or an expression e_1, with argument e_2.

A relational expression of the form $e_1 \; op \; e_2$ is equivalent to the method invocation $e_1.op(e_2)$. A relational expression of the form **super** $op \; e_2$ is equivalent to the method invocation **super**.$op(e_2)$.

10.22 Shift

Shift expressions invoke the shift operators on objects.

shiftExpression:
  additiveExpression (shiftOperator additiveExpression)* |
 super (shiftOperator additiveExpression)+
;

shiftOperator:
  '<<' |
 '>' '>' '>' ? |
 '>' '>' ?
;

A *shift expression* is either an additive expression (10.23), or an invocation of a shift operator on either **super** or an expression e_1, with argument e_2.

A shift expression of the form e_1 *op* e_2 is equivalent to the method invocation $e_1.op(e_2)$. A shift expression of the form **super** *op* e_2 is equivalent to the method invocation **super**.$op(e_2)$.

Note that this definition implies left-to-right evaluation order among shift expressions:

$e_1 << e_2 << e_3$

is evaluated as $(e_1 << e_2).<< (e_3)$ which is equivalent to $(e_1 << e_2) << e_3$. The same holds for additive and multiplicative expressions.

10.23 Additive Expressions

Additive expressions invoke the addition operators on objects.

additiveExpression:
  multiplicativeExpression (additiveOperator multiplicativeExpression)* |
 super (additiveOperator multiplicativeExpression)+
;

additiveOperator:
  '+' |
 '-'
;

An *additive expression* is either a multiplicative expression (10.24), or an invocation of an additive operator on either **super** or an expression e_1, with argument e_2.

An additive expression of the form e_1 *op* e_2 is equivalent to the method invocation $e_1.op(e_2)$. An additive expression of the form **super** *op* e_2 is equivalent to the method invocation **super**.$op(e_2)$.

10.24 Multiplicative Expressions

Multiplicative expressions invoke the multiplication operators on objects.

multiplicativeExpression:
  unaryExpression (multiplicativeOperator unaryExpression)* |
 super (multiplicativeOperator unaryExpression)+
;

multiplicativeOperator:
  '*' |
 '/' |
 '%' |
 '~/'
;

A *multiplicative expression* is either a unary expression (10.25), or an invocation of a multiplicative operator on either **super** or an expression e_1, with argument e_2.

A multiplicative expression of the form e_1 *op* e_2 is equivalent to the method invocation $e_1.op(e_2)$. A multiplicative expression of the form **super** *op* e_2 is equivalent to the method invocation **super**.$op(e_2)$.

10.25 Unary Expressions

Unary expressions invoke unary operators on objects.

unaryExpression:
  prefixExpression |
 postfixExpression |
 unaryOperator **super** |
 '-' **super**
;
|
 incrementOperator assignableExpression
;

A *unary expression* is either a prefix expression (10.26), a postfix expression (10.27), or an invocation of a unary operator on on either **super** or an expression e.

The expression $-e$ is equivalent to the method invocation e.**negate**(). The expressions -**super** is equivalent to the method invocation **super**.**negate**().

10.26 Prefix Expressions

Prefix Expressions invoke prefix operators on objects.

prefixExpression:
  prefixOperator unaryExpression
;

Evaluation of a prefix expression of the form $++e$ is equivalent to $e \mathrel{+}= 1$. Evaluation of a prefix expression of the form $-e$ is equivalent to $e \mathrel{-}= 1$.

10.27 Postfix Expressions

Postfix expressions invoke the postfix operators on objects.

postfixExpression:
  assignableExpression postfixOperator |
  primary selector*
;

postfixOperator:
  incrementOperator
;

incrementOperator:
  '++' |
  '--'
;

A *postfix expression* is either a primary expression, a function, method or getter invocation, or an invocation of a postfix operator on an expression e.

Evaluation of a postfix expression of the form $v{+}{+}$ is equivalent to $()\{\mathsf{var\ r} = v; v = \mathsf{r} + 1; \mathsf{return\ r}\}()$.

The above ensures that if v is a field, the getter gets called exactly once. Likewise in the cases below.

Evaluation of a postfix expression of the form $C.v{+}{+}$ is equivalent to $()\{\mathsf{var\ r} = C.v; C.v = \mathsf{r} + 1; \mathsf{return\ r}\}()$.

A postfix expression of the form $e_1.v{+}{+}$ is equivalent to $(\mathsf{x})\{\mathsf{var\ r} = \mathsf{x.v}; \mathsf{x.v} = \mathsf{r} + 1; \mathbf{return}\ \mathsf{r}\}(e_1)$.

Evaluation of a postfix expression of the form $e{-}{-}$ is equivalent to $e \mathrel{+}= (\text{-}1)$.

10.28 Assignable Expressions

Assignable expressions are expressions that can appear on the left hand side of an assignment.

Of course, if they always appeared as *the left hand side, one would have no need for their value, and the rules for evaluating them would be unnecessary. However, assignable expressions can be subexpressions of other expressions and therefore must be evaluated.*

assignableExpression:
primary (arguments* assignableSelector)+ |
super assignableSelector |
identifier
;

selector:
assignableSelector |
arguments
;

assignableSelector:
'[' expression ']' |
'.' identifier
;

An *assignable expression* is either:

- An identifier.
- An invocation of a method, getter (7.2) or list access operator on an expression e.
- An invocation of a getter or list access operator on **super**.

An assignable expression of the form *id* is evaluated as an identifier expression (10.29).

An assignable expression of the form $e.id(a_1, \ldots, a_n)$ is evaluated as a method invocation (10.14).

An assignable expression of the form $e.id$ is evaluated as a getter invocation (10.15).

An assignable expression of the form $e_1[e_2]$ is evaluated as a method invocation of the operator method [] on e_1 with argument e_2.

An assignable expression of the form super.id is evaluated as a getter invocation.

An assignable expression of the form **super**$[e_2]$ is equivalent to the method invocation **super**.$[e_2]$.

10.29 Identifier Reference

An *identifier expression* consists of a single identifier; it provides access to an object via an unqualified name.

identifier:
IDENTIFIER_NO_DOLLAR |
IDENTIFIER |
BUILT_IN_IDENTIFIER
;

IDENTIFIER_NO_DOLLAR:
IDENTIFIER_START_NO_DOLLAR IDENTIFIER_PART_NO_DOLLAR*
;

IDENTIFIER:
IDENTIFIER_START IDENTIFIER_PART*
;

BUILT_IN_IDENTIFIER:
abstract |
assert |
class |
extends |
factory |
get |
implements |
import |
interface |
is |
library |
negate, |
operator |
set |
source |
static |
typedef
;

IDENTIFIER_START:
IDENTIFIER_START_NO_DOLLAR |
'$'
;

IDENTIFIER_START_NO_DOLLAR:
LETTER |
'_'
;

IDENTIFIER_PART_NO_DOLLAR:
IDENTIFIER_START_NO_DOLLAR |
DIGIT
;

IDENTIFIER_PART:
IDENTIFIER_START |
DIGIT
;

qualified:
identifier ('.' identifier)?
;

Built-in identifiers are identifiers that are used as keywords in Dart, but are not reserved words in Javascript. To minimize incompatibilities when porting Javascript code to Dart, we do not make these into reserved words.

Evaluation of an identifier expression e of the form id proceeds as follows:

Let d be the innermost declaration in the enclosing lexical scope whose name is id. It is a compile-time error if d is a class, interface or type variable. If no such declaration exists in the lexical scope, let d be the declaration of the inherited member named id if it exists. If no such member exists, let d be the declaration of the static member name id declared in a superclass of the current class, if it exists.

- If d is a library variable, local variable, or formal parameter, then e evaluates to the current binding of id. This case also applies if d is a library or local function declaration, as these are equivalent to function-valued variable declarations.
- If d is a static method, then e evaluates to the function defined by d.
- If d is the declaration of a static variable or static getter declared in class C, then e is equivalent to the getter invocation (10.15) $C.id$.
- If d is the declaration of a top level getter, then e is equivalent to the getter invocation id.
- Otherwise, e is equivalent to the property extraction **this**.id.

10.30 Type Test

The *is-expression* tests if an object is a member of a type.

isOperator:
is '!'?
;

Evaluation of the is-expression e **is** T proceeds as follows:

The expression e is evaluated to a value v. Then, if the interface induced by the class of v is a subtype of T, the is-expression evaluates to true. Otherwise it evaluates to false.

It follows that e **is** Object is always true. This makes sense in a language where everything is an object.

Also note that **null is** T is false unless T = Object or T = Null. Since the class Null is not exported by the core library, the latter will not occur in user code. The former is useless, as is anything of the form e **is** Object. Users should test for a null value directly rather than via type tests.

The is-expression e **is!** T is equivalent to !(e **is** T).

The static type of an is-expression is bool.

11 Statements

statements:
  statement* |

statement:
  label* nonLabelledStatement
;

nonLabelledStatement:
  block |
  initializedVariableDeclaration ';' |
  forStatement |
  whileStatement |
  doStatement |
  switchStatement |
  ifStatement |
  tryStatement |
  breakStatement |
  continueStatement |
  returnStatement |
  throwStatement |
  expressionStatement |
  assertStatement |
  functionSignature functionBody
;

11.1 Blocks

A *block statement* supports sequencing of code.

Execution of a block statement $\{s_1, \ldots, s_n\}$ proceeds as follows:

For $i \in 1..n, s_i$ is executed.

11.2 Expression Statements

An *expression statement* consists of an expression.

expressionStatement:
expression? ';' |

;

Execution of an expression statement e; proceeds by evaluating e.

11.3 Variable Declaration

A variable declaration statement declares a new local variable.

A *variable declaration statement* $T\ id$; or $T\ id = e$; introduces a new variable id with static type T into the innermost enclosing scope. A variable declaration statement **var** id; or **var** $id = e$; introduces a new variable named id with static type **Dynamic** into the innermost enclosing scope. In all cases, iff the variable declaration is prefixed with the **final** modifier, the variable is marked as final.

Executing a variable declaration statement $T\ id = e$; is equivalent to evaluating the assignment expression $id = e$, except that the assignment is considered legal even if the variable is final.

However, it is still illegal to assign to a final variable from within its initializer.

A variable declaration statement of the form $T\ id$; is equivalent to $T\ id =$ **null**;.

This holds regardless of the type T. For example, int i; does not cause i to be initialized to zero. Instead, i is initialized to **null**, just as if we had written **var** i; or Object i; or Collection<String> i;.

To do otherwise would undermine the optionally typed nature of Dart, causing type annotations to modify program behavior.

11.4 If

The *if statement* allows for conditional execution of statements.

ifStatement:
if '(' expression ')' statement (**else** statement)?
;

Execution of an if statement of the form **if** $(b)s_1$ **else** s_2 proceeds as follows:

First, the expression b is evaluated to an object o. In checked mode, it is a dynamic type error if o is not of type bool. Otherwise, o is then subjected

to boolean conversion (10.4.1), producing an object r. If r is **true**, then the statement s_1 is executed, otherwise statement s_2 is executed.

It is a static type warning if the type of the expression b is not bool.

An if statement of the form **if** $(b)s_1$ is equivalent to the if statement **if** $(b)s_1$ **else** {}.

11.5 For

The *for statement* supports iteration.

forStatement:
  for '(' forLoopParts ')' statement
;

forLoopParts:
  forInitializerStatement expression? ';' expressionList? |
 declaredIdentifier **in** expression |
 identifier **in** expression
;

forInitializerStatement:
  initializedVariableDeclaration ';' |
 expression? ';'
;

The for statement has two forms - the traditional for loop and the foreach statement.

11.5.1 For Loop

A for statement of the form **for** $(e_1$; c; $e_2)$ s is equivalent to the the following code:

{e_1; **while** (c) { s e_2; }}

11.5.2 Foreach

A for statement of the form **for** $(finalVarOrType$ id **in** $e)$ s is equivalent to the the following code:

var n0 = e.iterator(); **while** (n0.hasNext()) { $finalVarOrType$ id = n0.next(); s } where n0 is an identifier that does not occur anywhere in the program.

11.6 While

The while statement supports conditional iteration, where the condition is evaluated prior to the loop.

whileStatement:
while '(' expression ')' statement
;

Execution of a while statement of the form **while** (e) s; proceeds as follows:

The expression e is evaluated to an object o. In checked mode, it is a dynamic type error if o is not of type bool. Otherwise, o is then subjected to boolean conversion (10.4.1), producing an object r. If r is **true**, then s is executed and then the while statement is re-executed recursively. If r is **false**, execution of the while statement is complete.

It is a static type warning if the type of e is not bool.

11.7 Do

The do statement supports conditional iteration, where the condition is evaluated after the loop.

doStatement:
do statement **while** '(' expression ')' ';'
;

Execution of a do statement of the form **do** s **while** (e); proceeds as follows:

The statement s is executed. Then, the expression e is evaluated to an object o. In checked mode, it is a dynamic type error if o is not of type bool. Otherwise, o is then subjected to boolean conversion (10.4.1), producing an object r. If r is **false**, execution of the do statement is complete. If r is **true**, then the do statement is re-executed recursively.

It is a static type warning if the type of e is not bool.

11.8 Switch

The switch statement supports dispatching control among a large number of cases.

switchStatement:
switch '(' expression ')' '{' switchCase* defaultCase? '}'
;

switchCase:
label? (case expression ':')+ statements
;

defaultCase:
label? (case expression ':')* default ':' statements

;

Execution of a switch statement **switch** (e) { **case** $e_1 : s_1 \ldots$ **case** $e_n : s_n$ **default** s_{n+1} } proceeds as follows:

The statement **var** n $= e$; is evaluated, where n is a variable whose name is distinct from any other variable in the program. Next, the case clause **case** $e_1 : s_1$ is executed if it exists. If **case** $e_1 : s_1$ does not exist, then the default clause is executed by executing s_{n+1}.

Execution of a **case** clause **case** $e_k : s_k$ of a switch statement **switch** (e) { **case** $e_1 : s_1 \ldots$ **case** $e_n : s_n$ **default** s_{n+1} } proceeds as follows:

The expression n $== e_k$ is evaluated to a value v.

If v is **false**, or if s_k is empty, the following case, **case** $e_{k+1} : s_{k+1}$ is executed if it exists. If **case** $e_{k+1} : s_{k+1}$ does not exist, then the **default** clause is executed by executing s_{n+1}. If v is **true**, the statement sequence s_k is executed.

A switch statement **switch** (e) { **case** $e_1 : s_1 \ldots$ **case** $e_n : s_n$ } is equivalent to switch statement **switch** (e) { **case** $e_1 : s_1 \ldots$ **case** $e_n : s_n$ **default** }

It is a static warning if the type of e is may not be assigned to the type of e_k for all $k \in 1..n$.

11.9 Try

The try statement supports the definition of exception handling code in a structured way.

tryStatement:
  try block (catchPart+ finallyPart? | finallyPart)
;

catchPart:
  catch '(' simpleFormalParameter (', ' simpleFormalParameter)?
')' block
;

finallyPart:
  finally block
;

A try statement consists of a block statement, followed by at least one of:

1. A set of **catch** clauses, each of which specifies one or two exception parameters and a block statement.
2. A **finally** clause, which consists of a block statement.

A **catch** clause **catch** (T p_1, T p_2) s matches an object o if o is **null** or if the type of o is a subtype of T.

The formulation below is an attempt to characterize exception handling without resorting to a normal/abrupt completion formulation. It has the advantage that one need not specify abrupt completion behavior for every compound statement. On the other hand, it is new different and needs more thought.

A try statement **try** s_1 $catch_1 \ldots catch_n$ **finally** s_f defines an exception handler h that executes as follows:

The **catch** clauses are examined in order, starting with $catch_1$, until either a **catch** clause that matches the current exception is found, or the list of **catch** clauses has been exhausted. If a **catch** clause $catch_k$ is found, then p_{k1} is bound to the current exception, p_{k2} is bound to the current stack trace, and then $catch_k$ is executed. If no **catch** clause is found, the **finally** clause is executed. Then, execution resumes at the end of the try statement.

A finally clause **finally** s defines an exception handler h that executes by executing the finally clause. Then, execution resumes at the end of the try statement.

Execution of a **catch** clause **catch** (p_1, p_2) s of a try statement t proceeds as follows: The statement s is executed in the dynamic scope of the exception handler defined by the finally clause of t. Then, the current exception and current stack trace both become undefined.

Execution of a **finally** clause **finally** s of a try statement proceeds as follows:

The statement s is executed. Then, if the current exception is defined, control is transferred to the nearest dynamically enclosing exception handler.

Execution of a try statement of the form **try** s_1 $catch_1 \ldots catch_n$ **finally** s_f; proceeds as follows:

The statement s_1 is executed in the dynamic scope of the exception handler defined by the try statement. Then, the **finally** clause is executed.

Whether any of the **catch** clauses is executed depends on whether a matching exception has been raised by s_1 (see the specification of the throw statement).

If s_1 has raised an exception, it will transfer control to the try statements handler, which will examine the catch clauses in order for a match as specified above. If no matches are found, the handler will execute the **finally** clause.

If a matching **catch** was found, it will execute first, and then the **finally** clause will be executed.

If an exception is raised during execution of a **catch** clause, this will transfer control to the handler for the **finally** clause, causing the **finally** clause to execute in this case as well.

If no exception was raised, the **finally** clause is also executed. Execution of the **finally** clause could also raise an exception, which will cause transfer of control to the next enclosing handler.

11.10 Return

The *return statement* returns a result to the caller of a function.

returnStatement:
  return expression? ';'
;

Executing a return statement
return e;
first causes evaluation of the expression e, producing an object o. Next, control is transferred to the caller of the current function activation, and the object o is provided to the caller as the result of the function call.

It is a static type warning if the type of e may not be assigned to the declared return type of the immediately enclosing function.

It is a compile-time error if a return statement of the form **return** e; appears in a generative constructor (7.5.1).

It is quite easy to forget to add the factory prefix for a constructor, accidentally converting a factory into a generative constructor. The static checker may detect a type mismatch in some, but not all, of these cases. The rule above helps catch such errors, which can otherwise be very hard to recognize. There is no real downside to it, as returning a value from a generative constructor is meaningless.

Let f be the function immediately enclosing a return statement of the form **return**; It is a static warning if both of the following conditions hold:

- f is not a generative constructor.
- The return type of f may not be assigned to **void**.

Hence, a static warning will not be issued if f has no declared return type, since the return type would be **Dynamic** and **Dynamic** may be assigned to **void**. However, any function that declares a return type must return an expression explicitly.

This helps catch situations where users forget to return a value in a return statement.

A return statement of the form **return**; is executed by executing the statement **return null**; if it occurs inside a method, getter, setter or factory; otherwise, the return statement necessarily occurs inside a generative constructor, in which case it is executed by executing **return this**;.

Despite the fact that **return**; is executed as if by a **return** e;, it is important to understand that it is not a static warning to include a statement of the form **return**; in a generative constructor. The rules relate only to the specific syntactic form **return** e;.

The motivation for formulating **return**; *in this way stems from the basic requirement that all function invocations indeed return a value. Function invocations are expressions, and we cannot rely on a mandatory typechecker to always prohibit use of* ***void*** *functions in expressions. Hence, a return statement must always return a value, even if no expression is specified.*

The question than becomes, what value should a return statement return when no return expression is given. In a generative constructor, it is obviously

*the object being constructed (**this**). A void function is not expected to participate in an expression, which is why it is marked **void** in the first place. Hence, this situation is a mistake which should be detected as soon as possible. The static rules help here, but if the code is executed, using **null** leads to fast failure, which is desirable in this case. The same rationale applies for function bodies that do not contain a return statement at all.*

11.11 Labels

A *label* is an identifier followed by a colon. A *labeled statement* is a statement prefixed by a label L. A *labeled case clause* is a case clause within a switch statement (11.8) prefixed by a label L.

The sole role of labels is to provide targets for the break (11.12) and continue (11.13) statements.

label:
  identifier ':'
;

The semantics of a labeled statement $L : s$ are identical to those of the statement s. The namespace of labels is distinct from the one used for types, functions and variables.

The scope of a label that labels a statement s is s. The scope of a label that labels a case clause of a switch statement s is s.

Labels should be avoided by programmers at all costs. The motivation for including labels in the language is primarily making Dart a better target for code generation.

11.12 Break

The *break statement* consists of the reserved word **break** and an optional label (11.11).

breakStatement:
  break identifier? ';'
;

Let s_b be a **break** statement. If s_b is of the form **break** L;, then let s_E be the the innermost labeled statement with label L enclosing s_b. If s_b is of the form **break**;, then let s_E be the the innermost do (11.7), for (11.5), switch (11.8) or while (11.6) statement enclosing s_b. It is a compile-time error if no such statement s_E exists within the innermost function in which s_b occurs. Furthermore, let $s_1, \ldots, s_n$ be those **try** statements that are both enclosed in s_E and that enclose s_b, and that have a **finally** clause. Lastly, let f_j be the **finally** clause of $s_j, 1 \leq j \leq n$. Executing s_b first executes $f_1, \ldots, f_n$ in innermost-clause-first order and then terminates s_E.

11.13 Continue

The *continue statement* consists of the reserved word **continue** and an optional label (11.11).

continueStatement:
  continue identifier? ';'
;

Let s_c be a **continue** statement. If s_c is of the form **continue** L;, then let s_E be the the innermost labeled statement or case clause with label L enclosing s_c. If s_c is of the form **continue**; then let s_E be the the innermost do (11.7), for (11.5) or while (11.6) statement enclosing s_c. It is a compile-time error if no such statement or case clause s_E exists within the innermost function in which s_c occurs. Furthermore, let $s_1, \ldots, s_n$ be those **try** statements that are both enclosed in s_E and that enclose s_c, and that have a **finally** clause. Lastly, let f_j be the **finally** clause of $s_j, 1 \leq j \leq n$. Executing s_c first executes $f_1, \ldots, f_n$ in innermost-clause-first order and then transfers control to s_E.

11.14 Throw

The throw statement is used to raise or re-raise an exception.

throwStatement:
  throw expression? ';'
;

The *current exception* is the last unhandled exception thrown. The *current stack trace* is a record of all the function activations within the current isolate that had not completed execution at the point where the current exception was thrown. For each such function activation, the current stack trace includes the name of the function, the bindings of all its formal parameters, local variables and **this**, and the position at which the function was executing.

Should we make it unambiguous that line numbers aren't good enough?

Execution of a throw statement of the form **throw** e; proceeds as follows:

The expression e is evaluated yielding a value v. Then, control is transferred to the nearest dynamically enclosing exception handler (11.9), with the current exception set to v and the current stack trace set to the series of activations that led to execution of the current function.

There is no requirement that the expression e evaluate to a special kind of exception or error object.

Execution of a statement of the form **throw**; proceeds as follows: Control is transferred to the nearest innermost enclosing exception handler (11.9).

No change is made to the current stack trace or the current exception.

It is a compile-time error if a statement of the form **throw**; is not enclosed within a catch clause.

11.15 Assert

An *assert statement* is used to disrupt normal execution if a given boolean condition does not hold.

assertStatement:
assert '(' conditionalExpression ')' ';' |

The assert statement has no effect in production mode. In checked mode, execution of an assert statement **assert**(e); proceeds as follows:

The conditional expression e is evaluated to an object o. In checked mode, it is a dynamic error if o is not of type bool. Otherwise, o is subjected to boolean conversion (10.4.1). If the resulting value r is **false**, we say that the assertion failed. If r is **true**, we say that the assertion succeeded. If the assertion succeeded, execution of the assert statement is complete. If the assertion failed, an AssertionError is thrown.

It is a static type warning if the type of e is not bool.

Why is this a statement, not a built in function call? Because it is handled magically so it has no effect and no overhead in production mode. Also, in the absence of final methods. one could not prevent it being overridden (though there is no real harm in that). Overall, perhaps it could be defined as a function, and the overhead issue could be viewed as an optimization.

12 Libraries and Scripts

A library consists of (a possibly empty) set of imports, and a set of top level declarations. A top level declaration is either a class (7), an interface (8), a type declaration, a function (6) or a variable declaration (5).

topLevelDefinition:
classDefinition |
interfaceDefinition |
functionTypeAlias |
functionSignature functionBody |
returnType? getOrSet identifier formalParameterList functionBody |
final type? staticFinalDeclarationList ';' |
variableDeclaration ';'
;

getOrSet:
get |

set
;

libraryDefinition:
scriptTag? libraryName import* include* resource* topLevelDefinition*
;

scriptTag:
'#!'
;
* NEWLINE |

libraryName:
'#' 'library' '(' stringLiteral ')' ';' |

A library may optionally begin with a script tag, which can be used to identify the interpreter of the script to whatever computing environment the script is embedded in. A script tag begins with the characters #! and ends at the end of the line. Any characters after #! are ignored by the Dart implementation.

Libraries are units of privacy. A private declaration declared within a library L can only be accessed by code within L. Any attempt to access a private member declaration from outside L will cause a method, getter or setter lookup failure.

Since top level privates are not imported, using them is a compile-time error and not an issue here.

The scope of a library L consists of the names introduced by all top level declarations declared in L, and the names added by L's imports (12.1).

Libraries may include extralinguistic resources (e.g., audio, video or graphics files)

resource:
'#' 'resource' '(' stringLiteral ')' ';'
;

12.1 Imports

An *import* specifies a library to be used in the scope of another library.

libraryImport:
'#' 'import' '(' stringLiteral (', ' 'prefix:
' stringLiteral)? ') ';'
;

An import specifies a URI where the declaration of the imported library is to be found. The effect of an *import of library B with prefix P within the declaration of library A* is"

- If P is the empty string, each non-private top level declaration d of B is added to the scope of A.
- Otherwise, each non-private top level declaration d of B is added to the scope of A under the name $P.d$, as is the name P.

Imports assume a global namespace of libraries (at least per isolate). They also assume the library is in control, rather than the other way around.

It is a compile-time error if a name N is introduced into the library scope of a library A, and either:

- N is declared by A, OR
- Another import introduces N into the scope of A.

This implies that it is a load-time error for a library to import itself, as the names of its members will be duplicated.

The *current library* is the library currently being compiled.

Compiling an import directive of the form #import(s_1, prefix: s_2); proceeds as follows:

- If the contents of the URI that is value of s_1 have not yet been compiled in the current isolate then they are compiled to yield a library B. It is a compile-time error if s_1 does not denote a URI that contains the source code for a Dart library.
- Otherwise, the contents of URI denoted by s_1 have been compiled into a library B within the current library.
- Then, the library B is imported into the current library with prefix p, where p is the value of s_2.

Compiling an include directive of the form #import(s) is equivalent to compiling the directive #import(s, prefix: '');.

It is a compile-time error to import two or more libraries that define the same name.

It is a compile-time error if either s_1 or s_2 is not a compile-time constant.

12.2 Includes

An *include directive* specifies a URI where a Dart compilation unit that should be incorporated into the current library may be found.

include:
'#' 'source' '(' stringLiteral ')' ';'

;

compilationUnit:
topLevelDefinition* EOF
;

A *compilation unit* is a sequence of top level declarations.

Compiling an include directive of the form #source(s); causes the Dart system to attempt to compile the contents of the URI that is the value of s. The top level declarations at that URI are then compiled by the Dart compiler in the scope of the current library. It is a compile-time error if the contents of the URI are not a valid compilation unit.

It is a compile-time error if s is not a compile-time constant.

12.3 Scripts

A *script* is a library with a top level function main().

scriptDefinition:
scriptTag? libraryName? import* include* resource* topLevelDefinition*
;

A script S may be executed as follows:

First, S is compiled as a library as specified above. Then, the top level function main() that is in scope in S is invoked with no arguments. It is a run time error if S does not declare or import a top level function main().

The names of scripts are optional, in the interests of interactive, informal use. However, any script of long term value should be given a name as a matter of good practice. Named scripts are composable: they can be used as libraries by other scripts and libraries.

13 Types

Dart supports optional typing based on interface types.

The type system is unsound, due to the covariance of generic types. This is a deliberate choice (and undoubtedly controversial). Experience has shown that sound type rules for generics fly in the face of programmer intuition. It is easy for tools to provide a sound type analysis if they choose, which may be useful for tasks like refactoring.

13.1 Static Types

Static type annotations are used in variable declarations (5) (including formal parameters (6.2)) and in the return types of functions (6). Static type annotations are used during static checking and when running programs in checked mode. They have no effect whatsoever in production mode.

type:
  qualified typeArguments?
;

typeArguments:
  '<' typeList '>'
;

typeList:
  type (', ' type)*
;

A Dart implementation must provide a static checker that detects and reports exactly those situations this specification identifies as static warnings. However:

- Running the static checker on a program P is not required for compiling and running P.
- Running the static checker on a program P must not prevent successful compilation of P nor may it prevent the execution of P, regardless of whether any static warnings occur.

Nothing precludes additional tools that implement alternative static analyses (e.g., interpreting the existing type annotations in a sound manner such as either non-variant generics, or inferring declaration based variance from the actual declarations). However, using these tools does not preclude successful compilation and execution of Dart code.

Should we do something with respect to non-nullable types?

13.2 Dynamic Type System

A Dart implementation must support execution in both production mode and checked mode. Those dynamic checks specified as occurring specifically in checked mode must be performed iff the code is executed in checked mode.

13.3 Type Declarations

13.3.1 Typedef

A *type alias* declares a name for a type expression.

functionTypeAlias:
  typedef functionPrefix typeParameters? formalParameterList ';'
;

The effect of a type alias of the form **typedef** T $id(T_1\ p_1, \ldots, T_n\ p_n, [T_{n+1}\ p_{n+1}, \ldots, T_{n+k}\ p_{n+k}])$ declared in a library L is is to introduce the name id into the scope of L, bound to the function type $(T_1, \ldots, T_n, [T_{n+1}\ p_{n+1}, \ldots, T_{n+k}\ p_{n+k}]) \rightarrow T$. If no return type is specified, it is taken to be **Dynamic**. Likewise, if a type annotation is omitted on a formal parameter, it is taken to be **Dynamic**.

Currently, type aliases are restricted to function types. It is a compile-time error if any default values are specified in the signature of a function type alias.

13.4 Interface Types

An interface I is a direct supertype of an interface J iff:

- If I is Object, and J has no **extends** clause
- if I is listed in the **extends** clause of J.

The supertypes of an interface are its direct supertypes and their supertypes.

A type T is more specific than a type S, written $T << S$, if one of the following conditions is met:

- T is S.
- T is $\bot$.
- S is **Dynamic**.
- S is a direct supertype of T.
- T is a type variable and S is the upper bound of T.
- T is of the form $I < T_1, \ldots, T_n >$ and S is of the form $I < S_1, \ldots, S_n >$ and: $T_i << S_i, 1 \leq i \leq n$
- $T << U$ and $U << S$.

$<<$ is a partial order on types. T is a subtype of S, written $T <: S$, iff $[\bot/\textbf{Dynamic}]T << S$.

Note that $<:$ is not a partial order on types, it is only binary relation on types. This is because $<:$ is not transitive. If it was, the subtype rule would have a cycle. For example: List $<:$ List $<$ String $>$ and List $<$ int $>><:$ List, but List $<$ int $>$ is not a subtype of List $<$ String $>$. Although $<:$ is not a partial order on types, it does contain a partial order, namely $<<$. This means that, barring raw types, intuition about classical subtype rules does apply.

S is a supertype of T, written $S :> T$, iff T is a subtype of S.

A type T may be assigned to a type S, written $T \Longleftrightarrow S$, iff either $T <: S$ or $S <: T$.

This rule may surprise readers accustomed to conventional typechecking. The intent of the $\Longleftrightarrow$ relation is not to ensure that an assignment is correct. Instead, it aims to only flag assignments that are almost certain to be erroneous, without precluding assignments that may work.

For example, assigning a value of static type Object to a variable with static type String, while not guaranteed to be correct, might be fine if the runtime value happens to be a string.

13.5 Function Types

A function type $(T_1, \ldots T_n, [T_{x_1}\ x_1, \ldots, T_{x_k}\ x_k]) \rightarrow T$ is a subtype of the function type $(S_1, \ldots, S_n, [S_{y_1} y_1, \ldots, S_{y_m} y_m]) \rightarrow S$, if all of the following conditions are met:

1. Either
   - S is **void**, Or
   - $T \Longleftrightarrow S$.
2. $\forall i \in 1..n, T_i \Longleftrightarrow S_i$.
3. $k \geq m$ and $x_i = y_i, i \in 1..m$. *It is necessary, but not sufficient, that the optional arguments of the subtype be a subset of those of the supertype. We cannot treat them as just sets, because optional arguments can be invoked positionally, so the order matters.*
4. For all $y \in \{y_1, \ldots, y_m\} S_y \Longleftrightarrow T_y$

We write $(T_1, \ldots, T_n) \rightarrow T$ as a shorthand for the type $(T_1, \ldots, T_n, []) \rightarrow T$.

All functions implement the interface Function, so all function types are a subtype of Function.

13.6 Type Dynamic

The type **Dynamic** denotes the unknown type.

If no static type annotation has been provided the type system assumes the declaration has the unknown type. If a generic type is used but the corresponding type arguments are not provided, then the missing type arguments default to the unknown type.

This means that given a generic declaration $G < T_1, \ldots, T_n >$, the type G is equivalent to $G <$ **Dynamic**$, \ldots,$ **Dynamic** $>$.

Type **Dynamic** has methods for every possible identifier and arity, with every possible combination of named parameters. These methods all have **Dynamic** as their return type, and their formal parameters all have type **Dynamic**. Type **Dynamic** has properties for every possible identifier. These properties all have type **Dynamic**.

From a usability perspective, we want to ensure that the checker does not issue errors everywhere an unknown type is used. The definitions above ensure that no secondary errors are reported when accessing an unknown type.

The current rules say that missing type arguments are treated as if they were the type ***Dynamic****. An alternative is to consider them as meaning* Object*. This would lead to earlier error detection in checked mode, and more aggressive errors during static typechecking. For example:*

(1) typedAPI(G<String>g){...}

(2) typedAPI(new G());

Under the alternative rules, (2) would cause a runtime error in checked mode. This seems desirable from the perspective of error localization. However, when a dynamic error is raised at (2), the only way to keep running is rewriting (2) into

(3) typedAPI(new G<String>());

This forces users to write type information in their client code just because they are calling a typed API. We do not want to impose this on Dart programmers, some of which may be blissfully unaware of types in general, and genericity in particular.

What of static checking? Surely we would want to flag (2) when users have explicitly asked for static typechecking? Yes, but the reality is that the Dart static checker is likely to be running in the background by default. Engineering teams typically desire a clean build free of warnings and so the checker is designed to be extremely charitable. Other tools can interpret the type information more aggressively and warn about violations of conventional (and sound) static type discipline.

13.7 Type Void

The special type **void** may only be used as the return type of a function: it is a compile-time error to use **void** in any other context.

For example, as a type argument, or as the type of a variable or parameter

Void is not an interface type.

The only subtype relations that pertain to void are therefore:

- **void** $<:$ **void** (by reflexivity)
- $\bot <:$ **void** (as bottom is a subtype of all types).
- **void** $<:$ **Dynamic** (as **Dynamic** is a supertype of all types)

Hence, the static checker will issue warnings if one attempts to access a member of the result of a void method invocation (even for members of **null**, such as $==$). Likewise, passing the result of a void method as a parameter or assigning it to a variable will cause a warning unless the variable/formal; parameter has type dynamic.

On the other hand, it is possible to return the result of a void method from within a void method. One can also return **null**; or a value of type **Dynamic**. Returning any other result will cause a type warning (or a dynamic type error in checked mode).

13.8 Parameterized Types

A *parameterized type* is an invocation of a generic type declaration.

Let $p = G < A_1, \ldots, A_n >$ be a parameterized type.

It is a compile-time error if G is not an accessible generic type declaration with n type parameters.

If S is the static type of of a member m of G, then the static type of the member m of $G < A_1, \ldots, A_n >$ is $[A_1, \ldots, A_n/T_1, \ldots, T_n]S$ where $T_1, \ldots, T_n$ are the formal type parameters of G. Let $B_{i1}, \ldots, B_{ik_i}$, be the bounds of $T_i, 1 \leq i \leq n$. It is a static type warning if A_i is not a subtype of $[A_1, \ldots, A_n/T_1, \ldots, T_n]B_{ij}, 1 \leq j \leq k_i$.

13.8.1 Actual Type of Declaration

A type T *depends on a type variable* U iff:

- T is U.
- T is a parameterized type, and one of the type arguments of T depends on U.

Let T be the declared type of a declaration d, as it appears in the program source. The *actual type* of d is

- $[A_1, \ldots, A_n/U_1, \ldots, U_n]T$ if d depends on type variables $U_1, \ldots, U_n$, and A_i is the value of $U_i, 1 \leq i \leq n$.
- T otherwise.

13.8.2 Least Upper Bounds

The least upper bound of two interfaces I & J is defined as:

- I if I i= J or I is a superinterface of J.
- J if J is a superinterface of I.
- **to be continued**

14 Reference

14.1 Lexical Rules

14.1.1 Reserved Words

break, **case**, **catch**, **const**, **continue**, **default**, **do**, **else**, **false**, **finally**, **for**, **if**, **in**, **new**, **null**, **return**, **super**, **switch**, **this**, **throw**, **true**, **try**, **var**, **void**, **while**.

LETTER:
‘a’ .. ‘z’ |
‘A’ ..‘Z’
;

DIGIT:
‘0’ .. ‘9’
;

WHITESPACE:
(‘\t’ | ‘ ’ | NEWLINE)+
;

14.1.2 Comments

Comments are sections of program text that are used for documentation.

SINGLE_LINE_COMMENT:
‘//’ ˜(NEWLINE)* (NEWLINE)?
;

MULTI_LINE_COMMENT:
‘/*’ (.)*
;

Dart supports both single-line and multi-line comments. A single line comments begins with the token //. Everything between // the end of line must be ignored by the Dart compiler.

A multi-line comment begins with the token /***** and ends with the token *****/. Everything between /* and *****/ must be ignored by the Dart compiler unless the comment is a documentation comment. Comments may nest.

Documentation comments are multi-line comments that begin with the tokens /******. Inside a documentation comment, the Dart compiler ignores all text unless it is enclosed in brackets.

14.2 Operator Precedence

Operator precedence is given implicitly by the grammar.