Dart Programming Language Specification
6th edition draft

Version 2.13-dev

April 20, 2023

Contents

1 Scope

This Ecma standard specifies the syntax and semantics of the Dart programming language. It does not specify the APIs of the Dart libraries except where those library elements are essential to the correct functioning of the language itself (e.g., the existence of class `Object` with methods such as `noSuchMethod`, `runtimeType`).

2 Conformance

A conforming implementation of the Dart programming language must provide and support all the APIs (libraries, types, functions, getters, setters, whether top-level, static, instance or local) mandated in this specification.

A conforming implementation is permitted to provide additional APIs, but not additional syntax, except for experimental features.

3 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

1. The Unicode Standard, Version 5.0, as amended by Unicode 5.1.0, or successor.
2. Dart API Reference, https://api.dartlang.org/

4 Terms and Definitions

Terms and definitions used in this specification are given in the body of the specification proper.

5 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.*

Reserved words and built-in identifiers (17.38) 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 "`::=`". On the right hand side, alternation is represented by vertical bars, and sequencing by spacing. As in PEGs, alternation gives priority to the left. 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 is represented by prefixing an element of a production with a tilde. Negation is similar to the not combinator of PEGs, but it consumes input if it matches. In the context of a lexical production it consumes a single character if there is one; otherwise, a single token if there is one.

An example would be:

⟨*aProduction*⟩ ::= ⟨*anAlternative*⟩
| ⟨*anotherAlternative*⟩
| ⟨*oneThing*⟩ ⟨*after*⟩ ⟨*another*⟩
| ⟨*zeroOrMoreThings*⟩*
| ⟨*oneOrMoreThings*⟩+
| ⟨*anOptionalThing*⟩?
| (⟨*some*⟩ ⟨*grouped*⟩ ⟨*things*⟩)
| ~⟨*notAThing*⟩
| '`aTerminal`'
| ⟨*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. Punctuation tokens appear in quotes.

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

A *term* is a syntactic construct. It may be considered to be a piece of text which is derivable in the grammar, and it may be considered to be a tree created by such a derivation. An *immediate subterm* of a given term t is a syntactic construct which corresponds to an immediate subtree of t considered as a derivation tree. A *subterm* of a given term t is t, or an immediate subterm of t, or a subterm of an immediate subterm of t. ◇ ◇ ◇

A list $x_1, \ldots, x_n$ denotes any list of n elements of the form $x_i, 1 \leq i \leq n$. $x_1, \ldots, x_n$

Note that n may be zero, in which case the list is empty. We use such lists extensively throughout this specification.

For $j \in 1..n$, let y_j be an atomic syntactic entity (like an identifier), x_j a composite syntactic entity (like an expression or a type), and E again a composite syntactic entity. The notation $[x_1/y_1, \ldots, x_n/y_n]E$ then denotes a copy of E in which each occurrence of $y_i, 1 \leq i \leq n$ has been replaced by x_i. j, y_j, x_j E ◇

This operation is also known as *substitution*, and it is the variant that avoids capture. That is, when E contains a construct that introduces y_i into a nested scope for some $i \in 1..n$, the substitution will not replace y_i in that scope. Conversely, if such a replacement would put an identifier *id* (a subterm of x_i) into a scope where *id* is declared, the relevant declarations in E are systematically renamed to fresh names. ◇

In short, capture freedom ensures that the "meaning" of each identifier is preserved during substitution.

We sometimes abuse list or map literal syntax, writing `[`o_1`, ..., `o_n`]` (respectively `{`k_1`: `o_1`, ..., `k_n`: `o_n`}`) where the o_i and k_i may be objects rather than expressions. The intent is to denote a list (respectively map) object whose elements are the o_i (respectively, whose keys are the k_i and values are the o_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 ◇

- 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.

When the specification refers to a *fresh variable*, it means a local variable with a name that doesn't occur anywhere in the current program. When the specification introduces a fresh variable bound to an object, the fresh variable is implicitly bound in a surrounding scope. ◇

References to otherwise unspecified names of program entities (such as classes or functions) are interpreted as the names of members of the Dart core library.

Examples would be the classes `Object` and `Type` representing, respectively, the root of the class hierarchy and the reification of run-time types. It would be possible to declare, e.g., a local variable named `Object`, so it is generally incorrect to assume that the name `Object` will actually resolve to said core class. However, we will generally omit mentioning this, for brevity.

When the specification says that one piece of syntax *is equivalent to* another piece of syntax, it means that it is equivalent in all ways, and the former syntax should generate the same compile-time errors and have the same run-time behavior as the latter, if any. Error messages, if any, should always refer to the original syntax. If execution or evaluation of a construct is said to be equivalent ◇

to execution or evaluation of another construct, then only the run-time behavior is equivalent, and compile-time errors apply only for the original syntax.

When the specification says that one piece of syntax s is *treated as* another piece of syntax s', it means that the static analysis of s is the static analysis of s' (in particular, exactly the same compile-time errors occur). Moreover, if s has no compile-time errors then the behavior of s at run time is exactly the behavior of s'. s, s' ◇

Error messages, *if any, should always refer to the original syntax* s.

In short, whenever s is treated as s', the reader should immediately switch to the section about s' in order to get any further information about the static analysis and dynamic semantics of s.

The notion of being 'treated as' is similar to the notion of syntactic sugar: "s *is treated as* s'*" could as well have been worded "*s *is desugared into* s'*". Of course, it should then actually be called "semantic sugar", because the applicability of the transformation and the construction of* s' *may rely on information from static analysis.*

The point is that we only specify the static analysis and dynamic semantics of a core language which is a subset of Dart (just slightly smaller than Dart), and desugaring transforms any given Dart program to a program in that core language. This helps keeping the language specification consistent and comprehensible, because it shows directly that some language features are introducing essential semantics, and others are better described as mere abbreviations of existing constructs.

The specification uses one syntactic construct, the ***let*** *expression*, which is not derivable in the grammar (that is, no Dart source code contains such an expression). This expression is helpful in specifying certain syntactic forms that are treated as other syntactic forms, because it allows for introducing and initializing one or more fresh variables, and using them in an expression. ◇

That is, a **let** expression is only introduced as a tool to define the evaluation semantics of an expression in terms of other expressions containing **let** expressions.

The syntax of a **let** expression is as follows:

⟨*letExpression*⟩ ::= **let** ⟨*staticFinalDeclarationList*⟩ **in** ⟨*expression*⟩

Let e_{let} be a **let** expression of the form **let** $v_1 = e_1$ **,** $\cdots$ **,** $v_k = e_k$ **in** e. It is tacitly assumed that v_j is a fresh variable, $j \in 1..k$, unless something is stated to the contrary. e_{let}, e_j, v_j, k

e_{let} contains k nested scopes, $S_1, \ldots, S_k$. The enclosing scope for S_1 is the current scope for e_{let}, and the enclosing scope for S_j is S_{j-1}, $j \in 2..k$. The current scope of e_1 is the current scope of e_{let}, the current scope of e_j is S_{j-1}, $j \in 2..k$, and the current scope of e is S_k. For $j \in 1..k$, v_j introduces a final, local variable into S_j, with the static type of e_j as its declared type. $S_1, \ldots, S_k$

Type inference of e_j and the context type used for inference of e_j are not relevant, it is assumed that type inference has occurred already (6).

Evaluation of e_{let} proceeds by evaluating e_j to an object o_j and binding v_j to o_j, where $j \in 1..k$, in that order. Finally, e is evaluated to an object o and then e_{let} evaluates to o.

The right margin of each page in this document is used to indicate referenced entities.

The document contains an index at the end. Each entry in the index refers to a page number, p. On page p there is a '$\diamond$' in the margin at the definition of the given indexed phrase, and the phrase itself is shown using *this typeface*. We have hereby introduced the *index marker* $\diamond$ itself. $\diamond$

The right margin also contains symbols. Whenever a symbol (say, C or x_j) is introduced and used in more than a few lines of text, it is shown in the margin. C x_j

The point is that it is easy to find the definition of a symbol by scanning back in the text until that symbol occurs in the margin. To avoid useless verbosity, some symbols are not mentioned in the margin. For instance, we may introduce $e_1, \ldots, e_k$, but only show e_j and k in the margin.

Note that it may be necessary to look at a few lines of text above the '$\diamond$' or symbol, because the margin markers can be pushed down one line when there is more than one marker for a single line.

6 Overview

Dart is a class-based, single-inheritance, pure object-oriented programming language. Dart is optionally typed (20) and supports reified generics. The run-time type of every object is represented as an instance of class `Type` which can be obtained by calling the getter `runtimeType` declared in class `Object`, the root of the Dart class hierarchy.

Dart programs may be statically checked. Programs with compile-time errors do not have a specified dynamic semantics. This specification makes no attempt to answer additional questions about a library or program at the point where it is known to have a compile-time error.

However, tools may choose to support execution of some programs with errors. For instance, a compiler may compile certain constructs with errors such that a dynamic error will be raised if an attempt is made to execute such a construct, or an IDE integrated runtime may support opening an editor window when such a construct is executed, allowing developers to correct the error. It is expected that such features would amount to a natural extension of the dynamic semantics of Dart as specified here, but, as mentioned, this specification makes no attempt to specify exactly what that means.

As specified in this document, dynamic checks are guaranteed to be performed in certain situations, and certain violations of the type system throw exceptions at run time.

An implementation is free to omit such checks whenever they are guaranteed to succeed, e.g., based on results from the static analysis.

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

1. Reified type information reflects the types of objects at run time and may always be queried by dynamic typechecking constructs (the analogs of instanceOf, casts, typecase etc. in other languages). Reified type information

includes access to instances of class Type representing types, the run-time type (aka class) of an object, and the actual values of type parameters to constructors and generic function invocations.

2. Type annotations declare the types of variables and functions (including methods and constructors).

3. Type annotations may be omitted, in which case they are generally filled in with the type **dynamic** (20.7).

Dart as implemented includes extensive support for inference of omitted types. This specification makes the assumption that inference has taken place, and hence inferred types are considered to be present in the program already. However, in some cases no information is available to infer an omitted type annotation, and hence this specification still needs to specify how to deal with that. A future version of this specification will also specify type inference.

Dart programs are organized in a modular fashion into units called *libraries* (19). 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.

A dart program execution may occur with assertions enabled or disabled. The method used to enable or disable assertions is implementation specific.

6.1 Scoping

A *compile-time namespace* is a partial function that maps names to namespace values. Compile-time namespaces are used much more frequently than run-time namespaces (defined later in this section), so when the word *namespace* is used alone, it means compile-time namespace. A *name* is a lexical token which is an ⟨*IDENTIFIER*⟩, an ⟨*IDENTIFIER*⟩ followed by '=', or an ⟨*operator*⟩, or `unary-`; and a *namespace value* is a declaration, a namespace, or the special value NAME_CONFLICT (19.4). ◇ ◇ ◇ ◇

If $NS(n) = V$ then we say that *NS maps* the *key* n to the *value* V, and that *NS has the binding* $n \mapsto V$. The fact that NS is a partial function just means that each name is mapped to at most one namespace value. That is, if NS has the bindings $n \mapsto V_1$ and $n \mapsto V_2$ then $V_1 = V_2$. ◇ ◇ ◇ ◇

Let *NS* be a namespace. We say that a name n *is in NS* if n is a key of *NS*. We say a declaration d *is in NS* if a key of *NS* is mapped to d. ◇ ◇

A scope S_0 has an associated namespace NS_0. The bindings of NS_0 is specified in this document by saying that a given declaration D named n *introduces* a specific entity V into S_0, which means that the binding $n \mapsto V$ is added to NS_0. D, n ◇ V

In some cases, the name of the declaration differs from the identifier that occurs in the declaration syntax used to declare it. Setters have names that are distinct from the corresponding getters because they always have an '=' automatically added at the end, and the unary minus operator has the special name `unary-`.

It is typically the case that V is the declaration D itself, but there are exceptions. For example, a variable declaration introduces an implicitly induced getter declaration, and in some cases also an implicitly induced setter declaration into the given scope.

Note that labels (18.13) are not included in the namespace of a scope. They are resolved lexically rather then being looked up in a namespace.

It is a compile-time error if there is more than one entity with the same name declared in the same scope.

It is therefore impossible, e.g., to define a class that declares a method and a getter with the same name in Dart. Similarly one cannot declare a top-level function with the same name as a library variable or a class which is declared in the same library.

We introduce the notion of a *run-time namespace*. This is a partial function from names to run-time entities, in particular storage locations and functions. Each run-time namespace corresponds to a namespace with the same keys, but with values that correspond to the semantics of the namespace values. ◇

A namespace typically maps a name to a declaration, and it can be used statically to figure out what that name refers to. For example, a variable is associated with an actual storage location at run time. We introduce the notion of a run-time namespace based on a namespace, such that the dynamic semantics can access run-time entities like that storage location. The same code may be executed multiple times with the same run-time namespace, or with different run-time namespaces for each execution. E.g., local variables declared inside a function are specific to each invocation of the function, and instance variables are specific to an object.

Dart is lexically scoped. Scopes may nest. A name or declaration d is *available in scope* S if d is in the namespace induced by S or if d is available in the lexically enclosing scope of S. We say that a name or declaration d is *in scope* if d is available in the current scope. ◇ ◇

If a declaration d named n is in the namespace induced by a scope S, then d *hides* any declaration named n that is available in the lexically enclosing scope of S. ◇

A consequence of these rules is that it is possible to hide a type with a method or variable. Naming conventions usually prevent such abuses. Nevertheless, the following program is legal:

```
class HighlyStrung {
  String() => "?";
}
```

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

The interaction of lexical scoping and inheritance is a subtle one. Ultimately, the question is whether lexical scoping takes precedence over inheritance or vice versa. Dart chooses the former.

Allowing inherited names to take precedence over locally declared names

could create unexpected situations as code evolves. Specifically, the behavior of code in a subclass could silently change if a new name is introduced in a superclass. Consider:

```
library L1;
class S {}
library L2;
import 'L1.dart';
foo() => 42;
class C extends S{ bar() => foo();}
```

Now assume a method `foo()` *is added to* `S`*.*

```
library L1;
class S {foo() => 91;}
```

If inheritance took precedence over the lexical scope, the behavior of `C` *would change in an unexpected way. Neither the author of* `S` *nor the author of* `C` *are necessarily aware of this. In Dart, if there is a lexically visible method* `foo()`*, it will always be called.*

Now consider the opposite scenario. We start with a version of `S` *that contains* `foo()`*, but do not declare* `foo()` *in library* `L2`*. Again, there is a change in behavior—but the author of* `L2` *is the one who introduced the discrepancy that effects their code, and the new code is lexically visible. Both these factors make it more likely that the problem will be detected.*

These considerations become even more important if one introduces constructs such as nested classes, which might be considered in future versions of the language.

Good tooling should of course endeavor to inform programmers of such situations (discreetly). For example, an identifier that is both inherited and lexically visible could be highlighted (via underlining or colorization). Better yet, tight integration of source control with language aware tools would detect such changes when they occur.

6.2 Privacy

Dart supports two levels of *privacy*: public and private. A declaration is ◇
private iff its name is private, otherwise it is *public*. A name q is *private* iff ◇
any one of the identifiers that comprise q is private, otherwise it is *public*. An ◇
identifier is *private* iff it begins with an underscore (the _ character) otherwise ◇
it is *public*. ◇

A declaration m is *accessible to a 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 applies only to declarations within a library, not to the library declaration as a whole. This is because libraries do not reference each other by name, and so the idea of a private library is meaningless (19.1). Thus, if the name of a library begins with an underscore, it has no effect on the accessibility of the library or its members.

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.

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.

6.3 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 (17.21.4). No state is ever shared between isolates. Isolates are created by spawning (17.14).

7 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.

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).

In a development environment a compiler should of course report compilation errors eagerly so as to best serve the programmer.

A Dart development environment might choose to support error eliminating program transformations, e.g., replacing an erroneous expression by the invocation of a debugger. It is outside the scope of this document to specify how such transformations work, and where they may be applied.

If an uncaught compile-time error occurs within the code of a running isolate A, A is immediately suspended. The only circumstance where a compile-time

error could be caught would be via code run reflectively, where the mirror system can catch it.

Typically, once a compile-time error is thrown and A is suspended, A will then be terminated. However, this depends on the overall environment. A Dart engine runs in the context of a runtime*, a program that interfaces between the engine and the surrounding computing environment. The runtime may be, for instance, a C++ program on the server. When an isolate fails with a compile-time error as described above, control returns to the runtime, along with an exception describing the problem. This is necessary so that the runtime can clean up resources etc. It is then the runtime's decision whether to terminate the isolate or not.* ◇

Static warnings are situations that do not preclude execution, but which are unlikely to be intended, and likely to cause bugs or inconveniences. A static warning must be reported by a Dart compiler before the associated code is executed. ◇

When this specification says that a *dynamic error* occurs, it means that a corresponding error object is thrown. When it says that a *dynamic type error* occurs, it represents a failed type check at run time, and the object which is thrown implements `TypeError`. ◇ ◇

Whenever we say that an exception *ex* is *thrown*, it acts like an expression had thrown (18.0.1) with *ex* as exception object and with a stack trace corresponding to the current system state. When we say that a *C is thrown*, where *C* is a class, we mean that an instance of class *C* is thrown. ◇ ◇

If an uncaught exception is thrown by a running isolate *A*, *A* is immediately suspended.

8 Variables

Variables are storage locations in memory.

⟨*finalConstVarOrType*⟩ ::= **late**? **final** ⟨*type*⟩?
  | **const** ⟨*type*⟩?
  | **late**? ⟨*varOrType*⟩

⟨*varOrType*⟩ ::= **var**
  | ⟨*type*⟩

⟨*initializedVariableDeclaration*⟩ ::=
    ⟨*declaredIdentifier*⟩ (‘=’ ⟨*expression*⟩)? (‘,’ ⟨*initializedIdentifier*⟩)*

⟨*initializedIdentifier*⟩ ::= ⟨*identifier*⟩ (‘=’ ⟨*expression*⟩)?

⟨*initializedIdentifierList*⟩ ::= ⟨*initializedIdentifier*⟩ (‘,’ ⟨*initializedIdentifier*⟩)*

An ⟨*initializedVariableDeclaration*⟩ that declares two or more variables is

equivalent to multiple variable declarations declaring the same set of variable names, in the same order, with the same initialization, type, and modifiers.

For example, **var** `x, y;` is equivalent to **var** `x;` **var** `y;` and **static final** `String s1, s2 = "foo";` is equivalent to **static final** `String s1;` **static final** `String s2 = "foo";`.

It is possible for a variable declaration to include the modifier **covariant**. The effect of doing this with an instance variable is described elsewhere (10.6). It is a compile-time error for the declaration of a variable which is not an instance variable to include the modifier **covariant**.

In a variable declaration of one of the forms $N\ v;$ $N\ v = e;$ where N is derived from $\langle metadata \rangle\ \langle finalConstVarOrType \rangle$, we say that v is the *declaring occurrence* of the identifier. For every identifier which is not a declaring occurrence, we say that it is an *referencing occurrence*. We also abbreviate that to say that an identifier is a *declaring identifier* respectively an *referencing identifier*. ◇

In an expression of the form $e.id$ it is possible that e has static type **dynamic** and id cannot be associated with any specific declaration named id at compile-time, but in this situation id is still a referencing identifier. ◇

An *initializing variable declaration* is a variable declaration whose declaring identifier is immediately followed by '=' and an *initializing expression*. ◇ ◇

A variable declared at the top-level of a library is referred to as either a *library variable* or a *top-level variable*. ◇

A *static variable* is a variable that is not associated with a particular instance, but rather with an entire library or class. Static variables include library variables and class variables. Class variables are variables whose declaration is immediately nested inside a class declaration and includes the modifier **static**. A library variable is implicitly static. It is a compile-time error to preface a top-level variable declaration with the built-in identifier (17.38) **static**. ◈

A *constant variable* is a variable whose declaration includes the modifier **const**. A constant variable must be initialized to a constant expression (17.3) or a compile-time error occurs. ◇

An initializing expression of a constant variable occurs in a constant context (17.3.2), which means that **const** modifiers need not be specified explicitly.

A *final variable* is a variable whose binding is fixed upon initialization; a final variable v will always refer to the same object after v has been initialized. A variable is final iff its declaration includes the modifier **final** or the modifier **const**. ◇

A *mutable variable* is a variable which is not final. ◇

The following rules on implicitly induced getters and setters apply to all static and instance variables.

A variable declaration of one of the forms $T\ v;$ $T\ v = e;$ **const** $T\ v = e;$ **final** $T\ v;$ or **final** $T\ v = e;$ induces an implicit getter function (10.3) with signature T **get** v whose invocation evaluates as described below (8.1). In these cases the static type of v is T.

A variable declaration of one of the forms **var** $v;$ **var** $v = e;$ **const** $v = e;$ **final** $v;$ or **final** $v = e;$ induces an implicit getter function with signature

dynamic get v whose invocation evaluates as described below (8.1). In these cases, the static type of v is **dynamic** (20.7).

A mutable variable declaration of the form $T\ v$**;** or $T\ v$ **=** e**;** induces an implicit setter function (10.4) with signature **void set** v**=(**$T\ x$**)** whose execution sets the value of v to the incoming argument x.

A mutable variable declaration of the form **var** v**;** or **var** v **=** e**;** induces an implicit setter function with signature **void set** v**=(dynamic** x**)** whose execution sets the value of v to the incoming argument x.

The scope into which the implicit getters and setters are introduced depends on the kind of variable declaration involved.

A library variable introduces a getter into the top level scope of the enclosing library. A class variable introduces a static getter into the immediately enclosing class. An instance variable introduces an instance getter into the immediately enclosing class.

A mutable library variable introduces a setter into the top level scope of the enclosing library. A mutable class variable introduces a static setter into the immediately enclosing class. A mutable instance variable introduces an instance setter into the immediately enclosing class.

Let v be variable declared in an initializing variable declaration, and let e be the associated initializing expression. It is a compile-time error if the static type of e is not assignable to the declared type of v. It is a compile-time error if a final instance variable whose declaration has an initializer expression is also initialized by a constructor, either by an initializing formal or an initializer list entry.

It is a compile-time error if a final instance variable that has been initialized by means of an initializing formal of a constructor k is also initialized in the initializer list of k (10.7.1).

A static final variable v does not induce a setter, so unless a setter named v= is in scope it is a compile-time error to assign to v.

Similarly, assignment to a final instance variable v is a compile-time error, unless a setter named v= is in scope, or the receiver has type **dynamic**. v can be initialized in its declaration or in initializer lists, but initialization and assignment is not the same thing. When the receiver has type **dynamic** such an assignment is not a compile-time error, but if there is no setter it will cause a dynamic error.

A variable that has no initializing expression has the null object (17.4) as its initial value. Otherwise, variable initialization proceeds as follows:

Static variable declarations with an initializing expression are initialized lazily (8.1).

The lazy semantics are given because we do not want a language where one tends to define expensive initialization computations, causing long application startup times. This is especially crucial for Dart, which must support the coding of client applications.

Initialization of an instance variable with no initializing expression takes place during constructor execution (10.7.1).

Initialization of an instance variable v with an initializing expression e pro-

ceeds as follows: e is evaluated to an object o and the variable v is bound to o.

It is specified elsewhere when this initialization occurs, and in which environment (p. 46, 18.3, 17.15.3).

If the initializing expression throws then access to the uninitialized variable is prevented, because the instance creation that caused this initialization to take place will throw.

It is a dynamic type error if the dynamic type of o is not a subtype of the actual type of the variable v (20.10.1).

8.1 Evaluation of Implicit Variable Getters

Let d be the declaration of a static or instance variable v. If d is an instance variable, then the invocation of the implicit getter of v evaluates to the object stored in v. If d is a static variable (which can be a library variable) then the implicit getter method of v executes as follows: d, v

- **Non-constant variable declaration with initializer**. If d is of one of the forms **var** v `=` e`;`, T v `=` e`;`, **final** v `=` e`;`, **final** T v `=` e`;`, **static** v `=` e`;`, **static** T v `=` e`;` , **static final** v `=` e`;` or **static final** T v `=` e`;` and no object has yet been stored into v then the initializing expression e is evaluated. If, during the evaluation of e, the getter for v is invoked, a `CyclicInitializationError` is thrown. If the evaluation of e throws an exception e and stack trace s, the null object (17.4) is stored into v; the execution of the getter then throws e and stack trace s. Otherwise, the evaluation of e succeeded yielding an object o; then o is stored into v and the execution of the getter completes by returning o. Otherwise, (when an object o has been stored in v) execution of the getter completes by returning o.
- **Constant variable declaration**. If d is of one of the forms **const** v `=` e`;`, **const** T v `=` e`;`, **static const** v `=` e`;` or **static const** T v `=` e`;` the result of the getter is the value of the constant expression e. Note that a constant expression cannot depend on itself, so no cyclic references can occur.
- **Variable declaration without initializer**. The result of executing the getter method is the object stored in v. This may be the initial value, that is, the null object.

9 Functions

Functions abstract over executable actions.

⟨*functionSignature*⟩ ::=
    ⟨*type*⟩? ⟨*identifier*⟩ ⟨*formalParameterPart*⟩

⟨*formalParameterPart*⟩ ::= ⟨*typeParameters*⟩? ⟨*formalParameterList*⟩

⟨*functionBody*⟩ ::= **async**? '**=>**' ⟨*expression*⟩ '**;**'
| (**async** '*****'? | **sync** '*****')? ⟨*block*⟩

⟨*block*⟩ ::= '**{**' ⟨*statements*⟩ '**}**'

Functions can be introduced by function declarations (9.1), method declarations (10.2, 10.8), getter declarations (10.3), setter declarations (10.4), and constructor declarations (10.7); and they can be introduced by function literals (17.11).

A function is *asynchronous* if its body is marked with the **async** or **async*** ◇ modifier. Otherwise the function is *synchronous*. A function is a *generator* if ◇ its body is marked with the **sync*** or **async*** modifier. Further details about ◇ these concepts are given below.

Whether a function is synchronous or asynchronous is orthogonal to whether it is a generator or not. Generator functions are a sugar for functions that produce collections in a systematic way, by lazily applying a function that *generates* individual elements of a collection. Dart provides such a sugar in both the synchronous case, where one returns an iterable, and in the asynchronous case, where one returns a stream. Dart also allows both synchronous and asynchronous functions that produce a single value.

Each declaration that introduces a function has a signature that specifies its return type, name, and formal parameter part, except that the return type may be omitted, and getters never have a formal parameter part. Function literals have a formal parameter part, but no return type and no name. The formal parameter part optionally specifies the formal type parameter list of the function, and it always specifies its formal parameter list. A function body is either:

- a block statement (18.1) containing the statements (18) executed by the function, optionally marked with one of the modifiers: **async**, **async*** or **sync***. Unless it is statically known that the body of the function cannot complete normally (that is, it cannot reach the end and "fall through", cf. 18.0.1), it is a compile-time error if the addition of **return;** at the end of the body would be a compile-time error. For instance, it is an error if the return type of a synchronous function is `int`, and the body may complete normally. The precise rules are given in section 18.12.

  Because Dart supports dynamic function invocations, we cannot guarantee that a function that does not return an object will not be used in the context of an expression. Therefore, every function must either throw or return an object. A function body that ends without doing a throw or return will cause the function to return the null object (17.4), as will a **return** without an expression. For generator functions, the situation is more subtle. See further discussion in section 18.12.

  OR

- of the form `=>` e or the form **async** `=>` e, which both return the value of the expression e as if by a `return` e. The other modifiers do not apply here, because they apply only to generators, discussed below. Generators are not allowed to explicitly return anything, objects are added to the generated stream or iterable using **yield** or **yield***. Let T be the declared return type of the function that has this body. It is a compile-time error if one of the following conditions hold:

  – The function is synchronous, T is not **void**, and it would have been a compile-time error to declare the function with the body `{` **return** e`;` `}` rather than `=>` e. In particular, e can have *any* type when the return type is **void**. *This enables concise declarations of **void** functions. It is reasonably easy to understand such a function, because the return type is textually near to the returned expression e. In contrast, **return** e; in a block body is only allowed for an e with one of a few specific static types, because it is less likely that the developer understands that the returned object will not be used (18.12).*

  – The function is asynchronous, *flatten*(T) is not **void**, and it would have been a compile-time error to declare the function with the body **async** `{` **return** e`;` `}` rather than **async** `=>` e. In particular, e can have *any* type when the flattened return type is **void**, *and the rationale is similar to the synchronous case.*

It is a compile-time error if an **async**, **async*** or **sync*** modifier is attached to the body of a setter or constructor.

An asynchronous setter would be of little use, since setters can only be used in the context of an assignment (17.23), and an assignment expression always evaluates to the value of the assignment's right hand side. If the setter actually did its work asynchronously, one might imagine that one would return a future that resolved to the assignment's right hand side after the setter did its work.

*An asynchronous constructor would, by definition, never return an instance of the class it purports to construct, but instead return a future. Calling such a beast via **new** would be very confusing. If you need to produce an object asynchronously, use a method.*

*One could allow modifiers for factories. A factory for `Future` could be modified by **async**, a factory for `Stream` could be modified by **async***, and a factory for `Iterable` could be modified by **sync***. No other scenario makes sense because the object returned by the factory would be of the wrong type. This situation is very unusual so it is not worth making an exception to the general rule for constructors in order to allow it.*

It is a compile-time error if the declared return type of a function marked **async** is not a supertype of `Future<`T`>` for some type T. It is a compile-time error if the declared return type of a function marked **sync*** is not a supertype of `Iterable<`T`>` for some type T. It is a compile-time error if the declared return type of a function marked **async*** is not a supertype of `Stream<`T`>` for

some type T. It is a compile-time error if the declared return type of a function marked **sync*** or **async*** is **void**.

We define the notion of the *element type of a generator* as follows: If the function f is a synchronous generator whose declared return type implements `Iterable<`U`>` for some U (11.2) then the element type of f is U. If the function f is an asynchronous generator whose declared return type implements `Stream<`U`>` for some U then the element type of f is U. Otherwise, if the function f is a generator (synchronous or asynchronous) then the element type of f is **dynamic**. ◇

In the latter case the return type is a top type, because the declaration of f would otherwise be a compile-time error. This implies that there is no information about the type of elements that the generator will yield.

9.1 Function Declarations

A *function declaration* is a function that is neither a member of a class nor a function literal. Function declarations include exactly the following: *library functions*, which are function declarations at the top level of a library, and *local functions*, which are function declarations declared inside other functions. Library functions are often referred to simply as top-level functions. ◇ ◇ ◇

A function declaration consists of an identifier indicating the function's name, possibly prefaced by a return type. The function name is followed by a signature and body. For getters, the signature is empty. The body is empty for functions that are external.

The scope of a library function is the scope of the enclosing library. The scope of a local function is described in section 18.4. In both cases, the name of the function is in scope in its formal parameter scope (9.2).

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

When we say that a function f_1 *forwards* to another function f_2, we mean that invoking f_1 causes f_2 to be executed with the same arguments and/or receiver as f_1, and returns the result of executing f_2 to the caller of f_1, unless f_2 throws an exception, in which case f_1 throws the same exception. Furthermore, we only use the term for synthetic functions introduced by the specification. ◇

9.2 Formal Parameters

Every non-getter function declaration includes a *formal parameter list*, which consists of a list of required positional parameters (9.2.1), followed by any optional parameters (9.2.2). The optional parameters may be specified either as a set of named parameters or as a list of positional parameters, but not both. ◇

Some function declarations include a *formal type parameter list* (9), in which case we say that it is a *generic function*. A *non-generic function* is a function which is not generic. ◇ ◇ ◇

The *formal parameter part* of a function declaration consists of the formal type parameter list, if any, and the formal parameter list. ◇

The following kinds of functions cannot be generic: Getters, setters, operators, and constructors.

The formal type parameter list of a function declaration introduces a new scope known as the function's *type parameter scope*. The type parameter scope of a generic function f is enclosed in the scope where f is declared. Every formal type parameter introduces a type into the type parameter scope. ◇

If it exists, the type parameter scope of a function f is the current scope for the signature of f, and for the formal type parameter list itself; otherwise the scope where f is declared is the current scope for the signature of f.

This means that formal type parameters are in scope in the bounds of parameter declarations, allowing for so-called F-bounded type parameters like `class C<X extends Comparable<X>> { ... }`, and the formal type parameters are in scope for each other, allowing dependencies like `class D<X extends Y, Y> { ... }`.

The formal parameter list of a function declaration introduces a new scope known as the function's *formal parameter scope*. The formal parameter scope of a non-generic function f is enclosed in the scope where f is declared. The formal parameter scope of a generic function f is enclosed in the type parameter scope of f. Every formal parameter introduces a local variable into the formal parameter scope. The current scope for the function's signature is the scope that encloses the formal parameter scope. ◇

This means that in a generic function declaration, the return type and parameter type annotations can use the formal type parameters, but the formal parameters are not in scope in the signature.

The body of a function declaration introduces a new scope known as the function's *body scope*. The body scope of a function f is enclosed in the scope introduced by the formal parameter scope of f. ◇

It is a compile-time error if a formal parameter is declared as a constant variable (8).

⟨*formalParameterList*⟩ ::= '(' ')'
| '(' ⟨*normalFormalParameters*⟩ ','? ')'
| '(' ⟨*normalFormalParameters*⟩ ',' ⟨*optionalOrNamedFormalParameters*⟩ ')'
| '(' ⟨*optionalOrNamedFormalParameters*⟩ ')'

⟨*normalFormalParameters*⟩ ::=
⟨*normalFormalParameter*⟩ (',' ⟨*normalFormalParameter*⟩)*

⟨*optionalOrNamedFormalParameters*⟩ ::= ⟨*optionalPositionalFormalParameters*⟩
| ⟨*namedFormalParameters*⟩

⟨*optionalPositionalFormalParameters*⟩ ::=
'[' ⟨*defaultFormalParameter*⟩ (',' ⟨*defaultFormalParameter*⟩)* ','? ']'

⟨*namedFormalParameters*⟩ ::=
'{' ⟨*defaultNamedParameter*⟩ (',' ⟨*defaultNamedParameter*⟩)* ','? '}'

Formal parameter lists allow an optional trailing comma after the last parameter (‘`,`’?). A parameter list with such a trailing comma is equivalent in all ways to the same parameter list without the trailing comma. All parameter lists in this specification are shown without a trailing comma, but the rules and semantics apply equally to the corresponding parameter list with a trailing comma.

9.2.1 Required Formals

A *required formal parameter* may be specified in one of three ways: ◇

- By means of a function signature that names the parameter and describes its type as a function type (20.5). It is a compile-time error if any default values are specified in the signature of such a function type.
- As an initializing formal, which is only valid as a parameter to a generative constructor (10.7.1).
- Via an ordinary variable declaration (8).

⟨*normalFormalParameter*⟩ ::=
  ⟨*metadata*⟩ ⟨*normalFormalParameterNoMetadata*⟩

⟨*normalFormalParameterNoMetadata*⟩ ::= ⟨*functionFormalParameter*⟩
  | ⟨*fieldFormalParameter*⟩
  | ⟨*simpleFormalParameter*⟩

⟨*functionFormalParameter*⟩ ::=
  covariant? ⟨*type*⟩? ⟨*identifier*⟩ ⟨*formalParameterPart*⟩ ‘`?`’?

⟨*simpleFormalParameter*⟩ ::= ⟨*declaredIdentifier*⟩
  | **covariant**? ⟨*identifier*⟩

⟨*declaredIdentifier*⟩ ::= **covariant**? ⟨*finalConstVarOrType*⟩ ⟨*identifier*⟩

⟨*fieldFormalParameter*⟩ ::=
  ⟨*finalConstVarOrType*⟩? **this** ‘`.`’ ⟨*identifier*⟩ (⟨*formalParameterPart*⟩ ‘`?`’?)?

It is a compile-time error if a formal parameter has the modifier **const** or the modifier **late**. It is a compile-time error if **var** occurs as the first token of a ⟨*fieldFormalParameter*⟩.

It is a compile-time error if a parameter derived from ⟨*fieldFormalParameter*⟩ occurs as a parameter of a function which is not a non-redirecting generative constructor.

A ⟨*fieldFormalParameter*⟩ declares an initializing formal, which is described elsewhere (10.7.1).

It is possible to include the modifier **covariant** in some forms of parameter

declarations. The effect of doing this is described in a separate section (9.2.3).

Note that the non-terminal ⟨*normalFormalParameter*⟩ is also used in the grammar rules for optional parameters, which means that such parameters can also be covariant.

It is a compile-time error if the modifier **covariant** occurs on a parameter of a function which is not an instance method, instance setter, or instance operator.

9.2.2 Optional Formals

Optional parameters may be specified and provided with default values.

⟨*defaultFormalParameter*⟩ ::= ⟨*normalFormalParameter*⟩ (‘=’ ⟨*expression*⟩)?

⟨*defaultNamedParameter*⟩ ::=
⟨*metadata*⟩ **required**? ⟨*normalFormalParameterNoMetadata*⟩
((‘=’ | ‘:’) ⟨*expression*⟩)?

The form ⟨*normalFormalParameter*⟩ ‘:’ ⟨*expression*⟩ is equivalent to the form ⟨*normalFormalParameter*⟩ ‘=’ ⟨*expression*⟩. The colon-syntax is included only for backwards compatibility. It is deprecated and will be removed in a later version of the language specification.

It is a compile-time error if the default value of an optional parameter is not a constant expression (17.3). If no default is explicitly specified for an optional parameter an implicit default of **null** is provided.

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.

9.2.3 Covariant Parameters

Dart allows formal parameters of instance methods, including setters and operators, to be declared **covariant**.

The syntax for doing this is specified in an earlier section (9.2.1).

It is a compile-time error if the modifier **covariant** occurs in the declaration of a formal parameter of a function which is not an instance method, an instance setter, or an operator.

As specified below, a parameter can also be covariant for other reasons. The overall effect of having a covariant parameter p in the signature of a given method m is to allow the type of p to be overridden covariantly, which means that the type

required at run time for a given actual argument may be a proper subtype of the type which is known at compile time at the call site.

This mechanism allows developers to explicitly request that a compile-time guarantee which is otherwise supported (namely: that an actual argument whose static type satisfies the requirement will also do so at run time) is replaced by dynamic type checks. In return for accepting these dynamic type checks, developers can use covariant parameters to express software designs where the dynamic type checks are known (or at least trusted) to succeed, based on reasoning that the static type analysis does not capture.

Let m be a method signature with formal type parameters $X_1, \ldots, X_s$, positional formal parameters $p_1, \ldots, p_n$, and named formal parameters $q_1, \ldots, q_k$. Let m' be a method signature with formal type parameters $X'_1, \ldots, X'_s$, positional formal parameters $p'_1, \ldots, p'_{n'}$, and named formal parameters $q'_1, \ldots, q'_{k'}$. Assume that $j \in 1..n'$, and $j \leq n$; we say that p'_j is the parameter in m' that *corresponds* to the formal parameter p_j in m. Assume that $j \in 1..k'$ and $l \in 1..k$; we say that q'_j is the parameter in m' that *corresponds* to the formal parameter q_l in m if $q'_j = q_l$. Similarly, we say that the formal type parameter X'_j from m' *corresponds* to the formal type parameter X_j from m, for all $j \in 1..s$.

m, X_j, s
p_j, n, q_j, k
m', X'_j
p'_j, n', q'_j, k'
$\diamond$
$\diamond$
$\diamond$

This includes the case where m respectively m' has optional positional parameters, in which case $k = 0$ respectively $k' = 0$ must hold, but we can have $n \neq n'$. The case where the numbers of formal type parameters differ is not relevant.

Let C be a class that declares a method m which has a parameter p whose declaration has the modifier **covariant**; in this case we say that the parameter p is *covariant-by-declaration*. In this case the interface of C has the method signature m, and that signature has the parameter p; we also say that the parameter p in this method signature is *covariant-by-declaration*. Finally, the parameter p of the method signature m of the interface of a class C is *covariant-by-declaration* if a direct superinterface of C has an accessible method signature m' with the same name as m, which has a parameter p' that corresponds to p, such that p' is covariant-by-declaration.

C, m, p
$\diamond$
$\diamond$
$\diamond$

Assume that C is a generic class with formal type parameter declarations X_1 **extends** $B_1 \ldots, X_s$ **extends** B_s, let m be a declaration of an instance method in C (which can be a method, a setter, or an operator), let p be a parameter declared by m, and let T be the declared type of p. The parameter p is *covariant-by-class* if, for any $j \in 1..s$, X_j occurs in a covariant or an invariant position in T. In this case the interface of C also has the method signature m, and that signature has the parameter p; we also say that the parameter p in this method signature is *covariant-by-class*. Finally, the parameter p of the method signature m of the interface of the class C is *covariant-by-class* if a direct superinterface of C has an accessible method signature m' with the same name as m, which has a parameter p' that corresponds to p, such that p' is covariant-by-class.

C, X_j, B_j, s
m, p, T
$\diamond$
$\diamond$
$\diamond$

A formal parameter p is *covariant* if p is covariant-by-declaration or p is covariant-by-class.

$\diamond$

It is possible for a parameter to be simultaneously covariant-by-declaration and covariant-by-class. Note that a parameter may be covariant-by-declaration

or covariant-by-class based on a declaration in any direct or indirect superinterface, including any superclass: The definitions above propagate these properties to an interface from each of its direct superinterfaces, but they will in turn receive the property from their direct superinterfaces, and so on.

9.3 Type of a Function

This section specifies the static type which is ascribed to the function denoted by a function declaration, and the dynamic type of the corresponding function object.

In this specification, the notation used to denote the type of a function, that is, a *function type*, follows the syntax of the language, except that **extends** is abbreviated to $\lhd$. This means that every function type is of one of the forms ◇

T_0 **Function**`<`$X_1 \lhd B_1, \ldots, X_s \lhd B_s$`>(`$T_1, \ldots, T_n$`, [`$T_{n+1}, \ldots, T_{n+k}$`])`

T_0 **Function**`<`$X_1 \lhd B_1, \ldots, X_s \lhd B_s$`>(`$T_1, \ldots, T_n$`, {`$T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}$`})`

where T_0 is the return type, X_j are the formal type parameters with bounds B_j, $j \in 1..s$, T_j are the formal parameter types for $j \in 1..n+k$, and x_{n+j} are the names of named parameters for $j \in 1..k$. Non-generic function types are covered by the case $s = 0$, where the type parameter declaration list `<...>` as a whole is omitted. Similarly, the optional brackets `[]` and `{}` are omitted when there are no optional parameters.

Both forms with optionals cover function types with no optionals when $k = 0$, and every rule in this specification is such that any of the two forms may be used without ambiguity to determine the treatment of function types with no optionals.

If a function declaration does not declare a return type explicitly, its return type is **dynamic** (20.7), unless it is a constructor, in which case it is not considered to have a return type, or it is a setter or operator `[]=`, in which case its return type is **void**.

A function declaration may declare formal type parameters. The type of the function includes the names of the type parameters and for each type parameter the upper bound, which is considered to be the built-in class `Object` if no bound is specified. When consistent renaming of type parameters can make two function types identical, they are considered to be the same type.

It is convenient to include the formal type parameter names in function types because they are needed in order to express such things as relations among different type parameters, F-bounds, and the types of formal parameters. However, we do not wish to distinguish between two function types if they have the same structure and only differ in the choice of names. This treatment of names is also known as alpha-equivalence.

In the following three paragraphs, if the number s of formal type parameters is zero then the type parameter list in the function type is omitted. s

Let F be a function with type parameters X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s, required formal parameter types $T_1, \ldots, T_n$, return type T_0, and no optional parameters. Then the static type of F is

T_0 **Function**`<`$X_1 \lhd B_1, \ldots, X_s \lhd B_s$`>(`$T_1, \ldots, T_n$`)`.

Let F be a function with type parameters X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s,

required formal parameter types $T_1, \ldots, T_n$, return type T_0 and positional optional parameter types $T_{n+1}, \ldots, T_{n+k}$. Then the static type of F is
T_0 **Function**<$X_1 \lhd B_1, \ldots, X_s \lhd B_s$>($T_1, \ldots, T_n$, [$T_{n+1}, \ldots, T_{n+k}$]).

Let F be a function with type parameters X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s, required formal parameter types $T_1, \ldots, T_n$, return type T_0, and named parameters $T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}$, where $x_{n+j}, j \in 1..k$ may or may not have a default value. Then the static type of F is
T_0 **Function**<$X_1 \lhd B_1, \ldots, X_s \lhd B_s$>($T_1, \ldots, T_n$, {$T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}$}).

Let T be the static type of a function declaration F. Let u be the run-time type of a function object o obtained by function closurization (17.16) or instance method closurization (17.22.3) applied to F, and let t be the actual type corresponding to T at the occasion where o was created (20.10.1). T may contain free type variables, but t contains their actual values. The following must then hold: u is a class that implements the built-in class **Function**; u is a subtype of t; and u is not a subtype of any function type which is a proper subtype of t. If we had omitted the last requirement then `f is int Function([int])` could evaluate to **true** with the declaration **void** `f() {}`, which is obviously not the intention.

It is up to the implementation to choose an appropriate representation for function objects. For example, consider that a function object produced via property extraction treats equality differently from other function objects, and is therefore likely a different class. Implementations may also use different classes for function objects based on arity and or type. Arity may be implicitly affected by whether a function is an instance method (with an implicit receiver parameter) or not. The variations are manifold and, e.g., one cannot assume that any two distinct function objects will necessarily have the same run-time type.

9.4 External Functions

An *external function* is a function whose body is provided separately from its declaration. An external function may be a top-level function (19), a method (10.2, 10.8), a getter (10.3), a setter (10.4), or a non-redirecting constructor (10.7.1, 10.7.2). External functions are introduced via the built-in identifier **external** (17.38) followed by the function signature. ◇

External functions allow us to introduce type information for code that is not statically known to the Dart compiler.

Examples of external functions might be foreign functions (defined in C, or Javascript etc.), primitives of the implementation (as defined by the Dart run-time system), or code that was dynamically generated but whose interface is statically known. However, an abstract method is different from an external function, as it has *no* body.

An external function is connected to its body by an implementation specific mechanism. Attempting to invoke an external function that has not been connected to its body will throw a `NoSuchMethodError` or some subclass thereof.

An implementation specific compile-time error can be raised at an **external** function declaration.

Such errors are intended to indicate that every invocation of that function would throw, e.g., because it is known that it will not be connected to a body.

The actual syntax is given in sections 10 and 19 below.

10 Classes

A *class* defines the form and behavior of a set of objects which are its *instances*. Classes may be defined by class declarations as described below, or via mixin applications (12.3). ◇ ◇

⟨*classDeclaration*⟩ ::= **abstract**? **class** ⟨*typeIdentifier*⟩ ⟨*typeParameters*⟩?
  ⟨*superclass*⟩? ⟨*interfaces*⟩?
  '{' (⟨*metadata*⟩ ⟨*classMemberDeclaration*⟩)* '}'
| **abstract**? **class** ⟨*mixinApplicationClass*⟩

⟨*typeNotVoidList*⟩ ::= ⟨*typeNotVoid*⟩ (',' ⟨*typeNotVoid*⟩)*

⟨*classMemberDeclaration*⟩ ::= ⟨*declaration*⟩ ';'
| ⟨*methodSignature*⟩ ⟨*functionBody*⟩

⟨*methodSignature*⟩ ::= ⟨*constructorSignature*⟩ ⟨*initializers*⟩?
| ⟨*factoryConstructorSignature*⟩
| **static**? ⟨*functionSignature*⟩
| **static**? ⟨*getterSignature*⟩
| **static**? ⟨*setterSignature*⟩
| ⟨*operatorSignature*⟩

⟨*declaration*⟩ ::= **external** ⟨*factoryConstructorSignature*⟩
| **external** ⟨*constantConstructorSignature*⟩
| **external** ⟨*constructorSignature*⟩
| (**external static**?)? ⟨*getterSignature*⟩
| (**external static**?)? ⟨*setterSignature*⟩
| (**external static**?)? ⟨*functionSignature*⟩
| **external**? ⟨*operatorSignature*⟩
| **static const** ⟨*type*⟩? ⟨*staticFinalDeclarationList*⟩
| **static final** ⟨*type*⟩? ⟨*staticFinalDeclarationList*⟩
| **static late final** ⟨*type*⟩? ⟨*initializedIdentifierList*⟩
| **static late**? ⟨*varOrType*⟩ ⟨*initializedIdentifierList*⟩
| **covariant late final** ⟨*type*⟩? ⟨*identifierList*⟩
| **covariant late**? ⟨*varOrType*⟩ ⟨*initializedIdentifierList*⟩
| **late**? **final** ⟨*type*⟩? ⟨*initializedIdentifierList*⟩
| **late**? ⟨*varOrType*⟩ ⟨*initializedIdentifierList*⟩
| ⟨*redirectingFactoryConstructorSignature*⟩
| ⟨*constantConstructorSignature*⟩ (⟨*redirection*⟩ | ⟨*initializers*⟩)?
| ⟨*constructorSignature*⟩ (⟨*redirection*⟩ | ⟨*initializers*⟩)?

⟨*staticFinalDeclarationList*⟩ ::=
⟨*staticFinalDeclaration*⟩ (‘**,**’ ⟨*staticFinalDeclaration*⟩)*

⟨*staticFinalDeclaration*⟩ ::= ⟨*identifier*⟩ ‘=’ ⟨*expression*⟩

It is possible to include the modifier **covariant** in some forms of declarations. The effect of doing this is described elsewhere (9.2.3).

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 class variables. The *members* of a class are its static and instance members. ◇

A class declaration introduces two scopes:

- A *type-parameter scope*, which is empty if the class is not generic (15). The enclosing scope of the type-parameter scope of a class declaration is the library scope of the current library. ◇
- A *body scope*. The enclosing scope of the body scope of a class declaration is the type parameter scope of the class declaration. ◇

The current scope of an instance member declaration, a static member declaration, or a constructor declaration is the body scope of the class in which it is declared.

The current instance (and hence its members) can only be accessed at specific locations in a class: We say that a location ℓ *has access to* ***this*** iff ℓ is inside the body of a declaration of an instance member or a generative constructor, or in the initializing expression of a **late** instance variable declaration. ◇

Note that an initializing expression for a non-**late** instance variable does not have access to **this**, and neither does any part of a declaration marked **static**.

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 (10.10).

An *abstract class declaration* is a class declaration that is explicitly declared with the **abstract** modifier. A *concrete class declaration* is a class declaration that is not abstract. An *abstract class* is a class whose declaration is abstract, and a *concrete class* is a class whose declaration is concrete. ◇ ◇ ◇ ◇

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.

The interface of a class C is an implicit interface that declares instance member signatures that correspond to the instance members declared by C, and whose direct superinterfaces are the direct superinterfaces of C (11, 10.10).

When a class name appears as a type, that name denotes the interface of the class.

It is a compile-time error if a class named C declares a member with base-

name (10.11) C. If a generic class named G declares a type variable named X, it is a compile-time error if X is equal to G, or if G has a member whose basename is X, or if G has a constructor named $G.X$.

Here are simple examples, that illustrate the difference between "has a member" and "declares a member". For example, `B` *declares* one member named `f`, but it *has* two such members. The rules of inheritance determine what members a class has. ◇ ◇

```
class A {
  var i = 0;
  var j;
  f(x) => 3;
}

class B extends A {
  int i = 1; // getter i and setter i= override versions from A
  static j; // compile-time error: static getter & setter conflict
  // with instance getter & setter

  // compile-time error: static method conflicts with instance method
  static f(x) => 3;
}
```

10.1 Fully Implementing an Interface

A concrete class must fully implement its interface. Let C be a concrete class declared in library L, with interface I. Assume that I has a member signature m which is accessible to L. It is a compile-time error if C does not have a concrete member with the same name as m and accessible to L, unless C has a non-trivial `noSuchMethod` (10.2.2). C, I, m

Each concrete member must have a suitable signature: Assume that C has a concrete member with the same name as m and accessible to L, and let m'' be its member signature. The concrete member may be declared in C or inherited from a superclass. Let m' be the member signature which is obtained from m'' by adding, if not present already, the modifier **covariant** (9.2.3) to each parameter p in m'' where the corresponding parameter in m has the modifier **covariant**. It is a compile-time error if m' is not a correct override of m (11.2.2), unless that concrete member is a `noSuchMethod` forwarder (10.2.2). m'' m'

Consider a concrete class `C`, and assume that `C` declares or inherits a member implementation with the same name for every member signature in its interface. It is still an error if one or more of those member implementations has parameters or types such that they do not satisfy the corresponding member signature in the interface. For this check, any missing **covariant** modifiers are implicitly added to the signature of an inherited member (this is how we get m' from m''). When the modifier **covariant** is added to one or more parameters (which will only happen

when the concrete member is inherited), an implementation may choose to implicitly induce a forwarding method with the same signature as m', in order to perform the required dynamic type check, and then invoke the inherited method.

It is an implementation specific choice whether or not an implicitly induced forwarding method is used when the modifier **covariant** is added to one or more parameters in m'.

This is true in spite of the fact that such forwarding methods can be observed. E.g., we can compare the run-time type of a tearoff of the method from a receiver of type `C` to the run-time type of a tearoff of the super-method from a location in the body of `C`.

With or without a forwarding method, the member signature in the interface of C is m.

The forwarding method does not change the interface of `C`, it is an implementation detail. In particular, this holds even in the case where an explicit declaration of the forwarding method would have changed the interface of `C`, because m' is a subtype of m.

When a class has a non-trivial `noSuchMethod`, the class may leave some members unimplemented, and the class is allowed to have a `noSuchMethod` forwarder which does not satisfy the class interface (in which case it will be overridden by another `noSuchMethod` forwarder).

Here is an example:

```
class B {
  void m(int i) {} // Signature m'': void m(int).
}

abstract class I {
  void m(covariant num n); // Signature: void m(covariant num).
}

class C extends B implements I {
  // Signature m: void m(covariant num).
  //
  // To check that this class fully implements its interface,
  // check that m', that is, void m(covariant int),
  // correctly overrides m: OK!
}
```

Parameters that are covariant-by-declaration must also satisfy the following constraint: Assume that the parameter p of m' has the modifier **covariant**. Assume that a direct or indirect superinterface of C has a method signature m_s with the same name as m' and accessible to L, such that m_s has a parameter p_s that corresponds to p. In this situation, a compile-time error occurs if the type of p is not a subtype and not a supertype of the type of p_s.

This ensures that an inherited method satisfies the same constraint for each formal parameter which is covariant-by-declaration as the constraint which is specified

for a declaration in C (10.2).

10.2 Instance Methods

Instance methods are functions (9) whose declarations are immediately contained within a class declaration and that are not declared **static**. The *instance methods of a class* C are the instance methods declared by C and the instance methods inherited by C from its superclass (10.9.1). ◇ ◇

Consider a class C and an instance member declaration D in C, with member signature m (11). It is a compile-time error if D overrides a declaration with member signature m' from a direct superinterface of C (11.2.1), unless m is a correct member override of m' (11.2.2). C, D, m

This is not the only kind of conflict that may exist: An instance member declaration D may conflict with another declaration D', even in the case where they do not have the same name or they are not the same kind of declaration. E.g., D could be an instance getter and D' a static setter (10.11).

For each parameter p of m where **covariant** is present, it is a compile-time error if there exists a direct or indirect superinterface of C which has an accessible method signature m'' with the same name as m, such that m'' has a parameter p'' that corresponds to p (9.2.3), unless the type of p is a subtype or a supertype of the type of p''.

This means that a parameter which is covariant-by-declaration can have a type which is a supertype or a subtype of the type of a corresponding parameter in a superinterface, but the two types cannot be unrelated. Note that this requirement must be satisfied for each direct or indirect superinterface separately, because that relationship is not transitive.

The superinterface may be the statically known type of the receiver, so this means that we relax the potential typing relationship between the statically known type of a parameter and the type which is actually required at run time to the subtype-or-supertype relationship, rather than the strict supertype relationship which applies to a parameter which is not covariant. It should be noted that it is not statically known at the call site whether any given parameter is covariant, because the covariance could be introduced in a proper subtype of the statically known type of the receiver. We chose to give priority to flexibility rather than safety here, because the whole point of covariant parameters is that developers can make the choice to increase the flexibility in a trade-off where some static type safety is lost.

10.2.1 Operators

Operators are instance methods with special names, except for operator '`[]`' which is an instance getter and operator '`[]=`' which is an instance setter. ◇

⟨*operatorSignature*⟩ ::=
    ⟨*type*⟩? **operator** ⟨*operator*⟩ ⟨*formalParameterList*⟩

⟨*operator*⟩ ::= '`~`'
| ⟨*binaryOperator*⟩
| '`[]`'
| '`[]=`'

⟨*binaryOperator*⟩ ::= ⟨*multiplicativeOperator*⟩
| ⟨*additiveOperator*⟩
| ⟨*shiftOperator*⟩
| ⟨*relationalOperator*⟩
| '`==`'
| ⟨*bitwiseOperator*⟩

An operator declaration is identified using the built-in identifier (17.38) **operator**.

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

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

The '`-`' operator is unique in that two overloaded versions are permitted. If the operator has no arguments, it denotes unary minus. If it has an argument, it denotes binary subtraction.

The name of the unary operator '`-`' is `unary-`.

This device allows the two methods to be distinguished for purposes of method lookup, override and reflection.

It is a compile-time error if the arity of the user-declared operator '`~`' is not 0.

It is a compile-time error to declare an optional parameter in an operator.

It is a compile-time error if a user-declared operator '`[]=`' declares a return type other than **void**.

If no return type is specified for a user-declared operator '`[]=`', its return type is **void** (9.3).

*The return type is **void** because a return statement in an implementation of operator '`[]=`' does not return an object. Consider a non-throwing evaluation of an expression e of the form e_1 `[`e_2`]` = e_3, and assume that the evaluation of e_3 yields an object o. e will then evaluate to o, and even if the executed body of operator '`[]=`' completes with an object o', that is, if o' is returned it is simply ignored. The rationale for this behavior is that assignments should be guaranteed to evaluate to the assigned object.*

10.2.2 The Method `noSuchMethod`

The method `noSuchMethod` is invoked implicitly during execution in situations where one or more member lookups fail (17.21.1, 17.22.1, 17.23).

We may think of `noSuchMethod` as a backup which kicks in when an invocation of a member m is attempted, but there is no member named m, or it exists, but the given invocation has an argument list shape that does not fit the declaration of m (passing fewer positional arguments than required or more than supported, or passing named arguments with names not declared by m). This can only occur for an ordinary method invocation when the receiver has static type **dynamic**, or for a function invocation when the invoked function has static type **Function** or **dynamic**. The method `noSuchMethod` can also be invoked in other ways, e.g., it can be called explicitly like any other method, and it can be invoked from a `noSuchMethod` forwarder, as explained below.

We say that a class C *has a non-trivial* ***`noSuchMethod`*** if C has a concrete member named `noSuchMethod` which is distinct from the one declared in the built-in class `Object`. ◇

Note that it must be a method that accepts one positional argument, in order to correctly override `noSuchMethod` in `Object`. For instance, it can have signature `noSuchMethod(Invocation i)` or `noSuchMethod(Object i, [String s = "])`, but not `noSuchMethod(Invocation i, String s)`. This implies that the situation where `noSuchMethod` is invoked (explicitly or implicitly) with one actual argument cannot fail for the reason that "there is no such method", such that we would enter an infinite loop trying to invoke `noSuchMethod`. It *is* possible, however, to encounter a dynamic error during an invocation of `noSuchMethod` because the actual argument fails to satisfy a type check, but that situation will give rise to a dynamic type error rather than a repeated attempt to invoke `noSuchMethod` (17.15.3). Here is an example where a dynamic type error occurs because an attempt is made to pass an `Invocation` where only the null object is accepted:

```
class A {
  noSuchMethod(covariant Null n) => n;
}

void main() {
  dynamic d = A();
  d.foo(42); // Dynamic type error when invoking noSuchMethod.
}
```

Let C be a concrete class, let L be the library that contains the declaration of C, and let m be a name. Then m is *noSuchMethod forwarded* in C iff one of the following is true: C, L, m ◇

- **Requested in program:** C has a non-trivial `noSuchMethod`, the interface of C contains a member signature S named m, and C has no concrete member named m and accessible to L that correctly overrides S (that is, no member named m is declared or inherited by C, or one is inherited, but it does not have the required signature). In this case we also say that S is noSuchMethod forwarded.

- **Forced by privacy:** There exists a direct or indirect superinterface D of C which is declared in a library L_2 different from L, the interface of D contains a member signature S named m, m is a private name, and no superclass of C has a concrete member named m accessible to L_2 that correctly overrides S. In this case we also say that S is noSuchMethod forwarded.

For a concrete class C, a *noSuchMethod forwarder* is implicitly induced for each member signature which is noSuchMethod forwarded. ◇

It is a compile-time error if the name m is noSuchMethod forwarded in a concrete class C, and a superclass of C has an accessible concrete declaration of m which is not a noSuchMethod forwarder.

A noSuchMethod forwarder is a concrete member of C with the signature taken from the interface of C, and with the same default value for each optional parameter. It can be invoked in an ordinary invocation and in a superinvocation, and when m is a method it can be closurized (17.22.3) using a property extraction (17.22).

The error concerned with an implictly induced forwarder that would override a human-written declaration can only occur if that concrete declaration does not correctly override S. Consider the following example:

```
class A {
  foo(int i) => null;
}
abstract class B {
  foo([int i]);
}
class C extends A implements B {
  noSuchMethod(Invocation i) => ...;
  // Error: noSuchMethod forwarder cannot override ‘A.foo‘.
}
```

In this example, an implementation with signature `foo(int i)` is inherited by `C`, and the superinterface `B` declares the signature `foo([int i])`. This is a compile-time error because `C` does not have a method implementation with signature `foo([int])`. We do not wish to implicitly induce a `noSuchMethod` forwarder with signature `foo([int])` because it would override `A.foo`, and that is likely to be highly confusing for developers. In particular, it would cause an invocation like `C().foo(42)` to invoke `noSuchMethod`, even though that is an invocation which is correct for the declaration of `foo` in `A`. Hence, we require developers to explicitly resolve the conflict whenever an implicitly induced `noSuchMethod` forwarder would override an explicitly declared inherited implementation. It is no problem, however, to let a `noSuchMethod` forwarder override another `noSuchMethod` forwarder, and hence there is no error in that situation.

This implies that a `noSuchMethod` forwarder has the same properties as an explicitly declared concrete member, except of course that a `noSuchMethod` for-

warder does not prevent itself or another noSuchMethod forwarder from being induced. We do not specify the body of a `noSuchMethod` forwarder, but it will invoke `noSuchMethod`, and we specify the dynamic semantics of executing it below.

At the beginning of this section we mentioned that implicit invocations of `noSuchMethod` can only occur with a receiver of static type **dynamic** or a function of static type **dynamic** or **Function**. With a `noSuchMethod` forwarder, `noSuchMethod` can also be invoked on a receiver whose static type is not **dynamic**. No similar situation exists for functions, because it is impossible to induce a `noSuchMethod` forwarder into the class of a function object.

For a concrete class C, we may think of a non-trivial `noSuchMethod` (declared in or inherited by C) as a request for "automatic implementation" of all unimplemented members in the interface of C as `noSuchMethod` forwarders. Similarly, there is an implicit request for automatic implementation of all unimplemented inaccessible members of any concrete class, whether or not there is a non-trivial `noSuchMethod`. Note that the latter cannot be written explicitly in Dart, because their names are inaccessible; but the language can still specify that they are induced implicitly, because compilers control the treatment of private names.

For the dynamic semantics, assume that a class C has an implicitly induced `noSuchMethod` forwarder named m, with formal type parameters $X_1, \ldots, X_r$, positional formal parameters $a_1, \ldots, a_k$ (some of which may be optional when $n = 0$), and named formal parameters with names $x_1, \ldots, x_n$ (with default values as mentioned above). C, m, X_j, r a_j, k, x_j, n

For this purpose we need not distinguish between a signature that has optional positional parameters and a signature that has named parameters, because the former is covered by $n = 0$.

The execution of the body of m creates an instance im of the predefined class `Invocation` such that: im

- im`.isMethod` evaluates to **true** iff m is a method.
- im`.isGetter` evaluates to **true** iff m is a getter.
- im`.isSetter` evaluates to **true** iff m is a setter.
- im`.memberName` evaluates to the symbol `m`.
- im`.positionalArguments` evaluates to an unmodifiable list whose dynamic type implements `List<Object>`, containing the same objects as the list resulting from evaluation of `<Object>[`$a_1, \ldots, a_k$`]`.
- im`.namedArguments` evaluates to an unmodifiable map whose dynamic type implements `Map<Symbol, Object>`, with the same keys and values as the map resulting from evaluation of

  `<Symbol, Object>{`$\#x_1$`:` $x_1, \ldots,$ $\#x_m$`:` x_m`}`.
- im`.typeArguments` evaluates to an unmodifiable list whose dynamic type implements `List<Type>`, containing the same objects as the list resulting from evaluation of `<Type>[`$X_1, \ldots, X_r$`]`.

Next, `noSuchMethod` is invoked with im as the actual argument, and the result obtained from there is returned by the execution of m.

This is an ordinary method invocation of `noSuchMethod` (17.21.1). That is, a `noSuchMethod` forwarder in a class C can invoke an implementation of `noSuchMethod` that is declared in a subclass of C.

Dynamic type checks on the actual arguments passed to m are performed in the same way as for an invocation of an explicitly declared method. In particular, an actual argument passed to a covariant parameter will be checked dynamically.

Also, like other ordinary method invocations, it is a dynamic type error if the result returned by a `noSuchMethod` forwarder has a type which is not a subtype of the return type of the forwarder.

One special case to be aware of is where a forwarder is torn off and then invoked with an actual argument list which does not match the formal parameter list. In that situation we will get an invocation of `Object.noSuchMethod` rather than the `noSuchMethod` in the original receiver, because this is an invocation of a function object (and they do not override `noSuchMethod`):

```
class A {
  noSuchMethod(Invocation i) => null;
  void foo();
}

void main() {
  A a = A();
  Function f = a.foo;
  // Invokes `Object.noSuchMethod`, which throws.
  f(42);
}
```

10.2.3 The Operator '==' and Primitive Equality

The operator '`==`' is used implicitly in certain situations, and in particular constant expressions (17.3) give rise to constraints on that operator. The situation is similar with the getter `hashCode`. In order to specify these constraints just once we introduce the notion of *primitive equality*. ◇

Certain constant expressions are known to have a value whose equality is primitive. This is useful to know because it allows the value of equality expressions and the value of invocations of `hashCode` to be computed at compile-time. In particular, this can be used to build constant collections at compile-time, and it can be used to check that all elements in a constant set are distinct, and all keys in a constant map are distinct.

- The null object has primitive equality (17.4).
- Every instance of type `bool`, `int`, and `String` has primitive equality.

- Every instance of type `Symbol` which was originally obtained by evaluation of a literal symbol or a constant invocation of a constructor of the `Symbol` class has primitive equality.
- Every instance of type `Type` which was originally obtained by evaluating a constant type literal (20.2) has primitive equality.
- Let o be an object obtained by evaluation of a constant list literal (17.9.4), a constant map literal (17.9.8), or a constant set literal (17.9.7), then o has primitive equality.
- A function object obtained by function closurization of a static method or a top-level function (17.16) as the value of a constant expression has primitive equality.
- An instance o has primitive equality if the dynamic type of o is a class C, and C has primitive equality.
- The class `Object` has primitive equality.
- A class C has primitive equality if it does not have an implementation of the operator '`==`' that overrides the one inherited from `Object`, and it does not have an implementation of the getter `hashCode` that overrides the one inherited from `Object`.

When we say that a given instance or class *does not have primitive equality*, it means that it is not true that said instance or class has primitive equality. ◇

10.3 Getters

Getters are functions (9) that are used to retrieve the values of object properties.

⟨*getterSignature*⟩ ::= ⟨*type*⟩? **get** ⟨*identifier*⟩

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.

The *instance getters of a class* C are those instance getters declared by C, either implicitly or explicitly, and the instance getters inherited by C from its superclass. The *static getters of a class* C are those static getters declared by C. ◇ ◇

A getter declaration may conflict with other declarations (10.11). In particular, a getter can never override a method, and a method can never override a getter or an instance variable. The rules for when a getter correctly overrides another member are given elsewhere (11.2.2).

10.4 Setters

Setters are functions (9) that are used to set the values of object properties.

⟨*setterSignature*⟩ ::= ⟨*type*⟩? **set** ⟨*identifier*⟩ ⟨*formalParameterList*⟩

If no return type is specified, the return type of the setter is **void** (9.3).

A setter definition that is prefixed with the **static** modifier defines a static setter. Otherwise, it defines an instance setter. The name of a setter is obtained by appending the string '=' to the identifier given in its signature.

Hence, a setter name can never conflict with, override or be overridden by a getter or method.

The *instance setters of a class* C are those instance setters declared by C ◇ either implicitly or explicitly, and the instance setters inherited by C from its superclass. The *static setters of a class* C are those static setters declared by ◇ C, either implicitly or explicitly.

It is a compile-time error if a setter's formal parameter list does not consist 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 setter declares a return type other than **void**. It is a compile-time error if a class has a setter named v= with argument type T and a getter named v with return type S, and S may not be assigned to T.

The rules for when a setter correctly overrides another member are given elsewhere (11.2.2). A setter declaration may conflict with other declarations as well (10.11).

10.5 Abstract Instance Members

An *abstract method* (respectively, *abstract getter* or *abstract setter*) is an ◇ instance method, getter or setter that is not declared **external** and does not ◇ provide an implementation. A *concrete method* (respectively, *concrete getter* or ◇ *concrete setter*) is an instance method, getter or setter that is not abstract. ◇

Abstract instance members are useful because of their interplay with classes. ◇ *Every Dart class induces an implicit interface, and Dart does not support specifying interfaces explicitly. Using an abstract class instead of a traditional interface has important advantages. An abstract class can provide default implementations. It can also provide static methods, obviating the need for service classes such as* `Collections` *or* `Lists`*, whose entire purpose is to group utilities related to a given type.*

Invocation of an abstract method, getter, or setter cannot occur, because lookup (17.18) will never yield an abstract member as its result. One way to think about this is that an abstract member declaration in a subclass does not override or shadow an inherited member implementation. It only serves to specify the signature of the given member that every concrete subtype must have an implementation of; that is, it contributes to the interface of the class, not to the class itself.

The purpose of an abstract method is to provide a declaration for purposes such as type checking and reflection. In mixins, it is often useful to introduce

such declarations for methods that the mixin expects will be provided by the superclass the mixin is applied to.

We wish to detect if one declares a concrete class with abstract members. However, code like the following should work:

```
class Base {
  int get one => 1;
}

abstract class Mix {
  int get one;
  int get two => one + one;
}

class C extends Base with Mix { }
```

At run time, the concrete method `one` *declared in* `Base` *will be executed, and no problem should arise. Therefore no error should be raised if a corresponding concrete member exists in the hierarchy.*

10.6 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 the instance variables declared by C and the instance variables inherited by C from its superclass. ◇ ◇

It is a compile-time error if an instance variable is declared to be constant.

The notion of a constant instance variable is subtle and confusing to programmers. An instance variable is intended to vary per instance. A constant instance variable would have the same value for all instances, and as such is already a dubious idea.

The language could interpret const instance variable declarations as instance getters that return a constant. However, a constant instance variable could not be treated as a true compile-time constant, as its getter would be subject to overriding.

Given that the value does not depend on the instance, it is better to use a class variable. An instance getter for it can always be defined manually if desired.

It is possible for the declaration of an instance variable to include the modifier **covariant** (8). The effect of this is that the formal parameter of the corresponding implicitly induced setter is considered to be covariant-by-declaration (9.2.3).

The modifier **covariant** on an instance variable has no other effects. In particular, the return type of the implicitly induced getter can already be overridden covariantly without **covariant**, and it can never be overridden to a supertype or an unrelated type, regardless of whether the modifier **covariant** is present.

10.7 Constructors

A *constructor* is a special function that is used in instance creation expressions (17.13) to obtain objects, typically by creating or initializing them. Constructors may be generative (10.7.1) or they may be factories (10.7.2). ◇

A *constructor name* always begins with the name of its immediately enclosing class, and may optionally be followed by a dot and an identifier *id*. It is a compile-time error if the name of a constructor is not a constructor name. ◇

The *function type of a constructor* k is the function type whose return type is the class that contains the declaration of k, and whose formal parameter types, optionality, and names of named parameters correspond to the declaration of k. ◇

Note that the function type F of a constructor k may contain type variables declared by the enclosing class C. In that case we can apply a substitution to F, as in $[T_1/X_1, \ldots, T_m/X_m]F$, where $X_j, j \in 1..m$ are the formal type parameters of C and $T_j, j \in 1..m$ are specified in the given context. We may also omit such a substitution when the given context is the body scope of C, where $X_1, \ldots, X_m$ are in scope.

A constructor declaration may conflict with static member declarations (10.11).

A constructor declaration does not introduce a name into a scope. If a function expression invocation (17.15.5) or an instance creation (17.13) denotes a constructor as C, $prefix.C$, $C.id$, or $prefix.C.id$, resolution relies on the library scope to determine the class (possibly via an import prefix). The class declaration is then directly checked for whether it has a constructor named C respectively $C.id$. It is not possible for an identifier to directly refer to a constructor, since the constructor is not in any scope used for resolving identifiers.

Iff no constructor is specified for a class C, it implicitly has a default constructor `C(): super() {}`, unless C is the built-in class `Object`.

10.7.1 Generative Constructors

A *generative constructor* declaration consists of a constructor name, a formal parameter list (9.2), and either a redirect clause or an initializer list and an optional body. ◇

⟨*constructorSignature*⟩ ::= ⟨*constructorName*⟩ ⟨*formalParameterList*⟩

⟨*constructorName*⟩ ::= ⟨*typeIdentifier*⟩ (‘`.`’ ⟨*identifier*⟩)?

See ⟨*declaration*⟩ and ⟨*methodSignature*⟩ for grammar rules introducing a redirection or an initializer list and a body.

A compile-time error occurs if a generative constructor declaration has a body of the form ‘`=>` e`;`’.

In other function declarations, this kind of body is taken to imply that the value of e is returned, but generative constructors do not return anything.

If a formal parameter declaration p is derived from ⟨*fieldFormalParameter*⟩, it declares an *initializing formal parameter*. A term of the form **this**`.`*id* is contained in p, and *id* is the *name* of p. It is a compile-time error if *id* is not ◇ ◇

also the name of an instance variable of the immediately enclosing class.

Note that it is a compile-time error for an initializing formal to occur in any function which is not a non-redirecting generative constructor (9.2.1), so there is always an enclosing class.

Assume that p is a declaration of an initializing formal parameter named id. Let T_{id} be the type of the instance variable named id in the immediately enclosing class. If p has a type annotation T then the declared type of p is T. Otherwise, the declared type of p is T_{id}. It is a compile-time error if the declared type of p is not a subtype of T_{id}.

Initializing formals constitute an exception to the rule that every formal parameter introduces a local variable into the formal parameter scope (9.2). When the formal parameter list of a non-redirecting generative constructor contains any initializing formals, a new scope is introduced, the *formal parameter initializer scope*, which is the current scope of the initializer list of the constructor, and which is enclosed in the scope where the constructor is declared. Each initializing formal in the formal parameter list introduces a final local variable into the formal parameter initializer scope, but not into the formal parameter scope; every other formal parameter introduces a local variable into both the formal parameter scope and the formal parameter initializer scope. ◇

This means that formal parameters, including initializing formals, must have distinct names, and that initializing formals are in scope for the initializer list, but they are not in scope for the body of the constructor. When a formal parameter introduces a local variable into two scopes, it is still one variable and hence one storage location. The type of the constructor is defined in terms of its formal parameters, including the initializing formals.

Initializing formals are executed during the execution of generative constructors detailed below. Executing an initializing formal **this**.*id* causes the instance variable *id* of the immediately surrounding class to be assigned the value of the corresponding actual parameter, unless the assigned object has a dynamic type which is not a subtype of the declared type of the instance variable *id*, in which case a dynamic error occurs.

The above rule allows initializing formals to be used as optional parameters:

```
class A {
  int x;
  A([this.x]);
}
```

is legal, and has the same effect as

```
class A {
  int x;
  A([int x]): this.x = x;
}
```

A *fresh instance* is an instance whose identity is distinct from any previously ◇

allocated instance of its class. A generative constructor always operates on 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 (17.13).

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

Redirecting Generative 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 which arguments. ◇

⟨*redirection*⟩ ::= '`:`' **this** ('`.`' ⟨*identifier*⟩)? ⟨*arguments*⟩

Assume that C`<`X_1 **extends** $B_1 \ldots,$ X_m **extends** B_m`>` is the name and formal type parameters of the enclosing class, **const**? stands for either **const** or nothing, N is C or $C.id_0$ for some identifier id_0, and id is an identifier. Consider a declaration of a redirecting generative constructor of one of the forms C, X_j, B_j, m **const**? N, id

const? N`(`$T_1\ x_1 \ldots,\ T_n\ x_n,$ `[`$T_{n+1}\ x_{n+1} = d_1 \ldots,\ T_{n+k}\ x_{n+k} = d_k$`]):` R`;`
const? N`(`$T_1\ x_1 \ldots,\ T_n\ x_n,$ `{`$T_{n+1}\ x_{n+1} = d_1 \ldots,\ T_{n+k}\ x_{n+k} = d_k$`}):` R`;`
where R is of one of the forms
this`(`$e_1 \ldots,\ e_p,\ x_1$`:` $e_{p+1}, \ldots,\ x_q$`:` e_{p+q}`)`
this$.id$`(`$e_1 \ldots,\ e_p,\ x_1$`:` $e_{p+1}, \ldots,\ x_q$`:` e_{p+q}`)`.

The *redirectee constructor* for this declaration is then the constructor denoted by C`<`$X_1 \ldots,\ X_m$`>` respectively C`<`$X_1 \ldots,\ X_m$`>.`id. It is a compile-time error if the static argument list type (17.15.1) of `(`$e_1 \ldots,\ e_p,\ x_1$`:` $e_{p+1}, \ldots,\ x_q$`:` e_{p+q}`)` is not an assignable match for the formal parameter list of the redirectee. ◇

Note that the case where no named parameters are passed is covered by letting q be zero, and the case where C is a non-generic class is covered by letting m be zero, in which case the formal type parameter list and actual type argument lists are omitted (15).

We require an assignable match rather than the stricter subtype match because a generative redirecting constructor k invokes its redirectee k' in a manner which resembles function invocation in general. For instance, k could accept an argument `x` *and pass on an expression e_j using* `x` *such as* `x.f(42)` *to k', and it would be surprising if e_j were subject to more strict constraints than the ones applied to actual arguments to function invocations in general.*

A redirecting generative constructor q' is *redirection-reachable* from a redirecting generative constructor q iff q' is the redirectee constructor of q, or q'' is the redirectee constructor of q and q' is redirection-reachable from q''. It is a compile-time error if a redirecting generative constructor is redirection-reachable from itself. ◇

When **const**? is **const**, it is a compile-time error if the redirectee is not a constant constructor. Moreover, when **const**? is **const**, each $e_i,\ i \in 1..p+q$, must be a potentially constant expression (10.7.3).

It is a dynamic type error if an actual argument passed in an invocation of a redirecting generative constructor k is not a subtype of the actual type (20.10.1) of the corresponding formal parameter in the declaration of k. It is a dynamic type error if an actual argument passed to the redirectee k' of a redirecting generative constructor is not a subtype of the actual type (20.10.1) of the corresponding formal parameter in the declaration of the redirectee.

Initializer Lists

An initializer list begins with a colon, and consists of a comma-separated list of individual *initializers*. ◇

There are three kinds of initializers.

- A *superinitializer* identifies 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 an object to an individual instance variable.
- An assertion.

⟨*initializers*⟩ ::= ‘`:`’ ⟨*initializerListEntry*⟩ (‘`,`’ ⟨*initializerListEntry*⟩)*

⟨*initializerListEntry*⟩ ::= **super** ⟨*arguments*⟩
| **super** ‘`.`’ ⟨*identifier*⟩ ⟨*arguments*⟩
| ⟨*fieldInitializer*⟩
| ⟨*assertion*⟩

⟨*fieldInitializer*⟩ ::=
(**this** ‘`.`’)? ⟨*identifier*⟩ ‘`=`’ ⟨*initializerExpression*⟩

⟨*initializerExpression*⟩ ::= ⟨*conditionalExpression*⟩ | ⟨*cascade*⟩

An initializer of the form v `=` e is equivalent to an initializer of the form **this**`.`v `=` e, both forms are called *instance variable initializers*. It is a compile-time error if the enclosing class does not declare an instance variable named v. It is a compile-time error unless the static type of e is assignable to the declared type of v. ◇

Consider a *superinitializer* s 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`(`$a_1, \ldots,\ a_n,\ x_{n+1}$`:`$\ a_{n+1}, \ldots,\ x_{n+k}$`:`$\ a_{n+k}$`)`.
Let S be the superclass of the enclosing class of s. It is a compile-time error if class S does not declare a generative constructor named S (respectively S`.`id). Otherwise, the static analysis of s is performed as specified in Section 17.15.3, as if **super** respectively **super**`.`id had had the function type of the denoted constructor, and substituting the formal type variables of the superclass for the corresponding actual type arguments passed to the superclass in the header of the current class.

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 a superinitializer appears in k's initializer list at any other position than at the end. 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. It is a compile-time error if k's initializer list contains an initializer for a final variable f whose declaration includes an initialization expression. It is a compile-time error if k includes an initializing formal for a final variable f whose declaration includes an initialization expression. k

Let f be a final instance variable declared in the immediately enclosing class. A compile-time error occurs unless f is initialized by one of the following means: f

- f is declared by an initializing variable declaration.
- f is initialized by means of an initializing formal of k.
- f has an initializer in k's initializer list.

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 Generative Constructors

Execution of a generative constructor k of type T to initialize a fresh instance i is always done with respect to a set of bindings for its formal parameters and the type parameters of the immediately enclosing class bound to a set of actual type arguments of T, $t_1, \ldots, t_m$. k, T, i $t_1, \ldots, t_m$

These bindings are usually determined by the instance creation expression that invoked the constructor (directly or indirectly). However, they may also be determined by a reflective call.

If k is redirecting then its redirect clause has the form

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

where g identifies another generative constructor of the immediately surrounding class. Then execution of k to initialize i proceeds by evaluating the argument list ($a_1, \ldots, a_n, x_{n+1}$: $a_{n+1}, \ldots, x_{n+k}$: a_{n+k}) to an actual argument list a of the form ($o_1, \ldots, o_n, x_{n+1}$: $o_{n+1}, \ldots, x_{n+k}$: o_{n+k}) in an environment where the type parameters of the enclosing class are bound to $t_1, \ldots, t_m$. g

Next, the body of g is executed to initialize i with respect to the bindings

that map the formal parameters of g to the corresponding objects in the actual argument list a, with **this** bound to i, and the type parameters of the immediately enclosing class bound to $t_1, \ldots, t_m$.

Otherwise, k is not redirecting. Execution then proceeds as follows:

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 ⟨*finalConstVarOrType*⟩ `v = e;` then e is evaluated to an object o and the instance variable v of i is bound to o.

Any initializing formals declared in k's parameter list are executed in the order they appear in the program text. Then, the initializers of k's initializer list are executed to initialize i in the order they appear in the program, as described below (p. 47).

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

Then if any instance variable of i declared by the immediately enclosing class is not yet bound to an object, all such variables are initialized with the null object (17.4).

Then, unless the enclosing class is `Object`, the explicitly specified or implicitly added superinitializer (10.7.1) is executed to further initialize i.

After the superinitializer has completed, the body of k is executed in a scope where **this** is bound to i.

*This process ensures that no uninitialized final instance variable is ever seen by code. Note that **this** is not in scope on the right hand side of an initializer (see 17.12) 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.*

Execution of Initializer Lists

During the execution of a generative constructor to initialize an instance i, execution of an initializer of the form **this**`.`v `=` e proceeds as follows: i

First, the expression e is evaluated to an object o. Then, the instance variable v of i is bound to o. It is a dynamic type error if the dynamic type of o is not a subtype of the actual type (20.10.1) of the instance variable v.

Execution of an initializer that is an assertion proceeds by executing the assertion (18.18).

Consider a superinitializer s of the form s
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}`)`.

Let C be the class in which s appears and let S be the superclass of C. If S is generic (15), let $u_1, \ldots, u_p$ be the actual type arguments passed to S, obtained by substituting the actual bindings $t_1, \ldots, t_m$ of the formal type parameters of C in the superclass as specified in the header of C. Let k be the constructor declared in S and named S respectively $S.id$. C, S, u_j, p k

Execution of s proceeds as follows: The argument list `(`$a_1, \ldots, a_n, x_{n+1}$`:` $a_{n+1}, \ldots, x_{n+k}$`:` a_{n+k}`)` is evaluated to an actual argument list a of the form

(*o*$_1$, ..., *o*$_n$, *x*$_{n+1}$: *o*$_{n+1}$, ..., *x*$_{n+k}$: *o*$_{n+k}$). Then the body of the super-constructor k is executed in an environment where the formal parameters of k are bound to the corresponding actual arguments from a, and the formal type parameters of S are bound to $u_1, \ldots, u_p$.

10.7.2 Factories

A *factory* is a constructor prefaced by the built-in identifier (17.38) **factory**. ◇

⟨*factoryConstructorSignature*⟩ ::=
  const? **factory** ⟨*constructorName*⟩ ⟨*formalParameterList*⟩

The return type of a factory whose signature is of the form **factory** M or the form **factory** M.*id* is M if M is not a generic type; otherwise the return type is M<$T_1, \ldots, T_n$> where $T_1, \ldots, T_n$ are the type parameters of the enclosing class.

It is a compile-time error if M is not the name of the immediately enclosing class.

It is a dynamic type error if a factory returns a non-null object whose type is not a subtype of its actual (20.10.1) return type.

It seems useless to allow a factory to return the null object (17.4). 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.

Redirecting Factory Constructors

A *redirecting factory constructor* specifies a call to a constructor of another class that is to be used whenever the redirecting constructor is called. ◇

⟨*redirectingFactoryConstructorSignature*⟩ ::=
  const? **factory** ⟨*constructorName*⟩ ⟨*formalParameterList*⟩ '='
  ⟨*constructorDesignation*⟩

⟨*constructorDesignation*⟩ ::= ⟨*typeIdentifier*⟩
  | ⟨*qualifiedName*⟩
  | ⟨*typeName*⟩ ⟨*typeArguments*⟩ ('.' ⟨*identifier*⟩)?

Assume that C<X_1 **extends** $B_1 \ldots, X_m$ **extends** B_m> is the name and formal type parameters of the enclosing class, **const**? is **const** or empty, N is C or $C.id_0$ for some identifier id_0, and id is an identifier, then consider a declaration of a redirecting factory constructor k of one of the forms

C, X_j, B_j, m
const?, N
id
k

const? factory
  N(*T*$_1$ *x*$_1$ ..., *T*$_n$ *x*$_n$, [*T*$_{n+1}$ *x*$_{n+1}$=*d*$_1$, ..., *T*$_{n+k}$ *x*$_{n+k}$=*d*$_k$]) = R;

const? factory

$$N(T_1\ x_1\ \ldots,\ T_n\ x_n,\ \{T_{n+1}\ x_{n+1}=d_1,\ \ldots,\ T_{n+k}\ x_{n+k}=d_k\})\ =\ R;$$

where R is of one of the forms $T<S_1 \ldots,\ S_p>$ or $T<S_1 \ldots,\ S_p>.id$. R, T

It is a compile-time error if T does not denote a class accessible in the current scope. If T does denote such a class D, it is a compile-time error if R does not denote a constructor. Otherwise, it is a compile-time error if R denotes a generative constructor and D is abstract. Otherwise, the *redirectee constructor* for this declaration is the constructor k' denoted by R. ◇ k'

A redirecting factory constructor q' is *redirection-reachable* from a redirecting factory constructor q iff q' is the redirectee constructor of q, or q'' is the redirectee constructor of q and q' is redirection-reachable from q''. It is a compile-time error if a redirecting factory constructor is redirection-reachable from itself. ◇

Let *Ts* be the static argument list type (17.15.1) $(T_1 \ldots,\ T_{n+k})$ when k takes no named arguments, and $(T_1 \ldots,\ T_n,\ T_{n+1}\ x_{n+1},\ \ldots,\ T_{n+k}\ x_{n+k})$ when k takes some named arguments. It is a compile-time error if *Ts* is not a subtype match for the formal parameter list of the redirectee.

We require a subtype match (rather than the more forgiving assignable match which is used with a generative redirecting constructor), because a factory redirecting constructor k always invokes its redirectee k′ with exactly the same actual arguments that k received. This means that a downcast on an actual argument "between" k and k′ would either be unused because the actual argument has the type required by k′, or it would amount to a dynamic error which is simply delayed a single step.

Note that the non-generic case is covered by letting m or p or both be zero, in which case the formal type parameter list of the class C and/or the actual type argument list of the redirectee constructor is omitted (15).

It is a compile-time error if k explicitly specifies a default value for an optional parameter.

Default values specified in k would be ignored, since it is the actual *parameters that are passed to k′. Hence, default values are disallowed.*

It is a compile-time error if a formal parameter of k' has a default value whose type is not a subtype of the type annotation on the corresponding formal parameter in k.

Note that it is not possible to modify the arguments being passed to k'.

At first glance, one might think that ordinary factory constructors could simply create instances of other classes and return them, and that redirecting factories are unnecessary. However, redirecting factories have several advantages:

- *An abstract class may provide a constant constructor that utilizes the constant constructor of another class.*
- *A redirecting factory constructor avoids the need for forwarders to repeat the formal parameters and their default values.*

It is a compile-time error if k is prefixed with the **const** modifier but k' is not a constant constructor (10.7.3).

Let $T_1, \ldots, T_m$ be the actual type arguments passed to k' in the declaration $T_1, \ldots, T_m$

of k. Let $X_1, \ldots, X_m$ be the formal type parameters declared by the class that contains the declaration of k'. Let F' be the function type of k' (10.7). It is a compile-time error if $[T_1/X_1, \ldots, T_m/X_m]F'$ is not a subtype of the function type of k.

In the case where the two classes are non-generic this is just a subtype check on the function types of the two constructors. In general, this implies that the resulting object conforms to the interface of the immediately enclosing class of k.

For the dynamic semantics, assume that k is a redirecting factory constructor and k' is the redirectee of k.

It is a dynamic type error if an actual argument passed in an invocation of k is not a subtype of the actual type (20.10.1) of the corresponding formal parameter in the declaration of k.

When the redirectee k' is a factory constructor, execution of k amounts to execution of k' with the actual arguments passed to k. The result of the execution of k' is the result of k.

When the redirectee k' is a generative constructor, let o be a fresh instance (10.7.1) of the class that contains k'. Execution of k then amounts to execution of k' to initialize o, governed by the same rules as an instance creation expression (17.13). If k' completed normally then the execution of k completes normally returning o, otherwise k completes by throwing the exception and stack trace thrown by k'.

10.7.3 Constant Constructors

A *constant constructor* may be used to create compile-time constant (17.3) objects. A constant constructor is prefixed by the reserved word **const**. ◇

⟨*constantConstructorSignature*⟩ ::=
const ⟨*constructorName*⟩ ⟨*formalParameterList*⟩

Constant constructors have stronger constraints than other constructors. For instance, all the work of a non-redirecting generative constant constructor must be done in its initializers and in the initializing expressions of the instance variables of the enclosing class (and the latter may already have happened earlier, because those initializing expressions must be constant).

Constant redirecting generative and factory constructors are specified elsewhere (p. 44, p. 48). This section is henceforth concerned with non-redirecting generative constant constructors.

It is a compile-time error if a non-redirecting generative constant constructor is declared by a class that has a mutable instance variable.

The above refers to both locally declared and inherited instance variables.

If a non-redirecting generative constant constructor k is declared by a class C, it is a compile-time error for an instance variable declared in C to have an initializing expression that is not a constant expression.

A superclass of C cannot have such an initializing expression e either. If it has a non-redirecting generative constant constructor then e is an error, and if it does

not have such a constructor then the (implicit or explicit) superinitializer in k is an error.

The superinitializer that appears, explicitly or implicitly, in the initializer list of a constant constructor must specify a generative constant constructor of the superclass of the immediately enclosing class, or a compile-time error occurs.

Any expression that appears within the initializer list of a constant constructor must be a potentially constant expression (17.3), or a compile-time error occurs.

When a constant constructor k is invoked from a constant object expression, k
it is a compile-time error if the invocation of k at run time would throw an exception, and it is a compile-time error if substitution of the actual arguments for the formal parameters yields an initializing expression e in the initializer list of k which is not a constant expression.

For instance, if e is `a.length` where `a` is a formal argument of k with type **dynamic**, e is potentially constant and can be used in the initializer list of k. It is an error to invoke k with an argument of type `C` if `C` is a class different from `String`, even if `C` has a `length` getter, and that same expression would evaluate without errors at run time.

10.8 Static Methods

Static methods are functions, other than getters or setters, whose declara- ◇
tions 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.

Static method declarations may conflict with other declarations (10.11).

10.9 Superclasses

The superclass S' of a class C whose declaration has a with clause **with** $M_1, \ldots, M_k$ and an extends clause **extends** S is the abstract class obtained by application of mixin composition (12) $M_k * \cdots * M_1$ to S. The name S' is a fresh identifier. If no **with** clause is specified then 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
- Class C is deemed to have an **extends** clause of the form **extends** `Object`, and the rules above apply.

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

⟨*superclass*⟩ ::= **extends** ⟨*typeNotVoid*⟩ ⟨*mixins*⟩?
  | ⟨*mixins*⟩

⟨*mixins*⟩ ::= **with** ⟨*typeNotVoidList*⟩

The scope of the **extends** and **with** clauses of a class C is the type-parameter scope of C.

It is a compile-time error if the type in the **extends** clause of a class C is a type variable (15), a type alias that does not denote a class (20.3), an enumerated type (14), a deferred type (20.1), type **dynamic** (20.7), or type `FutureOr<`T`>` for any T (20.8).

Note that **void** is a reserved word, which implies that the same restrictions apply for the type **void**, and similar restrictions are specified for other types like `Null` (17.4) and `String` (17.7).

The type parameters of a generic class are available in the lexical scope of the superclass clause, potentially shadowing classes in the surrounding scope. The following code is therefore illegal and should cause a compile-time error:

```
class T {}

/* Compilation error: Attempt to subclass a type parameter */
class G<T> extends T {}
```

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 the superclass of C.

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

10.9.1 Inheritance and Overriding

Let C be a class, let A be a superclass of C, and let $S_1, \ldots, S_k$ be superclasses of C that are also subclasses of A. C *inherits* all concrete, accessible instance ◇ members of A that have not been overridden by a concrete declaration in C or in at least one of $S_1, \ldots, S_k$.

It would be more attractive to give a purely local definition of inheritance, that depended only on the members of the direct superclass S. However, a class C can inherit a member m that is not a member of its superclass S. This can occur when the member m is private to the library L_1 of C, whereas S comes from a different library L_2, but the superclass chain of S includes a class declared in L_1.

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

Let $C = S_0$ be a class declared in library L, and let $\{S_1, \ldots, S_k\}$ be the set

of all superclasses of C, where S_i is the superclass of S_{i-1} for $i \in 1..k$. S_k is the built-in class `Object`. Let C declare a concrete member m, and let m' be a concrete member of $S_j, j \in 1..k$, that has the same name as m, such that m' is accessible to L. Then m overrides m' if m' is not already overridden by a concrete member of at least one of $S_1, \ldots, S_{j-1}$ and neither m nor m' are instance variables.

Instance variables never override each other. The getters and setters induced by instance variables do.

Again, a local definition of overriding would be preferable, but fails to account for library privacy.

Whether an override is legal or not is specified relative to all direct superinterfaces, not just the interface of the superclass, and that is described elsewhere (10.2). Static members never override anything, but they may participate in some conflicts involving declarations in superinterfaces (10.11).

For convenience, here is a summary of the relevant rules, using 'error' to denote compile-time errors. Remember that this is not normative. The controlling language is in the relevant sections of the specification.

1. There is only one namespace for getters, setters, methods and constructors (6.1). An instance or static variable f introduces a getter f, and a mutable instance or static variable f also introduces a setter f= (10.6, 8). When we speak of members here, we mean accessible instance or static variables, getters, setters, and methods (10).

2. You cannot have two members with the same name in the same class—be they declared or inherited (6.1, 10).

3. Static members are never inherited.

4. It is an error if you have a static member named m in your class and an instance member of the same basename (10.11).

5. It is an error if you have a static setter v=, and an instance member v (10.4).

6. It is an error if you have a static getter v and an instance setter v= (10.3).

7. If you define an instance member named m, and your superclass has an instance member of the same name, they override each other. This may or may not be legal.

8. If two members override each other, it is an error unless it is a correct override (11.2.2).

9. Setters, getters and operators never have optional parameters of any kind; it's an error (10.2.1, 10.3, 10.4).

10. It is an error if a member has the same name as its enclosing class (10).

11. A class has an implicit interface (10).

12. Superinterface members are not inherited by a class, but are inherited by its implicit interface. Interfaces have their own inheritance rules (11.2.1).

13. A member is abstract if it has no body and is not labeled **external** (10.5, 9.4).

14. A class is abstract iff it is explicitly labeled **abstract**.

15. It is an error if a concrete class does not implement some member of its interface, and there is no non-trivial `noSuchMethod` (10).

16. It is an error to call a non-factory constructor of an abstract class using an instance creation expression (17.13), such a constructor may only be invoked from another constructor using a superinvocation (17.21.3).

17. If a class defines an instance member named m, and any of its superinterfaces have a member signature named m, the interface of the class contains the m from the class itself.

18. An interface inherits all members of its superinterfaces that are not overridden and not members of multiple superinterfaces.

19. If multiple superinterfaces of an interface define a member with the same name as m, then at most one member is inherited. That member (if it exists) is the one whose type is a subtype of all the others. If there is no such member, an error occurs (11.2.1).

20. Rule 8 applies to interfaces as well as classes (11.2.1).

21. It is an error if a concrete class does not have an implementation for a method in its interface unless it has a non-trivial `noSuchMethod` (10.2.2).

22. The identifier of a named constructor cannot be the same as the basename of a static member declared in the same class (10.11).

10.10 Superinterfaces

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

⟨*interfaces*⟩ ::= **implements** ⟨*typeNotVoidList*⟩

The scope of the **implements** clause of a class C is the type-parameter scope of C.

It is a compile-time error if an element in the type list of the **implements** clause of a class C is a type variable (15), a type alias that does not denote a class (20.3), an enumerated type (14), a deferred type (20.1), type **dynamic** (20.7), or type `FutureOr<`T`>` for any T (20.8). It is a compile-time error if two

elements in the type list of the **implements** clause of a class C specifies the same type T. It is a compile-time error if the superclass of a class C is one of the elements of the type list of the **implements** clause of C.

One might argue that it is harmless to repeat a type in the superinterface list, so why make it an error? The issue is not so much that the situation 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.

It is a compile-time error if a class C has two superinterfaces that are different instantiations of the same generic class. For example, a class can not have both `List<`**void**`>` and `List<`**dynamic**`>` as superinterfaces, directly or indirectly.

When a generic class C declares a type parameter X, it is a compile-time error if X occurs in a non-covariant position in a type which specifies a superinterface of C. For example, the class can not have `List<`**void Function**`(`X`)>` in its **extends** or **implements** clause.

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

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

10.11 Class Member Conflicts

Some pairs of class, mixin, and extension member declarations cannot coexist, even though they do not both introduce the same name into the same scope. This section specifies these errors.

The *basename* of a getter or method named n is n; the basename of a setter named n`=` is n. The basename of an operator named n is n, except for operator `[]=` whose basename is `[]`. ◇

Let C be a class. It is a compile-time error if C declares a constructor named C`.`n and a static member with basename n. It is a compile-time error if C declares a static member with basename n and the interface of C has an instance member with basename n. It is a compile-time error if the interface of C has an instance method named n and an instance setter with basename n. It is a compile-time error if C declares a static method named n and a static setter with basename n. C

When C is a mixin or an extension, the compile-time errors occur according to the same rules. This is redundant in some cases. For instance, it is already an error for a mixin to declare a constructor. But useful cases exist as well, e.g., a conflict between a static member and an instance member. C

These errors occur when the getters or setters are defined explicitly as well as when they are induced by variable declarations.

Note that other errors which are similar in nature are covered elsewhere. For instance, if C is a class that has two superinterfaces I_1 and I_2, where I_1 has a method named m and I_2 has a getter named m, then it is an error because the computation of the interface of C includes a computation of the combined member

signature (11.1) of that getter and that method, and it is an error for a combined member signature to include a getter and a non-getter.

11 Interfaces

This section introduces the notion of interfaces. We define the notion of member signatures first, because that concept is needed in the definition of interfaces.

A *member signature* s can be derived from a class instance member declaration D. It contains the same information as D, except that s omits the body, if any; it contains the return type and parameter types even if they are implicit in D; it omits the names of positional parameters; it omits the modifier **final** from each parameter, if any; it omits metadata (16); and it omits information about whether the member is **external**, **async**, **async***, or **sync***. It makes no difference whether D is given as explicit syntax or it is induced implicitly, e.g., by a variable declaration. Finally, if s has formal parameters, each of them has the modifier **covariant** (9.2.1) if and only if that parameter is covariant-by-declaration (9.2.3). ◇

We use a syntax similar to that of an abstract member declaration to specify member signatures. The difference is that the names of positional parameters are omitted. This syntax is only used for the purposes of specification.

Member signatures are synthetic entities, that is, they are not supported as concrete syntax in a Dart program, they are computed entities used during static analysis. However, it is useful to be able to indicate the properties of a member signature in this specification via a syntactic representation. A member signature makes it explicit whether a parameter is covariant-by-declaration, but it remains implicit whether it is covariant-by-class (9.2.3). The reason for this is that the rule for determining whether a given override relation is correct (11.2.2) depends on the former and not on the latter.

Let m be a method signature of the form m

T_0 *id*<X_1 **extends** B_1, ..., X_s **extends** B_s>(
covariant? T_1, ..., **covariant**? T_n,
[**covariant**? $T_{n+1} = d_{n+1}$, ..., **covariant**? $T_{n+k} = d_{n+k}$]).

The *function type of* m is then ◇

T_0 **Function**<$X_1 \triangleleft B_1$, ..., $X_s \triangleleft B_s$>(T_1, ..., T_n, [T_{n+1}, ..., T_{n+k}]).

Let m be a method signature of the form m

T_0 *id*<X_1 **extends** B_1, ..., X_s **extends** B_s>(
covariant? T_1, ..., **covariant**? T_n,
{**covariant**? T_{n+1} $x_{n+1} = d_{n+1}$, ..., **covariant**? T_{n+k} $x_{n+k} = d_{n+k}$}).

The *function type of* m is then ◇

T_0 **Function**<$X_1 \triangleleft B_1$, ..., $X_s \triangleleft B_s$>(T_1, ..., T_n, {T_{n+1} x_{n+1}, ..., T_{n+k} x_{n+k}}).

Let m be a setter signature of the form **void set** *id*(**covariant**? T p). The m *function type of* m is then **void Function**(T). ◇

The function type of a member signature remains unchanged if some or all default values are omitted.

We do not specify the function type of a getter signature. For such signatures we will instead directly refer to the return type.

An *interface* is a synthetic entity that defines how one may interact with an object. An interface has method, getter and setter signatures, and a set of superinterfaces, which are again interfaces. Each interface is the implicit interface of a class, in which case we call it a *class interface*, or a combination of several other interfaces, in which case we call it a *combined interface*. ◇

Let C be a class. The *class interface* I of C is the interface that declares a member signature derived from each instance member declared by C. The *direct superinterfaces* of I are the direct superinterfaces of C (10.10). ◇

We say that the class interface 'declares' these member signatures, such that we can say that an interface 'declares' or 'has' a member, just like we do for classes. Note that a member signature s of the interface of class C may have a parameter p with modifier **covariant**, even though s was derived from a declaration D in C and the parameter corresponding to p in D does not have that modifier. This is because p may have "inherited" the property of being covariant-by-declaration from one of its superinterfaces (9.2.3).

For the purpose of performing static checks on ordinary method invocations (17.21.1) and property extractions (17.22), any type T which is T_0 bounded (17.15.3), where T_0 is a class with interface I, is also considered to have interface I. Similarly, when T is T_0 bounded where T_0 is a function type, T is considered to have a method named `call` with signature m, such that the function type of m is T_0.

The *combined interface* I of a list of interfaces $I_1, \ldots, I_k$ is the interface that declares the set of member signatures M, where M is determined as specified below. The *direct superinterfaces* of I is the set $I_1, \ldots, I_k$. ◇ $I, I_1, \ldots, I_k$

Let M_0 be the set of all member signatures declared by $I_1, \ldots, I_k$. M is then the smallest set satisfying the following: M

- For each name *id* and library L such that M_0 contains a member signature named *id* which is accessible to L, let m be the combined member signature named *id* from $I_1, \ldots, I_k$ with respect to L. It is a compile-time error if the computation of this combined member signature failed. Otherwise, M contains m.

Interfaces must be able to contain inaccessible member signatures, because they may be accessible from the interfaces associated with declarations of subtypes.

For instance, class C in library L may declare a private member named `_foo`, a class D in a different library L_2 may extend C, and a class E in library L may extend D; E may then declare a member that overrides `_foo` from C, and that override relation must be checked based on the interface of D. So we cannot allow the interface of D to "forget" inaccessible members like `_foo`.

For conflicts the situation is even more demanding: Classes C_1 and C_2 in library L may declare private members `String _foo(int i)` and `int get _foo`, and a subtype D_{12} in a different library L_2 may have an **implements** clause listing both

C_1 and C_2. In that case we must report a conflict even though the conflicting declarations are not accessible to L_2, because those member signatures are then noSuchMethod forwarded (10.2.2), and an invocation of `_foo` on an instance of D in L must return an 'int' according to the first member signature, and it must return a function object according to the second one, and an invocation of `_foo(42)` must return a `String` with the first member signature, and it must fail (at compile time or, for a dynamic invocation, run time) with the second.

It may not be possible to satisfy such constraints simultaneously, and it will inevitably be a complex semantics, so we have chosen to make it an error. It is unfortunate that the addition of a private declaration in one library may break existing code in a different library. But it should be noted that the conflicts can be detected locally in the library where the private declarations exist, because they only arise for private members with the same name and incompatible signatures. Renaming that private member to anything not used in that library will eliminate the conflict and will not break any clients.

11.1 Combined Member Signatures

This section specifies how to compute a member signature which will appropriately stand for a prioritized set of several member signatures, taken from a given list of interfaces.

In general, a combined member signature has a type which is a subtype of all the types given for that member. This is needed in order to ensure that the type of a member id of a class C is well-defined, even in the case where C inherits several different declarations of id and does not override id. In case of failure, it serves to specify the situations where a developer must add a declaration in order to resolve an ambiguity. The member signatures are prioritized in the sense that we will select a member signature from the interface with the lowest possible index in the case where several member signatures are equally suitable to be chosen as the combined member signature. That is, "the first interface wins".

For the purposes of computing a combined member signature, we need a special notion of *equality* of member signatures. Two member signatures m_1 and m_2 are equal iff they have the same name, are accessible to the same set of libraries, have the same same return type (for getters), or the same function type and the same occurrences of **covariant** (for methods and setters). ◇

In particular, private methods from different libraries are never equal. Top types differ as well. For instance, **dynamic Function**() and `Object` **Function**() are not equal, even though they are subtypes of each other. We need this distinction because management of top type discrepancies is one of the purposes of computing a combined interface.

Now we define combined member signatures. Let id be an identifier, L a library, $I_1, \ldots, I_k$ a list of interfaces, and M_0 the set of all member signatures from $I_1, \ldots, I_k$ named id and accessible to L. The *combined member signature named id from* $I_1, \ldots, I_k$ *with respect to* L is the member signature which is obtained as follows: id, L, I_j, k M_0 ◇

If M_0 is empty, computation of the combined member signature failed.

If M_0 contains exactly one member signature m', the combined member signature is m'.

Otherwise, M_0 contains more than one member signature $m_1, \ldots, m_q$. $m_1, \ldots, m_q$

Case ⟨Failing mixtures⟩**.** If M_0 contains at least one getter signature and at least one non-getter signature, the computation of the combined member signature failed. □

Case ⟨Getters⟩**.** If M_0 contains getter signatures only, the computation of the combined member signature proceeds as described below for methods and setters, except that it uses the return type of the getter signature where methods and setters use the function type of the member signature. □

Case ⟨Methods and setters⟩**.** In this case M_0 consists of setter signatures only, or method signatures only, because the name *id* in the former case always ends in '=', which is never true in the latter case.

Determine whether there exists a non-empty set $N \subseteq 1..q$ such that for each $i \in N$, the function type of m_i is a subtype of the function type of m_j for each $j \in 1..q$. If no such set exists, the computation of the combined member signature failed. A useful intuition about this situation is that the given member signatures do not agree on which type is suitable for the member named id. Otherwise we have a set of member signatures which are "most specific" in the sense that their function types are subtypes of them all. N

Otherwise, when a set N as specified above exists, let N_{all} be the greatest set satisfying the requirement on N, and let $M_{\text{all}} = \{m_i \mid i \in N_{\text{all}}\}$. That is, M_{all} contains all member signatures named id with the most specific type. Dart subtyping is a partial pre-order, which ensures that such a greatest set of least elements exists, if any non-empty set of least elements exist. We can have several such signatures because member signatures can be such that they are not equal, and yet their function types are subtypes of each other. We need to compute one member signature from M_{all}, and we do that by using the ordering of the given interfaces. M_{all}

Let $j \in 1..k$ be the smallest number such that $M_{\text{first}} = M_{\text{all}} \cap I_j$ is non-empty. Let m_i be the single element that M_{first} contains. This set contains exactly one element because it is non-empty and no interface contains more than one member signature named id. In other words, we choose m_i as the member signature from the first possible interface among the most specific member signatures M_{all}. M_{first}

The combined member signature is then m', which is obtained from m_i by adding the modifier **covariant** to each parameter p (if it is not already present) when there exists a $j \in 1..q$ such that the parameter corresponding to p (9.2.3) has the modifier **covariant**. In other words, each parameter in the combined member signature is marked covariant if any of the corresponding parameters are marked covariant, not just among the most specific signatures, but among *all* signatures named id (which are accessible to L) in the given list of interfaces. □

11.2 Superinterfaces

An interface has a set of direct superinterfaces (11). 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.

When we say that a type S *implements* another type T, this means that T is a superinterface of S, or S is S_0 bounded for some type S_0 (17.15.3), and T is a superinterface of S_0. Assume that G is a raw type (15.3) whose declaration declares s type parameters. When we say that a type S *implements* G, this means that there exist types $U_1, \ldots, U_s$ such that S implements G<$U_1, \ldots, U_s$>. ◇ ◇

Note that this is not the same as being a subtype. For instance, `List<int>` implements `Iterable<int>`, but it does not implement `Iterable<num>`. Similarly, `List<int>` implements `Iterable`. Also, note that when S implements T where T is not a subtype of `Null`, S cannot be a subtype of `Null`.

Assume that S is a type and G is a raw type such that S implements G. Then there exist unique types $U_1, \ldots, U_s$ such that S implements G<$U_1, \ldots, U_s$>. We then say that $U_1, \ldots, U_s$ are the *actual type arguments of* S *at* G, and we say that U_j is the j*th actual type argument of* S *at* G, for any $j \in 1..s$. ◇ ◇

For instance, the type argument of `List<int>` at `Iterable` is `int`. This concept is particularly useful when the chain of direct superinterfaces from S to G does not just pass all type arguments on unchanged, e.g., with a declaration like **class** `C<X, Y>` **extends** `B<List<Y>, Y, X> {}`.

11.2.1 Inheritance and Overriding

Let J be an interface and K be a library. We define *inherited*(J, K) to be the set of member signatures m such that all of the following hold: J, K m

- m is accessible to K and
- A is a direct superinterface of J and either
  - A declares a member signature m or
  - m is a member of *inherited*(A, K).
- m is not overridden by J.

Furthermore, we define *overrides*(J, K) to be the set of member signatures m' such that all of the following hold: m'

- J is the interface of a class C.
- C declares a member signature m.
- m' has the same name as m.
- m' is accessible to K.
- A is a direct superinterface of J and either
  - A declares a member signature m' or
  - m' is a member of *inherited*(A, K).

Let I be the interface of a class C declared in library L. I *inherits* all members of *inherited*(I, L) and I *overrides* m' if $m' \in$ *overrides*(I, L). ◇ ◇

All the compile-time errors pertaining to the overriding of instance members given in section 10 hold for overriding between interfaces as well.

If the above rule would cause multiple member signatures with the same name *id* to be inherited then exactly one member is inherited, namely the combined member signature named *id*, from the direct superinterfaces in the textual order that they are declared, with respect to L (11.1). It is a compile-time error if the computation of said combined member signature fails.

11.2.2 Correct Member Overrides

Let m and m' be member signatures with the same name *id*. Then m is a *correct override* of m' iff the following criteria are all satisfied: m, m', id ◇

- m and m' are both methods, both getters, or both setters.
- If m and m' are both getters: The return type of m must be a subtype of the return type of m'.
- If m and m' are both methods or both setters: Let F be the function type of m except that the parameter type is the built-in class `Object` for each parameter of m which is covariant-by-declaration (9.2.3). Let F' be the function type of m'. F must then be a subtype of F'.

  The subtype requirement ensures that argument list shapes that are admissible for an invocation of a method with signature m' are also admissible for an invocation of a method with signature m. For instance, m' may accept 2 or 3 positional arguments, and m may accept 1, 2, 3, or 4 positional arguments, but not vice versa. This is a built-in property of the function type subtype rules.

- If m and m' are both methods, p is an optional parameter of m, p' is the parameter of m' corresponding to p, p has default value d and p' has default value d', then d and d' must be identical, or a static warning occurs.

Note that a parameter which is covariant-by-declaration must have a type which satisfies one more requirement, relative to the corresponding parameters in all superinterfaces, both direct and indirect (10.2). We cannot make that requirement a part of the notion of correct overrides, because correct overrides are only concerned with the relation to a single superinterface.

12 Mixins

A mixin describes the difference between a class and its superclass. A mixin is either derived from an existing class declaration or introduced by a mixin declaration. It is a compile-time error to derive a mixin from a class that

declares a generative constructor, or from a class that has a superclass other than `Object`.

Mixin application occurs when one or more mixins are mixed into a class declaration via its **with** clause (12.3). Mixin application may be used to extend a class per section 10; alternatively, a class may be defined as a mixin application as described in the following section.

12.1 Mixin Classes

⟨*mixinApplicationClass*⟩ ::=
    ⟨*identifier*⟩ ⟨*typeParameters*⟩? '`=`' ⟨*mixinApplication*⟩ '`;`'

⟨*mixinApplication*⟩ ::= ⟨*typeNotVoid*⟩ ⟨*mixins*⟩ ⟨*interfaces*⟩?

It is a compile-time error if an element in the type list of the **with** clause of a mixin application is a type variable (15), a function type (20.5), a type alias that does not denote a class (20.3), an enumerated type (14), a deferred type (20.1), type **dynamic** (20.7), type **void** (20.9), or type `FutureOr<`T`>` for any T (20.8). If T is a type in a **with** clause, *the mixin of* T is either the mixin derived from T if T denotes a class, or the mixin introduced by T if T denotes a mixin declaration. ◇

Let D be a mixin application class declaration of the form

abstract? **class** N `=` S **with** M_1 `,` ... `,` M_n **implements** I_1 `,` ... `,` I_k `;`

It is a compile-time error if S is an enumerated type (14). It is a compile-time error if any of $M_1, \ldots, M_k$ is an enumerated type (14).

The effect of D in library L is to introduce the name N into the scope of L, bound to the class (10) defined by the clause S **with** M_1 `,` ... `,` M_n with name N, as described below. If $k > 0$ then the class also implements $I_1, \ldots, I_k$. Iff the class declaration is prefixed by the built-in identifier **abstract**, the class being defined is made an abstract class.

A clause of the form S **with** M_1 `,` ... `,` M_n with name N defines a class as follows:

If there is only one mixin ($n = 1$), then S **with** M_1 defines the class yielded by the mixin application (12.3) of the mixin of M_1 (12.2) to the class denoted by S with name N.

If there is more than one mixin ($n > 1$), then let X be the class defined by S **with** M_1 `,` ... `,` M_{n-1} with name F, where F is a fresh name, and make X abstract. Then S **with** M_1 `,` ... `,` M_n defines the class yielded by the mixin application of the mixin of M_n to the class X with name N.

In either case, let K be a class declaration with the same constructors, superclass, interfaces and instance members as the defined class. It is a compile-time error if the declaration of K would cause a compile-time error.

It is an error, for example, if M contains a member declaration d which overrides a member signature m in the interface of S, but which is not a correct override of m (11.2.2).

12.2 Mixin Declaration

A mixin defines zero or more *mixin member declarations*, zero or more *required superinterfaces*, one *combined superinterface*, and zero or more *implemented interfaces*.

The mixin derived from a class declaration:

abstract? **class** X **implements** I_1, ..., I_k {
  members
}

has `Object` as required superinterface and combined superinterface, $I_1, \ldots, I_k$ as implemented interfaces, and the instance members of *members* as mixin member declarations. If X is generic, so is the mixin.

A mixin declaration introduces a mixin and provides a scope for static member declarations.

⟨*mixinDeclaration*⟩ ::= **mixin** ⟨*typeIdentifier*⟩ ⟨*typeParameters*⟩?
  (**on** ⟨*typeNotVoidList*⟩)? ⟨*interfaces*⟩?
  '{' (⟨*metadata*⟩ ⟨*classMemberDeclaration*⟩)* '}'

It is a compile-time error to declare a constructor in a mixin-declaration.

A mixin declaration with no **on** clause is equivalent to one with the clause **on** `Object`.

Let M be a **mixin** declaration of the form

mixin N<X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s> **on** $T_1, \ldots, T_n$
  implements $I_1, \ldots, I_k$ {
  members
}

It is a compile-time error if any of the types T_1 through T_n or I_1 through I_k is a type variable (15), a function type (20.5), a type alias not denoting a class (20.3), an enumerated type (14), a deferred type (20.1), type **dynamic** (20.7), type **void** (20.9), or type `FutureOr<`T`>` for any T (20.8).

Let M_S be the interface declared by the class declaration

abstract class M_{super}<P_1, ..., P_m> **implements** T_1, ..., T_n {}

where M_{super} is a fresh name. It is a compile-time error for the mixin declaration if the M_S class declaration would cause a compile-time error, that is, if any member is declared by more than one declared superinterface, and there is not a most specific signature for that member among the super interfaces. The interface M_S

is called the *superinvocation interface* of the mixin declaration M. If the mixin declaration M has only one declared superinterface, T_1, then the superinvocation interface M_{super} has exactly the same members as the interface T_1. ◇

Let M_I be the interface that would be defined by the class declaration

abstract class N<X_1 **extends** B_1, ..., X_s **extends** B_s>
  implements T_1, ..., T_n, I_1, ..., I_k {
  members′
}

where *members*′ are the member declarations of the mixin declaration M except that all superinvocations are treated as if **super** was a valid expression with static type M_S. It is a compile-time error for the mixin M if this N class declaration would cause a compile-time error, that is, if the required superinterfaces, the implemented interfaces and the declarations do not define a consistent interface, if any member declaration contains a compile-time error other than a super-invocation, or if a super-invocation is not valid against the interface M_S. The interface introduced by the mixin declaration M has the same member signatures and superinterfaces as M_I.

The mixin declaration M introduces a mixin with the *required superinterfaces* T_1, ..., T_n, the *combined superinterface* M_S, *implemented interfaces* I_1, ..., I_k and the instance members declared in M as *mixin member declarations*. ◇ ◇ ◇ ◇

12.3 Mixin Application

A mixin may be applied to a superclass, yielding a new class.

Let S be a class, M be a mixin with *required superinterfaces* T_1, ..., T_n, *combined superinterface* M_S, *implemented interfaces* I_1, ..., I_k and *members* as *mixin member declarations*, and let N be a name. ◇ ◇ ◇ ◇

It is a compile-time error to apply M to S if S does not implement, directly or indirectly, all of T_1, ..., T_n. It is a compile-time error if any of *members* contains a super-invocation of a member m (for example `super.foo`, `super + 2`, or `super[1] = 2`), and S does not have a concrete implementation of m which is a valid override of the member m in the interface M_S. *We treat super-invocations in mixins as interface invocations on the combined superinterface, so we require the superclass of a mixin application to have valid implementations of those interface members that are actually super-invoked.*

The mixin application of M to S with name N introduces a new class, C, with name N, superclass S, implemented interface M and *members* as instance members. The class C has no static members. If S declares any generative constructors, then the application introduces generative constructors on C as follows:

Let L_C be the library containing the mixin application. That is, the library containing the clause S **with** M or the clause S_0 **with** M_1, ..., M_k, M giving rise to the mixin application.

Let S_N be the name of S.

For each generative constructor of the form S_q`(`T_1 a_1`, ...,` T_k a_k`)` of S that is accessible to L_C, C has an implicitly declared constructor of the form

C_q`(`T_1 a_1`, ...,` T_k a_k`):` **super**$_q$`(`a_1`, ...,` a_k`);`

where C_q is obtained from S_q by replacing occurrences of S_N, which denote the superclass, by N, and **super**$_q$ is obtained from S_q by replacing occurrences of S_N which denote the superclass by **super**. If S_q is a generative const constructor, and C does not declare any instance variables, C_q is also a const constructor.

For each generative constructor of the form S_q`(`T_1 a_1`, ...,` T_k a_k`, [`T_{k+1} a_{k+1} `=` d_1`, ...,` T_{k+p} a_{k+p} `=` d_p`])` of S that is accessible to L_C, C has an implicitly declared constructor of the form

C_q`(`T_1 a_1`, ... ,` T_k a_k`, [`T_{k+1} a_{k+1} `=` d'_1`, ... ,` T_{k+p} a_{k+p} `=` d'_p`])`
`:` **super**$_q$`(`a_1`, ... ,` a_k`,` a_{k+1}`, ...,` a_p`);`

where C_q is obtained from S_q by replacing occurrences of S_N, which denote the superclass, by N, **super**$_q$ is obtained from S_q by replacing occurrences of S_N which denote the superclass by **super**, and d'_i, $i \in 1..p$, is a constant expression evaluating to the same value as d_i. If S_q is a generative const constructor, and MC does not declare any instance variables, C_q is also a const constructor.

For each generative constructor of the form S_q`(`T_1 a_1`, ...,` T_k a_k`, {`T_{k+1} a_{k+1} `=` d_1`, ...,` T_{k+n} a_{k+n} `=` d_n`})` of S that is accessible to L_C, C has an implicitly declared constructor of the form

C_q`(`T_1 a_1`, ... ,` T_k a_k`, {`T_{k+1} a_{k+1} `=` d'_1`, ... ,` T_{k+n} a_{k+n} `=` d'_n`})`
`:` **super**$_q$`(`a_1`, ... ,` a_k`,` a_{k+1}`:` a_{k+1}`, ...,` a_p`:` a_p`);`

where C_q is obtained from S_q by replacing occurrences of S_N which denote the superclass by N, **super**$_q$ is obtained from S_q by replacing occurrences of S_N which denote the superclass by **super**, and d'_i, $i \in 1..n$, is a constant expression evaluating to the same value as d_i. If S_q is a generative const constructor, and M does not declare any fields, C_q is also a const constructor.

13 Extensions

This section specifies extensions. This mechanism supports the declaration of functions that are similar to instance methods in use, but similar to non-method functions in that they are declared outside the target class, and they are resolved statically. The resolution is based on whether the relevant extension is in scope, and whether the invocation satisfies several other requirements.

⟨*extensionDeclaration*⟩ ::=
extension ⟨*identifier*⟩? ⟨*typeParameters*⟩? **on** ⟨*type*⟩
'`{`' (⟨*metadata*⟩ ⟨*classMemberDeclaration*⟩)* '`}`'

A declaration derived from ⟨*extensionDeclaration*⟩ is known as an *extension declaration*. It introduces an *extension* with the given ⟨*identifier*⟩, if present, into the namespace of the enclosing library (and hence into the library scope), and provides a scope for the declaration of extension members. Additionally, the extension is introduced into the library namespace with a fresh, private name. The former is known as the *declared name* of the extension, and the latter is known as the *fresh name* of the extension. ◇

A fresh name is also introduced into the library namespace of the current library for each imported extension, even when it is imported with a prefix (19.1.1). The declared name of an extension E is introduced into the library scope of the current library following the same rules as the names of other locally declared or imported declarations like classes. A fresh name of E is introduced in these cases, but also in one additional case: when there is a name clash on the declared name of E.

The fresh name makes it possible for an extension to be used in an implicit invocation (13.2), even in the case where the declared name or an import prefix that provides access to the declared name is shadowed by a declaration in an intermediate scope, or conflicted by a name clash.

The fact that an extension can be used implicitly even in the case where it does not have a declared name or the declared name is shadowed or conflicted reflects the fact that the primary intended usage is implicit invocation. Even though a developer cannot know (and hence cannot use) the fresh name of a given extension, an implicit invocation can use it.

It is a compile-time error if the current library has a deferred import of a library L' such that the imported namespace from L' contains a name denoting an extension.

This implies that the import must use **hide** or **show** to eliminate the names of extensions from the deferred import.

This restriction ensures that no extensions are introduced using deferred imports, which allows us to introduce a semantics for such extensions in the future without affecting existing code.

The ⟨*type*⟩ in an extension declaration is known as the extension's ***on*** *type*. The **on** type can be any valid type, including a type variable. ◇

The basic intuition is that an extension `E` may have an **on** type T (specifying the type of receiver) and a set of members. If e is an expression whose static type is T and `foo()` is a member declared by `E`, e`.foo()` may invoke said member with the value of e bound to **this**. An explicitly resolved form `E(`e`).foo()` is available, such that `E.foo` can be invoked even in the case where e`.foo()` would invoke some other function because some other extension is more specific. Details of these concepts, rules, and mechanisms are given in this section and its subsections.

The declared name of an extension does not denote a type, but it can be used to denote the extension itself (e.g., in order to access static members of the extension, or in order to resolve an invocation explicitly). A compile-time error occurs if the declared name of an extension is a built-in identifier.

An extension declaration introduces two scopes:

- A *type-parameter scope*, which is empty if the extension is not generic ◇

(15). The enclosing scope of the type-parameter scope of an extension declaration is the library scope of the current library. The type parameter scope is the current scope for the type parameters and for the extension's **on** ⟨*type*⟩.

- A *body scope*. The enclosing scope of the body scope of an extension declaration is the type parameter scope of the extension declaration. The current scope for an extension member declaration is the body scope of the enclosing extension declaration. ◇

Let D be an extension declaration with declared name `E`. A member declaration in D with the modifier **static** is designated as a *static member* declaration. ◇ A member declaration in D without the modifier **static** is designated as an *instance member* declaration. ◇ Member naming conflict errors may occur in D in situations that also occur in classes and mixins (10.11). Moreover, a compile-time error occurs in the following situations: D, `E`

- D declares a member whose basename is `E`.
- D declares a type parameter named `E`.
- D declares a member whose basename is the name of a type parameter of D.
- D declares an instance member or a static member whose basename is `hashCode`, `noSuchMethod`, `runtimeType`, `toString`, or '`==`'. That is, a member whose basename is also the name of an instance member that every object has.
- D declares a constructor.
- D declares an instance variable.
- D declares an abstract member.
- D declares a method with a formal parameter with the modifier **covariant**.

Abstract members are not allowed because there is no support for providing an implementation. Constructors are not allowed since the extension does not introduce any type that can be constructed. Instance variables are not allowed because no memory is allocated for each object accessed as ***this*** *in the members. Developers can emulate per-****this*** *state if needed, e.g., using an* `Expando`*. Members with the same basename as members of* `Object` *are not allowed because they could only be invoked using explicit resolution, as in* `E(e).toString(14)`*, which would be confusing and error-prone.*

13.1 Explicit Invocation of an Instance Member of an Extension

Let E be a simple or qualified identifier that denotes an extension. An *extension application* is then an expression of the form E ⟨*typeArguments*⟩? '(' ⟨*expression*⟩ ')'. An extension member with a name accessible to the current library can be invoked explicitly on a particular object by performing a member invocation (17.20) where the receiver is an extension application. E ◇

Type inference is not yet specified in this document, and is assumed to have taken place already (6), but the following describes the intended treatment. This section and its subsections have similar commentary about type inference below, marked 'With type inference: ...'.

Let E be a simple or qualified identifier denoting an extension named `E` and declared as follows:

extension `E<`X_1 **extends** B_1, ..., X_s **extends** B_s`> on` T `{ ... }`

Type inference for an extension application of the form E`(`e`)` is done exactly the same as it would be for the same syntax considered as a constructor invocation where E is assumed to denote the following class, and the context type is empty (implying no requirements):

```
class E<X1 extends B1, ..., Xs extends Bs> {
  final T target;
  E(this.target);
}
```

This will infer type arguments for E`(`e`)`, and it will introduce a context type for the expression e. For example, if E is declared as **extension** `E<T> on Set<T> { ... }` then `E({})` will provide the expression `{}` with a context type that makes it a set literal.

Let a be an extension application of the form E`<`T_1, ..., T_s`>(`e`)`, where E denotes an extension declared as a, E, T_j, e

extension `E<`X_1 **extends** B_1, ..., X_s **extends** B_s`>` **on** T `{ ... }`. X_j, B_j, s, T

We define the *instantiated* ***on*** *type* of a as $[T_1/X_1, \ldots, T_s/X_s]T$. We define the *instantiation-to-bound* ***on*** *type* of a as $[U_1/X_1, \ldots, U_s/X_s]T$, where $U_1, \ldots, U_s$ is the result of instantiation to bound on the type parameters of E (15.3). A compile-time error occurs unless $T_j <: [T_1/X_1, \ldots, T_s/X_s]B_j$, $j \in 1..s$ (that is, the bounds cannot be violated). A compile-time error occurs unless the static type of e is assignable to the instantiated **on** type of a. Note that a compile-time error occurs as well if the static type of e is **void** (20.9). ◇ ◇

It is a compile-time error if an extension application occurs in a location where it is *not* the syntactic receiver of a simple or composite member invocation (17.20).

That is, the only valid use of an extension application is to invoke or tear off members on it. This is similar to how prefix names can also only be used as member

invocation targets, except that extensions can also declare operators. For instance, $E(e)$ `+ 1` can be a valid invocation of an operator '+' declared in an extension E.

An extension application does not have a type.

This is consistent with the fact that any use of an extension application where a type is needed is a compile-time error.

Let i be a simple member invocation (17.20) whose receiver r is an extension application of the form E`<`$T_1, \ldots, T_k$`>(`e`)` (which is $E(e)$ when k is zero) whose corresponding member name is n, and assume that r has no compile-time errors. A compile-time error occurs unless the extension denoted by E declares a member named n. Otherwise let $X_1, \ldots, X_k$ be the type parameters of said extension. Let s be the member signature of the member n declared by E. Exactly the same compile-time errors occur for i as the ones that would occur for a member invocation i_1 which is obtained from i by replacing r by a variable whose type is a class C declared in the same scope as E that declares a member n with member signature $s_1 = [T_1/X_1, \ldots, T_k/X_k]s$:

i, r
E, T_j, k, e
n
$X_1, \ldots, X_k$
s

```
abstract class C {
  D // Member declaration with signature s1.
}
```

The member signature s_1 is called the *invocation member signature* of i. The static type of i is the return type of the invocation member signature of i. For example: ◇

```
extension E<X> on List<X> {
  List<List<X>> split(int at) =>
      [this.sublist(0, at), this.sublist(at)];
}

void main() {
  List<num> xs = [1, 2, 3];
  var ys = E<num>(xs).split(1); // (*)
}

abstract class C {
  // Declaration with invocation member signature for (*).
  List<List<num>> split(int at);
}
```

With type inference: In the case where the invocation member signature s_1 is generic, type inference occurs on i in the same way as it would occur for an invocation of a function whose type is the function type of s_1.

For the dynamic semantics, let i be a simple, unconditional member invocation whose receiver r is an extension application of the form E`<`$T_1, \ldots, T_k$`>(`e`)` where the type parameters of E are $X_1, \ldots, X_k$ and the actual values of $T_1, \ldots, T_k$ are $t_1, \ldots, t_k$ (20.10.1), and whose corresponding member name

i, r
E, T_j, k, e
$X_1, \ldots, X_k$
$t_1, \ldots, t_k$
n, m

is n. Let m be the member of E that has the name n. Evaluation of i proceeds by evaluating e to an object o, evaluating and binding the actual arguments to the formal parameters (17.15.3), and finally executing m in a binding environment where $X_1, \ldots, X_k$ are bound to $t_1, \ldots, t_k$, **this** is bound to o, and each formal parameter is bound to the corresponding actual argument. The value of i is the value returned by the invocation of m.

When i is a conditional or composite member invocation, the static analysis and dynamic semantics is determined by member invocation desugaring (17.20).

Note that a cascade (17.21.2) whose receiver is an extension application a is a compile-time error. This is so because it implies that a denotes an object, which is not true, and also because it would force each ⟨cascadeSection⟩ to invoke a member of the same extension, which is unlikely to be desirable.

13.2 Implicit Invocation of an Instance Member of an Extension

Instance members of an extension can be invoked or closurized implicitly (without mentioning the name of the extension), as if they were instance members of the receiver of the given member invocation.

For instance, if `E<`T_1, T_2`>(`e_1`).m<`T_3`>(`e_2`)` is a correct explicit invocation of the instance member `m` of an extension `E`, then e_1`.m<`T_3`>(`e_2`)` may be a correct implicit invocation with the same meaning. In other words, the receiver r of a member invocation can be implicitly replaced by an extension application with receiver r, if a number of requirements that are detailed below are satisfied.

Implicit invocation is intended as the primary way to use extensions, with explicit invocation as a fallback in case the implicit invocation is an error, or the implicit invocation resolves to an instance member of a different extension than the intended one.

An implicit extension member invocation occurs for a member invocation i (17.20) with receiver r and corresponding member name m iff (1) r is not a type literal, (2) the interface of the static type of r does not have a member whose basename is the basename of m, and (3) there exists a unique most specific (13.2.3) extension denoted by E which is accessible (13.2.1) and applicable (13.2.2) to i. i, r, m

In the case where no compile-time error occurs, i is treated as i', which is obtained from i by replacing the leading r by E`(`r`)`.

With type inference: When E is generic, type inference applied to $E(r)$ may provide actual type arguments, yielding an i' of the form E`<`$T_1, \ldots, T_k$`>(`r`)`. If this type inference step fails then E is not applicable (13.2.2).

Implicit invocation of an instance member of an extension in a cascade is also possible, because a cascade is desugared to an expression that contains one or more member invocations.

13.2.1 Accessibility of an Extension

An extension E is *accessible* in a given scope S if there exists a name n such ◇

that a lexical lookup for n from S (17.37) yields E.

The name n can be the declared name of E or the fresh name of E, but since the fresh name is always in scope whenever the declared name is in scope, it is sufficient to consider the fresh name. When the fresh name of E is in the library scope, it is available in *any* scope, because the name is fresh and hence it cannot be shadowed by any declaration in any intermediate scope. This implies that if E is accessible anywhere in a given library L then it is accessible everywhere in L.

13.2.2 Applicability of an Extension

Let E be an extension. Let e be a member invocation (17.20) with a receiver r with static type S and with a corresponding member name whose basename is m. We say that E is *applicable* to e if the following conditions are all satisfied: $\diamond$ E, e, r, S, m

- r is not a type literal.

  This means that the invocation is an instance member invocation, in the sense that r denotes an object, so it may invoke an instance member or be an error, but it cannot be a static member access. Note that r also does not denote a prefix or an extension, and it is not an extension application, because they do not have a type.

- The type S does not have a member with the basename m, and S is neither **dynamic** nor `Never`.

  dynamic and `Never` are considered to have all members. Also, it is an error to access a member on a receiver of type **void** (20.9), so extensions are never applicable to receivers of any of the types **dynamic**, `Never`, or **void**.

  For the purpose of determining extension applicability, function types and the type `Function` are considered to have a member named `call`.

  Hence, extensions are never applicable to functions when the basename of the member is `call`. Instance members declared by the built-in class `Object` exist on all types, so no extension is ever applicable for members with such names.

- Consider an extension application a of the form $E(v)$, where v is a fresh variable with static type S. It is required that an occurrence of a in the scope which is the current scope for e is not a compile-time error.

  In other words, S must match the **on** type of E. With type inference, inferred actual type arguments may be added, yielding $E<S_1, \ldots, S_k>(v)$, which is then required to not be an error. If this inference step fails it is not an error, it just means that E is not applicable to e.

- The extension E declares an instance member with basename m.

With type inference: The context type of the invocation does not affect whether the extension is applicable, and neither the context type nor the method invocation affects the type inference of r, but if the extension method itself is generic, the context type may affect the member invocation.

13.2.3 Specificity of an Extension

When $E_1, \ldots, E_k$, $k > 1$, are extensions which are accessible and applicable to a member invocation e (17.20), we define the notion of *specificity*, which is a partial order on $E_1, \ldots, E_k$. E_j, k $\diamond$

Specificity is used to determine which extension method to execute in the situation where more than one choice is possible.

Let e be a member invocation with receiver r and correspsponding member name m, and let E_1 and E_2 denote two distinct accessible and applicable extensions for e. Let T_j be the instantiated **on** type of e with respect to E_j, and S_j be the instantiation-to-bound **on** type of e with respect to E_j, for $j \in 1..2$ (13.1). Then E_1 is more specific than E_2 with respect to e if at least one of the following conditions is satisfied: e, r, m E_1, E_2 T_j, S_j

- E_1 is not declared in a system library, but E_2 is declared in a system library.
- E_1 and E_2 are both declared in a system library, or neither of them is declared in a system library, and
  - $T_1 <: T_2$, but not $T_2 <: T_1$, or
  - $T_1 <: T_2$, $T_2 <: T_1$, and $S_1 <: S_2$, but not $S_2 <: S_1$.

  In other words, the instantiated **on** type determines the specificity, and the instantiation-to-bound **on** type is used as a tie breaker in the case where subtyping does not distinguish between the former.

The following examples illustrate implicit extension resolution when multiple applicable extensions are available.

```
extension ExtendIterable<T> on Iterable<T> {
  void myForEach(void Function(T) f) {
    for (var x in this) f(x);
  }
}
extension ExtendList<T> on List<T> {
  void myForEach(void Function(T) f) {
    for (int i = 0; i < length; i++) f(this[i]);
  }
}

void main() {
  List<int> x = [1];
  x.myForEach(print);
}
```

Here both of the extensions apply, but `ExtendList` is more specific than `ExtendIterable` because `List<int> <: Iterable<int>`.

```
extension BestCom<T extends num> on Iterable<T> { T best() {...}}
extension BestList<T> on List<T> { T best() {...}}
extension BestSpec on List<num> { num best() {...}}

void main() {
  List<int> x = ...;
  var v = x.best();
  List<num> y = ...;
  var w = y.best();
}
```

Here all three extensions apply to both invocations. For `x.best()`, `BestList` is most specific, because `List<int>` is a proper subtype of both `Iterable<int>` and `List<num>`. Hence, the type of `x.best()` is `int`.

For `y.best()`, `BestSpec` is most specific. The instantiated **on** types that are compared are `Iterable<num>` for `BestCom` and `List<num>` for the two other extensions. Using the instantiation-to-bound **on** types as a tie breaker, we find that `List<Object>` is less precise than `List<num>`, so `BestSpec` is selected. Hence, the type of `y.best()` is `num`.

In general, the definition of specificity aims to select the extension which has more precise type information available. This does not necessarily yield the most precise type of the result (for instance, `BestSpec.best` could have returned `Object`), but it is also important that the rule is simple.

In practice, we expect unintended extension member name conflicts to be rare. If the same author is providing more specialized versions of an extension for subtypes, the choice of an extension which has the most precise types available is likely to be a rather unsurprising and useful behavior.

13.3 Static analysis of Members of an Extension

Static analysis of the member declarations in an extension E relies on the scopes of the extension (13) and follows the normal rules except for the following:

When static analysis is performed on the body of an instance member of an extension E with **on** type T_{on}, the static type of **this** is T_{on}.

A compile-time error occurs if the body of an extension member contains **super**.

A lexical lookup in an extension E may yield a declaration of an instance method declared in E. As specified elsewhere (17.37), this implies that extension instance members will shadow class instance members when called from another instance member inside the same extension using an unqualified function invocation (that is, invoking it as `m()` and not **this**`.m()`, 17.15.4). This is the only situation where implicit invocation of an extension member with basename id can succeed even if the interface of the receiver has a member with basename id. On the other hand, it is consistent with the general property of Dart that lexically enclosing declarations shadow other declarations, e.g., an inherited declaration can be shadowed by a global declaration. Here is an example:

```
extension MyUnaryNumber on List<Object> {
  bool get isEven => length.isEven;
  bool get isOdd => !isEven;
  static bool isListEven(List<Object> list) => list.isEven;
  List<Object> get first => [];
  List<Object> get smallest => first;
}
```

With `list.isEven`, `isEven` resolves to the declaration in `MyUnaryNumber`, given that `List` does not have a member with basename `isEven`, and unless there are any other extensions creating a conflict.

The use of `length` in the declaration of `isEven` is not defined in the current lexical scope, so it is treated as **this**`.length`, because the interface of the **on** type `List<Object>` has a `length` getter.

The use of `isEven` in `isOdd` resolves lexically to the `isEven` getter above it, so it is treated as `MyUnaryNumber(this).isEven`, even if there are other extensions in scope which define an `isEven` on `List<Object>`.

The use of `first` in `smallest` resolves lexically to the `first` getter above it, even though there is a member with the same basename in the interface of **this**. The getter `first` cannot be called in an implicit invocation from anywhere outside of `MyUnaryNumber`. This is the exceptional case mentioned above, where a member of an extension shadows a regular instance member on **this**. In practice, extensions will very rarely introduce members with the same basename as a member of its **on** type's interface.

An unqualified identifier `id` which is not in scope is treated as **this**`.id` inside instance members as usual (17.37). If `id` is not declared by the static type of **this** (the **on** type) then it may be an error, or it may be resolved using a different extension.

13.4 Extension Method Closurization

An extension instance method is subject to closurization in a similar manner as class instance methods (17.22.2).

Let a be an extension application (13.1) of the form E<$S_1, \ldots, S_m$>(`(`e_1`)`). Let $Y_1, \ldots, Y_m$ be the formal type parameters of the extension E. An expression e of the form a`.`id where id is an identifier is then known as an *extension property extraction*. It is a compile-time error unless E declares an instance member named id. If said instance member is a method then e has the static type $[S_1/Y_1, \ldots, S_m/Y_m]F$, where F is the function type of said method declaration.

a, E, S_j, e_1
Y_j, m
e, id
◇

If id is a getter then e is a getter invocation, which is specified elsewhere (13.1).

If id is a method then e is known as an *instance method closurization* of id on a, and evaluation of e (which is E<$S_1, \ldots, S_m$>(`(`e_1`)`$.id$) proceeds as follows:

◇

Evaluate e_1 to an object o. Let u be a fresh final variable bound to o. Then e evaluates to a function object which is equivalent to:

- <X_1 **extends** $B'_1, \ldots, X_s$ **extends** B'_s>

$(T_1\ p_1, \ldots,\ T_n\ p_n,\ \{T_{n+1}\ p_{n+1}\ =\ d_1,\ \ldots,\ T_{n+k}\ p_{n+k}\ =\ d_k\})$ `=>`
E`<`$S_1, \ldots, S_m$`>(`u`)`
`.`id`<`$X_1, \ldots, X_s$`>(`$p_1, \ldots, p_n, p_{n+1}$`:` $p_{n+1}, \ldots, p_{n+k}$`:` p_{n+k}`);`

where id declares type parameters X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s, required parameters $p_1, \ldots, p_n$, and named parameters $p_{n+1}, \ldots, p_{n+k}$ with defaults $d_1, \ldots, d_k$, using `null` for parameters whose default value is not specified.

- `<`X_1 **extends** $B'_1, \ldots, X_s$ **extends** B'_s`>`
  $(T_1\ p_1, \ldots,\ T_n\ p_n,\ [T_{n+1}\ p_{n+1}\ =\ d_1,\ \ldots,\ T_{n+k}\ p_{n+k}\ =\ d_k])$ `=>`
  E`<`$S_1, \ldots, S_m$`>(`u`).`id`<`$X_1, \ldots, X_s$`>(`$p_1, \ldots, p_{n+k}$`);`

  where id declares type parameters X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s, required parameters $p_1, \ldots, p_n$, and optional positional parameters $p_{n+1}, \ldots, p_{n+k}$ with defaults $d_1, \ldots, d_k$, using `null` for parameters whose default value is not specified.

In the function literals above, $B'_j = [S_1/Y_1, \ldots, S_m/Y_m]B_j, j \in 1..s$, and $T_j = [S_1/Y_1, \ldots, S_m/Y_m]T'_j, j \in 1..n+k$, where T'_j is the type of the corresponding parameter in the declaration of id. Capture of type variables in $S_1, \ldots, S_m$ must be avoided, so X_j must be renamed if $S_1, \ldots, S_m$ contains any occurrences of X_j, for all $j \in 1..s$.

In other words, the closurization is the value of a function literal whose signature is the same as that of id, except that the actual type arguments are substituted for the formal type parameters of E, and then it simply forwards the invocation to id with the captured object u as the receiver.

Two extension instance method closurizations are never equal unless they are identical. Note that this differs from closurizations of class instance methods, which are equal when they tear off the same method of the same receiver.

The reason for this difference is that even if o_1 and o_2 are instance method closurizations of the same extension E applied to the same receiver o, they may have different actual type arguments passed to E, because those type arguments are determined by the call site (and with inference: by the static type of the expression yielding o), and not just by the properties of o and the torn-off method.

Note that an instance method closurization on an extension is not a constant expression, even in the case where the receiver is a constant expression. This is because it creates a new function object each time it is evaluated.

Extension method closurization can occur for an implicit invocation of an extension instance member.

This is a consequence of the fact that the implicit invocation is treated as the corresponding explicit invocation (13.2). For instance, $e\,.\,id$ may be implicitly transformed into E`<`T_1, T_2`>(`e`)`$\,.\,id$, which is then handled as specified above.

Extension method closurizations are subject to generic function instantiation

(17.17). For example:

```
extension on int {
  Set<T> asSet<T extends num>() => {if (this is T) this as T};
}

void main() {
  Set<double> Function() f = 1.asSet;
  print(f()); // Prints '{}'.
}
```

In this example {} is printed, because the function object obtained by extension method closurization was subject to a generic function instantiation which gave `T` the value `double`, which makes '**this is** `T`' evaluate to false.

13.5 The `call` Member of an Extension

An extension can provide a `call` method which is invoked implicitly, similarly to a function expression invocation (17.15.5).

E.g., e`()` is treated as e`.call()` when the static type of e is a non-function that has a method named `call`. Here is an example where the `call` method comes from an extension:

```
extension E on int {
  Iterable<int> call(int to) =>
      Iterable<int>.generate(to - this + 1, (i) => this + i);
}

void main() {
  for (var i in 1(3)) print(i); // Prints 1, 2, 3.
  for (var i in E(4)(5)) print(i); // Prints 4, 5.
}
```

This may look somewhat surprising, though similar to an approach using ***operator[]: for (var i in 1[3]) {...}****. We expect developers to use this power responsibly.*

Let a be an extension application (13.1), and i an expression of the form a, i
a`<`$A_1, \ldots, A_r$`>(`$a_1, \ldots, a_n, x_{n+1}$`:` $a_{n+1}, \ldots, x_{n+k}$`:` a_{n+k}`)`
(where the type argument list is omitted when r is zero). i is then treated as (5)
a`.call<`$A_1, \ldots, A_r$`>(`$a_1, \ldots, a_n, x_{n+1}$`:` $a_{n+1}, \ldots, x_{n+k}$`:` a_{n+k}`)`.

In other words, an invocation of an extension application is immediately treated as an invocation of an extension method named `call`.

Let e be an expression with static type S which is not a property extraction e, S, i
expression (17.22), and let i be an expression of the form
e`<`$A_1, \ldots, A_r$`>(`$a_1, \ldots, a_n, x_{n+1}$`:` $a_{n+1}, \ldots, x_{n+k}$`:` a_{n+k}`)`

(where the type argument list is again omitted when r is zero). If S is **dynamic**, **Function**, or a function type, or the interface of S has a method named `call`, i is specified elsewhere (17.15.5). Otherwise, if S has a non-method instance member with basename `call` then i is a compile-time error. Otherwise, i is treated as the expression i' which is
e`.call<`$A_1, \ldots, A_r$`>(`$a_1, \ldots, a_n, x_{n+1}$`:` $a_{n+1}, \ldots, x_{n+k}$`:` a_{n+k}`)`.

Note that i' can be an implicit invocation of an extension method named `call`, and it can be an error. In the latter case, error messages should be worded in terms of i, not i'.

It is a compile-time error unless i' is an implicit invocation of an extension instance method named `call`.

In particular, i' cannot be an invocation of an extension getter whose return type is a function type, **Function**, or **dynamic**.

Note that there is no support for an implicit property extraction which tears off an extension method named `call`. For instance, assuming the extension E declared in the previous example:

```
Iterable<int> Function(int) from2 = 2; // Error.
```

The implicit property extraction could be allowed, but it would come at a readability cost. A type like `int` *is well known as being non-callable, and an implicit* `.call` *tear-off would have no visible syntax. In an implicit* `call` *invocation, the arguments are visible to a reader, but for an implicit tear-off of a* `call` *function, there is no visible syntax at all.*

If desired, the property extraction can be expressed explicitly using `2.call`.

14 Enums

An *enumerated type*, or *enum*, is used to represent a fixed number of constant values. ◇ ◇

⟨*enumType*⟩ ::= **enum** ⟨*identifier*⟩
‘`{`’ ⟨*enumEntry*⟩ (‘`,`’ ⟨*enumEntry*⟩)* (‘`,`’)? ‘`}`’

⟨*enumEntry*⟩ ::= ⟨*metadata*⟩ ⟨*identifier*⟩

The declaration of an enum of the form m **enum** E `{`$m_0\ id_0, \ldots, m_{n-1}\ id_{n-1}$`}` has the same effect as a class declaration

```
m class E {
  final int index;
  const E(this.index);
  m0 static const E id0 = const E(0);
  ...
  mn-1 static const E idn-1 = const E(n - 1);
  static const List<E> values = const <E>[id0, ..., idn-1];
```

```
  String toString() => { 0: 'E.id_0', ..., n-1: 'E.id_{n-1}'}[index]
}
```

It is also a compile-time error to subclass, mix-in or implement an enum or to explicitly instantiate an enum. These restrictions are given in normative form in sections 10.9, 10.10, 12.3 and 17.13 as appropriate.

15 Generics

A declaration of a class (10), mixin (12), extension (13), type alias (20.3), or function (9) G may be *generic*, that is, G may have formal type parameters declared. ◇

When an entity in this specification is described as generic, and the special case is considered where the number of type arguments is zero, the type argument list should be omitted.

This allows non-generic cases to be included implicitly as special cases. For example, an invocation of a non-generic function arises as the special case where the function takes zero type arguments, and zero type arguments are passed. In this situation some operations are also omitted (have no effect), e.g., operations where formal type parameters are replaced by actual type arguments.

A *generic class declaration* introduces a generic class into the library scope of the current library. A *generic class* is a mapping that accepts a list of actual type arguments and maps them to a class. Consider a generic class declaration G named C with formal type parameter declarations X_1 **extends** $B_1, \ldots,$ X_m **extends** B_m, and a parameterized type T of the form C<$T_1, \ldots,$ T_l>. ◇ ◇

It is a compile-time error if $m \neq l$. It is a compile-time error if T is not well-bounded (15.2).

Otherwise, said parameterized type C<$T_1, \ldots,$ T_m> denotes an application of the generic class declared by G to the type arguments $T_1, \ldots, T_m$. This yields a class C' whose members are equivalent to those of a class declaration which is obtained from the declaration G by replacing each occurrence of X_j by T_j.

Other properties of C' such as the subtype relationships are specified elsewhere (20.4).

Generic type aliases are specified elsewhere (20.3).

A *generic type* is a type which is introduced by a generic class declaration or a generic type alias, or it is the type `FutureOr`. ◇

A *generic function declaration* introduces a generic function (9.2) into the current scope. ◇

Consider a function invocation expression of the form `f`<$T_1, \ldots,$ T_l>`(...)`, where the static type of `f` is a generic function type with formal type parameters X_1 **extends** $B_1, \ldots,$ X_m **extends** B_m. It is a compile-time error if $m \neq l$. It is a compile-time error if there exists a j such that T_j is not a subtype of $[T_1/X_1, \ldots, T_m/X_m]B_j$.

That is, if the number of type arguments is wrong, or if the jth actual type argument is not a subtype of the corresponding bound, where each formal type

parameter has been replaced by the corresponding actual type argument.

⟨*typeParameter*⟩ ::= ⟨*metadata*⟩ ⟨*identifier*⟩ (**extends** ⟨*typeNotVoid*⟩)?

⟨*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`. It is a compile-time error if a type parameter is a supertype of its upper bound when that upper bound is itself a type variable. ◇

This prevents circular declarations like `X` **extends** `X` and `X` **extends** `Y, Y` **extends** `X`.

Type parameters are declared in the type parameter scope of a class or function. The type parameters of a generic G are in scope in the bounds of all of the type parameters of G. The type parameters of a generic class declaration G are also in scope in the **extends** and **implements** clauses of G (if these exist) and in the body of G.

However, a type parameter of a generic class is considered to be a malformed type when referenced by a static member (20.1). The scopes associated with the type parameters of a generic function are described in (9.2).

The restriction on static members is necessary since a type variable has no meaning in the context of a static member, because statics are shared among all generic instantiations of a generic class. However, a type variable may be referenced from an instance initializer, even though ***this*** *is not available.*

Because type parameters are in scope in their bounds, we support F-bounded quantification. This enables typechecking code such as:

```
class Ordered<T> {
  operator >(T x);
}
class Sorter<T extends Ordered<T>> {
   sort(List<T> l) {... l[n] < l[n+1] ...}
}
```

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 (17.13).
- A type parameter cannot be used as a superclass or superinterface (10.9, 10.10, 11.2).
- A type parameter cannot be used as a generic type.

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

15.1 Variance

We say that a type S *occurs covariantly* in a type T iff S occurs in a covariant position in T, but not in a contravariant position, and not in an invariant position. ◇

We say that a type S *occurs contravariantly* in a type T iff S occurs in a contravariant position in T, but not in a covariant position, and not in an invariant position. ◇

We say that a type S *occurs invariantly* in a type T iff S occurs in an invariant position in T, or S occurs in a covariant position as well as a contravariant position. ◇

We say that a type S occurs in a *covariant position* in a type T iff one of the following conditions is true: ◇

- T is S
- T is of the form $G\texttt{<}S_1,\ \ldots,\ S_n\texttt{>}$ where G denotes a generic class and S occurs in a covariant position in S_j for some $j \in 1..n$.
- T is of the form S_0 **Function**$\texttt{<}X_1$ **extends** $B_1, \ldots\texttt{>(}S_1\ x_1, \ldots\texttt{)}$ where the type parameter list may be omitted, and S occurs in a covariant position in S_0.
- T is of the form

  S_0 **Function**$\texttt{<}X_1$ **extends** $B_1, \ldots\texttt{>}$

  $\texttt{(}S_1\ x_1, \ldots,\ S_k\ x_k,\ \texttt{[}S_{k+1}\ x_{k+1} = d_{k+1}, \ldots,\ S_n\ x_n = d_n\texttt{])}$

  or of the form

  S_0 **Function**$\texttt{<}X_1$ **extends** $B_1, \ldots\texttt{>}$

  $\texttt{(}S_1\ x_1, \ldots,\ S_k\ x_k,\ \texttt{\{}S_{k+1}\ x_{k+1} = d_{k+1}, \ldots,\ S_n\ x_n = d_n\texttt{\}})$

  where the type parameter list and each default value may be omitted, and S occurs in a contravariant position in S_j for some $j \in 1..n$.
- T is of the form $G\texttt{<}S_1, \ldots,\ S_n\texttt{>}$ where G denotes a generic type alias such that $j \in 1..n$, the formal type parameter corresponding to S_j is covariant, and S occurs in a covariant position in S_j.
- T is of the form $G\texttt{<}S_1, \ldots,\ S_n\texttt{>}$ where G denotes a generic type alias such that $j \in 1..n$, the formal type parameter corresponding to S_j is contravariant, and S occurs in a contravariant position in S_j.

We say that a type S occurs in a *contravariant position* in a type T iff one of the following conditions is true: ◇

- T is of the form $G\texttt{<}S_1,\ \ldots,\ S_n\texttt{>}$ where G denotes a generic class and S occurs in a contravariant position in S_j for some $j \in 1..n$.

- T is of the form S_0 **Function**<X_1 **extends** $B_1, \ldots$>(S_1 $x_1, \ldots$) where the type parameter list may be omitted, and S occurs in a contravariant position in S_0.

- T is of the form

  S_0 **Function**<X_1 **extends** $B_1, \ldots$>

  (S_1 $x_1, \ldots,$ S_k $x_k,$ [S_{k+1} $x_{k+1} = d_{k+1}, \ldots,$ S_n $x_n = d_n$])

  or of the form

  S_0 **Function**<X_1 **extends** $B_1, \ldots$>

  (S_1 $x_1, \ldots,$ S_k $x_k,$ {S_{k+1} $x_{k+1} = d_{k+1}, \ldots,$ S_n $x_n = d_n$})

  where the type parameter list and each default value may be omitted, and S occurs in a covariant position in S_j for some $j \in 1..n$.

- T is of the form G<$S_1, \ldots,$ S_n> where G denotes a generic type alias such that $j \in 1..n$, the formal type parameter corresponding to S_j is covariant, and S occurs in a contravariant position in S_j.

- T is of the form G<$S_1, \ldots,$ S_n> where G denotes a generic type alias such that $j \in 1..n$, the formal type parameter corresponding to S_j is contravariant, and S occurs in a covariant position in S_j.

We say that a type S occurs in an *invariant position* in a type T iff one of the following conditions is true: ◇

- T is of the form G<$S_1,$ $\ldots,$ S_n> where G denotes a generic class or a generic type alias, and S occurs in an invariant position in S_j for some $j \in 1..n$.

- T is of the form

  S_0 **Function**<X_1 **extends** $B_1, \ldots,$ X_m **extends** B_m>

  (S_1 $x_1, \ldots,$ S_k $x_k,$ [S_{k+1} $x_{k+1} = d_{k+1}, \ldots,$ S_n $x_n = d_n$])

  or of the form

  S_0 **Function**<X_1 **extends** $B_1, \ldots,$ X_m **extends** B_m>

  (S_1 $x_1, \ldots,$ S_k $x_k,$ {S_{k+1} $x_{k+1} = d_{k+1}, \ldots,$ S_n $x_n = d_n$})

  where the type parameter list and each default value may be omitted, and S occurs in an invariant position in S_j for some $j \in 0..n$, or S occurs in B_i for some $i \in 1..m$.

- T is of the form G<$S_1, \ldots,$ S_n> where G denotes a generic type alias, $j \in 1..n$, the formal type parameter corresponding to S_j is invariant, and S occurs in S_j.

Consider a generic type alias declaration G with formal type parameter declarations X_1 **extends** $B_1, \ldots, X_m$ **extends** B_m, and right hand side T. Let $j \in 1..m$. We say that *the formal type parameter X_j is invariant* iff X_j occurs invariantly in T, X_j *is covariant* iff X_j occurs covariantly in T, and X_j *is contravariant* iff X_j occurs contravariantly in T. ◇

Variance gives a characterization of the way a type varies as the value of a subterm varies, e.g., a type variable: Assume that T is a type where a type variable X occurs, and L and U are types such that L is a subtype of U. If X occurs covariantly in T then $[L/X]T$ is a subtype of $[U/X]T$. Similarly, if X occurs contravariantly in T then $[U/X]T$ is a subtype of $[L/X]T$. If X occurs invariantly then $[L/X]T$ and $[U/X]T$ are not guaranteed to be subtypes of each other in any direction. In short: with covariance, the type covaries; with contravariance, the type contravaries; with invariance, all bets are off.

15.2 Super-Bounded Types

This section describes how the declared upper bounds of formal type parameters are enforced, including some cases where a limited form of violation is allowed.

A *top type* is a type T such that `Object` is a subtype of T. For instance, `Object`, **dynamic**, and **void** are top types, and so are `FutureOr<`**void**`>` and `FutureOr<FutureOr<`**dynamic**`>>`. ◇

Every type which is not a parameterized type is *regular-bounded*. ◇

In particular, every non-generic class and every function type is a regular-bounded type.

Let T be a parameterized type of the form G`<`$S_1, \ldots, S_n$`>` where G denotes a generic class or a generic type alias. Let X_1 **extends** $B_1, \ldots, X_n$ **extends** B_n be the formal type parameter declarations of G. T is *regular-bounded* iff S_j is a subtype of $[S_1/X_1, \ldots, S_n/X_n]B_j$, for all $j \in 1..n$. ◇

This means that regular-bounded types are those types that do not violate their type parameter bounds.

Let T be a parameterized type of the form G`<`$S_1, \ldots, S_n$`>` where G denotes a generic class or a generic type alias. T is *super-bounded* iff the following conditions are both true: ◇

- T is not regular-bounded.
- Let T' be the result of replacing every occurrence in T of a top type in a covariant position by `Null`, and every occurrence in T of `Null` in a contravariant position by `Object`. It is then required that T' is regular-bounded. Moreover, if G denotes a generic type alias with body U, it is required that every type that occurs as a subterm of $[S_1/X_1, \ldots, S_n/X_n]U$ is well-bounded (defined below).

In short, at least one type argument violates its bound, but the type is regular-bounded after replacing all occurrences of an extreme type by an opposite extreme type, depending on their variance.

A type T is *well-bounded* iff it is either regular-bounded or super-bounded. ◇

Any use of a type T which is not well-bounded is a compile-time error.

It is a compile-time error if a parameterized type T is super-bounded when it is used in any of the following ways:

- T is an immediate subterm of a new expression (17.13.1) or a constant object expression (17.13.2).
- T is an immediate subterm of a redirecting factory constructor signature (10.7.2).
- T is an immediate subterm of an **extends** clause of a class (10.9), or it occurs as an element in the type list of an **implements** clause (10.10), or a **with** clause (10).

It is *not* an error if a super-bounded type occurs as an immediate subterm of an **extends** clause that specifies the bound of a type variable (15).

Types of members from super-bounded class types are computed using the same rules as types of members from other types. Types of function applications involving super-bounded types are computed using the same rules as types of function applications involving other types. Here is an example:

```
class A<X extends num> {
  X x;
}

A<Object> a;
```

With this, `a.x` has static type `Object`, even though the upper bound on the type variable `X` is `num`.

Super-bounded types enable the expression of informative common supertypes of some sets of types whose common supertypes would otherwise be much less informative.

For example, consider the following class:

```
class C<X extends C<X>> {
  X next;
}
```

Without super-bounded types, there is no type T which makes `C<`T`>` a common supertype of all types of the form `C<`S`>` (noting that all types must be regular-bounded when we do not have the notion of super-bounded types). So if we wish to allow a variable to hold any instance "of type `C`" then that variable must use `Object` or another top type as its type annotation, which means that a member like `next` is not known to exist (which is what we mean by saying that the type is 'less informative').

We could introduce a notion of recursive (infinite) types, and express the least upper bound of all types of the form `C<S>` as some syntax whose meaning could be approximated by `C<C<C<C<...>>>>`. However, we expect that any such

concept in Dart would incur a significant cost on developers and implementations in terms of added complexity and subtlety, so we have chosen not to do that. Super-bounded types are finite, but they offer a useful developer-controlled approximation to such infinite types.

For example, `C<Object>` and `C<C<C<`**`void`**`>>>` are types that a developer may choose to use as a type annotation. This choice serves as a commitment to a finite level of unfolding of the infinite type, and it allows for a certain amount of control at the point where the unfolding ends: If `c` has type `C<C<`**`dynamic`**`>>` then `c.next.next` has type **dynamic** and `c.next.next.whatever` has no compile-time error, but if `c` has type `C<C<`**`void`**`>>` then already `Object x = c.next.next;` is a compile-time error. It is thus possible for developers to get a more or less strict treatment of expressions whose type proceeds beyond the given finite unfolding.

15.3 Instantiation to Bound

This section describes how to compute type arguments that are omitted from a type, or from an invocation of a generic function.

Note that type inference is assumed to have taken place already (6), so type arguments are not considered to be omitted if they are inferred. This means that instantiation to bound is a backup mechanism, which will be used when no information is available for inference.

Consider the situation where a term t of the form $\langle typeName \rangle$ denotes a generic type declaration, and it is used as a type or as an expression in the enclosing program. This implies that type arguments are accepted, but not provided. We use the phrase *raw type* respectively *raw type expression* to identify such terms. In the following we only mention raw types, but everything said about raw types applies to raw type expressions in the obvious manner. ◇ ◇

For instance, with the declaration `Type listType() => List;`, evaluation of the raw type expression `List` in the body yields an instance of class `Type` reifying `List<dynamic>`, because `List` is subject to instantiation to bound. Note that `List<dynamic>` is not syntactically an expression, but it is still possible to get access to a `Type` instance reifying `List<dynamic>` without instantiation to bound, because it can be the value of a type variable.

We can unambiguously define raw types to denote the result of applying the generic type to a list of implicitly provided actual type arguments, and instantiation to bound is a mechanism which does just that. This is because Dart does not, and will not, support higher-kinded types; for example, the value of a type variable X will be a type, it cannot be the generic class `List` as such, and it cannot be applied to type arguments, e.g., X`<int>`.

In the typical case where only covariance is encountered, instantiation to bound will yield a supertype *of all the regular-bounded types that can be expressed. This allows developers to consider a raw type as a type which is used to specify that "the actual type arguments do not matter".* For example, assuming the declaration **class** `C<X extends num> {...}`, instantiation to bound on `C` yields `C<num>`, and this means that `C x;` can be used to declare a variable `x` whose value can be a `C<`T`>` for *all possible* values of T.

Conversely, consider the situation where a generic type alias denotes a function type, and it has one type parameter which is contravariant. Instantiation to bound on that type alias will then yield a subtype *of all the regular-bounded types that can be expressed by varying that type argument. This allows developers to consider such a type alias used as a raw type as a function type which allows the function to be passed to clients "where it does not matter which values for the type argument the client expects".* E.g., with **typedef** `F<X> = Function(X);` instantiation to bound on `F` yields `F<dynamic>`, and this means that `F f;` can be used to declare a variable `f` whose value will be a function that can be passed to clients expecting an `F<`T`>` for *all possible* values of T.

15.3.1 Auxiliary Concepts for Instantiation to Bound

Before we specify instantiation to bound we need to define two auxiliary concepts. Let T be a raw type. A type S then *raw-depends on* T if one or more of the following conditions hold: ◇

- S is of the form ⟨*typeName*⟩, and S is T. Note that this case is not applicable if S is a subterm of a term of the form S <typeArguments>, that is, if S receives any type arguments. Also note that S cannot be a type variable, because then 'S is T' cannot hold. See the discussion below and the reference to 20.4.2 for more details about why this is so.
- S is of the form ⟨*typeName*⟩ ⟨*typeArguments*⟩, and one of the type arguments raw-depends on T.
- S is of the form ⟨*typeName*⟩ ⟨*typeArguments*⟩? where ⟨*typeName*⟩ denotes a type alias F, and the body of F raw-depends on T.
- S is of the form ⟨*type*⟩? **Function** ⟨*typeParameters*⟩? ⟨*parameterTypeList*⟩ and ⟨*type*⟩? raw-depends on T, or a bound in ⟨*typeParameters*⟩? raw-depends on T, or a type in ⟨*parameterTypeList*⟩ raw-depends on T.

Meta-variables (20.4.1) like S and T are understood to denote types, and they are considered to be equal (as in 'S is T') in the same sense as in the section about subtype rules (20.4.2). In particular, even though two identical pieces of syntax may denote two distinct types, and two different pieces of syntax may denote the same type, the property of interest here is whether they denote the same type and not whether they are spelled identically.

The intuition behind the situation where a type raw-depends on another type is that we need to compute any missing type arguments for the latter in order to be able to tell what the former means.

In the rule about type aliases, F may or may not be generic, and type arguments may or may not be present. However, there is no need to consider the result of substituting actual type arguments for formal type parameters in the body of F (or even the correctness of passing those type arguments to F), because we only need to inspect all types of the form ⟨*typeName*⟩ in its body, and they are not affected

by such a substitution. In other words, raw-dependency is a relation which is simple and cheap to compute.

Let G be a generic class or a generic type alias with k formal type parameter declarations containing formal type parameters $X_1, \ldots, X_k$ and bounds $B_1, \ldots, B_k$. For any $j \in 1..k$, we say that the formal type parameter X_j has a *simple bound* when one of the following requirements is satisfied: ◇

- B_j is omitted.
- B_j is included, but does not contain any of $X_1, \ldots, X_k$. If B_j raw-depends on a raw type T then every type parameter of T must have a simple bound.

The notion of a simple bound must be interpreted inductively rather than coinductively, i.e., if a bound B_j of a generic class or generic type alias G is reached during an investigation of whether B_j is a simple bound, the answer is no.

For example, with **class** `C<X` **extends** `C> {}`, the type parameter `X` does not have a simple bound: A raw `C` is used as a bound for `X`, so `C` must have simple bounds, but one of the bounds of `C` is the bound of `X`, and that bound is `C`, so `C` must have simple bounds: That was a cycle, so the answer is "no", `C` does not have simple bounds.

Let G be a generic class or a generic type alias. We say that G *has simple bounds* iff every type parameter of G has simple bounds. ◇

We can now specify in which sense instantiation to bound requires the involved types to be "simple enough". We impose the following constraint on all type parameter bounds, because all type parameters may be subject to instantiation to bound.

It is a compile-time error if a formal type parameter bound B contains a raw type T, unless T has simple bounds.

So type arguments on bounds can only be omitted if they themselves have simple bounds. In particular, **class** `C<X` **extends** `C> {}` is a compile-time error, because the bound `C` is raw, and the formal type parameter `X` that corresponds to the omitted type argument does not have a simple bound.

Let T be a type of the form $\langle typeName \rangle$ which denotes a generic class or a generic type alias (so T is raw). Then T is equivalent to the parameterized type which is the result obtained by applying instantiation to bound to T. It is a compile-time error if the instantiation to bound fails.

This rule is applicable for all occurrences of raw types, e.g., when it occurs as a type annotation of a variable or a parameter, as a return type of a function, as a type which is tested in a type test, as the type in an $\langle onPart \rangle$, etc.

15.3.2 The Instantiation to Bound Algorithm

We now specify how the *instantiation to bound* algorithm proceeds. Let T be a raw type. Let $X_1, \ldots, X_k$ be the formal type parameters in the declaration of G, and let $B_1, \ldots, B_k$ be their bounds. For each $i \in 1..k$, let S_i denote ◇

the result of instantiation to bound on B_i; in the case where the ith bound is omitted, let S_i be **dynamic**.

If B_i for some i is raw (in general: if it raw-depends on some type U) then all its (respectively U's) omitted type arguments have simple bounds. This limits the complexity of instantiation to bound for B_i, and in particular it cannot involve a dependency cycle where we need the result from instantiation to bound for G in order to compute the instantiation to bound for G.

Let $U_{i,0}$ be S_i, for all $i \in 1..k$. This is the "current value" of the bound for type variable i, at step 0; in general we will consider the current step, m, and use data for that step, e.g., the bound $U_{i,m}$, to compute the data for step $m+1$.

Let $\rightarrow_m$ be a relation among the type variables $X_1, \ldots, X_k$ such that $X_p \rightarrow_m X_q$ iff X_q occurs in $U_{p,m}$. So each type variable is related to, that is, depends on, every type variable in its bound, which might include itself. Let $\rightarrow_m^+$ be the transitive (but not reflexive) closure of $\rightarrow_m$. For each m, let $U_{i,m+1}$, for $i \in 1..k$, be determined by the following iterative process, where V_m denotes $G\texttt{<}U_{1,m}, \ \ldots, \ U_{k,m}\texttt{>}$:

1. If there exists a $j \in 1..k$ such that $X_j \rightarrow_m^+ X_j$ (that is, if the dependency graph has a cycle) let $M_1, \ldots, M_p$ be the strongly connected components (SCCs) with respect to $\rightarrow_m$. That is, the maximal subsets of $X_1, \ldots, X_k$ where every pair of variables in each subset are related in both directions by $\rightarrow_m^+$; note that the SCCs are pairwise disjoint; also, they are uniquely defined up to reordering, and the order does not matter for this algorithm. Let M be the union of $M_1, \ldots, M_p$ (that is, all variables that participate in a dependency cycle). Let $i \in 1..k$. If X_i does not belong to M then $U_{i,m+1}$ is $U_{i,m}$. Otherwise there exists a q such that $X_i \in M_q$; $U_{i,m+1}$ is then obtained from $U_{i,m}$ by substituting **dynamic** for every occurrence of a variable in M_q that is in a position in V_m which is not contravariant, and substituting `Null` for every occurrence of a variable in M_q which is in a contravariant position in V_m.

2. Otherwise (when there are no dependency cycles), let j be the lowest number such that X_j occurs in $U_{p,m}$ for some p and $X_j \not\rightarrow_m X_q$ for all q in $1..k$ (that is, the bound of X_j does not contain any type variables, but X_j occurs in the bound of some other type variable). Then, for all $i \in 1..k$, $U_{i,m+1}$ is obtained from $U_{i,m}$ by substituting $U_{j,m}$ for every occurrence of X_j that is in a position in V_m which is not contravariant, and substituting `Null` for every occurrence of X_j which is in a contravariant position in V_m.

3. Otherwise (when there are no dependencies at all), terminate with the result $\texttt{<}U_{1,m}, \ \ldots, \ U_{k,m}\texttt{>}$.

This process will always terminate, because the total number of occurrences of type variables from $\{ X_1, \ldots, X_k \}$ in the current bounds is strictly decreasing with each step, and we terminate when that number reaches zero.

It may seem somewhat arbitrary to treat unused and invariant parameters in the same way as covariant parameters, in particular because invariant parame-

ters fail to satisfy the expectation that a raw type denotes a supertype of all the expressible regular-bounded types.

We could easily have made every instantiation to bound an error when applied to a type where invariance occurs anywhere during the run of the algorithm. However, there are a number of cases where this choice produces a usable type, and we decided that it is not helpful to outlaw such cases.

```
typedef Inv<X> = X Function(X);
class B<Y extends num, Z extends Inv<Y>> {}

B b; // The raw B means B<num, Inv<num>>.
```

For example, the value of `b` can have dynamic type `B<int, int Function(num)>`. However, the type arguments have to be chosen carefully, or the result will not be a subtype of `B`. For instance, `b` cannot have dynamic type `B<int, Inv<int>>`, because `Inv<int>` is not a subtype of `Inv<num>`.

A raw type T is a compile-time error if instantiation to bound on T yields a type which is not well-bounded (15.2).

This kind of error can occur, as demonstrated by the following example:

```
class C<X extends C<X>> {}
typedef F<X extends C<X>> = X Function(X);

F f; // Compile-time error.
```

With these declarations, the raw `F` which is used as a type annotation is a compile-time error: The algorithm yields `F<C<dynamic>>`, and that is neither a regular-bounded nor a super-bounded type. The resulting type can be specified explicitly as `C<dynamic> Function(C<dynamic>)`. That type exists, we just cannot express it by passing a type argument to `F`, so we make it an error rather than allowing it implicitly.

The core reason why it makes sense to make such a raw type an error is that there is no subtype relationship between the relevant parameterized types. For instance, `F<T1>` and `F<T2>` are unrelated, even when `T1 <: T2` or vice versa. In fact, there is no type `T` whatsoever such that a variable with declared type `F<T>` could be assigned to a variable of type `C<dynamic> Function(C<dynamic>)`. So the raw `F`, if permitted, would not be "a supertype of `F<T>` for all possible `T`", it would be a type which is unrelated to `F<T>` for *every single* `T` that satisfies the bound of `F`. This is so useless that we made it an error.

When instantiation to bound is applied to a type, it proceeds recursively: For a parameterized type G`<`$T_1, \ldots, T_k$`>` it is applied to $T_1, \ldots, T_k$. For a function type

T_0 **Function**`<`$X_1 \triangleleft B_1, \ldots, X_s \triangleleft B_s$`>(`$T_1, \ldots, T_n$`, [`$T_{n+1}, \ldots, T_{n+k}$`])`

and a function type

T_0 **Function**`<`$X_1 \triangleleft B_1, \ldots, X_s \triangleleft B_s$`>(`$T_1, \ldots, T_n$`, {`$T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}$`})`

it is applied to $T_0, \ldots, T_{n+k}$.

This means that instantiation to bound has no effect on a type that does not contain any raw types. Conversely, instantiation to bound acts on types which are syntactic subterms, also when they are deeply nested.

Instantiation to bound on a generic function f also uses the algorithm described above, taking the formal parameters $X_1, \ldots, X_k$ from the declaration of f, with bounds $B_1, \ldots, B_k$, and, for each $i \in 1..k$, letting S_i denote the result of instantiation to bound on B_i, and letting S_i be **dynamic** when the ith bound is omitted. ◇

Let f be a generic function declaration. If instantiation to bound on f yields a list of type arguments $T_1, \ldots, T_k$ such that, for some $j \in 1..k$, T_j is or contains a type which is not well-bounded, or if $T_1, \ldots, T_k$ does not satisfy the bounds on the formal type parameters of f, then we say that f *does not have default type arguments*. ◇

16 Metadata

Dart supports metadata which is used to attach user defined annotations to program structures.

⟨*metadata*⟩ ::= (‘`@`’ ⟨*metadatum*⟩)*

⟨*metadatum*⟩ ::=
⟨*identifier*⟩ | ⟨*qualifiedName*⟩ | ⟨*constructorDesignation*⟩ ⟨*arguments*⟩

Metadata consists of a series of annotations, each of which begin with the character ‘`@`’, followed by a constant expression e derivable from ⟨*metadatum*⟩. It is a compile-time error if e is not one of the following:

- A reference to a constant variable.
- A call to a constant constructor.

The expression e occurs in a constant context (17.3.2), which means that **const** modifiers need not be specified explicitly.

Metadata is associated with the abstract syntax tree of the program construct p that immediately follows the metadata, and which is not itself metadata or a comment. Metadata can be retrieved at run time via a reflective call, provided the annotated program construct p is accessible via reflection.

Obviously, metadata can also be retrieved statically by parsing the program and evaluating the constants via a suitable interpreter. In fact, many if not most uses of metadata are entirely static.

It is important that no run-time overhead be incurred by the introduction of metadata that is not actually used. Because metadata only involves constants, the time at which it is computed is irrelevant. So implementations may skip the metadata during ordinary parsing and execution, and evaluate it lazily.

It is possible to associate metadata with constructs that may not be accessible via reflection, such as local variables (though it is conceivable that in the future,

richer reflective libraries might provide access to these as well). This is not as useless as it might seem. As noted above, the data can be retrieved statically if source code is available.

Metadata can appear before a library, part header, class, typedef, type parameter, constructor, factory, function, parameter, or variable declaration, and before an import, export, or part directive.

The constant expression given in an annotation is type checked and evaluated in the scope surrounding the declaration being annotated.

17 Expressions

An *expression* is a fragment of Dart code that can be evaluated at run time. ◇

Every expression has an associated static type (20.1) and may have an associated static context type which may affect the static type and evaluation of the expression. Every object has an associated dynamic type (20.2).

⟨*expression*⟩ ::= ⟨*assignableExpression*⟩ ⟨*assignmentOperator*⟩ ⟨*expression*⟩
  | ⟨*conditionalExpression*⟩
  | ⟨*cascade*⟩
  | ⟨*throwExpression*⟩

⟨*expressionWithoutCascade*⟩ ::=
    ⟨*assignableExpression*⟩ ⟨*assignmentOperator*⟩ ⟨*expressionWithoutCascade*⟩
  | ⟨*conditionalExpression*⟩
  | ⟨*throwExpressionWithoutCascade*⟩

⟨*expressionList*⟩ ::= ⟨*expression*⟩ (‘,’ ⟨*expression*⟩)*

⟨*primary*⟩ ::= ⟨*thisExpression*⟩
  | **super** ⟨*unconditionalAssignableSelector*⟩
  | **super** ⟨*argumentPart*⟩
  | ⟨*functionExpression*⟩
  | ⟨*literal*⟩
  | ⟨*identifier*⟩
  | ⟨*newExpression*⟩
  | ⟨*constObjectExpression*⟩
  | ⟨*constructorInvocation*⟩
  | ‘(’ ⟨*expression*⟩ ‘)’

⟨*literal*⟩ ::= ⟨*nullLiteral*⟩
  | ⟨*booleanLiteral*⟩
  | ⟨*numericLiteral*⟩
  | ⟨*stringLiteral*⟩
  | ⟨*symbolLiteral*⟩
  | ⟨*listLiteral*⟩
  | ⟨*setOrMapLiteral*⟩

An expression e may always be enclosed in parentheses, but this never has any semantic effect on e.

However, it may have an effect on the surrounding expression. For instance, given a class `C` with a static method `m() => 42`, `C.m()` returns 42, but `(C).m()` is a compile-time error. The point is that the meaning of `C.m()` is specified in terms of several parts, rather than being specified in a strictly compositional manner. Concretely, the meaning of `C` and `(C)` as expressions is the same, but the meaning of `C.m()` is not defined in terms of the meaning of `C` as an expression, and it differs from the meaning of `(C).m()`.

17.1 Expression Evaluation

Evaluation of an expression either *produces an object* or it *throws* an exception object and an associated stack trace. In the former case, we also say that the expression *evaluates to an object*. ◇ ◇ ◇

If evaluation of one expression, e, is defined in terms of evaluation of another expression e_1, typically a subexpression of e, and the evaluation of e_1 throws an exception and a stack trace, the evaluation of e stops at that point and throws the same exception object and stack trace.

17.2 Object Identity

The predefined Dart function `identical()` is defined such that `identical(`c_1`,` c_2`)` iff:

- c_1 evaluates to either the null object (17.4) or an instance of `bool` and c_1 `==` c_2, OR
- c_1 and c_2 are instances of `int` and c_1 `==` c_2, OR
- c_1 and c_2 are constant strings and c_1 `==` c_2, OR
- c_1 and c_2 are instances of `double` and one of the following holds:
  - c_1 and c_2 are non-zero and c_1 `==` c_2.
  - Both c_1 and c_2 are $+0.0$.
  - Both c_1 and c_2 are -0.0.
  - Both c_1 and c_2 represent a NaN value with the same underlying bit pattern.

  OR
- c_1 and c_2 are constant lists that are defined to be identical in the specification of literal list expressions (17.9.4), OR
- c_1 and c_2 are constant maps that are defined to be identical in the specification of literal map expressions (17.9.8), OR

- c_1 and c_2 are constant objects of the same class C and the value of each instance variable of c_1 is identical to the value of the corresponding instance variable of c_2. OR
- c_1 and c_2 are the same object.

The definition of `identity` for doubles differs from that of equality in that a NaN is identical to itself, and that negative and positive zero are distinct.

The definition of equality for doubles is dictated by the IEEE 754 standard, which posits that NaNs do not obey the law of reflexivity. Given that hardware implements these rules, it is necessary to support them for reasons of efficiency.

The definition of identity is not constrained in the same way. Instead, it assumes that bit-identical doubles are identical.

The rules for identity make it impossible for a Dart programmer to observe whether a boolean or numerical value is boxed or unboxed.

17.3 Constants

All usages of 'constant' in Dart are associated with compile time. A potentially constant expression is an expression that will generally yield a constant value when the values of certain parameters are given. The constant expressions is a subset of the potentially constant expressions that *can* be evaluated at compile time.

The constant expressions are restricted to expressions that perform only simple arithmetic operations, boolean conditions, and string and instance creation. No user-written function body is executed during constant expression evaluation, only members of the system classes `Object`, `bool`, `int`, `double`, `String`, `Type`, `Symbol`, or `Null`.

The *potentially constant expressions* and *constant expressions* are the following: ◇ ◇

- A literal boolean, **true** or **false** (17.6), is a potentially constant and constant expression.
- A literal number (17.5) is a potentially constant and constant expression if it evaluates to an instance of type `int` or `double`.
- A literal string (17.7) with string interpolations (17.7.1) with expressions $e_1, \ldots, e_n$ is a potentially constant expression if $e_1, \ldots, e_n$ are potentially constant expressions. The literal is further a constant expression if $e_1, \ldots, e_n$ are constant expressions evaluating to instances of `int`, `double`, `String`, `bool`, or `Null`. These requirements hold trivially if there are no interpolations in the string. *It would be tempting to allow string interpolation where the interpolated value is any compile-time constant. However, this would require running the `toString()` method for constant objects, which could contain arbitrary code.*
- A literal symbol (17.8) is a potentially constant and constant expression.

- The literal **null** (17.4) is a potentially constant and constant expression.
- An identifier that denotes a constant variable is a potentially constant and constant expression.
- A qualified reference to a static constant variable (8) that is not qualified by a deferred prefix, is a potentially constant and constant expression. For example, if class C declares a constant static variable v, $C.v$ is a constant. The same is true if C is accessed via a prefix p; $p.C.v$ is a constant unless p is a deferred prefix.
- A simple or qualified identifier denoting a class, a mixin or a type alias that is not qualified by a deferred prefix, is a potentially constant and constant expression. The constant expression always evaluates to a `Type` object. For example, if C is the name of a class or type alias, the expression C is a constant, and if C is imported with a prefix p, $p.C$ is a constant `Type` instance representing the type of C unless p is a deferred prefix.
- Let e be a simple or qualified identifier denoting a top-level function (9) or a static method (10.8) that is not qualified by a deferred prefix. If e is not subject to generic function instantiation (17.17) then e is a potentially constant and constant expression. If generic function instantiation does apply to e and the provided actual type arguments are $T_1, \ldots, T_s$ then e is a potentially constant and constant expression iff each $T_j, j \in 1..s$, is a constant type expression (17.3).
- An identifier expression denoting a parameter of a constant constructor (10.7.3) that occurs in the initializer list of the constructor, is a potentially constant expression.
- A constant object expression (17.13.2), **const** `<`T_1, ..., T_k`>(`*arguments*`)` or **const** `<`T_1, ..., T_k`>.`*id*`(`*arguments*`)`, or either expression without the leading **const** that occurs in a constant context, is a potentially constant expression. It is further a constant expression if the invocation evaluates to an object. It is a compile-time error if a constant object expression is not a constant expression (17.13.2).
- A constant list literal (17.9.4), **const** `<`T`>[`e_1`, ..., `e_n`]`, or `<`T`>[`e_1`, ..., `e_n`]` that occurs in a constant context, is a potentially constant expression if T is a constant type expression, and $e_1, \ldots, e_n$ are constant expressions. It is further a constant expression if the list literal evaluates to an object.
- A constant set literal (17.9.7), **const** `<`T`>{`e_1`, ..., `e_n`}`, or `<`T`>{`e_1`, ..., `e_n`}` that occurs in a constant context, is a potentially constant expression if T is a constant type expression, and $e_1, \ldots, e_n$ are constant expressions. It is further a constant expression if the list literal evaluates to an object.

- A constant map literal (17.9.8), **const** `<`K`,` V`>{`k_1`:` v_1`,` ... `,` k_n`:` v_n`}`, or `<`K`,` V`>{`k_1`:` v_1`,` ... `,` k_n`:` v_n`}` that occurs in a constant context, is a potentially constant expression. It is further a constant expression if the map literal evaluates to an object.

- A parenthesized expression `(`e`)` is a potentially constant expression if e is a potentially constant expression. It is further a constant expression if e is a constant expression.

- An expression of the form `identical(`e_1`,` e_2`)` is a potentially constant expression if e_1 and e_2 are potentially constant expressions and `identical` is statically bound to the predefined dart function `identical()` discussed above (17.2). It is further a constant expression if e_1 and e_2 are constant expressions.

- An expression of the form e_1 `!=` e_2 is equivalent to `!(`e_1 `==` e_2`)` in every way, including whether it is potentially constant or constant.

- An expression of the form e_1 `==` e_2 is potentially constant if e_1 and e_2 are both potentially constant expressions. It is further constant if both e_1 and e_2 are constant, and either e_1 evaluates to an instance of `double` or an instance that that has primitive equality (10.2.3), or e_2 evaluates to the null object (17.4).

- An expression of the form `!`e_1 is potentially constant if e_1 is potentially constant. It is further constant if e_1 is a constant expression that evaluates to an instance of type `bool`.

- An expression of the form e_1 `&&` e_2 is potentially constant if e_1 and e_2 are both potentially constant expressions. It is further constant if e_1 is a constant expression and either

  1. e_1 evaluates to **false**, or
  2. e_1 evaluates to **true** and e_2 is a constant expression that evaluates to an instance of type `bool`.

- An expression of the form e_1 `||` e_2 is potentially constant if e_1 and e_2 are both potentially constant expressions. It is further constant if e_1 is a constant expression and either

  1. e_1 evaluates to **true**, or
  2. e_1 evaluates to **false** and e_2 is a constant expression that evaluates to an instance of type `bool`.

- An expression of the form `~`e_1 is a potentially constant expression if e_1 is a potentially constant expression. It is further a constant expression if e_1 is a constant expression that evaluates to an instance of type `int` such that `~` denotes an instance operator invocation.

- An expression of one of the forms e_1 `&` e_2, e_1 `|` e_2, or e_1 `^` e_2 is potentially constant if e_1 and e_2 are both potentially constant expressions. It is further constant if both e_1 and e_2 are constant expressions that both evaluate to instances of `int`, or both to instances of `bool`, such that the operator symbol '`&`', '`|`', respectively '`^`' denotes an instance operator invocation.

- An expression of one of the forms e_1 `~/` e_2, e_1 `>>` e_2, e_1 `>>>` e_2, or e_1 `<<` e_2 is potentially constant if e_1 and e_2 are both potentially constant expressions. It is further constant if both e_1 and e_2 are constant expressions that both evaluate to an instance of `int`, such that the operator symbol '`~/`', '`>>`', '`>>>`', respectively '`<<`' denotes an instance operator invocation.

- An expression of the form e_1 `+` e_2 is a potentially constant expression if e_1 and e_2 are both potentially constant expressions. It is further a constant expression if both e_1 and e_2 are constant expressions and either both evaluate to an instance of `int` or `double`, or both evaluate to an instance of `String`, such that '`+`' denotes an instance operator invocation.

- An expression of the form `-`e_1 is a potentially constant expression if e_1 is a potentially constant expression. It is further a constant expression if e_1 is a constant expression that evaluates to an instance of type `int` or `double`, such that '`-`' denotes an instance operator invocation.

- An expression of the form 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 is potentially constant if e_1 and e_2 are both potentially constant expressions. It is further constant if both e_1 and e_2 are constant expressions that evaluate to instances of `int` or `double`, such that the given operator symbol denotes an invocation of an instance operator.

- An expression of the form e_1 `?` e_2 `:` e_3 is potentially constant if e_1, e_2, and e_3 are all potentially constant expressions. It is constant if e_1 is a constant expression and either

  1. e_1 evaluates to **true** and e_2 is a constant expression, or
  2. e_1 evaluates to **false** and e_3 is a constant expression.

- An expression of the form e_1 `??` e_2 is potentially constant if e_1 and e_2 are both potentially constant expressions. It is further constant if e_1 is a constant expression and either

  1. e_1 evaluates to an object which is not the null object, or
  2. e_1 evaluates to the null object, and e_2 is a constant expression.

- An expression of the form e`.length` is potentially constant if e is a potentially constant expression. It is further constant if e is a constant expression that evaluates to an instance of `String`, such that `length` denotes an instance getter invocation.

- An expression of the form e `as` T is potentially constant if e is a potentially constant expression and T is a constant type expression, and it is further constant if e is constant. It is a compile-time error to evaluate the constant expression if the cast operation would throw, that is, if e evaluates to an object which is not the null object and not of type T.

- An expression of the form e `is` T is potentially constant if e is a potentially constant expression and T is a constant type expression, and it is further constant if e is constant.

- An expression of the form e `is!` T is equivalent to `!(`e `is` T`)` in every way, including whether it's potentially constant or constant.

A *constant type expression* is one of: ◇

- An simple or qualified identifier denoting a type declaration (a type alias, class or mixin declaration) that is not qualified by a deferred prefix, optionally followed by type arguments of the form `<`T_1`, ..., `T_n`>` where T_1, ..., T_n are constant type expressions.

- A type of the form `FutureOr<`T`>` where T is a constant type expression.

- A function type R `Function<`*typeParameters*`>(`*argumentTypes*`)` (where R and `<`*typeParameters*`>` may be omitted) and where R, *typeParameters* and *argumentTypes* (if present) contain only constant type expressions.

- The type **void**.

- The type **dynamic**.

It is a compile-time error if an expression is required to be a constant expression, but its evaluation would throw an exception. It is a compile-time error if an assertion is evaluated as part of a constant object expression evaluation, and the assertion would throw an exception.

It is a compile-time error if the value of a constant expression depends on itself.

As an example, consider:

```
class CircularConsts {
  // Illegal program - mutually recursive compile-time constants
  static const i = j; // a compile-time constant
  static const j = i; // a compile-time constant
}
```

17.3.1 Further Remarks on Constants and Potential Constants

There is no requirement that every constant expression evaluate correctly. Only when a constant expression is required (e.g., to initialize a constant variable, or as a default value of a formal parameter, or as metadata) do we insist that a constant expression actually be evaluated successfully at compile time.

The above is not dependent on program control-flow. The mere presence of a required compile-time constant whose evaluation would fail within a program is an error. This also holds recursively: since compound constants are composed out of constants, if any subpart of a constant would throw an exception when evaluated, that is an error. On the other hand, since implementations are free to compile code late, some compile-time errors may manifest quite late:

```
const x = 1 ~/ 0;
final y = 1 ~/ 0;

class K {
  m1() {
    var z = false;
    if (z) { return x; }
    else { return 2; }
  }

  m2() {
    if (true) { return y; }
    else { return 3; }
  }
}
```

An implementation is free to immediately issue a compilation error for `x`, but it is not required to do so. It could defer errors if it does not immediately compile the declarations that reference `x`. For example, it could delay giving a compilation error about the method `m1` until the first invocation of `m1`. However, it could not choose to execute `m1`, see that the branch that refers to `x` is not taken, and return 2 successfully.

The situation with respect to an invocation of `m2` is different. Because `y` is not a compile-time constant (even though its value is), one need not give a compile-time error upon compiling `m2`. An implementation may run the code, which will cause the getter for `y` to be invoked. At that point, the initialization of `y` must take place, which requires the initializer to be compiled, which will cause a compilation error.

The treatment of ***null*** *merits some discussion. Consider* ***null*** *+ 2. This expression always causes an error. We could have chosen not to treat it as a constant expression (and in general, not to allow* ***null*** *as a subexpression of numeric or boolean constant expressions). There are two arguments for including it: First, it is constant so we* can *evaluate it at compile time. Second, it seems more useful to give the error stemming from the evaluation explicitly.*

One might reasonably ask why e_1 ? e_2 : e_3 and e_1 ?? e_2 have constant forms. If e_1 is known statically, why do we need to test it? The answer is that there are contexts where e_1 is a variable, e.g., in constant constructor initializers such as `const C(foo): this.foo = foo ?? someDefaultValue;`

The difference between a potentially constant expression and a constant expression deserves some explanation. The key issue is how one treats the formal parameters of a constructor.

If a constant constructor is invoked from a constant object expression, the actual arguments will be required to be constant expressions. Therefore, if we were assured that constant constructors were always invoked from constant object expressions, we could assume that the formal parameters of a constructor were compile-time constants.

However, constant constructors can also be invoked from ordinary instance creation expressions (17.13.1), and so the above assumption is not generally valid.

Nevertheless, the use of the formal parameters of a constant constructor is of considerable utility. The concept of potentially constant expressions is introduced to facilitate limited use of such formal parameters. Specifically, we allow the usage of the formal parameters of a constant constructor for expressions that involve built-in operators, but not for constant objects, lists and maps. For instance:

```
class C {
  final x, y, z;
  const C(p, q): x = q, y = p + 100, z = p + q;
}
```

The assignment to `x` is allowed under the assumption that `q` is constant (even though `q` is not, in general a compile-time constant). The assignment to `y` is similar, but raises additional questions. In this case, the superexpression of `p` is `p + 100`, and it requires that `p` be a numeric constant expression for the entire expression to be considered constant. The wording of the specification allows us to assume that `p` evaluates to an integer, for an invocation of this constructor in a constant expression. A similar argument holds for `p` and `q` in the assignment to `z`.

However, the following constructors are disallowed:

```
class D {
  final w;
  const D.makeList(p): w = const [p]; // compile-time error
  const D.makeMap(p): w = const {"help": q}; // compile-time error
  const D.makeC(p): w = const C(p, 12); // compile-time error
}
```

The problem is that all these run afoul of the rules for constant lists (17.9.4), maps (17.9.8), and objects (17.13.2), all of which independently require their subexpressions to be constant expressions.

All of the illegal constructors of `D` above could not be sensibly invoked via **new**, *because an expression that must be constant cannot depend on a formal*

parameter, which may or may not be constant. In contrast, the legal examples make sense regardless of whether the constructor is invoked via ***const*** *or via* ***new****.*

Careful readers will of course worry about cases where the actual arguments to `C()` *are constants, but are not numeric. This is precluded by the rules on constant constructors (10.7.3), combined with the rules for evaluating constant objects (17.13.2).*

17.3.2 Constant Contexts

Let e be an expression; e occurs in a *constant context* iff one of the following applies: ◇

- e is an element of a list or set literal whose first token is **const**, or e is a key or a value of an entry of a map literal whose first token is **const**.
- e occurs as `@`e in a construct derived from ⟨*metadata*⟩.
- e is an actual argument in an expression derived from ⟨*constObjectExpression*⟩.
- e is the initializing expression of a constant variable declaration (8).
- e is a switch case expression (18.9).
- e is an immediate subexpression of an expression e_0 which occurs in a constant context, where e_0 is not a function literal (17.11).

A constant context is introduced in situations where an expression is required to be constant. This is used to allow the ***const*** *modifier to be omitted in cases where it does not contribute any new information.*

17.4 Null

The reserved word **null** evaluates to the *null object*. ◇

⟨*nullLiteral*⟩ ::= **null**

The null object is the sole instance of the built-in class `Null`. Attempting to instantiate `Null` causes a compile-time error. It is a compile-time error for a class to extend, mix in or implement `Null`. The `Null` class extends the `Object` class and declares no methods except those also declared by `Object`.

The null object has primitive equality (10.2.3).

The static type of **null** is the `Null` type.

17.5 Numbers

A *numeric literal* is either a decimal or hexadecimal numeral representing an integer value, or a decimal double representation. ◇

⟨*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*⟩

A numeric literal starting with ‘0x’ or ‘0X’ is a *hexadecimal integer literal*. ◇ It has the numeric integer value of the hexadecimal numeral following ‘0x’ (respectively ‘0X’).

A numeric literal that contains only decimal digits is a *decimal integer literal*. ◇ It has the numeric integer value of the decimal numeral.

An *integer literal* is either a hexadecimal integer literal or a decimal integer literal. ◇

Let l be an integer literal that is not the operand of by a unary minus operator, and let T be the static context type of l. If `double` is assignable to T and `int` is not assignable to T, then the static type of l is `double`; otherwise the static type of l is `int`.

This means that an integer literal denotes a `double` when it would satisfy the type requirement, and an `int` would not. Otherwise it is an `int`, even in situations where that is an error.

A numeric literal that is not an integer literal is a *double literal*. A double literal always contains either a decimal point or an exponent part. ◇ The static type of a double literal is `double`.

If l is an integer literal with numeric value i and static type `int`, and l is not the operand of a unary minus operator, then evaluation of l proceeds as follows:

- If l is a hexadecimal integer literal, $2^{63} \leq i < 2^{64}$ and the `int` class is implemented as signed 64-bit two's complement integers, then l evaluates to an instance of the `int` class representing the numeric value $i - 2^{64}$,
- Otherwise l evaluates to an instance of the `int` class representing the numeric value i. It is a compile-time error if the integer i cannot be represented exactly by an instance of `int`.

Integers in Dart are designed to be implemented as 64-bit two's complement integer representations. In practice, implementations may be limited by other considerations. For example, Dart compiled to JavaScript may use the JavaScript number type, equivalent to Dart `double`, to represent integers, and if so, integer literals with more than 53 bits of precision cannot be represented exactly.

A double literal evaluates to a an instance of the `double` class representing a 64 bit double precision floating point number as specified by the IEEE 754 standard.

An integer literal with static type `double` and numeric value i evaluates to an instance of the `double` class representing the value i. It is a compile-time error if the value i cannot be represented *precisely* by an instance of `double`.

A 64 bit double precision floating point number is usually taken to represent a range of real numbers around the precise value denoted by the number's sign, mantissa and exponent. For integer literals evaluating to `double` values we insist that the integer literal's numeric value is the precise value of the `double` instance.

It is a compile-time error for a class to extend, mix in or implement `int`. It is a compile-time error for a class to extend, mix in or implement `double`. It is a compile-time error for any class other than `int` and `double` to extend, mix in or implement `num`.

The instances of `int` and `double` all override the '`==`' operator inherited from the `Object` class.

17.6 Booleans

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

⟨*booleanLiteral*⟩ ::= **true**
| **false**

Both *true* and *false* are instances of the built-in class `bool`, and there are no other objects that implement `bool`. It is a compile-time error for a class to extend, mix in or implement `bool`. ◇ ◇

The **true** and **false** objects have primitive equality (10.2.3).

Invoking the getter `runtimeType` on a boolean value returns the `Type` object that is the value of the expression `bool`. The static type of a boolean literal is `bool`.

17.7 Strings

A *string* is a sequence of UTF-16 code units. ◇

This decision was made for compatibility with web browsers and Javascript. Earlier versions of the specification required a string to be a sequence of valid Unicode code points. Programmers should not depend on this distinction.

⟨*stringLiteral*⟩ ::= (⟨*multilineString*⟩ | ⟨*singleLineString*⟩)+

A string can be a sequence of single line strings and multiline strings.

⟨*singleLineString*⟩ ::= ⟨*RAW_SINGLE_LINE_STRING*⟩
  | ⟨*SINGLE_LINE_STRING_SQ_BEGIN_END*⟩
  | ⟨*SINGLE_LINE_STRING_SQ_BEGIN_MID*⟩ ⟨*expression*⟩
    (⟨*SINGLE_LINE_STRING_SQ_MID_MID*⟩ ⟨*expression*⟩)*
    ⟨*SINGLE_LINE_STRING_SQ_MID_END*⟩
  | ⟨*SINGLE_LINE_STRING_DQ_BEGIN_END*⟩
  | ⟨*SINGLE_LINE_STRING_DQ_BEGIN_MID*⟩ ⟨*expression*⟩
    (⟨*SINGLE_LINE_STRING_DQ_MID_MID*⟩ ⟨*expression*⟩)*
    ⟨*SINGLE_LINE_STRING_DQ_MID_END*⟩

⟨*RAW_SINGLE_LINE_STRING*⟩ ::= 'r' '' (~('' | '\r' | '\n'))* ''
  | 'r' '"' (~('"' | '\r' | '\n'))* '"'

⟨*STRING_CONTENT_COMMON*⟩ ::= ~('\' | '' | '"' | '$' | '\r' | '\n')
  | ⟨*ESCAPE_SEQUENCE*⟩
  | '\' ~('n' | 'r' | 'b' | 't' | 'v' | 'x' | 'u' | '\r' | '\n')
  | ⟨*SIMPLE_STRING_INTERPOLATION*⟩

⟨*STRING_CONTENT_SQ*⟩ ::= ⟨*STRING_CONTENT_COMMON*⟩ | '"'

⟨*SINGLE_LINE_STRING_SQ_BEGIN_END*⟩ ::=
    '' ⟨*STRING_CONTENT_SQ*⟩* ''

⟨*SINGLE_LINE_STRING_SQ_BEGIN_MID*⟩ ::=
    '' ⟨*STRING_CONTENT_SQ*⟩* '${'

⟨*SINGLE_LINE_STRING_SQ_MID_MID*⟩ ::=
    '}' ⟨*STRING_CONTENT_SQ*⟩* '${'

⟨*SINGLE_LINE_STRING_SQ_MID_END*⟩ ::=
    '}' ⟨*STRING_CONTENT_SQ*⟩* ''

⟨*STRING_CONTENT_DQ*⟩ ::= ⟨*STRING_CONTENT_COMMON*⟩ | ''

⟨*SINGLE_LINE_STRING_DQ_BEGIN_END*⟩ ::=
    '"' ⟨*STRING_CONTENT_DQ*⟩* '"'

⟨*SINGLE_LINE_STRING_DQ_BEGIN_MID*⟩ ::=
    '"' ⟨*STRING_CONTENT_DQ*⟩* '${'

⟨*SINGLE_LINE_STRING_DQ_MID_MID*⟩ ::=
    '}' ⟨*STRING_CONTENT_DQ*⟩* '${'

⟨*SINGLE_LINE_STRING_DQ_MID_END*⟩ ::=
    '}' ⟨*STRING_CONTENT_DQ*⟩* '"'

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"`.

The grammar ensures that a single line string cannot span more than one line of source code, unless it includes an interpolated expression that spans multiple lines.

Adjacent strings are implicitly concatenated to form a single string literal.

Here is an example:

```
print("A string" "and then another"); // A stringand then another
```

Dart also supports the operator + for string concatenation.

The + operator on Strings requires a String argument. It does not coerce its argument into a string. This helps avoid puzzlers such as

```
print("A simple sum: 2 + 2 = " +
            2 + 2);
```

which would print 'A simple sum: 2 + 2 = 22' rather than 'A simple sum: 2 + 2 = 4'. However, for efficiency reasons, the recommended Dart idiom is to use string interpolation.

```
print("A simple sum: 2 + 2 = ${2+2}");
```

String interpolation works well for most cases. The main situation where it is not fully satisfactory is for string literals that are too large to fit on a line. Multiline strings can be useful, but in some cases, we want to visually align the code. This can be expressed by writing smaller strings separated by whitespace, as shown here:

```
'Imagine this is a very long string that does not fit on a line. What shall we do? '
'Oh what shall we do? '
'We shall split it into pieces '
'like so'.
```

An auxiliary *string interpolation state stack* is maintained outside the parser, in order to ensure that string interpolations are matched up correctly. ◇

This is necessary because the expression of a non-simple string interpolation may itself contain string literals with their own non-simple string interpolations.

For rules with names ⟨*..._BEGIN_MID*⟩, a marker is pushed on the auxiliary stack to indicate that a string interpolation of the given kind has started, where the kind is '', "", '''', or """". For rules with names ⟨*..._MID_MID*⟩, only the rule with the kind on the top of the auxiliary stack can be used. For rules with names ⟨*..._MID_END*⟩, only the rule with the kind on the top of the auxiliary stack can be used, and the marker is then popped.

⟨*multilineString*⟩ ::= ⟨*RAW_MULTI_LINE_STRING*⟩
| ⟨*MULTI_LINE_STRING_SQ_BEGIN_END*⟩
| ⟨*MULTI_LINE_STRING_SQ_BEGIN_MID*⟩ ⟨*expression*⟩
(⟨*MULTI_LINE_STRING_SQ_MID_MID*⟩ ⟨*expression*⟩)*
⟨*MULTI_LINE_STRING_SQ_MID_END*⟩
| ⟨*MULTI_LINE_STRING_DQ_BEGIN_END*⟩
| ⟨*MULTI_LINE_STRING_DQ_BEGIN_MID*⟩ ⟨*expression*⟩
(⟨*MULTI_LINE_STRING_DQ_MID_MID*⟩ ⟨*expression*⟩)*
⟨*MULTI_LINE_STRING_DQ_MID_END*⟩

⟨*RAW_MULTI_LINE_STRING*⟩ ::= 'r' ''''' .*? '''''
| 'r' '"""' .*? '"""'

⟨*QUOTES_SQ*⟩ ::= | ''' | '''

⟨*STRING_CONTENT_TSQ*⟩ ::=
⟨*QUOTES_SQ*⟩ (⟨*STRING_CONTENT_COMMON*⟩ | '"' | '\r' | '\n')

⟨*MULTI_LINE_STRING_SQ_BEGIN_END*⟩ ::=
''''' ⟨*STRING_CONTENT_TSQ*⟩* '''''

⟨*MULTI_LINE_STRING_SQ_BEGIN_MID*⟩ ::=
''''' ⟨*STRING_CONTENT_TSQ*⟩* ⟨*QUOTES_SQ*⟩ '${'

⟨*MULTI_LINE_STRING_SQ_MID_MID*⟩ ::=
'}' ⟨*STRING_CONTENT_TSQ*⟩* ⟨*QUOTES_SQ*⟩ '${'

⟨*MULTI_LINE_STRING_SQ_MID_END*⟩ ::=
'}' ⟨*STRING_CONTENT_TSQ*⟩* '''''

⟨*QUOTES_DQ*⟩ ::= | '"' | '""'

⟨*STRING_CONTENT_TDQ*⟩ ::=
⟨*QUOTES_DQ*⟩ (⟨*STRING_CONTENT_COMMON*⟩ | ''' | '\r' | '\n')

⟨*MULTI_LINE_STRING_DQ_BEGIN_END*⟩ ::=
'"""' ⟨*STRING_CONTENT_TDQ*⟩* '"""'

⟨*MULTI_LINE_STRING_DQ_BEGIN_MID*⟩ ::=
'"""' ⟨*STRING_CONTENT_TDQ*⟩* ⟨*QUOTES_DQ*⟩ '${'

⟨*MULTI_LINE_STRING_DQ_MID_MID*⟩ ::=
'}' ⟨*STRING_CONTENT_TDQ*⟩* ⟨*QUOTES_DQ*⟩ '${'

⟨*MULTI_LINE_STRING_DQ_MID_END*⟩ ::=
'}' ⟨*STRING_CONTENT_TDQ*⟩* '"""'

⟨*ESCAPE_SEQUENCE*⟩ ::= '`\n`' | '`\r`' | '`\f`' | '`\b`' | '`\t`' | '`\v`'
| '`\x`' ⟨*HEX_DIGIT*⟩ ⟨*HEX_DIGIT*⟩
| '`\u`' ⟨*HEX_DIGIT*⟩ ⟨*HEX_DIGIT*⟩ ⟨*HEX_DIGIT*⟩ ⟨*HEX_DIGIT*⟩
| '`\u{`' ⟨*HEX_DIGIT_SEQUENCE*⟩ '`}`'

⟨*HEX_DIGIT_SEQUENCE*⟩ ::=
⟨*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. If the first line of a multiline string consists solely of the whitespace characters defined by the production ⟨*WHITESPACE*⟩ (21.1), possibly prefixed by '`\`', then that line is ignored, including the line break at its end.

The idea is to ignore a whitespace-only first line of a multiline string, where whitespace is defined as tabs, spaces and the final line break. These can be represented directly, but since for most characters prefixing by backslash is an identity in a non-raw string, we allow those forms as well.

In the rule for ⟨*RAW_MULTI_LINE_STRING*⟩, the two occurrences of '`.*?`' denote a non-greedy token recognition step: It terminates as soon as the lookahead is the specified next token (that is, '`'''`' or '`"""`').

Note that multi-line string interpolation relies on the auxiliary string interpolation state stack, just like single-line string interpolation.

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`' ⟨*HEX_DIGIT*⟩$_1$ ⟨*HEX_DIGIT*⟩$_2$, equivalent to
  '`\u{`' ⟨*HEX_DIGIT*⟩$_1$ ⟨*HEX_DIGIT*⟩$_2$ '`}`'.
- '`\u`' ⟨*HEX_DIGIT*⟩$_1$ ⟨*HEX_DIGIT*⟩$_2$ ⟨*HEX_DIGIT*⟩$_3$ ⟨*HEX_DIGIT*⟩$_4$, equivalent to
  '`\u{`' ⟨*HEX_DIGIT*⟩$_1$ ⟨*HEX_DIGIT*⟩$_2$ ⟨*HEX_DIGIT*⟩$_3$ ⟨*HEX_DIGIT*⟩$_4$ '`}`'.
- '`\u{`' ⟨*HEX_DIGIT_SEQUENCE*⟩ '`}`' is the Unicode code point represented by the ⟨*HEX_DIGIT_SEQUENCE*⟩. It is a compile-time error if the value of the ⟨*HEX_DIGIT_SEQUENCE*⟩ is not a valid Unicode code point. For example, '`\u{0A}`' is the code point U+000A.

- '`$`' 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`'}.

Any string may be prefixed with the character '`r`', indicating that it is a *raw string*, in which case no escapes or interpolations are recognized. ◇

Line breaks in a multiline string are represented by the ⟨*LINE_BREAK*⟩ production. A line break introduces a single newline character (U+000A) into the string value.

It is a compile-time error if a non-raw string literal 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 non-raw string literal 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.

⟨*LINE_BREAK*⟩ ::= '`\n`'
| '`\r\n`'
| '`\r`'

All string literals evaluate to instances of the built-in class `String`. It is a compile-time error for a class to extend, mix in or implement `String`. The `String` class overrides the '`==`' operator inherited from the `Object` class. The static type of a string literal is `String`.

17.7.1 String Interpolation

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

⟨*stringInterpolation*⟩ ::= ⟨*SIMPLE_STRING_INTERPOLATION*⟩
| '`${`' ⟨*expression*⟩ '`}`'

⟨*SIMPLE_STRING_INTERPOLATION*⟩ ::=
'`$`' (⟨*IDENTIFIER_NO_DOLLAR*⟩ | ⟨*BUILT_IN_IDENTIFIER*⟩ | **this**)

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 does not contain the '`$`' character (but it can be a built-in identifier), or the reserved word **this**.
- An expression e delimited by curly braces.

The form `$`*id* is equivalent to the form `${`*id*`}`. An interpolated string, s, with content 's_0`${`e_1`}`$s_1 \ldots s_{n-1}$`${`e_n`}`s_n' (where any of $s_0, \ldots, s_n$ can be empty) is evaluated by evaluating each expression e_i $(1 \leq i \leq n)$ into a string r_i in the order they occur in the source text, as follows:

- Evaluate e_i to an object o_i.
- Invoke the `toString` method on o_i with no arguments, and let r_i be the returned object.
- If r_i is the null object, a dynamic error occurs.

Finally, the result of the evaluation of s is the concatenation of the strings s_0, r_1, ..., r_n, and s_n.

17.8 Symbols

A *symbol literal* denotes a name that would be either a valid declaration name, a valid library name, or **void**. ◇

⟨*symbolLiteral*⟩ ::= '`#`' (⟨*identifier*⟩ ('`.`' ⟨*identifier*⟩)* | ⟨*operator*⟩ | **void**)

The static type of a symbol literal is `Symbol`.

Let *id* be an identifier that does not begin with an underscore ('`_`'). The symbol literal `#`*id* evaluates to an instance of `Symbol` representing the identifier *id*.

A symbol literal `#`id_1`.`id_2`.`$\cdots$`.`id_n where $id_1, \ldots, id_n$ are identifiers evaluates to an instance of `Symbol` representing that particular sequence of identifiers. This kind of symbol literal denotes the name of a library declaration, as specified in a ⟨*libraryName*⟩. Library names are not subject to library privacy, even if some of its identifiers begin with an underscore.

A symbol literal `#`*op* where *op* is derived from ⟨*operator*⟩ evaluates to an instance of `Symbol` representing that particular operator name.

The symbol literal `#void` evaluates to an instance of `Symbol` representing the reserved word **void**.

For the value o of a symbol literal representing a source code term as specified in the previous paragraphs, we say that o is a *non-private symbol based on* the string whose contents is the characters of that term, without whitespace. ◇

Note that this does not apply for private symbols, which are discussed below. A private symbol is not based on any string.

If o is the value of an invocation of the `Symbol` constructor of the form `Symbol(`e`)`, **new** `Symbol(`e`)`, or **const** `Symbol(`e`)`, where e is an expression (constant if necessary) that evaluates to a string s, we say that o is a *non-private symbol based on* s. ◇

Note that `Symbol('_foo')` is a non-private symbol, and it is distinct from `#_foo`, as described below.

Assume that $i \in 1, 2$, and that o_i is the value of a constant expression which

is a symbol based on the string s_i. If s_1 `==` s_2 then o_1 and o_2 is the same object. That is, symbol instances are canonicalized.

If o_1 and o_2 are non-private symbols (not necessarily constant) based on strings s_1 and s_2 then o_1 and o_2 are equal according to operator '`==`' if and only if s_1 `==` s_2 (17.27).

A symbol literal `#`*id* where *id* is an identifier evaluates to an instance of `Symbol` representing the private identifier *id* of the enclosing library. All occurrences of `#`*id in the same library* evaluate to the same object, and no other symbol literal or `Symbol` constructor invocation evaluates to the same object, nor to a `Symbol` instance that is equal to that object according to the '`==`' operator.

One may well ask what is the motivation for introducing literal symbols? In some languages, symbols are canonicalized whereas strings are not. However literal strings are already canonicalized in Dart. Symbols are slightly easier to type compared to strings and their use can become strangely addictive, but this is not nearly sufficient justification for adding a literal form to the language. The primary motivation is related to the use of reflection and a web specific practice known as minification.

Minification compresses identifiers consistently throughout a program in order to reduce download size. This practice poses difficulties for reflective programs that refer to program declarations via strings. A string will refer to an identifier in the source, but the identifier will no longer be used in the minified code, and reflective code using these would fail. Therefore, Dart reflection uses objects of type `Symbol` rather than strings. Instances of `Symbol` are guaranteed to be stable with respect to minification. Providing a literal form for symbols makes reflective code easier to read and write. The fact that symbols are easy to type and can often act as convenient substitutes for enums are secondary benefits.

17.9 Collection Literals

This section specifies several literal expressions denoting collections. Some syntactic forms may denote more than one kind of collection, in which case a disambiguation step is performed in order to determine the kind (17.9.5).

The subsections of this section are concerned with mechanisms that are common to all kinds of collection literals (17.9.1, 17.9.2), followed by a specification of list literals (17.9.3, 17.9.4), followed by a specification of how to disambiguate and infer types for sets and maps (17.9.5, 17.9.6), and finally a specification of sets (17.9.7) and maps (17.9.8).

⟨*listLiteral*⟩ ::= **const**? ⟨*typeArguments*⟩? '`[`' ⟨*elements*⟩? '`]`'

⟨*setOrMapLiteral*⟩ ::= **const**? ⟨*typeArguments*⟩? '`{`' ⟨*elements*⟩? '`}`'

⟨*elements*⟩ ::= ⟨*element*⟩ ('`,`' ⟨*element*⟩)* '`,`'?

⟨*element*⟩ ::= ⟨*expressionElement*⟩
  | ⟨*mapElement*⟩
  | ⟨*spreadElement*⟩
  | ⟨*ifElement*⟩
  | ⟨*forElement*⟩

⟨*expressionElement*⟩ ::= ⟨*expression*⟩

⟨*mapElement*⟩ ::= ⟨*expression*⟩ '`:`' ⟨*expression*⟩

⟨*spreadElement*⟩ ::= ('`...`' | '`...?`') ⟨*expression*⟩

⟨*ifElement*⟩ ::= **if** '`(`' ⟨*expression*⟩ '`)`' ⟨*element*⟩ (**else** ⟨*element*⟩)?

⟨*forElement*⟩ ::= **await**? **for** '`(`' ⟨*forLoopParts*⟩ '`)`' ⟨*element*⟩

Syntactically, a *collection literal* can be a ⟨*listLiteral*⟩ or a ⟨*setOrMapLiteral*⟩. ◇
The contents of the collection is specified as a sequence of *collection literal ele-* ◇
ments, in short *elements*. Each element may be a declarative specification of a ◇
single entity, such as an ⟨*expressionElement*⟩ or a ⟨*mapElement*⟩, it may specify a collection which is to be included, of the form ⟨*spreadElement*⟩, or it may be a computational element specifying how to obtain zero or more entities, of the form ⟨*ifElement*⟩ or ⟨*forElement*⟩.

Terms derived from ⟨*element*⟩, and the ability to build collections from them, is also known as *UI-as-code*. ◇

The *leaf elements* of an element ℓ derived from ⟨*expressionElement*⟩ or ◇
⟨*mapElement*⟩ is $\{\ell\}$. The leaf elements of an element of the form **if** `(`e`)` ℓ or **for** `(`*forLoopParts*`)` ℓ is the leaf elements of ℓ. The leaf elements of an element of the form **if** `(`e`)` ℓ_1 **else** ℓ_2 is the union of the leaf elements of ℓ_1 and ℓ_2. The leaf elements of a ⟨*spreadElement*⟩ is the empty set.

The leaf elements of a collection literal is always a set of expression elements and/or map elements.

In order to allow collection literals to occur as constant expressions, we specify what it means for an element ℓ to be *constant* or *potentially constant*: ◇
◇

- When ℓ is an ⟨*expressionElement*⟩ of the form e:

  ℓ is a potentially constant element if e is a potentially constant expression, and ℓ is a constant element if e is a constant expression.

- When ℓ is a ⟨*mapElement*⟩ of the form 'e_1`:` e_2':

  ℓ is a potentially constant element if both e_1 and e_2 are potentially constant expressions, and it is a constant element if they are constant expressions.

- When ℓ is a ⟨*spreadElement*⟩ of the form '`...`e' or '`...?`e':

  ℓ is a potentially constant element if e is a potentially constant expression.

ℓ is a constant element if e is a constant expression that evaluates to a `List`, `Set`, or `Map` instance originally created by a list, set, or map literal.

Moreover, ℓ is a constant element if it is '`...?`e', where e is a constant expression that evaluates to the null object.

- When ℓ is an ⟨*ifElement*⟩ of the form **if** `(`b`)` ℓ_1 or the form **if** `(`b`)` ℓ_1 **else** ℓ_2:

  ℓ is a potentially constant element if b is a potentially constant expression, ℓ_1 is potentially constant, and so is ℓ_2, if present.

  ℓ is a constant element if b is a constant expression and:

  - ℓ is **if** `(`b`)` ℓ_1 and either b evaluates to **true** and ℓ_1 is constant, or b evaluates to **false** and ℓ_1 is potentially constant.
  - ℓ is **if** `(`b`)` ℓ_1 **else** ℓ_2 and either b evaluates to **true**, ℓ_1 is constant, and ℓ_2 is potentially constant; or b evaluates to **false**, ℓ_1 is potentially constant, and ℓ_2 is constant.

A ⟨*forElement*⟩ can never occur in a constant collection literal.

17.9.1 Type Promotion

An ⟨*ifElement*⟩ interacts with type promotion in the same way that **if** statements do. Let ℓ be an ⟨*ifElement*⟩ of the form **if** `(`b`)` ℓ_1 or **if** `(`b`)` ℓ_1 **else** ℓ_2. If b shows that a local variable v has type T, then the type of v is known to be T in ℓ_1, unless any of the following are true:

- v is potentially mutated in ℓ_1,
- v is potentially mutated within a function other than the one where v is declared, or
- v is accessed by a function defined in ℓ_1 and v is potentially mutated anywhere in the scope of v.

Type promotion will likely get more sophisticated in a future version of Dart. When that happens, ⟨*ifElement*⟩s will continue to match **if** statements (18.5).

17.9.2 Collection Literal Element Evaluation

The evaluation of a sequence of collection literal elements (17.9) yields a *collection literal object sequence*, also called an *object sequence* when no ambiguity can arise. ◇ ◇

We use the notation $[\![\ldots]\!]$ to denote an object sequence with explicitly listed elements, and we use '+' to compute the concatenation of object sequences (as in $s_1 + s_2$), which is an operation that will succeed and has no side-effects. Each element in the sequence is an object o or a pair $o_1 : o_2$. There is no notion of an element type for an object sequence, and hence no notion of dynamic errors arising from a type mismatch during concatenation. ◇

Object sequences can safely be treated as a low-level mechanism which may omit otherwise required actions like dynamic type checks because every access to an object sequence occurs in code created by language defined desugaring on statically checked constructs. It is left unspecified how an object sequence is implemented, it is only required that it contains the indicated objects or pairs in the given order. For each kind of collection, the sequence is used in the given order to populate the collection, in a manner which is specific to the kind, and which is specified separately (17.9.4, 17.9.7, 17.9.8).

There may be an actual data structure representing the object sequence at run time, but the object sequence could also be eliminated, e.g., because each element is inserted directly into the target collection as soon as it has been computed. Note that each object sequence will exclusively contain objects, or it will exclusively contain pairs, because any attempt to create a mixed sequence would cause an error at compile time or at run time (the latter may occur for a spread element with static type **dynamic**).

Assume that a literal collection *target* is given, and the object sequence obtained as described below will be used to populate *target*. Let T_{target} denote the dynamic type of *target*.

Access to the type of *target* is needed below in order to raise dynamic errors at specific points during the evaluation of an object sequence. Note that the dynamic type of *target* is statically known, except for the binding of any type variables in its ⟨*typeArguments*⟩. This implies that some questions can be answered at compile-time, e.g., whether or not `Iterable` occurs as a superinterface of T_{target}. In any case, T_{target} is guaranteed to implement `Iterable` (when *target* is a list or a set) or `Map` (when *target* is a map), but never both.

Assume that a location in code and a dynamic context is given, such that ordinary expression evaluation is possible. *Evaluation of a collection literal element sequence* at that location and in that context is specified as follows: ◇

Let s_{syntax} of the form $\ell_1, \ldots, \ell_k$ be a sequence of collection literal elements. The sequence of objects s_{object} obtained by evaluating s_{syntax} is the concatenation of the sequences of objects obtained by evaluating each element ℓ_j, $j \in 1..k$: $s_{object} = \mathit{evaluateElement}(\ell_1) + \ldots + \mathit{evaluateElement}(\ell_k)$, where $\mathit{evaluateElement}(\ell_j)$ denotes the object sequence yielded by evaluation of a single collection literal element ℓ_j.

When a pseudo-statement of the form $s := s + \mathit{evaluateElement}(\ell)$**;** is used in normative code below, it denotes the extension of s with the object sequence yielded by evaluation of ℓ, but it also denotes the specification of actions taken to produce said object sequence, and to produce the side effects associated with this computation, as implied by evaluation of expressions and execution of statements as specified below for the evaluation of $\mathit{evaluateElement}(\ell)$.

When a pseudo-statement of the form $\mathit{evaluateElement}(\ell) := s$**;** occurs in normative code below, it occurs at a point where the computation is complete and it specifies that the value of $\mathit{evaluateElement}(\ell)$ is s. *Evaluation of a collection literal element* ℓ in the given context to an object sequence $\mathit{evaluateElement}(\ell)$ is then specified as follows: ◇ ◇

Case ⟨Expression element⟩**.** In this case ℓ is an expression e; e is evaluated

to an object o and $evaluateElement(\ell) := [\![o]\!]$. □

Case ⟨Map element⟩**.** In this case ℓ is pair of expressions e_1`:`e_2; first e_1 is evaluated to an object o_1, then e_2 is evaluated to an object o_2, and $evaluateElement(\ell) := [\![o_1 : o_2]\!]$. □

Case ⟨Spread element⟩**.** The element ℓ is of the form '`...`e' or '`...?`e'. Evaluate e to an object o_{spread}.

1. When ℓ is '`...`e': If o_{spread} is the null object then a dynamic error occurs. Otherwise evaluation proceeds with step 2.

   When ℓ is '`...?`e': If o_{spread} is the null object then $evaluateElement(\ell) := [\![]\!]$. Otherwise evaluation proceeds with step 2.

2. Let T_{spread} be the dynamic type of o_{spread}. Let S be the static type of e. When S is not a top type (15.2), let S_{spread} be S. When S is a top type: If *target* is a list or a set then let S_{spread} be `Iterable<`**`dynamic`**`>`; otherwise (where *target* is a map), let S_{spread} be `Map<`**`dynamic, dynamic`**`>`.

   - When *target* is a list or a set and T_{spread} implements (11.2) `Iterable`, the following code is executed in the context where ℓ occurs, where `spread`, s, `v`, and `value` are fresh variables, and `Value` is a fresh type variable bound to the actual type argument of T_{target} at `Iterable` (11.2):

     ```
     S_spread spread = o_spread;
     var s = [;]
     for (var v in spread) {
       Value value = v;
       s := s + [[value]];
     }
     evaluateElement(ℓ) := s;
     ```

     The code makes use of a pseudo-variable s denoting an object sequence. We do not specify the type of s, this variable is only used to indicate the required semantic actions taken to gather the resulting object sequence. In the case where the implementation does not have a representation of s at all, the action may be to extend *target* immediately. A similar approach is used in subsequent cases.

   - When *target* is a map and T_{spread} implements `Map`, the following code is executed in the context where ℓ occurs, where `spread`, s, `v`, `key`, and `value` are fresh variables, and `Key` and `Value` are fresh type variables bound to the first respectively second actual type argument of T_{target} at `Map`:

```
S_spread spread = o_spread;
var s = [;]
for (var v in spread) {
  Key key = v.key;
  Value value = v.value;
  s := s + [key : value];
}
evaluateElement(ℓ) := s;
```

It is allowed for an implementation to delay the dynamic errors that occur if the given `key` does not have the type `Key`, or the given `value` does not have the type `Value`, but it cannot occur after the pair has been appended to s.

- Otherwise, a dynamic error occurs.

  This occurs when the target is an iterable respectively a map, and the spread is not, which is possible for a spread whose static type is **dynamic**.

This may not be the most efficient way to traverse the items in a collection, and implementations may of course use any other approach with the same observable behavior. However, in order to give implementations more room to optimize we also allow the following.

If o_{spread} is an object whose dynamic type implements (11.2) `List`, `Queue`, or `Set`, an implementation may choose to call `length` on the object. If o_{spread} is an object whose dynamic type implements `List`, an implementation may choose to call operator '`[]`' in order to access elements from the list. If it does so, it will only pass indices that are non-negative and less than the value returned by `length`.

This may allow for more efficient code for allocating the collection and accessing its parts. The given classes are expected to have an efficient and side-effect free implementation of `length` and operator '`[]`'. A Dart implementation may detect whether these options apply at compile time based on the static type of e, or at runtime based on the actual value. □

Case ⟨If element⟩**.** When ℓ is an ⟨*ifElement*⟩ of the form **if** `(`b`)` ℓ_1 or **if** `(`b`)` ℓ_1 **else** ℓ_2, the condition b is evaluated to a value o_b. If o_b is **true** then *evaluateElement*(ℓ) := *evaluateElement*(ℓ_1). If o_b is **false** and ℓ_2 is present then *evaluateElement*(ℓ) := *evaluateElement*(ℓ_2), and if ℓ_2 is not present then *evaluateElement*(ℓ) := [[]]. If o_b is neither **true** nor **false** then a dynamic error occurs. □

Case ⟨For element⟩**.** Let P be derived from ⟨*forLoopParts*⟩ and let ℓ be a ⟨*forElement*⟩ of the form **await? for** `(`P`)` ℓ_1, where '**await**?' indicates that **await** may be present or absent. To evaluate ℓ, the following code is executed in the context where ℓ occurs, where **await** is present if and only if it is present in ℓ:

```
var s = [;]
await? for (P) {
  s := s + evaluateElement(ℓ1);
}
evaluateElement(ℓ) := s;
```

□

17.9.3 List Literal Inference

This section specifies how a list literal *list* is traversed and an *inferred element type* for *list* is determined. We specify first how to infer the element type of a single element, then how to use that result to infer the element type of *list* as a whole. ◇

The context type P (17.9.5) for each element of *list* is obtained from the context type of *list*. If downwards inference constrains the type of *list* to `List<`P_e`>` or `Iterable<`P_e`>` for some P_e then P is P_e. Otherwise, P is ⍰ (17.9.5).

Let ℓ be a term derived from ⟨*element*⟩. Inference of the element type of ℓ with context type P proceeds as follows, where the context type for inference of an element type is always P, unless anything is said to the contrary:

Case ⟨Expression element⟩**.** In this case ℓ is an expression e. The inferred element type of ℓ is the inferred type of e in context P. □

Case ⟨Map element⟩**.** This cannot occur: it is a compile-time error when a leaf element of a list literal is a map element (17.9.4). □

Case ⟨Spread element⟩**.** Let e be the expression of ℓ. If ℓ is '`...`e', let S be the inferred type of e in context `Iterable<`P`>`. Otherwise (when ℓ is '`...?`e'), let S be the non-nullable type of the inferred type of e in context `Iterable<`P`>?`.

- If S implements `Iterable`, the inferred element type of ℓ is the type argument of S at `Iterable`.
- If S is **dynamic**, the inferred element type of ℓ is **dynamic**.
- If S is `Null` and the spread operator is '`...?`', the inferred element type of ℓ is `Null`.
- Otherwise, a compile-time error occurs.

□

Case ⟨If element⟩**.** In this case ℓ is of the form **if** `(`b`)` ℓ_1 or **if** `(`b`)` ℓ_1 **else** ℓ_2. The condition b is always inferred with a context type of `bool`.

Assume that '**else** ℓ_2' is not present. Then, if the inferred element type of ℓ_1 is S, the inferred element type of ℓ is S.

Otherwise, '**else** ℓ_2' is present. If the inferred element type of ℓ_1 is S_1 and the inferred element type of ℓ_2 is S_2, the inferred element type of ℓ is the least upper bound of S_1 and S_2. □

Case ⟨For element⟩**.** In this case ℓ is of the form **await? for** `(`P`)` ℓ_1 where P

is derived from ⟨*forLoopParts*⟩ and '**await**?' indicates that **await** may be present or absent.

The same compile-time errors occur for ℓ as the errors that would occur with the corresponding **for** statement **await**? **for** `(`P`) {}`, located in the same scope as ℓ. Moreover, the errors and type analysis of ℓ is performed as if it occurred in the body scope of said **for** statement. For instance, if P is of the form **var** v **in** e_1 then the variable v is in scope for ℓ.

Inference for the parts (such as the iterable expression of a for-in, or the ⟨*forInitializerStatement*⟩ of a for loop) is done as for the corresponding **for** statement, including **await** if and only if the element includes **await**. Then, if the inferred element type of ℓ_1 is S, the inferred element type of ℓ is S.

In other words, inference flows upwards from the body element. □

Finally, we define *type inference on a list literal* as a whole. Assume that ◇
list is derived from ⟨*listLiteral*⟩ and contains the elements $\ell_1, \ldots, \ell_n$, and the context type for *list* is P.

- If P is ? then the inferred element type for *list* is T, where T is the least upper bound of the inferred element types of $\ell_1, \ldots, \ell_n$.
- Otherwise, the inferred element type for *list* is T, where T is determined by downwards inference.

In both cases, the static type of *list* is `List<`T`>`.

17.9.4 Lists

A *list literal* denotes a list object, which is an integer indexed collection of ◇
objects. The grammar rule for ⟨*listLiteral*⟩ is specified elsewhere (17.9).

When a given list literal e has no type arguments, the type argument T is selected as specified elsewhere (17.9.3), and e is henceforth treated as (5) `<`T`>`e.

The static type of a list literal of the form `<`T`>`e is `List<`T`>` (17.9.3).

Let e be a list literal of the form `<`T`>[`$\ell_1, \ldots, \ell_m$`]`. It is a compile-time error if a leaf element of e is a ⟨*mapElement*⟩. It is a compile-time error if, for some $j \in 1..m$, ℓ_j does not have an element type, or the element type of ℓ_j may not be assigned to T.

A list may contain zero or more objects. The number of objects 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 dynamic error to attempt to access a list using an index that is not a member of its set of indices.

The system libraries define many members for the type `List`*, but we specify only the minimal set of requirements which are used by the language itself.*

If a list literal e begins with the reserved word **const** or e occurs in a constant context (17.3.2), it is a *constant list literal*, which is a constant expression (17.3) ◇
and therefore evaluated at compile time. Otherwise, it is a *run-time list literal* ◇
and it is evaluated at run time. Only run-time list literals can be mutated after

they are created. Attempting to mutate a constant list literal will result in a dynamic error.

Note that the collection literal elements of a constant list literal occur in a constant context (17.3.2), which means that **const** modifiers need not be specified explicitly.

It is a compile-time error if an element of a constant list literal is not constant. It is a compile-time error if the type argument of a constant list literal (no matter whether it is explicit or inferred) is not a constant type expression (17.3).

The binding of a formal type parameter of an enclosing class or function is not known at compile time, so we cannot use such type parameters inside constant expressions.

The value of a constant list literal **const**`? <`T`>[`ℓ_1, ..., ℓ_m`]` is an object o whose class implements the built-in class `List<`t`>` where t is the actual value of T (20.10.1), and whose contents is the object sequence o_1, ..., o_n obtained by evaluation of ℓ_1, ..., ℓ_m (17.9.2). The ith object of o (at index $i-1$) is then o_i.

Let **const**`? <`T_1`>[`$\ell_{11}, \ldots, \ell_{1m_1}$`]` and **const**`? <`$T_2$`>[`$\ell_{21}, \ldots, \ell_{2m_2}$`]` be two constant list literals. Let o_1 with contents $o_{11}, \ldots, o_{1n}$ and actual type argument t_1 respectively o_2 with contents $o_{21}, \ldots, o_{2n}$ and actual type argument t_2 be the result of evaluating them. Then `identical(`o_1`, `o_2`)` evaluates to **true** iff t_1 `==` t_2 and `identical(`o_{1i}`, `o_{2i}`)` evaluates to **true** for all $i \in 1..n$.

In other words, constant list literals are canonicalized. There is no need to consider canonicalization for other instances of type `List`, because such instances cannot be the result of evaluating a constant expression.

A run-time list literal `<`T`>[`ℓ_1, ..., ℓ_m`]` is evaluated as follows:

- The elements ℓ_1, ..., ℓ_m are evaluated (17.9.2), to an object sequence $[\![o_1, \ldots, o_n]\!]$.
- A fresh instance (10.7.1) o, of size n, whose class implements the built-in class `List<`t`>` is allocated, where t is the actual value of T (20.10.1).
- The operator '`[]=`' is invoked on o with first argument i and second argument $o_{i+1}, 0 \leq i < n$.
- The result of the evaluation is o.

The objects created by list literals do not override the '`==`' operator inherited from the `Object` class.

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 as follows (and may not be relied upon): 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}.

17.9.5 Set and Map Literal Disambiguation

Some terms like `{}` and `{ ...`*id*`}` are ambiguous: they may be either a set literal or a map literal. This ambiguity is eliminated in two steps. The first step

uses only the syntax and context type, and is described in this section. The second step uses expression types and is described next (17.9.6).

Let e be a ⟨*setOrMapLiteral*⟩ with leaf elements $\mathcal{L}$ and context type C. If C is ? then let S be undefined. Otherwise let S be the greatest closure of *futureOrBase*(C) (20.8).

A future version of this document will specify context types. The basic intuition is that a *context type* is the type declared for a receiving entity such as a formal parameter p or a declared variable v. That type will be the context type for an actual argument passed to p, respectively an initializing expression for v. In some situations the context has no constraints, e.g., when a variable is declared with **var** rather than a type annotation. This gives rise to an *unconstrained context type*, ?, which may also occur in a composite term, e.g., `List<?>`. The greatest closure of a context type C is approximately the least common supertype of all types obtainable by replacing ? by a type. ◇ ◇ ◇

The disambiguation step of this section is the first applicable entry in the following list:

- When e has type arguments $T_1, \ldots, T_k$, $k > 0$: If $k = 1$ then e is a set literal with static type `Set<`T_1`>`. If $k = 2$ then e is a map literal with static type `Map<`T_1`,` T_2`>`. Otherwise a compile-time error occurs.
- When S implements (11.2) `Iterable` but not `Map`, e is a set literal. When S implements `Map` but not `Iterable`, e is a map literal.
- When $\mathcal{L} \neq \emptyset$ (that is, e has leaf elements): If $\mathcal{L}$ contains a ⟨*mapElement*⟩ as well as an ⟨*expressionElement*⟩, a compile-time error occurs. Otherwise, if $\mathcal{L}$ contains an ⟨*expressionElement*⟩, e is a set literal. Otherwise $\mathcal{L}$ contains a ⟨*mapElement*⟩, and e is a map literal.
- When e is of the form `{}` and S is undefined, e is a map literal. *There is no deeper reason for this choice, but the fact that `{}` is a map by default was useful when set literals were introduced, because it would be a breaking change to make it a set.*
- Otherwise, e is still ambiguous. In this case e is non-empty, but contains only spreads wrapped zero or more times in ⟨*ifElement*⟩s or ⟨*forElement*⟩s. Disambiguation will then occur during inference (17.9.6).

When this step does not determine a static type, it will be determined by type inference (17.9.6).

If this process successfully disambiguates the literal then we say that e is *unambiguously a set* or *unambiguously a map*, as appropriate. ◇ ◇

17.9.6 Set and Map Literal Inference

This section specifies how a ⟨*setOrMapLiteral*⟩ e is traversed and an associated *inferred element type* and/or an associated *inferred key and value type pair* is determined. ◇ ◇

If e has an element type then it may be a set, and if it has a key and value type pair then it may be a map. However, if the literal e contains a spread element of type **dynamic**, that element cannot be used to determine whether e is a set or a map. The ambiguity is represented as having *both* an element type and a key and value type pair.

It is an error if the ambiguity is not resolved by some other elements, but if it is resolved then the dynamic spread element is required to evaluate to a suitable instance (implementing `Iterable` when e is a set, and implementing `Map` when e is a map), which means that it is a dynamic error if there is a mismatch. In other situations it is a compile-time error to have both an element type and a key and value type pair, because e must be both a set and a map. Here is an example:

```
dynamic x = <int, int>{};
Iterable l = [];
Map m = {};

void main() {
  var v1 = {...x};        // Compile-time error: ambiguous
  var v2 = {...x, ...l}; // A set, dynamic error when 'x' is evaluated
  var v3 = {...x, ...m}; // A map, no dynamic errors
  var v4 = {...l, ...m}; // Compile-time error: must be set and map
}
```

Let *collection* be a collection literal derived from ⟨*setOrMapLiteral*⟩. The inferred type of an ⟨*element*⟩ is an element type T, a pair of a key and value type (K, V), or both. It is computed relative to a context type P (17.9.5), which is determined as follows:

- If *collection* is unambiguously a set (17.9.5) then P is `Set<`P_e`>`, where P_e is determined by downwards inference, and may be ⍰ (17.9.5) if downwards inference does not constrain it.

  A future version of this document will specify inference, the notion of downwards inference, and constraining. The brief intuition is that inference selects values for type parameters in generic constructs where no type arguments have been provided, aiming at a type which matches a given context type; downwards inference does this by passing information from a given expression into its subexpressions, and upwards inference propagates information in the opposite direction. Constraints are expressed in terms of context types; being unconstrained means having ⍰ as the context type. Having a context type that *contains* one or more occurrences of ⍰ provides a partial constraint on the inferred type.

- If *collection* is unambiguously a map then P is `Map<`P_k`,` P_v`>` where P_k and P_v are determined by downwards inference, and may be ⍰ if the downwards context does not constrain one or both.

- Otherwise, *collection* is ambiguous, and the downwards context for the elements of *collection* is ⍰.

We say that a collection literal element *can be a set* if it has an element type; ◇
it *can be a map* if it has a key and value type pair; it *must be a set* if it can be ◇
a set and has and no key and value type pair; and it *must be a map* if can be a ◇
map and has no element type. ◇

Let ℓ be a term derived from ⟨*element*⟩. *Inference of the type of* ℓ with ◇
context type P then proceeds as follows:

Case ⟨Expression element⟩**.** In this case ℓ is an expression e. If P is ⍰, the inferred element type of ℓ is the inferred type of e in context ⍰. If P is `Set<`P_e`>`, the inferred element type of ℓ is the inferred type of e in context P_e. □

Case ⟨Map element⟩**.** In this case ℓ is a pair of expressions e_k`:` e_v. If P is ⍰, the inferred key and value type pair of ℓ is (K, V), where K and V is the inferred type of e_k respectively e_v, in context ⍰. If P is `Map<`P_k`,` P_v`>`, the inferred key and value type pair of ℓ is (K, V), where K is the inferred type of e_k in context P_k, and the V is the inferred type of e_v in context P_v. □

Case ⟨Spread element⟩**.** In this case ℓ is of the form '`...`e' or '`...?`e'. If P is ⍰ then let S be the inferred type of e in context ⍰. Then:

- If S implements `Iterable`, the inferred element type of ℓ is the type argument of S at `Iterable`.

  This is the result of constraint matching for X using the constraint S `<: Iterable<`X`>`. Note that when S implements a class like `Map` or `Iterable`, it cannot be a subtype of `Null` (11.2).

- If S implements `Map`, the inferred key and value type pair of ℓ is (K, V), where K is the first and V the second type argument of S at `Map`.

  This is the result of constraint matching for X and Y using the constraint S `<: Map<`X`,` Y`>`.

  Note that this case and the previous case can match on the same element simultaneously when S implements both `Iterable` and `Map`. The same situation arises several times below. In such cases we rely on other elements to disambiguate.

- If S is **dynamic** then the inferred element type of ℓ is **dynamic**, and the inferred key and value type pair of ℓ is (**dynamic**, **dynamic**).

  We produce both an element type and a key and value type pair here, and rely on other elements to disambiguate.

- If S is `Null` and the spread operator is '`...?`' then the inferred element type of ℓ is `Null`, and the inferred key and value type pair (`Null`, `Null`).

- Otherwise, a compile-time error occurs.

Otherwise, if P is `Set<`P_e`>` then let S be the inferred type of e in context `Iterable<`P_e`>`, and then:

- If S implements `Iterable`, the inferred element type of ℓ is the type argument of S at `Iterable`. This is the result of constraint matching for X using the constraint S <: `Iterable<`X`>`.
- If S is **dynamic**, the inferred element type of ℓ is **dynamic**.
- If S is `Null` and the spread operator is '`...?`', the inferred element type of ℓ is `Null`.
- Otherwise, a compile-time error occurs.

Otherwise, if P is `Map<`P_k`,` P_v`>` then let S be the inferred type of e in context P, and then:

- If S implements `Map`, the inferred key and value type pair of ℓ is (K, V), where K is the first and V the second type argument of S at `Map`. This is the result of constraint matching for X and Y using the constraint S <: `Map<`X`,` Y`>`.
- If S is **dynamic**, the inferred key and value type pair of ℓ is (**dynamic**, **dynamic**).
- If S is `Null` and the spread operator is '`...?`', the inferred key and value type pair (`Null`, `Null`).
- Otherwise, a compile-time error occurs.

□

Case ⟨If element⟩**.** In this case ℓ is of the form **if** `(`b`)` ℓ_1 or **if** `(`b`)` ℓ_1 **else** ℓ_2. The condition b is always inferred with a context type of `bool`.

Assume that '**else** ℓ_2' is not present. Then:

- If the inferred element type of ℓ_1 is S, the inferred element type of ℓ is S.
- If the inferred key and value type pair of ℓ_1 is (K, V), the inferred key and value type pair of ℓ is (K, V).

Otherwise, '**else** ℓ_2' is present. It is a compile error if ℓ_1 must be a set and ℓ_2 must be a map, or vice versa.

This means that one cannot spread a map on one branch and a set on the other. Since **dynamic** provides both an element type and a key and value type pair, a **dynamic** spread in either branch does not cause the error to occur.

Then:

- If the inferred element type of ℓ_1 is S_1 and the inferred element type of ℓ_2 is S_2, the inferred element type of ℓ is the least upper bound of S_1 and S_2.
- If the inferred key and value type pair of e_1 is (K_1, V_1) and the inferred key and value type pair of e_2 is (K_2, V_2), the inferred key and value type pair of ℓ is (K, V), where K is the least upper bound of K_1 and K_2, and and V is the least upper bound of V_1 and V_2.

□

Case ⟨For element⟩**.** In this case ℓ is of the form **await? for** `(`P`)` ℓ_1 where P is derived from ⟨*forLoopParts*⟩ and '**await**?' indicates that **await** may be present or absent.

The same compile-time errors occur for ℓ as the errors that would occur with the corresponding **for** statement **await? for** `(`P`) {}`, located in the same scope as ℓ. Moreover, the errors and type analysis of ℓ is performed as if it occurred in the body scope of said **for** statement.

For instance, if P is of the form **var** v **in** e_1 then the variable v is in scope for ℓ.

Inference for the parts (such as the iterable expression of a for-in, or the ⟨*forInitializerStatement*⟩ of a for loop) is done as for the corresponding **for** statement, including **await** if and only if the element includes **await**.

- If the inferred element type of ℓ_1 is S then the inferred element type of ℓ is S.
- If the inferred key and value type pair of e_1 is (K, V), the inferred key and value type pair of ℓ is (K, V).

In other words, inference flows upwards from the body element. □

Finally, we define *type inference on a set or map literal* as a whole. Assume ◇
that *collection* is derived from ⟨*setOrMapLiteral*⟩, and the context type for *collection* is P.

- If *collection* is unambiguously a set:
  - If P is ⍰ then the static type of *collection* is `Set<`T`>` where T is the least upper bound of the inferred element types of the elements.
  - Otherwise, the static type of *collection* is T where T is determined by downwards inference.

    Note that the inference will never produce a key and value type pair with the given context type.

  The static type of *collection* is then `Set<`T`>`.
- If *collection* is unambiguously a map where P is `Map<`P_k`,` P_v`>` or P is ⍰ and the inferred key and value type pairs are $(K_1, V_1), \ldots, (K_n, V_n)$:

  If P_k is ⍰ or P is ⍰, the static key type of *collection* is K where K is the least upper bound of $K_1, \ldots, K_n$. Otherwise the static key type of *collection* is K where K is determined by downwards inference.

  If P_v is ⍰ or P is ⍰, the static value type of *collection* is V where V is the least upper bound of $V_1, \ldots, V_n$. Otherwise the static value type of *collection* is V where V is determined by downwards inference.

  Note that inference will never produce a element type here given this downwards context.

  The static type of *collection* is then `Map<`K`,` V`>`.

- Otherwise, *collection* is still ambiguous, the downwards context for the elements of *collection* is ⍰, and the disambiguation is done using the immediate elements of *collection* as follows:

  - If all elements can be a set, and at least one element must be a set, then *collection* is a set literal with static type `Set<`T`>` where T is the least upper bound of the element types of the elements.
  - If all elements can be a map, and at least one element must be a map, then e is a map literal with static type `Map<`K`, `V`>` where K is the least upper bound of the key types of the elements and V is the least upper bound of the value types.
  - Otherwise, a compile-time error occurs. In this case the literal cannot be disambiguated.

This last error can occur if the literal *must* be both a set and a map. Here is an example:

```
var iterable = [1, 2];
var map = {1: 2};
var ambiguous = {...iterable, ...map}; // Compile-time error
```

Or, if there is nothing indicates that it is *either* a set or a map:

```
dynamic dyn;
var ambiguous = {...dyn}; // Compile-time error
```

17.9.7 Sets

A *set literal* denotes a set object. The grammar rule for ⟨*setOrMapLiteral*⟩ ◇ which covers set literals as well as map literals occurs elsewhere (17.9). A set literal consists of zero or more collection literal elements (17.9). A term derived from ⟨*setOrMapLiteral*⟩ may be a set literal or a map literal, and it is determined via a disambiguation step whether it is a set literal or a map literal (17.9.5, 17.9.6).

When a given set literal e has no type arguments, the type argument T is selected as specified elsewhere (17.9.6), and e is henceforth treated as (5) `<`T`>`e.

The static type of a set literal of the form `<`T`>`e is `Set<`T`>` (17.9.6).

Let e be a set literal of the form `<`T`>{`ℓ_1`, ..., `ℓ_m`}`. It is a compile-time error if a leaf element of e is a ⟨*mapElement*⟩. It is a compile-time error if, for some $j \in 1..m$, ℓ_j does not have an element type, or the element type of ℓ_j may not be assigned to T.

A set may contain zero or more objects. Sets have a method which can be used to insert objects; this will incur a dynamic error if the set is not modifiable. Otherwise, when inserting an object o_{new} into a set s, if an object o_{old} exists in s such that o_{old} `==` o_{new} evaluates to **true** then the insertion makes no changes

to s; if no such object exists, o_{new} is added to s; in both cases the insertion completes successfully.

A set is ordered: iteration over the elements of a set occurs in the order the elements were added to the set.

The system libraries define many members for the type `Set`, but we specify only the minimal set of requirements which are used by the language itself.

Note that an implementation may require consistent definitions of several members of a class implementing `Set` in order to work correctly. For instance, there may be a getter `hashCode` which is required to have a behavior which is in some sense consistent with operator '==='. Such constraints are documented in the system libraries.

If a set literal e begins with the reserved word **const** or e occurs in a constant context (17.3.2), it is a *constant set literal* which is a constant expression (17.3) ◇ and therefore evaluated at compile time. Otherwise, it is a *run-time set literal* ◇ and it is evaluated at run time. Only run-time set literals can be mutated after they are created. Attempting to mutate a constant set literal will result in a dynamic error.

Note that the element expressions of a constant set literal occur in a constant context (17.3.2), which means that **const** modifiers need not be specified explicitly.

It is a compile-time error if a collection literal element in a constant set literal is not a constant expression. It is a compile-time error if an element in a constant set literal does not have primitive equality (10.2.3). It is a compile-time error if two elements of a constant set literal are equal according to their '==' operator (17.27). It is a compile-time error if the type argument of a constant set literal (no matter whether it is explicit or inferred) is not a constant type expression (17.3).

The binding of a formal type parameter of an enclosing class or function is not known at compile time, so we cannot use such type parameters inside constant expressions.

The value of a constant set literal **const**`? <`T`>{`$\ell_1, \ldots, \ell_m$`}` is an object o whose class implements the built-in class `Set<`t`>` where t is the actual value of T (20.10.1), and whose contents is the set of objects in the object sequence $o_1, \ldots, o_n$ obtained by evaluation of $\ell_1, \ldots, \ell_m$ (17.9.2). The elements of o occur in the same order as the objects in said object sequence (which can be observed by iteration).

Let **const**`? <`T_1`>{`$\ell_{11}, \ldots, \ell_{1m_1}$`}` and **const**`? <`$T_2$`>{`$\ell_{21}, \ldots, \ell_{2m_2}$`}` be two constant set literals. Let o_1 with contents $o_{11}, \ldots, o_{1n}$ and actual type argument t_1 respectively o_2 with contents $o_{21}, \ldots, o_{2n}$ and actual type argument t_2 be the result of evaluating them. Then `identical(`o_1`, `o_2`)` evaluates to **true** iff t_1 `==` t_2 and `identical(`o_{1i}`, `o_{2i}`)` evaluates to **true** for all $i \in 1..n$.

In other words, constant set literals are canonicalized if they have the same type argument and the same values in the same order. Two constant set literals are never identical if they have a different number of elements. There is no need to consider canonicalization for other instances of type `Set`, because such instances cannot be the result of evaluating a constant expression.

A run-time set literal `<`T`>{`$\ell_1, \ldots, \ell_n$`}` is evaluated as follows:

- The elements $\ell_1, \ldots, \ell_m$ are evaluated (17.9.2), to an object sequence $[\![o_1, \ldots, o_n]\!]$.
- A fresh object (10.7.1) o implementing the built-in class `Set<t>` is created, where t is the actual value of T (20.10.1).
- For each object o_j in $o_1, \ldots, o_n$, in order, o_j is inserted into o. Note that this leaves o unchanged when o already contains and object o which is equal to o_j according to operator '=='.
- The result of the evaluation is o.

The objects created by set literals do not override the '==' operator inherited from the `Object` class.

17.9.8 Maps

A *map literal* denotes a map object, which is a mapping from keys to values. ◇ The grammar rule for ⟨*setOrMapLiteral*⟩ which covers both map literals and set literals occurs elsewhere (17.9). A map literal consists of zero or more collection literal elements (17.9). A term derived from ⟨*setOrMapLiteral*⟩ may be a set literal or a map literal, and it is determined via a disambiguation step whether it is a set literal or a map literal (17.9.5, 17.9.6).

When a given map literal e has no type arguments, the type arguments K and V are selected as specified elsewhere (17.9.6), and e is henceforth treated as (5) `<`K`,` V`>`e.

The static type of a map literal of the form `<`K`,` V`>`e is `Map<`K`,` V`>` (17.9.6).

Let e be a map literal of the form `<`K`,` V`>{`$\ell_1, \ldots, \ell_m$`}`. It is a compile-time error if a leaf element of e is an ⟨*expressionElement*⟩. It is a compile-time error if, for some $j \in 1..m$, ℓ_j does not have a key and value type pair; or the key and value type pair of ℓ_j is (K_j, V_j), and K_j may not be assigned to K or V_j may not be assigned to V.

A map object consists of zero or more map entries. Each entry has a *key* ◇ and a *value*, and we say that the map *binds* or *maps* the key to the value. A ◇ key and value pair is added to a map using operator '`[]=`', and the value for a ◇ given key is retrieved from a map using operator '`[]`'. The keys of a map are ◇ treated similarly to a set (17.9.7): When binding a key k_{new} to a value v in a map m (as in m`[`k_{new}`]=`v), if m already has a key k_{old} such that k_{old} `==` k_{new} evaluates to **true**, m will bind k_{old} to v; otherwise (when no such key k_{old} exists), a binding from k_{new} to v is added to m.

A map is ordered: iteration over the keys, values, or key/value pairs occurs in the order in which the keys were added to the set.

The system libraries support many operations on an instance whose type implements `Map`, but we specify only the minimal set of requirements which are used by the language itself.

Note that an implementation may require consistent definitions of several members of a class implementing `Map` in order to work correctly. For instance, there

may be a getter `hashCode` which is required to have a behavior which is in some sense consistent with operator '==='. Such constraints are documented in the system libraries.

If a map literal e begins with the reserved word **const**, or if e occurs in a constant context (17.3.2), it is a *constant map literal* which is a constant ◇ expression (17.3) and therefore evaluated at compile time. Otherwise, it is a *run-time map literal* and it is evaluated at run time. Only run-time map literals ◇ can be mutated after they are created. Attempting to mutate a constant map literal will result in a dynamic error.

Note that the key and value expressions of a constant map literal occur in a constant context (17.3.2), which means that **const** modifiers need not be specified explicitly.

It is a compile-time error if a collection literal element in a constant map literal is not constant. It is a compile-time error if a key in a constant map literal does not have primitive equality (10.2.3). It is a compile-time error if two keys of a constant map literal are equal according to their '==' operator (17.27). It is a compile-time error if a type argument of a constant map literal (no matter whether it is explicit or inferred) is not a constant type expression (17.3).

The binding of a formal type parameter of an enclosing class or function is not known at compile time, so we cannot use such type parameters inside constant expressions.

The value of a constant map literal **const**`?` `<`T_1`,` T_2`>{`$\ell_1, \ldots, \ell_m$`}` is an object o whose class implements the built-in class `Map<`t_1`,` t_2`>`, where t_1 and t_2 is the actual value of T_1 respectively T_2 (20.10.1). The key and value pairs of o is the pairs of the object sequence $k_1 : v_1, \ldots, k_n : v_n$ obtained by evaluation of $\ell_1, \ldots, \ell_m$ (17.9.2), in that order.

Let **const**`?` `<`U_1`,` V_1`>{`$\ell_1, \ldots, \ell_{m_1}$`}` and **const**`?` `<`U_2`,` V_2`>{`$\ell_1, \ldots, \ell_{m_2}$`}` be two constant map literals. Let o_1 with contents $k_{11} : v_{11}, \ldots, k_{1n} : v_{1n}$ and actual type arguments u_1, v_1 respectively o_2 with contents $k_{21} : v_{21}, \ldots, k_{2n} : v_{2n}$ and actual type argument u_2, v_2 be the result of evaluating them. Then `identical(`o_1`,` o_2`)` evaluates to **true** iff u_1 `==` u_2, v_1 `==` v_2, `identical(`k_{1i}`,` k_{2i}`)`, and `identical(`v_{1i}`,` v_{2i}`)` for all $i \in 1..n$.

In other words, constant map literals are canonicalized. There is no need to consider canonicalization for other instances of type `Map`, because such instances cannot be the result of evaluating a constant expression.

A run-time map literal `<`T_1, T_2`>{`$\ell_1, \ldots, \ell_m$`}` is evaluated as follows:

- The elements $\ell_1, \ldots, \ell_m$ are evaluated (17.9.2), to an object sequence $[\![k_1 : v_1, \ldots, k_n : v_n]\!]$.
- A fresh instance (10.7.1) o whose class implements the built-in class `Map<`t_1`,` t_2`>` is allocated, where t_1 and t_2 are the actual values of T_1 respectively T_2 (20.10.1).
- The operator '`[]=`' is invoked on o with first argument k_i and second argument v_i, for each $i \in 1..n$, in that order.

- The result of the evaluation is o.

The objects created by map literals do not override the '`==`' operator inherited from the `Object` class.

17.10 Throw

The *throw expression* is used to throw an exception. ◇

⟨*throwExpression*⟩ ::= **throw** ⟨*expression*⟩

⟨*throwExpressionWithoutCascade*⟩ ::= **throw** ⟨*expressionWithoutCascade*⟩

Evaluation of a throw expression of the form **throw** e**;** proceeds as follows: The expression e is evaluated to an object v (17.1).

There is no requirement that the expression e must evaluate to any special kind of object.

If v is the null object (17.4), then a `NullThrownError` is thrown. Otherwise let t be a stack trace corresponding to the current execution state, and the **throw** statement throws with v as exception object and t as stack trace (17.1).

If v is an instance of class `Error` or a subclass thereof, and it is the first time that `Error` object is thrown, the stack trace t is stored on v so that it will be returned by the `stackTrace` getter inherited from `Error`.

If the same `Error` object is thrown more than once, its `stackTrace` getter will return the stack trace from the *first* time it was thrown.

The static type of a throw expression is $\bot$.

17.11 Function Expressions

A *function literal* is an anonymous declaration and an expression that encapsulates an executable unit of code. ◇

⟨*functionExpression*⟩ ::= ⟨*formalParameterPart*⟩ ⟨*functionExpressionBody*⟩

⟨*functionExpressionBody*⟩ ::= **async**? '**=>**' ⟨*expression*⟩
| (**async** '*****'? | **sync** '*****')? ⟨*block*⟩

The grammar does not allow a function literal to declare a return type, but it is possible for a function literal to have a *declared return type*, because it can be obtained by means of type inference. Such a return type is included when we refer to the declared return type of a function. ◇

Type inference will be specified in a future version of this document. Currently we consider type inference to be a phase that has completed, and this document specifies the meaning of Dart programs where inferred types have already been added.

We say that S is the *future type of a type* T in the following cases, using the first applicable case: ◇

- T implements S (11.2), and there is a U such that S is `Future<`U`>`.
- T is S bounded (17.15.3), and there is a U such that S is `FutureOr<`U`>`, `Future<`U`>?`, or `FutureOr<`U`>?`.

When none of these cases are applicable, we say that T does not have a future type.

Note that if T has a future type F then $T <: F$, and F is always of the form G`<...>` or G`<...>?`, where G is `Future` or `FutureOr`.

We define the auxiliary function *flatten*(T) as follows, using the first applicable case: ◇

- If T is S`?` for some S then *flatten*(T) $\triangleq$ *flatten*(S)`?`.
- If T is X `&` S for some type variable X and type S then
  - if S has future type U then *flatten*(T) $\triangleq$ *flatten*(U).
  - otherwise, *flatten*(T) $\triangleq$ *flatten*(X).
- If T has future type `Future<`S`>` or `FutureOr<`S`>` then *flatten*(T) $\triangleq$ S.
- If T has future type `Future<`S`>?` or `FutureOr<`S`>?` then *flatten*(T) $\triangleq$ S`?`.
- Otherwise, *flatten*(T) $\triangleq$ T.

*This definition guarantees that for any type T, $T <:$ `FutureOr<`*flatten*(T)`>`.*

Case ⟨Positional, arrow⟩**.** The static type of a function literal of the form
`<`X_1 **extends** B_1, ..., X_s **extends** B_s`>`
`(`T_1 a_1, ..., T_n a_n, `[`T_{n+1} $x_{n+1} = d_1$, ..., T_{n+k} $x_{n+k} = d_k$`]) =>` e
is
T_0 **Function**`<`$X_1 \triangleleft B_1$, ..., $X_s \triangleleft B_s$`>(`T_1, ..., T_n`, [`T_{n+1}, ..., T_{n+k}`])`,
where T_0 is the static type of e. □

Case ⟨Positional, arrow, future⟩**.** The static type of a function literal of the form
`<`X_1 **extends** B_1, ..., X_s **extends** B_s`>`
`(`T_1 a_1, ..., T_n a_n, `[`T_{n+1} $x_{n+1} = d_1$, ..., T_{n+k} $x_{n+k} = d_k$`]) async =>` e
is
`Future<`*flatten*(T_0)`>`
Function`<`$X_1 \triangleleft B_1$, ..., $X_s \triangleleft B_s$`>(`T_1, ..., T_n`, [`T_{n+1}, ..., T_{n+k}`])`,
where T_0 is the static type of e. □

Case ⟨Named, arrow⟩**.** The static type of a function literal of the form
`<`X_1 **extends** B_1, ..., X_s **extends** B_s`>`
`(`T_1 a_1, ..., T_n a_n, `{`T_{n+1} $x_{n+1} = d_1$, ..., T_{n+k} $x_{n+k} = d_k$`}) =>` e
is
T_0 **Function**`<`$X_1 \triangleleft B_1$, ..., $X_s \triangleleft B_s$`>(`T_1, ..., T_n`, {`T_{n+1} x_{n+1}, ..., T_{n+k} x_{n+k}`})`,
where T_0 is the static type of e. □

Case ⟨Named, arrow, future⟩**.** The static type of a function literal of the

form
`<`X_1 **extends** B_1, ..., X_s **extends** B_s`>`
`(`T_1 a_1, ..., T_n a_n, `{`T_{n+1} $x_{n+1} = d_1$, ..., T_{n+k} $x_{n+k} = d_k$`}) async =>` e
is
`Future<`*flatten*(T_0)`>`
Function`<`X_1 `◁` B_1, ..., X_s `◁` B_s`>(`T_1, ..., T_n`, {`T_{n+1} x_{n+1}, ..., T_{n+k} x_{n+k}`})`,
where T_0 is the static type of e. □

Case ⟨Positional, block⟩**.** The static type of a function literal of the form
`<`X_1 **extends** B_1, ..., X_S **extends** B_S`>`
`(`T_1 a_1, ..., T_n a_n, `[`T_{n+1} $x_{n+1} = d_1$, ..., T_{n+k} $x_{n+k} = d_k$`]) { `s `}`
is
dynamic
Function`<`$X_1 \triangleleft B_1$, ..., $X_s \triangleleft B_s$`>(`T_1, ..., T_n`, [`T_{n+1}, ..., T_{n+k}`])` □

Case ⟨Positional, block, future⟩**.** The static type of a function literal of the form
`<`X_1 **extends** B_1, ..., X_s **extends** B_s`>`
`(`T_1 a_1, ..., T_n a_n, `[`T_{n+1} $x_{n+1} = d_1$, ..., T_{n+k} $x_{n+k} = d_k$`]) async { `s `}`
is
`Future`
Function`<`$X_1 \triangleleft B_1$, ..., $X_s \triangleleft B_s$`>(`T_1, ..., T_n`, [`T_{n+1}, ..., T_{n+k}`]).` □

Case ⟨Positional, block, stream⟩**.** The static type of a function literal of the form
`<`X_1 **extends** B_1, ..., X_s **extends** B_s`>`
`(`T_1 a_1, ..., T_n a_n, `[`T_{n+1} $x_{n+1} = d_1$, ..., T_{n+k} $x_{n+k} = d_k$`]) async* { `s `}`
is
`Stream`
Function`<`$X_1 \triangleleft B_1$, ..., $X_s \triangleleft B_s$`>(`T_1, ..., T_n`, [`T_{n+1}, ..., T_{n+k}`]).` □

Case ⟨Positional, block, iterable⟩**.** The static type of a function literal of the form
`<`X_1 **extends** B_1, ..., X_s **extends** B_s`>`
`(`T_1 a_1, ..., T_n a_n, `[`T_{n+1} $x_{n+1} = d_1$, ..., T_{n+k} $x_{n+k} = d_k$`]) sync* { `s `}`
is
`Iterable`
Function`<`$X_1 \triangleleft B_1$, ..., $X_s \triangleleft B_s$`>(`T_1, ..., T_n`, [`T_{n+1}, ..., T_{n+k}`]).` □

Case ⟨Named, block⟩**.** The static type of a function literal of the form
`<`X_1 **extends** B_1, ..., X_s **extends** B_s`>`
`(`T_1 a_1, ..., T_n a_n, `[`T_{n+1} $x_{n+1} = d_1$, ..., T_{n+k} $x_{n+k} = d_k$`]) { `s `}`
is
dynamic
Function`<`$X_1 \triangleleft B_1$, ..., $X_s \triangleleft B_s$`>(`T_1, ..., T_n`, [`T_{n+1}, ..., T_{n+k}`]).` □

Case ⟨Named, block, future⟩**.** The static type of a function literal of the form
`<`X_1 **extends** B_1, ..., X_s **extends** B_s`>`
`(`T_1 a_1, ..., T_n a_n, `{`T_{n+1} $x_{n+1} = d_1$, ..., T_{n+k} $x_{n+k} = d_k$`}) async { `s `}`
is

`Future`
Function`<`$X_1 \lhd B_1, \ldots, X_s \lhd B_s$`>(`$T_1, \ldots, T_n$`, {`$T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}$`}).`
□

Case ⟨Named, block, stream⟩**.** The static type of a function literal of the form
`<`X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s`>`
`(`$T_1\ a_1, \ldots, T_n\ a_n$`, {`$T_{n+1}\ x_{n+1} = d_1, \ldots, T_{n+k}\ x_{n+k} = d_k$`}) async* { `$s$` }`
is
`Stream`
Function`<`$X_1 \lhd B_1, \ldots, X_s \lhd B_s$`>(`$T_1, \ldots, T_n$`, {`$T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}$`}).`
□

Case ⟨Named, block, iterable⟩**.** The static type of a function literal of the form
`<`X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s`>`
`(`$T_1\ a_1, \ldots, T_n\ a_n$`, {`$T_{n+1}\ x_{n+1} = d_1, \ldots, T_{n+k}\ x_{n+k} = d_k$`}) sync* { `$s$` }`
is
`Iterable`
Function`<`$X_1 \lhd B_1, \ldots, X_s \lhd B_s$`>(`$T_1, \ldots, T_n$`, {`$T_{n+1}\ x_{n+1}, \ldots, T_{n+k}\ x_{n+k}$`}).`
□

In all of the above cases, the type argument lists are omitted when $m = 0$, and whenever T_i is not specified, $i \in 1..n + k$, it is considered to have been specified as **dynamic**.

Evaluation of a function literal yields a function object o.

The run-time type of o is specified based on the static type T of the function literal and the binding of type variables occurring in T at the occasion where the evaluation occurred (9.3).

17.12 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.

It is a compile-time error if **this** appears, implicitly or explicitly, in a top-level function or variable initializer, in a factory constructor, or in a static method or variable initializer, or in the initializer of an instance variable.

17.13 Instance Creation

Instance creation expressions generally produce instances and invoke constructors to initialize them.

The exception is that a factory constructor invocation works like a regular function call. It may of course evaluate an instance creation expression and thus produce

a fresh instance, but no fresh instances are created as a direct consequence of the factory constructor invocation.

It is a compile-time error if the type T in an instance creation expression of one of the forms
new $T.id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$,
new $T(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$,
const $T.id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$,
const $T(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$
is an enumerated type (14).

17.13.1 New

The *new expression* invokes a constructor (10.7). ◇

⟨*newExpression*⟩ ::= **new** ⟨*constructorDesignation*⟩ ⟨*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 a parameterized type accessible in the current scope, or if T is a parameterized type which is not a class. For instance, **new** `F<int>()` is an error if `F` is a type alias that does not denote a class.

If T is a parameterized type (20.10) S<$U_1, \ldots, U_m$>, let R be the generic class S, and let X_1 **extends** $B_1, \ldots, X_p$ **extends** B_p be the formal type parameters of S. If T is not a parameterized type, let R be T.

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

Let q be the above-mentioned constructor named $R.id$ or R.

It is a compile-time error if R is abstract and q is not a factory constructor. It is a compile-time error if R is a non-generic class and T is a parameterized type. It is a compile-time error if R is a generic class and T is not a parameterized type. It is a compile-time error if R is a generic class, T is a parameterized type, and $m \neq p$. That is, the number of type arguments is incorrect. It is a compile-time error if R is a generic class, T is a parameterized type, and T is not regular-bounded (15.2).

If q is a redirecting factory constructor, it is a compile-time error if q in some number of redirecting factory redirections redirects to itself. It is possible and allowed for a redirecting factory q' to enter an infinite loop, e.g., because q' redirects to a non-redirecting factory constructor q'' whose body uses q' in an instance creation expression. Only loops that consist exclusively of redirecting factory redirections are detected at compile time.

Let S_i be the static type of the formal parameter of the constructor $R.id$ (respectively R) corresponding to the actual argument a_i, $i \in 1..n+k$. It is a compile-time error if the static type of $a_i, i \in 1..n+k$ is not assignable to $[U_1/X_1, \ldots, U_m/X_m]S_i$. The non-generic case is covered with $m = 0$.

The static type of e is T.

Evaluation of e proceeds as follows:

First, the argument part

<$U_1, \ldots,\ U_m$>($a_1, \ldots,\ a_n,\ x_{n+1}$: $a_{n+1}, \ldots,\ x_{n+k}$: a_{n+k})

is evaluated, yielding the evaluated actual argument part

<$u_1, \ldots,\ u_m$>($o_1, \ldots,\ o_n,\ x_{n+1}$: $o_{n+1},\ \ldots,\ x_{n+k}$: o_{n+k}).

Note that the non-generic case is covered by letting $m = 0$. If for any $j \in 1..n+k$ the run-time type of o_j is not a subtype of $[u_1/X_1, \ldots, u_m/X_m]S_j$, a dynamic type error occurs.

Case ⟨Non-loaded deferred constructors⟩**.** If T is a deferred type with prefix p, then if p has not been successfully loaded, a dynamic error occurs. □

Case ⟨Generative constructors⟩**.** When q is a generative constructor (10.7.1) evaluation proceeds to allocate a fresh instance (10.7.1), i, of class T. Then q is executed to initialize i with respect to the bindings that resulted from the evaluation of the argument list, and, if R is a generic class, with its type parameters bound to $u_1, \ldots, u_m$.

If execution of q completes normally (18.0.1), e evaluates to i. Otherwise execution of q throws an exception object x and stack trace t, and then evaluation of e also throws exception object x and stack trace t (17.1). □

Case ⟨Redirecting factory constructors⟩**.** When q is a redirecting factory constructor (10.7.2) of the form **const**? T(`(`$p_1, \ldots,\ p_{n+k}$`)` `=` c`;` or of the form **const**? $T.id$(`(`$p_1, \ldots,\ p_{n+k}$`)` `=` c`;` where **const**? indicates that **const** may be present or absent, the remaining evaluation of e is equivalent to evaluating **new** c(`(`$v_1, \ldots,\ v_n,\ x_{n+1}$: $v_{n+1}, \ldots,\ x_{n+k}$: v_{n+k}`)`) in an environment where v_j is a fresh variable bound to o_j for $j \in 1..n+k$, and X_j is bound to u_j for $j \in 1..m$. We need access to the type variables because c may contain them. □

Case ⟨Non-redirecting factory constructors⟩**.** When q is a non-redirecting factory constructor, the body of q is executed with respect to the bindings that resulted from the evaluation of the argument list, and with the type parameters, if any, of q bound to the actual type arguments $u_1, \ldots, u_m$. If this execution returns an object (18.0.1) then e evaluates to the returned object. Otherwise, if the execution completes normally or returns with no object, then e evaluates to the null object (17.4). Otherwise the execution throws an exception x and stack trace t, and then evaluation of e also throws x and t (17.1).

A factory constructor can be declared in an abstract class and used safely, as it will either produce a valid instance or throw. □

17.13.2 Const

A *constant object expression* invokes a constant constructor (10.7.3). ◇

⟨*constObjectExpression*⟩ ::= **const** ⟨*constructorDesignation*⟩ ⟨*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 a parameterized type accessible in the current scope, or if T is a parameterized type which is not a class. It is a compile-time error if T is a deferred type (20.1). In particular, T must not be a type variable.

It is a compile-time error if a_i is not a constant expression for some $i \in 1..n+k$.

If T is a parameterized type (20.10) $S<U_1, \ldots, U_m>$, let R be the generic class S, and let X_1 **extends** $B_1, \ldots, X_p$ **extends** B_p be the formal type parameters of S. If T is not a parameterized type, let R be T.

If T is a parameterized type, it is a compile-time error if U_j is not a constant type expression for any $j \in 1..m$.

- If e is of the form **const** $T.id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ it is a compile-time error if $R.id$ is not the name of a constant constructor declared by R, or id is not accessible.
- If e is 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 R is not the name of a constant constructor declared by R.

Let q be the above-mentioned constant constructor named $R.id$ or R.

It is a compile-time error if R is abstract and q is not a factory constructor. It is a compile-time error if R is a non-generic class and T is a parameterized type. It is a compile-time error if R is a generic class and T is not a parameterized type. It is a compile-time error if R is a generic class, T is a parameterized type, and $m \neq p$. That is, the number of type arguments is incorrect. It is a compile-time error if R is a generic class, T is a parameterized type, and T is not regular-bounded (15.2).

Let S_i be the static type of the formal parameter of the constructor $R.id$ (respectively R) corresponding to the actual argument a_i, $i \in 1..n+k$. It is a compile-time error if the static type of $a_i, i \in 1..n+k$ is not assignable to $[U_1/X_1, \ldots, U_m/X_m]S_i$. The non-generic case is covered with $m = 0$.

The static type of e is T.

Evaluation of e proceeds as follows:

If e is of the form **const** $T.id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$ let i be the value of the expression e':
new $T.id(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$.

Let o be the result of an evaluation of e', at some point in time of some execution of the program in the library L where e occurs. The result of an evaluation of e' in L at some other time and/or in some other execution will yield a result o', such that o' would be replaced by o by canonicalization as described below. This means that the value is well-defined.

If e is of the form **const** $T(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$,

let i be the value of **new** $T(a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k})$. Which is well-defined for the same reason.

- If during execution of the program, a constant object expression has already evaluated to an instance j of class R with type arguments $U_i, 1 \leq i \leq m$, then:
  - For each instance variable f of i, let v_{if} be the value of the instance variable f in i, and let v_{jf} be the value of the instance variable f in j. If `identical(`v_{if}`,` v_{jf}`)` for all instance variables 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 type arguments and identical instance variables. 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.

It is a compile-time error if evaluation of a constant object results in an uncaught exception being thrown.

To see how such situations might arise, consider the following examples:

```
class A {
  final x;
  const A(p): x = p * 10;
}

class IntPair {
  const IntPair(this.x, this.y);
  final int x;
  final int y;
  operator *(v) => new IntPair(x*v, y*v);
}

const a1 = const A(true); // compile-time error
const a2 = const A(5); // legal
const a3 = const A(const IntPair(1,2)); // compile-time error
```

Due to the rules governing constant constructors, evaluating the constructor `A()` with the argument `"x"` or the argument **const** `IntPair(1, 2)` would cause it to throw an exception, resulting in a compile-time error. In the latter case, the error is caused by the fact that **operator** `*` can only be used with a few "well-known" types, which is required in order to avoid running arbitrary code during the evaluation of constant expressions.

17.14 Spawning an Isolate

Spawning an isolate is accomplished via what is syntactically an ordinary method call, invoking one of the static methods `spawnUri` or `spawn` defined in the `Isolate` class in the library `dart:isolate`. However, such calls have the semantic effect of creating a new isolate with its own memory and thread of control.

An isolate's memory is finite, as is the space available to its thread's call stack. It is possible for a running isolate to exhaust its memory or stack, resulting in a dynamic error that cannot be effectively caught, which will force the isolate to be suspended.

As discussed in section 7, the handling of a suspended isolate is the responsibility of the runtime.

17.15 Function Invocation

Function invocation occurs in the following cases: when a function expression (17.11) is invoked (17.15.5), when a method (17.21), getter (17.19, 17.22) or setter (17.23) is invoked, or when a constructor is invoked (either via instance creation (17.13), constructor redirection (10.7.1), 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** (17.12) is bound. Once f has been determined, formal type parameters of f are bound to the corresponding actual type arguments, and the formal parameters of f are bound to corresponding actual arguments. When the body of f is executed it will be executed with the aforementioned bindings.

Executing a body of the form `=>` e is equivalent to executing a body of the form `{ return` e`; }`. Execution a body of the form **async** `=>` e is equivalent to executing a body of the form **async** `{ return` e`; }`.

If f is synchronous and is not a generator (9) then execution of the body of f begins immediately. If the execution of the body of f returns an object v (18.0.1), the invocation evaluates to v. If the execution completes normally or it returns without an object, the invocation evaluates to the null object (17.4). If the execution throws an exception object and stack trace, the invocation throws the same exception object and stack trace (17.1).

A complete function body can never break or continue (18.0.1) because a **break** or **continue** statement must always occur inside the statement that is the target of the **break** or **continue**. This means that a function body can only either complete normally, throw, or return. Completing normally or returning without an object is treated the same as returning with the null object (17.4), so the result of executing a function body can always be used as the result of evaluating an expression, either by evaluating to an object, or by the evaluation throwing.

If f is marked **sync*** (9), then a fresh instance (10.7.1) i implementing `Iterable<`U`>` is immediately returned, where U is determined as follows: Let T be the actual return type of f (20.10.1). If T is `Iterable<`S`>` for some type S, then U is S, otherwise U is `Object`.

A Dart implementation will need to provide a specific implementation of `Iterable` that will be returned by **sync*** methods. A typical strategy would be to produce an instance of a subclass of class `IterableBase` defined in `dart:core`. The only method that needs to be added by the Dart implementation in that case is `iterator`.

The iterable implementation must comply with the contract of `Iterable` and should not take any steps identified as exceptionally efficient in that contract.

The contract explicitly mentions a number of situations where certain iterables could be more efficient than normal. For example, by precomputing their length. Normal iterables must iterate over their elements to determine their length. This is certainly true in the case of a synchronous generator, where each element is computed by a function. It would not be acceptable to pre-compute the results of the generator and cache them, for example.

When iteration over the iterable is started, by getting an iterator j from the iterable and calling `moveNext()`, execution of the body of f will begin. When execution of the body of f completes (18.0.1),

- If it returns without an object or it completes normally (18.0.1), j is positioned after its last element, so that its current value is the null object (17.4) and the current call to `moveNext()` on j returns false, as must all further calls.
- If it throws an exception object e and stack trace t then the current value of j is the null object (17.4) and the current call to `moveNext()` throws e and t as well. Further calls to `moveNext()` must return false.

Each iterator starts a separate computation. If the **sync*** function is impure, the sequence of objects yielded by each iterator may differ.

One can derive more than one iterator from a given iterable. Note that operations on the iterable itself can create distinct iterators. An example would be `length`. It is conceivable that different iterators might yield sequences of different length. The same care needs to be taken when writing **sync*** functions as when writing an `Iterator` class. In particular, it should handle multiple simultaneous iterators gracefully. If the iterator depends on external state that might change, it should check that the state is still valid after every yield (and maybe throw a `ConcurrentModificationError` if it isn't).

Each iterator runs with its own shallow copies of all local variables; in particular, each iterator has the same initial arguments, even if their bindings are modified by the function. Two executions of an iterator interact only via state outside the function.

If f is marked **async** (9), then a fresh instance (10.7.1) o is associated with the invocation, where the dynamic type of o implements `Future<`$flatten(T)$`>`, and T is the actual return type of f (20.10.1). Then the body of f is executed until it either suspends or completes, at which point o is returned. The body of f may suspend during the evaluation of an **await** expression or execution of an asynchronous **for** loop. The future o is completed when execution of the body of f completes (18.0.1). If execution of the body returns an object, o is completed

with that object. If it completes normally or returns without an object, o is completed with the null object (17.4), and if it throws an exception e and stack trace t, o is completed with the error e and stack trace t. If execution of the body throws before the body suspends the first time, completion of o happens at some future time after the invocation has returned. *The caller needs time to set up error handling for the returned future, so the future is not completed with an error* before *it has been returned.*

If f is marked **async*** (9), then a fresh instance (10.7.1) s implementing `Stream<`U`>` is immediately returned, where U is determined as follows: Let T be the actual return type of f (20.10.1). If T is `Stream<`S`>` for some type S, then U is S, otherwise U is `Object`. When s is listened to, execution of the body of f will begin. When execution of the body of f completes:

- If it completes normally or returns without an object (18.0.1), then if s has been canceled then its cancellation future is completed with the null object (17.4).
- If it throws an exception object e and stack trace t:
  - If s has been canceled then its cancellation future is completed with error e and stack trace t.
  - otherwise the error e and stack trace t are emitted by s.
- s is closed.

The body of an asynchronous generator function cannot break, continue or return with an object (18.0.1). The first two are only allowed in contexts that will handle the break or continue, and return statements with an expression are not allowed in generator functions.

When an asynchronous generator's stream has been canceled, cleanup will occur in the ***finally*** *clauses (18.11) inside the generator. We choose to direct any exceptions that occur at this time to the cancellation future rather than have them be lost.*

17.15.1 Actual Argument Lists

Actual argument lists have the following syntax:

⟨*arguments*⟩ ::= '`(`' (⟨*argumentList*⟩ '`,`'?)? '`)`'

⟨*argumentList*⟩ ::= ⟨*namedArgument*⟩ ('`,`' ⟨*namedArgument*⟩)*
  | ⟨*expressionList*⟩ ('`,`' ⟨*namedArgument*⟩)*

⟨*namedArgument*⟩ ::= ⟨*label*⟩ ⟨*expression*⟩

Argument lists allow an optional trailing comma after the last argument ('`,`'?). An argument list with such a trailing comma is equivalent in all ways to the same parameter list without the trailing comma. All argument lists in this

specification are shown without a trailing comma, but the rules and semantics apply equally to the corresponding argument list with a trailing comma.

Let L be an argument list of the form $(e_1 \ldots,\ e_m,\ y_{m+1}\texttt{:}\ \ e_{m+1} \ldots,\ y_{m+p}\texttt{:}\ e_{m+p})$ and assume that the static type of e_i is S_i, $i \in 1..m+p$. The *static argument list type* of L is then $(S_1 \ldots,\ S_m,\ S_{m+1}\ y_{m+1} \ldots,\ S_{m+p}\ y_{m+p})$. ◇

Let Ss be the static argument list type
$(S_1 \ldots,\ S_m,\ S_{m+1}\ y_{m+1} \ldots,\ S_{m+p}\ y_{m+p})$
and let Ps be the formal parameter list
$(T_1\ x_1 \ldots,\ T_n\ x_n,\ \texttt{[}T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k\texttt{]})$
where each parameter may be marked **covariant** (not shown, but allowed).

We say that Ss is a *subtype match* for Ps iff $p = 0$, $n \leq m \leq n + k$, and S_i is a subtype of T_i for all $i \in 1..m$. We say that Ss is an *assignable match* for Ps iff $p = 0$, $n \leq m \leq n + k$, and S_i is assignable to T_i for all $i \in 1..m$. ◇ ◇

Let Ss be the static argument list type
$(S_1 \ldots,\ S_m,\ S_{m+1}\ y_{m+1} \ldots,\ S_{m+p}\ y_{m+p})$
and let Ps be the formal parameter list
$(T_1\ x_1 \ldots,\ T_n\ x_n,\ \texttt{\{}T_{n+1}\ x_{n+1} = d_1, \ldots,\ T_{n+k}\ x_{n+k} = d_k\texttt{\}})$
where each parameter may be marked **covariant** (not shown, but allowed).

We say that Ss is a *subtype match* for Ps iff $m = n$, $\{y_{m+1} \ldots,\ y_{m+p}\} \subseteq \{x_{n+1} \ldots,\ x_{n+k}\}$, S_i is a subtype of T_i for all $i \in 1..m$, and S_i is a subtype of T_j whenever $y_i = x_j$ and $j \in n+1..n+k$, for all $i \in m+1..m+p$. We say that Ss is an *assignable match* for Ps iff $m = n$, $\{y_{m+1} \ldots,\ y_{m+p}\} \subseteq \{x_{n+1} \ldots,\ x_{n+k}\}$, S_i is assignable to T_i for all $i \in 1..m$, and S_i is assignable to T_j whenever $y_i = x_j$ and $j \in n + 1..n + k$, for all $i \in m + 1..m + p$. ◇ ◇

In short, an actual argument list is a match for a formal parameter list whenever the former can safely be passed to the latter.

17.15.2 Actual Argument List Evaluation

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

When parsing an argument list, an ambiguity may arise because the same source code could be one generic function invocation, and it could be two or more relational expressions and/or shift expressions. In this situation, the expression is always parsed as a generic function invocation.

An example is `f(a<B, C>(d))`, which may be an invocation of `f` passing two actual arguments of type `bool`, or an invocation of `f` passing the result returned by an invocation of the generic function `a`. Note that the ambiguity can be eliminated by omitting the parentheses around the expression d, or adding parentheses around one of the relational expressions.

When the intention is to pass several relational or shift expressions as actual arguments and there is an ambiguity, the source code can easily be adjusted to a form which is unambiguous. Also, we expect that it will be more common to have generic function invocations as actual arguments than having relational or shift expressions that happen to match up and have parentheses at the end, such that the ambiguity arises.

Evaluation of an actual argument part of the form
<$A_1, \ldots, A_r$>($a_1, \ldots, a_m, q_1: a_{m+1}, \ldots, q_l: a_{m+l}$) proceeds as follows:

The type arguments $A_1, \ldots, A_r$ are evaluated in the order they appear in the program, producing types $t_1, \ldots, t_r$. The arguments $a_1, \ldots, a_{m+l}$ are evaluated in the order they appear in the program, producing objects $o_1, \ldots, o_{m+l}$.

Simply stated, an argument part consisting of s type arguments, m positional arguments, and l named arguments is evaluated from left to right. Note that the type argument list is omitted when $r = 0$ (15).

17.15.3 Binding Actuals to Formals

In the following, the non-generic case is covered implicitly: When the number of actual type arguments is zero the entire type argument list <...> is omitted, and similarly for empty type parameter lists (15).

Consider an invocation i of a function f with an actual argument part of the form <$A_1, \ldots, A_r$>($a_1, \ldots, a_m, q_1: a_{m+1}, \ldots, q_l: a_{m+l}$).

Note that f denotes a function in a semantic sense, rather than a syntactic construct. A reference to this section is used in other sections when the static analysis of an invocation is specified, and the static type of f has been determined. The function itself may have been obtained from a function declaration, from an instance bound to **this** and an instance method declaration, or as a function object obtained by evaluation of an expression. Because of that, we cannot indicate here which syntactic construct corresponds to f. A reference to this section is also used in other sections when actual arguments are to be bound to the corresponding formal parameters, and f is about to be invoked, to specify the dynamic semantics.

We do not call f a 'function object' here, because we do not wish to imply that every function invocation must involve a separate evaluation of an expression that yields a function object, followed by an invocation of that function object. For instance, an implementation should be allowed to compile the invocation of a top-level function as a series of steps whereby a stack frame is created, followed by a low-level jump to the generated code for the body. So, in this section, the word 'function' is more low-level than 'function object', but 'function' still denotes a semantic entity which is associated with a function declaration, even though there may not be a corresponding entity in the heap at run time.

It is a compile-time error if $q_j = q_k$ for any $j \neq k$.

For a given type T_0, we introduce the notion of a T_0 *bounded type*: T_0 itself ◇
is T_0 bounded; if B is T_0 bounded and X is a type variable with bound B then
X is T_0 bounded; finally, if B is T_0 bounded and X is a type variable then $X\&B$
is T_0 bounded. In particular, a ***dynamic*** *bounded type* is either **dynamic** itself ◇
or a type variable whose bound is **dynamic** bounded, or an intersection whose
second operand is **dynamic** bounded. Similarly for a ***Function*** *bounded type*. ◇

A *function-type bounded type* is a type T which is T_0 bounded where T_0 ◇
is a function type (20.5). A function-type bounded type T has an *associated* ◇
function type which is the unique function type T_0 such that T is T_0 bounded.

If the static type of f is **dynamic** bounded or **Function** bounded, no further static checks are performed on the invocation i (apart from separate static checks

on subterms like arguments), and the static type of i is **dynamic**. Otherwise, it is a compile-time error if the static type of f is not function-type bounded.

If no error occurred and the static analysis of i is not complete then the static type T_f of f is function-type bounded; let F be the associated function type of T_f.

Let S_0 be the return type of F, let X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s be the formal type parameters, let h be the number of required parameters, let $p_1, \ldots, p_n$ be the positional parameters, and let $p_{h+1}, \ldots, p_{h+k}$ be the optional parameters of F. Let S_i be the static type of the formal parameters $p_i, i \in 1..h+k$, and for each q let S_q be the type of the parameter named q, where each parameter type is obtained by replacing X_j by $A_j, j \in 1..s$, in the given parameter type annotation. Finally, let T_i be the static type of a_i.

We have an actual argument list consisting of r type arguments, m positional arguments, and l named arguments. We have a function with s type parameters, h required parameters, and k optional parameters. Figure 1 shows how this situation arises.

Actual arguments:

$$\langle r \text{ type arguments} \rangle \left(\begin{array}{c} m \text{ positional arguments, } l \text{ named arguments} \end{array} \right)$$

Declaration with named parameters: $n = h$

$$\langle s \text{ type parameters} \rangle \left(\begin{array}{c} h \text{ required parameters, } k \text{ optional parameters} \\ \textit{which may also be viewed as} \\ n \text{ positional parameters, } k \text{ named parameters} \end{array} \right)$$

Declaration with optional positional parameters: $n = h + k$

$$\langle s \text{ type parameters} \rangle \left(\begin{array}{c} h \text{ required parameters, } k \text{ optional parameters} \\ \textit{which may also be viewed as} \\ n \text{ positional parameters} \end{array} \right)$$

Figure 1: Possible actual argument parts and formal parameter parts.

It is a compile-time error if $r \neq s$. It is a compile-time error if $r = s$ and for some $j \in 1..s$, $A_j \not<: [A_1/X_1, \ldots, A_r/X_s]B_j$. It is a compile-time error unless $h \leq m \leq n$. If $l > 0$, it is a compile-time error unless F has named parameters and $q_j \in \{p_{h+1}, \ldots, p_{h+k}\}, j \in 1..l$.

That is, the number of type arguments must match the number of type parameters, and the bounds must be respected. We must receive at least the required number of positional arguments, and not more than the total number of positional parameters. For each named argument there must be a named parameter with the same name.

The static type of i is $[A_1/X_1, \ldots, A_r/X_s]S_0$.

It is a compile-time error if T_j may not be assigned to $S_j, j \in 1..m$. It is a

compile-time error if T_{m+j} may not be assigned to $S_{q_j}, j \in 1..l$.

Consider the case where the function invocation in focus here is an instance method invocation. In that case, for each actual argument, the corresponding parameter may be covariant. However, the above assignability requirements apply equally both when the parameter is covariant and when it is not.

Parameter covariance in an instance method invocation can be introduced by a subtype of the statically known receiver type, which means that any attempt to flag a given actual argument as dangerous due to the dynamic type check that it will be subjected to will be incomplete: some actual arguments can be subjected to such a dynamic type check even though this is not known statically at the call site. This is not surprising for a mechanism like parameter covariance which is designed for the very purpose of allowing developers to explicitly request that this specific kind of compile-time safety is violated. The point is that this mechanism postpones the enforcement of the underlying invariant to run time, and in return allows some useful program designs that would otherwise be rejected at compile-time.

For the dynamic semantics, let f be a function with s type parameters and h required parameters; let $p_1, \ldots, p_n$ be the positional parameters of f; and let $p_{h+1}, \ldots, p_{h+k}$ be the optional parameters declared by f.

An evaluated actual argument part

$\texttt{<}t_1, \ldots, \ t_r\texttt{>(}o_1, \ldots, \ o_m, \ q_1\texttt{:} \ \ o_{m+1}, \ \ldots, \ q_l\texttt{:} \ \ o_{m+l}\texttt{)}$

derived from an actual argument part of the form

$\texttt{<}A_1, \ldots, \ A_r\texttt{>(}a_1, \ldots, \ a_m, \ q_1\texttt{:} \ \ a_{m+1}, \ldots, \ q_l\texttt{:} \ \ a_{m+l}\texttt{)}$

is bound to the formal type parameters and formal parameters of f as follows:

If $r = 0$ and $s > 0$ then if f does not have default type arguments (15.3) then a dynamic error occurs. Otherwise replace the actual type argument list: Let r be s and let t_i for $i \in 1..s$ be the result of instantiation to bound (15.3) on the formal type parameters of f, substituting the actual values of any free type variables (20.10.1). Otherwise, if $r \neq s$, a `NoSuchMethodError` is thrown.

If $l > 0$ and $n \neq h$, a `NoSuchMethodError` is thrown. If $m < h$, or $m > n$, a `NoSuchMethodError` is thrown. Furthermore, each $q_i, i \in 1..l$, must have a corresponding named parameter in the set $\{p_{h+1}, \ldots, p_{h+k}\}$, or a `NoSuchMethodError` is thrown. 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 t_i is not a subtype of the actual bound (20.10.1) of the ith type argument of f, for actual type arguments $t_1, \ldots, t_r$. It is a dynamic type error if o_i is not the null object (17.4) and the actual type (20.10.1) of p_i is not a supertype of the dynamic type of $o_i, i \in 1..m$. It is a dynamic type error if o_{m+j} is not the null object and the actual type (20.10.1) of q_j is not a supertype of the dynamic type of $o_{m+j}, j \in 1..l$.

17.15.4 Unqualified Invocation

An unqualified function invocation i has the form
id<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k}$),
where *id* is an identifier.

Note that the type argument list is omitted when $r = 0$ (15).

Perform a lexical lookup of *id* (17.37) from the location of i.

Case ⟨Lexical lookup yields a declaration⟩**.** Let D be the declaration yielded by the lexical lookup of *id*.

- When D is a type declaration, that is, a declaration of a class, mixin, type alias, or type parameter, the following applies: If D is a declaration of a class C that has a constructor named C then the meaning of i depends on the context: If i occurs in a constant context (17.3.2), then i is treated as (5) **const** i; if i does not occur in a constant context then i is treated as **new** i. If D is not a class declaration, or it declares a class named C that has no constructor named C, a compile-time error occurs.

- Otherwise, if D is a declaration of a local function, a library function, or a library or static getter, or a variable, then i is treated as (5) a function expression invocation (17.15.5).

- Otherwise, if D is a static method or getter (which may be implicitly induced by a class variable) in the enclosing class or mixin C, i is treated as (5) $C.i$ (17.21.1).

- If D is an instance member of an extension E with type parameters $X_1, \ldots, X_k$, i is treated as E<$X_1, \ldots, X_k$>(**this**).i. Both the static analysis and evaluation proceeds with the transformed expression, so there is no need to further specify the treatment of i.

  In other words, inside E the instance members of E will shadow the instance members of the **on** type, that is, the extension has higher priority than the object interface. The opposite is true for invocations everywhere outside E.

  There is no need to consider an instance member of a class, because a lexical lookup in a class will never yield a declaration which is an instance member.

  □

Case ⟨Lexical lookup yields an import prefix⟩**.** When the lexical lookup of *id* yields an import prefix, a compile-time error occurs. □

Case ⟨Lexical lookup yields nothing⟩**.** When the lexical lookup of *id* yields nothing, i is treated as (5) the ordinary method invocation **this**.i (17.21.1).

This occurs when the lexical lookup has determined that i must invoke an instance member of a class or an extension, and the location of i can access **this**, and the interface of the enclosing class has a member named id, or there is an applicable extension with such a member. Both the static analysis and evaluation proceeds with **this**.i, so there is no need to further specify the treatment of i. □

Note that an unqualified invocation does not specify an evaluation semantics. This is because every case which is not an error ends in the conclusion that the unqualified invocation should be treated as some other construct, which is specified elsewhere.

17.15.5 Function Expression Invocation

A function expression invocation i has the form
$e_f\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:} \; a_{n+1}, \ldots, x_{n+k}\texttt{:} \; a_{n+k}\texttt{)}$,
where e_f is an expression.

Note that the type argument list is omitted when $r = 0$ (15).

Consider the situation where e_f denotes a class C that contains a declaration of a constructor named C, or it is of the form $e'_f\texttt{.}id$ where e'_f denotes a class C that contains a declaration of a constructor named $C\texttt{.}id$. If i occurs in a constant context (17.3.2) then i is treated as **const** i, and if i does not occur in a constant context then i is treated as **new** i.

When i is treated as another construct i', both the static analysis and the dynamic semantics is specified in the section about i' (5).

Otherwise, it is a compile-time error if e_f is a type literal.

This error was already specified elsewhere (17.15.4) for the case where e_f is an identifier, but e_f may also have other forms, e.g., `p.C`.

Otherwise, 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 would not have been treated as a function expression invocation.

If e_f is a property extraction expression (17.22) then i treated as an ordinary method invocation (17.21.1).

$a.b(x)$ is treated as a method invocation of method $b()$ on object a, not as an invocation of getter b on a followed by a function call $(a.b)(x)$. If a method or getter b exists, the two will be equivalent. However, if b is not defined on a, the resulting invocation of `noSuchMethod()` would differ. The `Invocation` passed to `noSuchMethod()` would describe a call to a method b with argument x in the former case, and a call to a getter b (with no arguments) in the latter.

Let F be the static type of e_f. If F is an interface type that has a method named `call`, i is treated as (5) the ordinary invocation
$e_f\texttt{.call<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:} \; a_{n+1}, \ldots, x_{n+k}\texttt{:} \; a_{n+k}\texttt{)}$.

Otherwise, the static analysis of i is performed as specified in Section 17.15.3, using F as the static type of the invoked function, and the static type of i is as specified there.

Evaluation of a function expression invocation
$e_f\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:} \; a_{n+1}, \ldots, x_{n+k}\texttt{:} \; a_{n+k}\texttt{)}$
proceeds to evaluate e_f, yielding an object o. Let f be a fresh variable bound to o. If o is a function object then the function invocation
$f\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:} \; a_{n+1}, \ldots, x_{n+k}\texttt{:} \; a_{n+k}\texttt{)}$
is evaluated by binding actuals to formals as specified in Section 17.15.3, and executing the body of f with those bindings; the returned result is then the result of evaluating i.

Otherwise o is not a function object. If o has a method named `call` the following ordinary method invocation is evaluated, and its result is then the result of evaluating i:
$f\texttt{.call<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:} \; a_{n+1}, \ldots, x_{n+k}\texttt{:} \; a_{n+k}\texttt{)}$.

Otherwise o has no method named `call`. A new instance im of the predefined

class `Invocation` is created, such that:

- im`.isMethod` evaluates to **true**.
- im`.memberName` evaluates to the symbol `#call`.
- im`.positionalArguments` evaluates to an unmodifiable list whose dynamic type implements `List<Object>`, containing the objects resulting from evaluation of `<Object>[`$a_1, \ldots, a_n$`]`.
- im`.namedArguments` evaluates to an unmodifiable map whose dynamic type implements `Map<Symbol, Object>`, with the keys and values resulting from evaluation of

  `<Symbol, Object>{`$\#x_{n+1}$`:` $a_{n+1}, \ldots, \#x_{n+k}$`:` a_{n+k}`}`.
- im`.typeArguments` evaluates to an unmodifiable list whose dynamic type implements `List<Type>`, with the values resulting from evaluation of

  `<Type>[`$A_1, \ldots, A_r$`]`.

Then the method invocation `f.noSuchMethod(`im`)` is evaluated, and its result is then the result of evaluating i.

The situation where `noSuchMethod` is invoked can only arise when the static type of e_f is **dynamic**. The run-time semantics ensures that a function invocation may amount to an invocation of the instance method `call`. However, an interface type with a method named `call` is not itself a subtype of any function type (20.4.2).

17.16 Function Closurization

Let f be an expression denoting a declaration of a top-level function, a local function, a static method of a class, of a mixin, or of an extension (17.38); or let f be a function literal (17.11). Evaluation of f yields a function object which is the outcome of a *function closurization* applied to the declaration denoted by f respectively to the function literal f considered as a function declaration. *Closurization* denotes instance method closurization (17.22.3) as well as function closurization, and it is also used as a shorthand for either of them when there is no ambiguity. ◇ ◇

Function closurization applied to a function declaration f amounts to the creation of a function object o which is an instance of a class C whose interface is a subtype of the actual type F (20.10.1) corresponding to the signature in the function declaration f, using the current bindings of type variables, if any. There does not exist a function type F' which is a proper subtype of F such that C is a subtype of F'.

If f is a static method or a top-level function then o has primitive equality (10.2.3).

In other words, C has the freedom to be a proper subtype of the function type that we can read off of the declaration of f because it may need to be a specific internal platform defined class, but C does not have the freedom to be a subtype of a different and more special function type, and it cannot be `Null`.

An invocation of o with a given argument list will bind actuals to formals in the same way as an invocation of f (17.15.3), and then execute the body of f in the captured scope amended with the bound parameter scope, yielding the same completion (18.0.1) as the invocation of f would have yielded.

Let e_1 and e_2 be two constant expressions that both evaluate to a function object which is obtained by function closurization of the same function declaration. In this case `identical(`e_1`, `e_2`)` shall evaluate to true.

That is, constant expressions whose evaluation is a function closurization are canonicalized.

17.17 Generic Function Instantiation

Generic function instantiation is a mechanism that yields a non-generic function object based on a given generic function.

The essence of generic function instantiation is to allow for "curried" invocations, in the sense that a generic function can receive its actual type arguments separately (it must then receive all *type arguments, not just some of them), and that yields a non-generic function object. The type arguments are passed implicitly, based on type inference; a future version of Dart may allow for passing them explicitly.* Here is an example:

```
X fg<X extends num>(X x) => x;

class A {
  static X fs<X extends num>(X x) => x;
}

void main() {
  X fl<X extends num>(X x) => x;
  List<int Function(int)> functions = [fg, A.fs, fl];
}
```

Each function object stored in `functions` has dynamic type `int Function(int)`, and it is obtained by implicitly "passing the actual type argument `int`" to the corresponding generic function.

Let f be an expression whose static type G is a generic function type of the form T_0 **Function**`<`$X_1 \lhd B_1, \ldots, X_s \lhd B_s$`>(`$p$`)` where `(`$p$`)` is derived from ⟨*formalParameterList*⟩. Note that $s > 0$ because G is generic. Assume that the context type is a non-generic function type F. In this situation a compile-time error occurs (8, 9, 10.7.1, 10.7.1, 10.7.1, 17.13.1, 17.13.2, 17.15.3, 17.23, 18.3, 18.9, 18.12, 18.17), except when the following step succeeds: f, G T_0, X_j, B_j, s, p

Generic function type instantiation: Type inference is applied to G with context type F, and it yields the actual type argument list $T_1, \ldots, T_s$. $\diamond$ T_j

The generic function type instantiation fails in the case where type inference fails, in which case the above mentioned compile-time error occurs. It will be specified in a future version of this document how type inference computes $T_1, \ldots, T_s$ (6).

Assume that the generic function type instantiation succeeded. Let F' denote the type $[T_1/X_1, \ldots, T_s/X_s](T_0$ **Function**$(p))$. Note that it is guaranteed that F' is assignable to F, or inference would have failed. Henceforth in the static analysis, this occurrence of f is considered to have static type F'. F'

Execution of f proceeds as follows. Evaluate f to an object o. A dynamic error occurs if o is the null object. Let S_0 **Function**<$Y_1 \triangleleft B'_1, \ldots, Y_s \triangleleft B'_s$>$(q)$ be the dynamic type of o (by soundness, this type is a subtype of G). f then evaluates to a function object o' with dynamic type $[t_1/Y_1, \ldots, t_s/Y_s](S_0$ **Function**$(q))$, where t_j is the actual value of T_j, for $j \in 1..k$. An invocation of o' with actual arguments *args* has the same effect as an invocation of o with actual type arguments $t_1, \ldots, t_s$ and actual arguments *args*.

Let f_1 and f_2 be two constant expressions that are subject to generic function instantiation. Assume that f_1 and f_2 without a context type evaluate to o_1 respectively o_2 such that `identical(`o_1`, `o_2`)` is true. Assume that the given context types give rise to a successful generic function type instantiation with the same actual type arguments for f_1 and f_2, yielding the non-generic function objects o'_1 respectively o'_2. In this case `identical(`o'_1`, `o'_2`)` shall evaluate to true.

That is, constant expressions whose evaluation is a generic function instantiation are canonicalized, based on the underlying function and on the actual type arguments. As a consequence, they are also equal according to operator '=='.

Let g_1 and g_2 be two expressions (which may or may not be constant) that are subject to generic function instantiation. Assume that g_1 and g_2 without a context type evaluate to o_1 respectively o_2 such that o_1 `==` o_2`)` is true. Assume that the given context types give rise to a successful generic function type instantiation with the same actual type arguments for g_1 and g_2, yielding the non-generic function objects o'_1 respectively o'_2. In this case o'_1 `==` o'_2 shall evaluate to true.

When one or both of the expressions is not constant, it is unspecified whether `identical(`o_1`, `o_2`)` evaluates to **true** or **false**, but operator '==' yields true for equal function objects instantiated with the same actual type arguments.

No notion of equality is appropriate when the type arguments differ, even if the resulting function objects turn out to have exactly the same type at run time, because execution of two function objects that differ in these ways can have different side-effects and return different results when executed starting from exactly the same state.

17.18 Lookup

A *lookup* is a procedure which selects a concrete instance member declaration based on a traversal of a sequence of classes, starting with a given class C and proceeding with the superclass of the current class at each step. A lookup may be part of the static analysis, and it may be performed at run time. It may succeed or fail. ◇

We define several kinds of lookup with a very similar structure. We spell out each of them in spite of the redundancy, in order to avoid introducing meta-level

abstraction mechanisms just for this purpose. The point is that we must indicate for each lookup which kind of member it is looking for, because, e.g., a 'method lookup' and a 'getter lookup' are used in different situations.

Let m be an identifier, o an object, L a library, and C a class which is the class of o or a superclass thereof.

The result of a *method lookup for m in o with respect to L starting in class C* is the result of a method lookup for m in C with respect to L. The result of a *method lookup for m in C with respect to L* is: If C declares a concrete instance method named m that is accessible to L, then that method declaration is the result of the method lookup, and we say that the method was *looked up in C*. Otherwise, if C has a superclass S, the result of the method lookup is the result of a method lookup for m in S with respect to L. Otherwise, we say that the method lookup has failed.

The result of a *getter lookup for m in o with respect to L starting in class C* is the result of a getter lookup for m in C with respect to L. The result of a *getter lookup for m in C with respect to L* is: If C declares a concrete instance getter named m that is accessible to L, then that getter declaration is the result of the getter lookup, and we say that the getter was *looked up in C*. Otherwise, if C has a superclass S, the result of the getter lookup is the result of a getter lookup for m in S with respect to L. Otherwise, we say that the getter lookup has failed.

The result of a *setter lookup for m in o with respect to L starting in class C* is the result of a setter lookup for m in C with respect to L. The result of a *setter lookup for m in C with respect to L* is: If C declares a concrete instance setter named m that is accessible to L, then that setter declaration is the result of the setter lookup, and we say that the setter was *looked up in C*. Otherwise, if C has a superclass S, the result of the setter lookup is the result of a setter lookup for m in S with respect to L. Otherwise, we say that the setter lookup has failed.

Let m be an identifier, o an object, and L a library. The result of a *method lookup for m in o with respect to L* is the result of a method lookup for m in o with respect to L starting with the class of o. The result of a *getter lookup for m in o with respect to L* is the result of a getter lookup for m in o with respect to L starting with the class of o. The result of a *setter lookup for m in o with respect to L* is the result of a setter lookup for m in o with respect to L starting with the class of o.

Note that for getter (setter) lookup, the result may be a getter (setter) which has been induced by an instance variable declaration.

Note that we sometimes use phrases like 'looking up method m' to indicate that a method lookup is performed, and similarly for setter lookups and getter lookups.

The motivation for ignoring abstract members during lookup is largely to allow smoother mixin composition.

17.19 Top level Getter Invocation

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. Note that the invocation is always defined. Per the rules for identifier references, an identifier will not be treated as a top-level getter invocation unless the getter i is defined.

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

17.20 Member Invocations

Simple member invocation	Corresponding member name	Composite member invocation	Corresponding member name
r`.`*id*`, `r`?.`*id*	*id*	r`.`*id* $\otimes$`= `e	*id*
r`.`*id*` = `e`, `r`?.`*id*` = `e	*id*`=`	r`?.`*id* $\otimes$`= `e	*id*
r`.`*id*`(`*args*`), `r`?.`*id*`(`*args*`)`	*id*	r`[`e_1`] `$\otimes$`= `e_2	`[]`
r`.`*id*`<`*types*`>(`*args*`)`	*id*	`++`r`.`*id*`, --`r`.`*id*	*id*
r`?.`*id*`<`*types*`>(`*args*`)`	*id*	r`.`*id*`++, `r`.`*id*`--`	*id*
`-`r	`unary-`	`++`r`[`e`], --`r`[`e`]`	`[]`
`~`r	`~`	r`[`e`]++, `r`[`e`]--`	`[]`
r $\oplus$ e	$\oplus$		
r`[`e`]`	`[]`		
r`[`e_1`] = `e_2	`[]=`		
r`(`*args*`)`	`call`		
r`<`*types*`>(`*args*`)`	`call`		

In the tables above, r, e, e_1, e_2 are expressions; *args* is derived from ⟨*arguments*⟩; *types* is derived from ⟨*typeArguments*⟩; '$\oplus$' is an ⟨*operator*⟩ which is not '`==`'; and '$\otimes$`=`' is a ⟨*compoundAssignmentOperator*⟩.

Figure 2: Member invocations with syntactic receiver r.

A *member invocation* is an expression with a specific syntactic form whose dynamic semantics involves invocation of one or two instance members of a given receiver, or invocation of extension members. This section specifies which syntactic forms are member invocations, and defines some terminology which is needed in order to denote specific parts of several syntactic forms collectively. ◇

The static analysis and dynamic semantics of each of the syntactic forms that are member invocations is specified separately, this section is only concerned with the syntactic classification and terminology.

For example, one kind of member invocation is an ordinary method invocation (17.21.1).

A *simple member invocation* respectively *composite member invocation* on a *syntactic receiver* expression r is an expression of one of the forms shown in Fig. 2. Each member invocation has a *corresponding member name* as shown in the figure. ◇ ◇ ◇ ◇

Each member invocation in Fig. 2 that contains '`?`' is a *conditional member* ◇

Source	Desugared form
r`?.`id	r' `==null ? null :` r'`.`id
r`?.`id `=` e	r' `==null ? null :` r'`.`id `=` e
r`?.`id`(`$args$`)`	r' `==null ? null :` r'`.`id`(`$args$`)`
r`?.`id`<`$types$`>(`$args$`)`	r' `==null ? null :` r`.`id`<`$types$`>(`$args$`)`
r`?[`e`]`	r' `==null ? null :` r'`[`e`]`
r`?[`e_1`] =` e_2	r' `==null ? null :` r'`[`e_1`] =` e_2
r`.`id `??=` e	`let` v`=`r'`.`id `in` v `==null ?` r'`.`id`=`e `:` v
r`.`id $\otimes$`=` e	r'`.`id `=` r'`.`id $\otimes$ e
r`?.`id $\otimes$`=` e	r' `==null ? null :` r'`.`$id$$\otimes$`=` e
r`[`e_1`] ??=` e_2	`let` v`=`e_1`,` v'`=`r'`[`v`]` `in` v' `==null ?` r'`[`v`]=`e_2 `:` v'
r`[`e_1`]` $\otimes$`=` e_2	`let` v`=`e_1 `in` r'`[`v`] =` r'`[`v`]` $\otimes$ e_2
`++`r`.`id	r`.`id `+= 1`
`--`r`.`id	r`.`id `-= 1`
r`.`$id`++`	`let` v`=`r'`.`id`,` v'`=`r'`.`id`=`v`+1` `in` v
r`.`$id`--`	`let` v`=`r'`.`id`,` v'`=`r'`.`id`=`v`-1` `in` v
`++`r`[`e`]`	r`[`e`] += 1`
`--`r`[`e`]`	r`[`e`] -= 1`
r`[`e`]++`	`let` v`=`e`,` v'`=`r'`[`v`],` v''`=`r'`[`v`]=`v'`+1` `in` v'
r`[`e`]--`	`let` v`=`e`,` v'`=`r'`[`v`],` v''`=`r'`[`v`]=`v'`-1` `in` v'

Figure 3: Desugaring of member invocations. '$\otimes$=' is a ⟨*compoundAssignmentOperator*⟩. The first applicable rule is used. In particular, '$\otimes$=' cannot be '`??=`' when an earlier rule with '`??=`' matches. r' is known as the *replacement receiver* of the composite member invocation, and it is specified in the main text (17.20).

invocation. An *unconditional member invocation* is a member invocation which is not conditional. ◇

For a simple member invocation the corresponding member name is the name of the member which is invoked in the case where the member invocation invokes an instance member. For a composite member invocation it is the name of the getter and the basename of both the getter and the setter.

Note that r cannot be ***super*** *even though* ***super***`.m()` *invokes an instance method. This is because the semantics of a superinvocation is different from that of other invocations. Among the binary operators, '==' is not included. This is because evaluation of e_1 == e_2 involves more steps than an instance member invocation. Similarly, '&&', and '||' are not included because their evaluation does not involve method invocation.*

A composite member invocation is an abbreviated form whose meaning is reduced to simple member invocations as shown in Fig. 3. This step is known as a *desugaring* transformation, and we say that the resulting expression has been *desugared*. ◇ Fig. 3 contains several occurrences of r' which is the *replacement receiver* of the composite method invocation. ◇ The meaning of each occurrence ◇

of r' is determined as follows:

When the receiver r is an extension application (13.1) of the form $E\texttt{<}T_1, \ldots, T_k\texttt{>(}e_r\texttt{)}$ (where $k = 0$ means that the type argument list is absent): Let v_r be a fresh variable bound to the value of e_r and with the same static type as e_r, then r' is $E\texttt{<}T_1, \ldots, T_k\texttt{>(}v_r\texttt{)}$ when it occurs as the receiver of a member invocation, and otherwise r' is v_r.

When r is not an extension application, r' is a fresh variable bound to the value of r, with the same static type as r.

This corresponds to an extra outermost **let** in each rule in Fig. 3 where r' occurs, and an explicit distinction between the two forms of r, but the figure would be considerably more verbose if it had been specified in that manner.

17.21 Method Invocation

Method invocation can take several forms as specified below.

17.21.1 Ordinary Invocation

An ordinary method invocation can be conditional or unconditional.

Case $\langle e\texttt{?.}m\texttt{<}\cdots\texttt{>(}\cdots\texttt{)}\rangle$**.** Consider a *conditional ordinary method invocation* i of the form $e\texttt{?.}m\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:}\ a_{n+1}, \ldots, x_{n+k}\texttt{:}\ a_{n+k}\texttt{)}$. ◇

Note that non-generic invocations arise as the special case where the number of type arguments is zero, in which case the type argument list is omitted, and similarly for formal type parameter lists (15).

The static type of i is the same as the static type of $e\texttt{.}m\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:}\ a_{n+1}, \ldots, x_{n+k}\texttt{:}\ a_{n+k}\texttt{)}$. Exactly the same compile-time errors that would be caused by $e\texttt{.}m\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:}\ a_{n+1}, \ldots, x_{n+k}\texttt{:}\ a_{n+k}\texttt{)}$ are also generated in the case of i.

Evaluation of i proceeds as follows:

If e is a type literal or denotes an extension, i is treated as $e\texttt{.}m\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:}\ a_{n+1}, \ldots, x_{n+k}\texttt{:}\ a_{n+k}\texttt{)}$.

Otherwise, evaluate e to an object o. If o is the null object, i evaluates to the null object (17.4). Otherwise let v be a fresh variable bound to o and evaluate $v\texttt{.}m\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:}\ a_{n+1}, \ldots, x_{n+k}\texttt{:}\ a_{n+k}\texttt{)}$ to an object r. Then e evaluates to r. □

Case $\langle C\texttt{.}m\texttt{<}\cdots\texttt{>(}\cdots\texttt{)}\rangle$**.** A *static member invocation* i is an invocation of the form $C\texttt{.}m\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:}\ a_{n+1}, \ldots, x_{n+k}\texttt{:}\ a_{n+k}\texttt{)}$, where C is a type literal, or C denotes an extension. ◇

Non-generic invocations arise as the special case where the number of type arguments is zero (15).

A compile-time error occurs unless C denotes a class, a mixin, or an extension that declares a static member named m, which we will call the *denoted member* of i. When the denoted member is a static method, let F be its function type; ◇

when the denoted member is a static getter, let F be its return type; when the denoted member is neither, a compile-time error occurs.

The static analysis of i is then performed as specified in Section 17.15.3, considering F to be the static type of the function to call, and the static type of i is as specified there.

Evaluation of a static method invocation i of the form
$C\texttt{.}m\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:} a_{n+1}, \ldots, x_{n+k}\texttt{:} a_{n+k}\texttt{)}$
proceeds as follows:

If the denoted member of i is a static method, let f be the function declared by that member. The binding of actual arguments to formal parameters is performed as specified in Section 17.15.3. The body of f is then executed with respect to the bindings that resulted from the evaluation of the argument part. The value of i is the object returned by the execution of f's body.

If the denoted member of i is a static getter, invoke said getter and let v_f be a fresh variable bound to the returned object. Then the value of i is the value of $v_f\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:} a_{n+1}, \ldots, x_{n+k}\texttt{:} a_{n+k}\texttt{)}$. □

Case $\langle e\texttt{.}m\texttt{<}\cdots\texttt{>(}\cdots\texttt{)}\rangle$. An *unconditional ordinary method invocation* i has the form $e\texttt{.}m\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:} a_{n+1}, \ldots, x_{n+k}\texttt{:} a_{n+k}\texttt{)}$, where e is an expression that is not a type literal, and does not denote an extension. ◇

Non-generic invocations arise as the special case where the number of type arguments is zero (15).

Let T be the static type of e. It is a compile-time error if T does not have an accessible (6.2) instance member named m, unless either:

- T is **dynamic** bounded (17.15.3); in this case no further static checks are performed on i (apart from separate static checks on subterms like arguments) and the static type of i is **dynamic**. Or
- T is **Function** bounded (17.15.3) and m is `call`; in this case no further static checks are performed on i (apart from separate static checks on subterms like arguments) and the static type of i is **dynamic**. *This means that for invocations of an instance method named `call`, a receiver of type **Function** is treated like a receiver of type **dynamic**. The expectation is that any concrete subclass of **Function** will implement `call`, but there is no method signature which can be assumed for `call` in **Function** because every signature will conflict with some potential overriding declarations.*

If T did not have an accessible member named m the static type of i is **dynamic**, and no further static checks are performed on i (except that subexpressions of i are subject to their own static analysis).

Otherwise $T\texttt{.}m$ denotes an instance member. Let L be the library that contains i. Let d be the result of method lookup for m in T with respect to L, and if the method lookup succeeded then let F be the static type of d.

Otherwise, let d be the result of getter lookup for m in T with respect to L, and let F be the return type of d. (Since $T\texttt{.}m$ exists we cannot have a failure in both lookups.) If the getter return type F is an interface type that has a method named `call`, i is treated as (5) the ordinary invocation

e`.`m`.call<`$A_1, \ldots, A_r$`>(`$a_1, \ldots, a_n, x_{n+1}$`:` $a_{n+1}, \ldots, x_{n+k}$`:` a_{n+k}`)`, which determines any further static analysis.

Otherwise, the static analysis of i is performed as specified in Section 17.15.3, considering F to be the static type of the function to call, and the static type of i is as specified there, except that invocations of methods named `remainder` or `clamp` on subtypes of `num` that are not subtypes of `Never` have special rules similar to those for additive (17.31) and multiplicative (17.32) operators.

Let i be an invocation of the form e`.remainder(`e_2`)` and let C be the context type of i. The context type of e_2 is then determined as follows: If T <: `num` and not T <: `Never`, then:

- If `int` <: C and not `num` <: C, and T <: `int` then the context type of e_2 is `int`.
- If `double` <: C and not `num` <: C, and not T <: `double` then the context type of e_2 is `double`.
- Otherwise the context type of e_2 is `num`.

Let further S be the static type of e_2. If T <: `num` and not T <: `Never` and S is assignable to `num`, then the static type of i is determined as follows:

- If T <: `double` then the static type of i is T.
- Otherwise, if S <: `double` and not S <: `Never`, then the static type of i is `double`.
- Otherwise, if T <: `int`, S <: `int` and not S <: `Never`, then the static type of i is `int`.
- Otherwise the static type of i is `num`.

Let i be an invocation of the form e`.clamp(`e_2`,` e_3`)`, where T <: `num` and not T <: `Never`, and let C be the context type of i. The context type of e_2 and e_3 is then determined as follows:

- If T <: `int`, `int` <: S and not `num` <: S, then the context type of e_2 and e_3 is `int`.
- If T <: `double`, `double` <: S and not `num` <: S, then the context type of e_2 and e_3 is `double`.
- Otherwise the context type of e_2 and e_3 is `num`.

Let further T_2 be the static type of e_2 and T_3 be the static type of e_3.

- If all of T, T_2 and T_3 are subtypes of `int`, but not subtypes of `Never`, then the static type of i is `int`.
- If all of T, T_2 and T_3 are subtypes of `double`, but not subtypes of `Never`, then the static type of i is `double`.

- Otherwise the static type of i is `num`.

It is a compile-time error to invoke an instance method on a type literal that is immediately followed by the token '.' (a period). For instance, `int.toString()` is an error.

The reason for this rule is that member access on a type literal is reserved for invocation of static members. Invocation of a static member of a class, mixin, or extension uses said entity as a namespace*, not as an actual class, mixin, or extension. In particular, the syntactic receiver is not evaluated to an object—that would not even be possible for an extension.*

A member access on a type literal (e.g., `C.id()`, `C.id`, or `C?.id()`), always treats the declaration denoted by the literal as a namespace for accessing static members or constructors. For instance, `int.toString()` is an error because `int` does not declare a static member named `toString`. It will not evaluate `int` to a `Type` object and then call its `toString` instance method. To do that, you can use `(int).toString()`. Note that cascades are different: they *always* evaluate their receiver to an object first.

As a natural consequence, a type literal cannot be the receiver in an implicit invocation of an extension method (13.2).

Evaluation of an unconditional ordinary method invocation i of the form e`.`m`<`$A_1, \ldots, A_r$`>(`$a_1, \ldots, a_n, x_{n+1}$`:` $a_{n+1}, \ldots, x_{n+k}$`:` a_{n+k}`)` proceeds as follows:

First, the expression e is evaluated to an object o. Let f be the result of looking up (17.18) method m in o with respect to the current library L.

If the method lookup succeeded, the binding of actual arguments to formal parameters is performed as specified in Section 17.15.3. The body of f is then executed with respect to the bindings that resulted from the evaluation of the argument list, and with **this** bound to o. The value of i is the object returned by the execution of f's body.

If the method lookup failed, then let g be the result of looking up getter (17.18) m in o with respect to L.

If the getter lookup succeeded then invoke the getter $o.m$ and let v_g be the returned object. Then the value of i is the value of
v_g`<`$A_1, \ldots, A_r$`>(`$a_1, \ldots, a_n, x_{n+1}$`:` $a_{n+1}, \ldots, x_{n+k}$`:` a_{n+k}`)`.

If getter lookup has also failed, then a new instance im of the predefined class `Invocation` is created, such that:

- im`.isMethod` evaluates to **true**.
- im`.memberName` evaluates to the symbol `m`.
- im`.positionalArguments` evaluates to an unmodifiable list whose dynamic type implements `List<Object>`, containing the objects resulting from the evaluation of `<Object>[`$a_1, \ldots, a_n$`]`.
- im`.namedArguments` evaluates to an unmodifiable map whose dynamic type implements `Map<Symbol, Object>`, with the keys and values resulting from the evaluation of

`<Symbol, Object>{`$\#x_{n+1}$`:` $a_{n+1}, \ldots,$ $\#x_{n+k}$`:` a_{n+k}`}`.

- im`.typeArguments` evaluates to an unmodifiable list whose dynamic type implements `List<Type>`, containing the objects resulting from the evaluation of `<Type>[`$A_1, \ldots,$ A_r`]`.

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.

The situation where `noSuchMethod` is invoked can only arise when the static type of e is **dynamic**. Notice that the wording avoids re-evaluating the receiver o and the arguments a_i. □

17.21.2 Cascades

A *cascade* is a kind of expression that allows for performing multiple operations on a given object without storing it in a variable and accessing it via a name. In general, a cascade can be recognized by the use of exactly two periods, '`..`'. ◇

⟨*cascade*⟩ ::= ⟨*cascade*⟩ '`..`' ⟨*cascadeSection*⟩
  | ⟨*conditionalExpression*⟩ ('`?..`' | '`..`') ⟨*cascadeSection*⟩

⟨*cascadeSection*⟩ ::= ⟨*cascadeSelector*⟩ ⟨*cascadeSectionTail*⟩

⟨*cascadeSelector*⟩ ::= '`[`' ⟨*expression*⟩ '`]`'
  | ⟨*identifier*⟩

⟨*cascadeSectionTail*⟩ ::= ⟨*cascadeAssignment*⟩
  | ⟨*selector*⟩* (⟨*assignableSelector*⟩ ⟨*cascadeAssignment*⟩)?

⟨*cascadeAssignment*⟩ ::= ⟨*assignmentOperator*⟩ ⟨*expressionWithoutCascade*⟩

A *cascaded member access* is an expression derived from ⟨*cascade*⟩. ◇

A ⟨cascadeSection⟩ allows for accessing members, including setters. The motivation for having a cascaded member access is that it allows for performing a chain of operations based on an object while preserving a reference to that object for further processing.

Let e_0 be an extension application (13). Note that it is then a compile-time error to have a ⟨*cascade*⟩ of the form e_0`..`c or e_0`?..`c, where c is a ⟨*cascadeSection*⟩.

For example, `C()..foo.bar = 2` allows us to obtain a reference to the object o which is the result of evaluating `C()`, and at the same time use o to invoke the getter `foo` and the setter `bar=` on the value returned by that getter.

An expression of the form e_0`?..`s where e_0 is a ⟨*conditionalExpression*⟩ and s is a ⟨*cascadeSection*⟩ is an *initially conditional cascaded member access*. Moreover, if e_0 is an initially conditional cascaded member access and s is derived from ⟨*cascadeSection*⟩ then e_0`..`s is also initially conditional. ◇

In short, a cascade is initially conditional if the "innermost dots" are '`?..`' rather than '`..`'. Note that only the innermost dots can have the '`?`'. All the non-innermost

ones are implicitly skipped if the receiver is null, so any '?' on a non-innermost '..' would be useless.

The static analysis and dynamic semantics of a cascaded member access is specified in terms of the following desugaring step.

Let e be a cascaded member access which is not initially conditional. This implies that there exist terms $c_1, \ldots, c_k$ derived from ⟨*cascadeSection*⟩ and a ⟨*conditionalExpression*⟩ e_0 which is not an extension application, and e is $e_0..c_1 \ \cdots \ ..c_k$. In this case, e is desugared to
let $v = e_0$, $v_1 = v.c_1$, $\cdots$, $v_k = v.c_k$ **in** v.

Let e be a cascaded member access which is initially conditional. This implies that there exist terms $c_1, \ldots, c_k$ derived from ⟨*cascadeSection*⟩ and a ⟨*conditionalExpression*⟩ e_0 which is not an extension application, and e is $e_0?..c_1 \ \cdots \ ..c_k$. In this case, e is desugared to
let $v = e_0$ **in** $v ==$ **null** ? **null** : **let** $v_1 = v.c_1$, $\cdots$, $v_k = v.c_k$ **in** v.

Note that the grammar is such that $v.c_j$ is a syntactically correct expression for all j.

17.21.3 Superinvocations

A *method superinvocation* i has the form ◇
super.m<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k}$). i, m, A_j, a_j, x_j

Note that non-generic invocations arise as the special case where the number of type arguments is zero, in which case the type argument list is omitted, and similarly for formal type parameter lists (15).

It is a compile-time error if a method superinvocation occurs in a top-level function or variable initializer, in an instance variable initializer or initializer list, in class `Object`, in a factory constructor, or in a static method or variable initializer.

Let S_{super} be the superclass (10.9) of the immediately enclosing class for i, and let L be the library that contains i. Let the declaration D be the result of looking up the method m in S_{super} with respect to L (17.18), and let F be the static type of D. Otherwise, if the method lookup failed, let the declaration D be the result of looking up the getter m with respect to L in S_{super} (17.18), and let F be the return type of D. If both lookups failed, a compile-time error occurs. S_{super}, L D, F

Otherwise (when one of the lookups succeeded), the static analysis of i is performed as specified in Section 17.15.3, considering the function to have static type F, and the static type of i is as specified there.

Note that member lookups ignore abstract declarations, which means that there will be a compile-time error if the targeted member m is abstract, as well as when it does not exist at all.

An *implicit* `call` *superinvocation* has the form ◇
super<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k}$),
and it is treated as
super.`call`<$A_1, \ldots, A_r$>($a_1, \ldots, a_n, x_{n+1}: a_{n+1}, \ldots, x_{n+k}: a_{n+k}$).

The type argument list is again omitted when $r = 0$.

Evaluation of i proceeds as follows: Let o be the current binding of **this**, let C be the enclosing class for i, and let S_{super} be the superclass (10.9) of C. Let L be the library that contains C. Let the declaration D be the result of looking up the method m with respect to L in o starting with S_{super} (17.18). If the lookup succeeded, let f denote the function associated with D. Otherwise (when method lookup failed), let the declaration D be the result of looking up the getter m with respect to L in o starting with S_{super} (17.18). If the getter lookup succeeded, invoke said getter with **this** bound to o, and let f denote the returned object. o, C, S_{super} L D, f

It cannot occur that both lookups fail, because the corresponding lookups would then have failed at compile-time, in which case the program has a compile-time error.

Otherwise perform the binding of actual arguments to formal parameters for $f\texttt{<}A_1, \ldots, A_r\texttt{>(}a_1, \ldots, a_n, x_{n+1}\texttt{:}\ a_{n+1}, \ldots, x_{n+k}\texttt{:}\ a_{n+k}\texttt{)}$ as specified in Section 17.15.3, and execute the body of f with said bindings plus a binding of **this** to o. The result returned by f is then the result of evaluating i.

17.21.4 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.

17.22 Property Extraction

Property extraction allows for a member to be accessed as a property rather than a function. A property extraction can be either: ◇

1. An instance method closurization, which converts a method into a function object (17.22.3). Or

2. A getter invocation, which returns the result of invoking of a getter method (17.22.1).

Function objects derived from members via closurization are colloquially known as tear-offs.

Property extraction comes in several forms, as described below.

Case ⟨Conditional⟩**.** Consider a *conditional property extraction expression* i of the form $e\texttt{?.}id$. ◇

If e is a type literal, i is treated as $e\texttt{.}id$.

Otherwise, the static type of i is the same as the static type of $e\texttt{.}id$. Let T be the static type of e, and let y be a fresh variable of type T. Except for errors inside e and references to the name y, exactly the same compile-time errors that would be caused by $y\texttt{.}id$ are also generated in the case of $e\texttt{?.}id$.

Evaluation of a conditional property extraction expression i of the form $e\texttt{?.}id$

proceeds as follows:

If e is a type literal or e denotes an extension, evaluation of i amounts to evaluation of $e.id$.

Otherwise evaluate e to an object o. If o is the null object, i evaluates to the null object (17.4). Otherwise let x be a fresh variable bound to o and evaluate $x.id$ to an object r. Then i evaluates to r. □

Case ⟨Static⟩**.** Let id be an identifier; a *static property extraction* i is an ◇ expression of the form $C.id$, where C is a type literal or C denotes an extension.

A compile-time error occurs unless C denotes a class, a mixin, or an extension that declares a static member named m, which we will call the *denoted member* ◇ of i. If the denoted member is a static getter, the static type of i is the return type of said getter; if the denoted member is a static method, the static type of i is the function type of said method; if the denoted member is neither, a compile-time error occurs.

Evaluation of a static property extraction i of the form $C.id$ proceeds as follows: If the denoted member of i is a static getter, said getter is invoked, yielding an object o, and i then evaluates to o. If the denoted member of i is a static method, i evaluates to a function object by function closurization (17.16) applied to said static method. □

Case ⟨Unconditional⟩**.** Let id be an identifier; an *unconditional property* ◇ *extraction* is an expression of the form $e.id$ where e is an expression that is not a type literal, and does not denote an extension (17.22.1); or it is an expression of the form **super**$.id$ (17.22.2). □

Case ⟨Implicit⟩**.** Let e be an expression whose static type is an interface type that has a method named `call`. In the case where the context type for e is a function type or the type **Function**, e is treated as e`.call`.

This means that a "callable object" may be treated as a function that supports a mechanism similar to function closurization (17.16) by desugaring it to a method closurization on `call`. This only occurs when it is statically known that it is a callable object, and when the context type requires a function. □

17.22.1 Getter Access and Method Extraction

Consider an unconditional property extraction i (17.22) of the form $e.id$. It is a compile-time error if id is the name of an instance member of the built-in class `Object` and e is a type literal.

This means that we cannot use `int.toString` to obtain a function object for the `toString` method of the `Type` object for `int`. But we can use `(int).toString`: e is then not a type literal, but a parenthesized expression.

This is a pragmatic trade-off. The ability to tear off instance methods on instances of `Type` *was considered less useful, and it was considered more useful to insist on the simple rule that a method tear-off on a type literal is* always *a tear-off of a static method on the denoted class.*

Let T be the static type of e. It is a compile-time error if T does not have a method or getter named id unless T is **dynamic** bounded (17.15.3), or T is **Function** bounded and id is `call`. The static type of i is:

- The declared return type of $T.id$, if T has an accessible instance getter named id.
- The function type of the method signature $T.id$, if T has an accessible instance method named id.
- **Function** if T is **Function** bounded and id is `call`.
- The type **dynamic** otherwise. This only occurs when T is **dynamic** bounded.

Note that the type of a method tear-off ignores whether any given parameter is covariant. However, the dynamic type of a function object thus obtained does take parameter covariance into account.

Evaluation of a property extraction i of the form $e.id$ proceeds as follows:

First, the expression e is evaluated to an object o. Let f be the result of looking up (17.18) method (10.2) id in o with respect to the current library L. If method lookup succeeds then i evaluates to the closurization of method f on object o (17.22.3).

Note that f is never an abstract method, because method lookup skips abstract methods. If the method lookup failed, e.g., because there is an abstract declaration of id, but no concrete declaration, we will continue to the next step. However, since methods and getters never override each other, getter lookup will necessarily fail as well, and `noSuchMethod()` will ultimately be invoked. The regrettable implication is that the error will refer to a missing getter rather than an attempt to closurize an abstract method.

Otherwise, i is a getter invocation. Let f be the result of looking up (17.18) getter (10.3) id in o with respect to L. Otherwise, the body of f is executed with **this** bound to o. The value of i 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 class `Invocation` is created, such that:

- im`.isGetter` evaluates to **true**.
- im`.memberName` evaluates to the symbol `m`.
- im`.positionalArguments` evaluates to an object whose dynamic type implements `List<Object>`, and which is empty and unmodifiable.
- im`.namedArguments` evaluates to an object whose dynamic type implements `Map<Symbol, Object>`, and which is empty and unmodifiable.
- im`.typeArguments` evaluates to an object whose dynamic type implements `List<Type>`, and which is empty and unmodifiable.

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.

The situation where `noSuchMethod` is invoked can only arise when the static type of e is **dynamic**.

17.22.2 Super Getter Access and Method Closurization

Consider a property extraction i of the form **super**.id.

Let S be the superclass of the immediately enclosing class. It is a compile-time error if S does not have an accessible instance method or getter named id. The static type of i is:

- The declared return type of $S.id$, if S has an accessible instance getter named id.
- The function type of the method signature $S.id$, if S has an accessible instance method named id.
- The type **dynamic** otherwise. This only occurs when T is **dynamic** or **Function**.

Note that the type of a method tear-off ignores whether any given parameter is covariant. However, the dynamic type of a function object thus obtained does take parameter covariance into account.

Evaluation of a property extraction i of the form **super**.m proceeds as follows:

Let g be the method implementation currently executing, and let C be the class in which g is declared. Let S be the superclass of C. Let f be the result of looking up method id in S with respect to the current library L. If method lookup succeeds then i evaluates to the closurization of method f with respect to superclass S (17.22.4).

Otherwise, i is a getter invocation. Let f be the result of looking up getter id in S with respect to L. The body of f is executed with **this** bound to the current value of **this**. The value of i is the result returned by the call to the getter function.

The getter lookup will not fail, because it is a compile-time error to have a super property extraction of a member id when the superclass S does not have a concrete member named id.

17.22.3 Instance Method Closurization

This section specifies the dynamic semantics of instance method closurizations.

Note that the non-generic case is covered implicitly using $s = 0$, in which case the type parameter declaration lists and the actual type argument lists passed in invocations are omitted (15).

An *instance method closurization* is a closurization of some method on some object, defined below, or a super closurization (17.22.4). ◇

Let o be an object, and let u be a fresh final variable bound to o. The *closurization of method f* on object o is defined to be equivalent (except for equality, as noted below) to: ◇

- <X_1 **extends** B'_1, ..., X_s **extends** B'_s>

$(T_1\ p_1, \ldots,\ T_n\ p_n,\ \{T_{n+1}\ p_{n+1}\ =\ d_1,\ \ldots,\ T_{n+k}\ p_{n+k}\ =\ d_k\})$ `=>`
$u.m\langle X_1, \ldots,\ X_s\rangle(p_1, \ldots,\ p_n,\ p_{n+1}: p_{n+1},\ \ldots,\ p_{n+k}: p_{n+k});$

where f is an instance method named m which has type parameter declarations X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s, required parameters $p_1, \ldots, p_n$, and named parameters $p_{n+1}, \ldots, p_{n+k}$ with defaults $d_1, \ldots, d_k$, using `null` for parameters whose default value is not specified.

- $\langle X_1$ **extends** $B'_1, \ldots,\ X_s$ **extends** $B'_s\rangle$
  $(T_1\ p_1, \ldots,\ T_n\ p_n,\ [T_{n+1}\ p_{n+1}\ =\ d_1,\ \ldots,\ T_{n+k}\ p_{n+k}\ =\ d_k])$ `=>`
  $u.m\langle X_1, \ldots,\ X_s\rangle(p_1, \ldots,\ p_{n+k});$

  where f is an instance method named m which has type parameter declarations X_1 **extends** $B_1, \ldots, X_s$ **extends** B_s, required parameters $p_1, \ldots, p_n$, and optional positional parameters $p_{n+1}, \ldots, p_{n+k}$ with defaults $d_1, \ldots, d_k$, using `null` for parameters whose default value is not specified.

$B'_j, j \in 1..s$, are determined as follows: If o is an instance of a non-generic class, $B'_j = B_j, j \in 1..s$. Otherwise, let $X'_1, \ldots, X'_{s'}$ be the formal type parameters of the class of o, and $t'_1, \ldots, t'_{s'}$ be the actual type arguments. Then $B'_j = [t'_1/X'_1, \ldots, t'_{s'}/X'_{s'}]B_j, j \in 1..s$.

That is, we replace the formal type parameters of the enclosing class, if any, by the corresponding actual type arguments.

The parameter types $T_j, j \in 1..n+k$, are determined as follows: Let the method declaration D be the implementation of m which is invoked by the expression in the body. Let T be the class that contains D.

Note that T is the dynamic type of o, or a superclass thereof.

For each parameter p_j, $j \in 1..n+k$, if p_j is covariant (9.2.3) then T_j is the built-in class `Object`.

This is concerned with the dynamic type of the function object obtained by the member closurization. The static type of the expression that gives rise to the member closurization is specified elsewhere (17.22, 17.22.1). Note that for the static type it is ignored whether a parameter is covariant.

If T is a non-generic class then for $j \in 1..n+k$, T_j is a type annotation that denotes the same type as that which is denoted by the type annotation on the corresponding parameter declaration in D. If that parameter declaration has no type annotation then T_j is **dynamic**.

Otherwise T is a generic instantiation of a generic class G. Let $X''_1, \ldots, X''_{s''}$ be the formal type parameters of G, and $t''_1, \ldots, t''_{s''}$ be the actual type arguments of o at T. Then T_j is a type annotation that denotes $[t''_1/X''_1, \ldots, t''_{s''}/X''_{s''}]S_j$, where S_j is the type annotation of the corresponding parameter in D. If that parameter declaration has no type annotation then T_j is **dynamic**.

There is one way in which the function object yielded by the instance method closurization differs from the function object obtained by function closurization

on the above mentioned function literal: Assume that o_1 and o_2 are objects, m is an identifier, and c_1 and c_2 are function objects obtained by closurization of m on o_1 respectively o_2. Then c_1 `==` c_2 evaluates to true if and only if o_1 and o_2 is the same object.

In particular, two closurizations of a method m from the same object are equal, and two closurizations of a method m from non-identical objects are not equal. Assuming that v_i is a fresh variable bound to an object, $i \in 1..2$, it also follows that `identical(`$v_1.m, v_2.m$`)` must be false when v_1 and v_2 are not bound to the same object. However, Dart implementations are not required to canonicalize function objects, which means that `identical(`$v_1.m, v_2.m$`)` is not guaranteed to be true, even when it is known that v_1 and v_2 are bound to the same object.

The special treatment of equality in this case facilitates the use of extracted property functions in APIs where callbacks such as event listeners must often be registered and later unregistered. A common example is the DOM API in web browsers.

17.22.4 Super Closurization

This section specifies the dynamic semantics of super closurizations.

Note that the non-generic case is covered implicitly using $s = 0$, in which case the type parameter declarations are omitted (15).

Consider expressions in the body of a class T which is a subclass of a given class S, where a method declaration that implements f exists in S, and there is no class U which is a subclass of S and a superclass of T which implements f.

In short, consider a situation where a superinvocation of f will execute f as declared in S.

A *super closurization* is a closurization of a method with respect to a class, ◇
as defined next. The *closurization of a method f* with respect to the class S is ◇
defined to be equivalent (except for equality, as noted below) to:

- `<`X_1 **extends** B'_1, ..., X_s **extends** B'_s`>`
  `(`T_1 p_1, ..., T_n p_n, `{`T_{n+1} p_{n+1} `=` d_1, ..., T_{n+k} p_{n+k} `=` d_k`}) =>`
  super`.`m`<`X_1, ..., X_s`>(`p_1, ..., p_n, p_{n+1}`:` p_{n+1}, ..., p_{n+k}`:` p_{n+k}`);`

  where f is an instance method named m which has type parameter declarations X_1 **extends** B_1, ..., X_s **extends** B_s, required parameters p_1, ..., p_n, and named parameters p_{n+1}, ..., p_{n+k} with defaults d_1, ..., d_k.

- `<`X_1 **extends** B'_1, ..., X_s **extends** B'_s`>`
  `(`T_1 p_1, ..., T_n p_n, `[`T_{n+1} p_{n+1} `=` d_1, ..., T_{n+k} p_{n+k} `=` d_k`]) =>`
  super`.`m`<`X_1, ..., X_s`>(`p_1, ..., p_{n+k}`);`

where f is an instance method named m which has type parameter declarations X_1 **extends** B_1, ..., X_s **extends** B_s, required parameters p_1, ..., p_n, and optional positional parameters p_{n+1}, ..., p_{n+k} with defaults d_1, ..., d_k.

Note that a super closurization is an instance method closurization, as defined in (17.22.3).

$B'_j, j \in 1..s$, are determined as follows: If S is a non-generic class then $B'_j = B_j, j \in 1..s$. Otherwise, let $X'_1, \ldots, X'_{s'}$ be the formal type parameters of S, and $t'_1, \ldots, t'_{s'}$ be the actual type arguments of **this** at S. Then $B'_j = [t'_1/X'_1, \ldots, t'_{s'}/X'_{s'}]B_j, j \in 1..s$.

That is, we replace the formal type parameters of the enclosing class, if any, by the corresponding actual type arguments. We need to consider the type arguments with respect to a specific class because it is possible for a class to pass different type arguments to its superclass than the ones it receives itself.

The parameter types $T_j, j \in 1..n+k$, are determined as follows: Let the method declaration D be the implementation of m in S.

For each parameter p_j, $j \in 1..n+k$, if p_j is covariant (9.2.3) then T_j is the built-in class `Object`.

This is concerned with the dynamic type of the function object obtained by the super closurization. The static type of the expression that gives rise to the super closurization is specified elsewhere (17.22, 17.22.2). Note that for the static type it is ignored whether a parameter is covariant.

If S is a non-generic class then for $j \in 1..n+k$, T_j is a type annotation that denotes the same type as that which is denoted by the type annotation on the corresponding parameter declaration in D. If that parameter declaration has no type annotation then T_j is **dynamic**.

Otherwise S is a generic instantiation of a generic class G. Let $X''_1, \ldots, X''_{s''}$ be the formal type parameters of G, and $t''_1, \ldots, t''_{s''}$ be the actual type arguments of o at S. Then T_j is a type annotation that denotes $[t''_1/X''_1, \ldots, t''_{s''}/X''_{s''}]S_j$, where S_j is the type annotation of the corresponding parameter in D. If that parameter declaration has no type annotation then T_j is **dynamic**.

There is one way in which the function object yielded by the super closurization differs from the function object obtained by function closurization on the above mentioned function literal: Assume that an occurrence of the expression **super**.m in a given class is evaluated on two occasions where **this** is bound to o_1 respectively o_2, and the resulting function objects are c_1 respectively c_2: c_1 `==` c_2 is then true if and only if o_1 and o_2 is the same object.

17.22.5 Generic Method Instantiation

Generic method instantiation is a mechanism that yields a non-generic function object, based on a property extraction which denotes an instance method closurization (17.22.3, 17.22.4).

It is a mechanism which is very similar to instance method closurization, but it only occurs in situations where a compile-time error would otherwise occur.

The essence of generic method instantiation is to allow for "curried" invocations, in the sense that a generic instance method can receive its actual type

arguments separately during closurization (it must then receive all *type arguments, not just some of them), and that yields a non-generic function object. The type arguments are passed implicitly, based on type inference; a future version of Dart may allow for passing them explicitly.* Here is an example:

```
class A {
  X fi<X extends num>(X x) => x;
}

class B extends /* or implements */ A {
  X fi<X extends num>(X x, [List<X> xs]) => x;
}

void main() {
  A a = B();
  int Function(int) f = a.fi;
}
```

The function object which is stored in `f` at the end of `main` has dynamic type `int Function(int, [List<int>])`, and it is obtained by implicitly "passing the actual type argument `int`" to the denoted generic instance method, thus obtaining a non-generic function object of the specified type. Note that this function object accepts an optional positional argument, even though this is not part of the statically known type of the corresponding instance method, nor of the context type.

In other words, generic method instantiation yields a function whose signature matches the context type as far as possible, but with respect to its parameter list shape (that is, the number of positional parameters and their optionality, or the set of names of named parameters), it will be determined by the method signature of the actual instance method of the given receiver. Of course, the difference can only be such that the actual type is a subtype of the given context type, otherwise the declaration of that instance method would have been a compile-time error.

Let i be a property extraction expression of the form e`?.` id, e`.` id, or **super**`.` id (17.22, 17.22.2), which is statically resolved to denote an instance method named id, and let G be the static type of i. Consider the situation where G is a function type of the form T_0 **Function**`<`$X_1 \triangleleft B_1, \ldots, X_s \triangleleft B_s$`>(`*parameters*`)` with $s > 0$ (that is, G is a generic function type), and the context type is a non-generic function type F. In this situation a compile-time error occurs (8, 9, 10.7.1, 10.7.1, 10.7.1, 17.13.1, 17.13.2, 17.15.3, 17.23, 18.3, 18.9, 18.12, 18.17), except when generic function type instantiation (17.17) succeeds, that is:

Type inference is applied to G with context type F, and it succeeds, yielding the actual type argument list $T_1, \ldots, T_s$.

Consider the situation where generic function type instantiation succeeded. Let $gmiName_{id}$ be a fresh name which is associated with id, which is private if and only if id is private. An implementation could use, say, `foo_*` when id is

`foo`, which is known to be fresh because user-written identifiers cannot contain '`*`'. The program is then modified as follows:

- When i is e`?.`id: Replace i by e`?.`$gmiName_{id}$`<`T_1`, ..., `T_s`>()`.
- When i is e`.`id: Replace i by e`.`$gmiName_{id}$`<`T_1`, ..., `T_s`>()`.
- When i is **super**`.`id: Replace i by **super**`.`$gmiName_{id}$`<`T_1`, ..., `T_s`>()`.

The inserted expressions have no compile-time error and can be executed, because the corresponding generic method is induced implicitly. We use the phrase *generic instantiation method* to denote these implicitly induced methods, and designate the method that induced it as its *target*. ◇ ◇

Assume that a class C declares a generic instance method named id, with a method signature corresponding to a generic function type G, formal type parameters X_1 **extends** B_1, ..., X_s **extends** B_s, and formal parameter declarations *parameters*. Let *arguments* denote the corresponding actual argument list, passing these parameters. For instance, *parameters* could be
$T_1\ p_1, \ldots,\ T_n\ p_n,\ \{T_{n+1}\ p_{n+1} = d_1, \ldots,\ T_{n+k}\ p_{n+k} = d_k\}$
in which case *arguments* would be $p_1, \ldots,\ p_n$`,` p_{n+1}`:` p_{n+1}`,` p_{n+k}`:` p_{n+k}.

Let G' be the same function type as G, except that it omits the formal type parameter declarations. For instance, if G is **void Function**`<`X`,` Y **extends** `num>(X x, List<Y> ys)` then G' is **void Function**`(X x, List<Y> ys)`. Note that G' will typically contain free type variables.

An instance method with the name $gmiName_{id}$ is then implicitly induced, with the same behavior as the following declaration (except for equality of the returned function object, which is specified below):

```
G' gmiName_id<X_1 extends B_1, ..., X_s extends B_s>() {
  return (parameters) => this.id<X_1, ..., X_s>(arguments);
}
```

Let o be an instance of a class which contains an implicitly induced declaration of $gmiName_{id}$ as described above. Consider the situation where the program evaluates two invocations of this method with the same receiver o, and with actual type arguments whose actual values are the same types $t_1, \ldots, t_s$ for both invocations, and assume that the invocations returned the instances o_1 respectively o_2. It is then guaranteed that o_1 and o_2 are equal according to operator '`==`'. It is unspecified whether `identical(`o_1`,` o_2`)` evaluates to **true** or **false**.

No notion of equality is appropriate with different receivers, nor when different type arguments are provided, because execution of two function objects that differ in these ways can have different side-effects and return different results when executed starting from exactly the same state.

17.23 Assignment

An assignment changes the value associated with a mutable variable, or invokes a setter.

⟨*assignmentOperator*⟩ ::= '`=`'
  | ⟨*compoundAssignmentOperator*⟩

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

Case ⟨*id* `=` *e*⟩**.** Consider an assignment a of the form id `=` e, where id is an identifier. Perform a lexical lookup of id`=` from the location of id.

- When the lexical lookup yields a declaration D of a local variable v (which may be a formal parameter), a compile-time error occurs if v is final or if the static type of e is not assignable to the declared type of v.
- When the lexical lookup yields a declaration D which is not a local variable, it is guaranteed to be a setter (that may be explicit or induced implicitly by a variable) because other declarations do not have a name of the form id`=`.

  If D is the declaration of a static setter in class or mixin C then a is treated as (5) the assignment C`.`id `=` e.

  Further analysis as well as evaluation of C`.`id `=` e proceeds as specified elsewhere.

  Otherwise, a compile-time error occurs, unless the static type of e is assignable to the parameter type of D.
- When the lexical lookup yields nothing, a is treated as (5) **this**`.`id `=` e.

  In this case it is known that a has access to **this** (10), and the interface of the enclosing class has a member named id`=`, or there is an applicable extension with a member named id`=`. Both the static analysis and evaluation proceeds with **this**`.`id `=` e, so there is no need to further specify the treatment of a.
- The lexical lookup can never yield an import prefix, because they never have a name of the form id`=`.

In all cases (whether or not id is a local variable, etc.), the static type of a is the static type of e.

Evaluation of an assignment a of the form id `=` e proceeds as follows. Perform a lexical lookup of id`=` from the location of id.

- In the case where the lexical lookup yields a declaration D of a local variable v, (which may be a formal parameter), the expression e is evaluated to an object o, and the variable v is bound to o. Then a evaluates to the object o (17.1).
- In the case where the lexical lookup of id`=` from the location of id yields a declaration D, D is necessarily a top level setter s (possibly implicitly induced by a variable).

  The expression e is evaluated to an object o. Then the setter s is invoked with its formal parameter bound to o. Then a evaluates to the object o.

  D cannot be a static setter in a class C, because a is then treated as C`.`id `=` e, which is specified elsewhere.

- The case where the lexical lookup of id`=` yields nothing cannot occur, because that case is treated as **this**`.`id `=` e, whose evaluation is specified elsewhere.

□

Case ⟨p`.`id `=` e⟩**.** Consider an assignment a of the form p`.`id `=` e, where p is an import prefix and id is an identifier.

A compile-time error occurs, unless p has a member which is a setter s named `id=` (which may be implicitly induced by a variable declaration) such that the static type of e is assignable to the parameter type of s.

The static type of a is the static type of e.

Evaluation of an assignment a of the form p`.`id `=` e proceeds as follows: The expression e is evaluated to an object o. Then the setter denoted by p`.`id is invoked with its formal parameter bound to o. Then a evaluates to the object o. □

Case ⟨e_1`?.`v `=` e_2⟩**.** Consider an assignment a of the form e_1`?.`v `=` e_2. Exactly the same compile-time errors that would be caused by e_1`.`v `=` e_2 are also generated in the case of a. The static type of a is the static type of e_2.

Evaluation of an assignment a of the form e_1`?.`v `=` e_2 proceeds as follows: If e_1 is a type literal, a is equivalent to e_1`.`v `=` e_2. Otherwise evaluate e_1 to an object o. If o is the null object, a evaluates to the null object (17.4). Otherwise let x be a fresh variable bound to o and evaluate x`.`v `=` e_2 to an object r. Then a evaluates to r. □

Case ⟨e_1`.`v `=` e_2⟩**.** Consider an assignment a of the form e_1`.`v `=` e_2. Let T be the static type of e_1. If T is **dynamic**, no further checks are performed. Otherwise, it is a compile-time error unless T has an accessible instance setter named v`=`. It is a compile-time error unless the static type of e_2 may be assigned to the declared type of the formal parameter of said setter. Whether or not T is **dynamic**, the static type of a is the static type of e_2.

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 (17.18) in o_1 with respect to the current library. It is a dynamic type error if the dynamic type of o_2 is not a subtype of the actual parameter type of said setter (20.10.1). Otherwise, the body of the setter 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 class `Invocation` is created, such that:

- im`.isSetter` evaluates to **true**.
- im`.memberName` evaluates to the symbol `v=`.
- im`.positionalArguments` evaluates to an object whose dynamic type implements `List<Object>`, which is unmodifiable, and which contains the same objects as `<Object>`$[o_2]$.
- im`.namedArguments` evaluates to an object whose dynamic type implements `Map<Symbol, Object>`, and which is empty and unmodifiable.
- im`.typeArguments` evaluates to an object whose dynamic type implements `List<Type>`, and which is empty and unmodifiable.

Then the method `noSuchMethod()` is looked up in o_1 and invoked with argument im.

The situation where `noSuchMethod` is invoked can only arise when the static type of e_1 is **dynamic**.

The value of the assignment expression is o_2 irrespective of whether setter lookup has failed or succeeded. □

Case ⟨**super**`.`v `=` e⟩**.** Consider an assignment a of the form **super**`.`v `=` e. Let S_{static} be the superclass of the immediately enclosing class. It is a compile-time error if S_{static} does not have a concrete accessible instance setter named v`=`. Otherwise, it is a compile-time error if the static type of e may not be assigned to the static type of the formal parameter of said setter. The static type of a is the static type of e.

Evaluation of an assignment of the form **super**`.`v `=` e proceeds as follows: Let g be the currently executing method, and let C be the class in which g was looked up. Let $S_{dynamic}$ be the superclass of C. The expression e is evaluated to an object o. Then, the setter v`=` is looked up (17.18) in $S_{dynamic}$ with respect to the current library. The body of v`=` is executed with its formal parameter bound to o and **this** bound to the current value of **this**.

The setter lookup will not fail, because it is a compile-time error when no concrete setter named v`=` exists in S_{static}.

The value of the assignment expression is o.

It is a dynamic type error if o is not the null object (17.4) and the dynamic type of o is not a subtype of the actual type of the formal parameter of v`=` (20.10.1) in S_{static}. □

Case $\langle e_1\texttt{[}e_2\texttt{]} \texttt{ = } e_3\rangle$**.** Consider an assignment a of the form $e_1\texttt{[}e_2\texttt{]} \texttt{ = } e_3$. Let T be the static type of e_1. If T is **dynamic**, no further checks are performed. Otherwise, it is a compile-time error unless T has a method named `[]=`. Let S_2 be the static type of the first formal parameter of the method `[]=`, and S_3 the static type of the second. It is a compile-time error unless the static type of e_2 respectively e_3 may be assigned to S_2 respectively S_3. Whether or not T is **dynamic**, the static type of a is the static type of e_3.

Evaluation of an assignment a of the form $e_1\texttt{[}e_2\texttt{]} \texttt{ = } e_3$ proceeds as follows: Evaluate e_1 to an object o, then evaluate e_2 to an object i, and finally evaluate e_3 to an object v. Call the method `[]=` on o with i as first argument and v as second argument. Then a evaluates to v. □

Case $\langle$**super**`[`e_1`] = `$e_2\rangle$**.** Consider an assignment a of the form **super**`[`e_1`] = `e_2. Let S_{static} be the superclass of the immediately enclosing class. It is a compile-time error if S_{static} does not have a method `[]=`. Otherwise, let S_1 be the static type of the first formal parameter of the method `[]=`, and S_2 the static type of the second. It is a compile-time error if the static type of e_1 respectively e_2 may not be assigned to S_1 respectively S_2. The static type of a is the static type of e_2.

For evaluation, an assignment of the form **super**`[`e_1`] = `e_2 is equivalent to the expression **super**`.[`e_1`] = `e_2. □

17.23.1 Compound Assignment

Case $\langle v \texttt{ ??= } e\rangle$**.** Consider a compound assignment a of the form $v \texttt{ ??= } e$ where v is an identifier or an identifier qualified by an import prefix. Exactly the same compile-time errors that would be caused by $v \texttt{ = } e$ are also generated in the case of a. The static type of a is the least upper bound of the static type of v and the static type of e.

Evaluation of a compound assignment a of the form $v \texttt{ ??= } e$ proceeds as follows: Evaluate v to an object o. If o is not the null object (17.4), a evaluates to o. Otherwise evaluate $v \texttt{ = } e$ to an object r, and then a evaluates to r. □

Case $\langle C\texttt{.}v \texttt{ ??= } e\rangle$**.** Consider a compound assignment a of the form $C\texttt{.}v \texttt{ ??= } e$ where C is a type literal that may or may not be qualified by an import prefix. Exactly the same compile-time errors that would be caused by $C\texttt{.}v \texttt{ = } e$ are also generated in the case of a. The static type of a is the least upper bound of the static type of $C\texttt{.}v$ and the static type of e.

Evaluation of a compound assignment a of the form $C\texttt{.}v \texttt{ ??= } e$ where C is a type literal proceeds as follow: Evaluate $C\texttt{.}v$ to an object o. If o is not the null object (17.4), a evaluates to o. Otherwise evaluate $C\texttt{.}v \texttt{ = } e$ to an object r, and then a evaluates to r. □

Case $\langle e_1\texttt{.}v \texttt{ ??= } e_2\rangle$**.** Consider a compound assignment a of the form $e_1\texttt{.}v \texttt{ ??= } e_2$. Let T be the static type of e_1 and let x be a fresh variable of type T. Except for errors inside e_1 and references to the name x, exactly the same compile-time errors that would be caused by $x\texttt{.}v \texttt{ = } e_2$ are also generated in the case of a. Moreover, it is a compile-time error if T does not have a getter named

v. The static type of a is the least upper bound of the static type of $e_1.v$ and the static type of e_2.

Evaluation of a compound assignment a of the form $e_1.v$ `??=` e_2 proceeds as follows: Evaluate e_1 to an object u. Let x be a fresh variable bound to u. Evaluate $x.v$ to an object o. If o is not the null object (17.4), a evaluates to o. Otherwise evaluate $x.v$ `=` e_2 to an object r, and then a evaluates to r. □

Case $\langle e_1[e_2]$ `??=` $e_3\rangle$**.** Consider a compound assignment a of the form $e_1[e_2]$ `??=` e_3. Exactly the same compile-time errors that would be caused by $e_1[e_2]$ `=` e_3 are also generated in the case of a. Moreover, it is a compile-time error if the static type of e_1 does not have an '`operator []`'. The static type of a is the least upper bound of the static type of $e_1[e_2]$ and the static type of e_3.

Evaluation of a compound assignment a of the form $e_1[e_2]$ `??=` e_3 proceeds as follows: Evaluate e_1 to an object u and then evaluate e_2 to an object i. Call the `[]` method on u with argument i, and let o be the returned object. If o is not the null object (17.4), a evaluates to o. Otherwise evaluate e_3 to an object v and then call the `[]=` method on u with i as first argument and v as second argument. Then a evaluates to v. □

Case $\langle$**super**$.v$ `??=` $e\rangle$**.** Consider a compound assignment a of the form **super**$.v$ `??=` e. Exactly the same compile-time errors that would be caused by **super**$.v$ `=` e are also generated in the case of a. Moreover, exactly the same compile-time errors that would be caused by evaluation of the expression **super**$.v$ are also generated in the case of a. The static type of a is the least upper bound of the static type of **super**$.v$ and the static type of e.

Evaluation of a compound assignment a of the form **super**$.v$ `??=` e proceeds as follows: Evaluate **super**$.v$ to an object o. If o is not the null object (17.4) then a evaluates to o. Otherwise evaluate **super**$.v$ `=` e to an object r, and then a evaluates to r. □

Case $\langle e_1$`?.`v `??=` $e_2\rangle$**.** Consider a compound assignment a of the form e_1`?.`v `??=` e_2. Exactly the same compile-time errors that would be caused by $e_1.v$ `??=` e_2 are also generated in the case of a. The static type of a is the least upper bound of the static type of e_1`?.`v and the static type of e_2.

Evaluation of a compound assignment a of the form e_1`?.`v `??=` e_2 proceeds as follows: Evaluate e_1 to an object u. If u is the null object (17.4) then a evaluates to the null object. Otherwise, let x be a fresh variable bound to u. Evaluate $x.v$ to an object o. If o is not the null object (17.4) then a evaluates to o. Otherwise evaluate $x.v$ `=` e_2 to an object r, and then a evaluates to r. □

Case $\langle C$`?.`v `??=` $e_2\rangle$**.** A compound assignment of the form C`?.`v `??=` e_2 where C is a type literal that may or may not be qualified by an import prefix is equivalent to the expression $C.v$ `??=` e. □

Case $\langle v$ *op*`=` $e\rangle$**.** For any other valid operator *op*, a compound assignment of the form v *op*`=` e is equivalent to v `=` v *op* e, where v is an identifier or an identifier qualified by an import prefix. □

Case $\langle C.v$ *op*`=` $e\rangle$**.** A compound assignment of the form $C.v$ *op*`=` e where C is a type literal that may or may not be qualified by an import prefix is equivalent to $C.v$ `=` $C.v$ *op* e. □

Case $\langle e_1.v$ *op*`=` $e_2\rangle$**.** Consider a compound assignment a of the form $e_1.v$

op= e_2. Let x be a fresh variable whose static type is the static type of e_1. Except for errors inside e_1 and references to the name x, exactly the same compile-time errors that would be caused by x `.`v `=` x `.`v *op* e_2 are also generated in the case of a. The static type of a is the static type of e_1 `.`v *op* e_2.

Evaluation of a compound assignment a of the form e_1 `.`v *op*= e_2 proceeds as follows: Evaluate e_1 to an object u and let x be a fresh variable bound to u. Evaluate x `.`v `=` x `.`v *op* e_2 to an object r and then a evaluates to r. □

Case ⟨e_1`[`e_2`]` *op*= e_3⟩**.** Consider a compound assignment a of the form e_1`[`e_2`]` *op*= e_3. Let x and i be fresh variables where the static type of the former is the static type of e_1 and the static type of the latter is the static type of e_2. Except for errors inside e_1 and e_2 and references to the names x and i, exactly the same compile-time errors that would be caused by x`[`i`]` `=` x`[`i`]` *op* e_3 are also generated in the case of a. The static type of a is the static type of x`[`i`]` *op* e_3.

Evaluation of s compound assignment a of the form e_1`[`e_2`]` *op*= e_3 proceeds as follows: Evaluate e_1 to an object u and evaluate e_2 to an object v. Let x and i be fresh variables bound to u and v respectively. Evaluate x`[`i`]` `=` x`[`i`]` *op* e_3 to an object r, and then a evaluates to r. □

Case ⟨e_1`?.`v *op*= e_2⟩**.** Consider a compound assignment a of the form e_1`?.`v *op*= e_2. Exactly the same compile-time errors that would be caused by e_1 `.`v *op*= e_2 are also generated in the case of a. The static type of a is the static type of e_1 `.`v *op*= e_2.

Evaluation of a compound assignment a of the form e_1`?.`v *op*= e_2 proceeds as follows: Evaluate e_1 to an object u. If u is the null object, then a evaluates to the null object (17.4). Otherwise let x be a fresh variable bound to u. Evaluate x `.`v *op*= e_2 to an object r. Then a evaluates to r. □

Case ⟨C`?.`v *op* `=` e_2⟩**.** A compound assignment of the form C`?.`v *op* `=` e_2 where C is a type literal is equivalent to the expression C `.`v *op* `=` e_2. □

17.24 Conditional

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

⟨*conditionalExpression*⟩ ::= ⟨*ifNullExpression*⟩
('`?`' ⟨*expressionWithoutCascade*⟩ '`:`' ⟨*expressionWithoutCascade*⟩)?

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. It is a dynamic error if the run-time type of o_1 is not `bool`. 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.

If e_1 shows that a local variable v has type T, then the type of v is known to be T in e_2, unless any of the following are true:

- v is potentially mutated in e_2,

- v is potentially mutated within a function other than the one where v is declared, or
- v is accessed by a function defined in e_2 and v is potentially mutated anywhere in the scope of v.

It is a compile-time error if the static type of e_1 may not be assigned to `bool`. The static type of c is the least upper bound (20.10.2) of the static type of e_2 and the static type of e_3.

17.25 If-null Expressions

An *if-null expression* evaluates an expression and if the result is the null object (17.4), evaluates another. ◇

⟨*ifNullExpression*⟩ ::= ⟨*logicalOrExpression*⟩ (‘`??`’ ⟨*logicalOrExpression*⟩)*

Evaluation of an if-null expression e of the form e_1 `??` e_2 proceeds as follows:

Evaluate e_1 to an object o. If o is not the null object (17.4), then e evaluates to o. Otherwise evaluate e_2 to an object r, and then e evaluates to r.

The static type of e is the least upper bound (20.10.2) of the static type of e_1 and the static type of e_2.

17.26 Logical Boolean Expressions

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

⟨*logicalOrExpression*⟩ ::=
⟨*logicalAndExpression*⟩ (‘`||`’ ⟨*logicalAndExpression*⟩)*

⟨*logicalAndExpression*⟩ ::= ⟨*equalityExpression*⟩ (‘`&&`’ ⟨*equalityExpression*⟩)*

A *logical boolean expression* is either an equality expression (17.27), 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 to an object o_1. It is a dynamic error if the run-time type of o_1 is not `bool`. If o_1 is **true**, the result of evaluating b is **true**, otherwise e_2 is evaluated to an object o_2. It is a dynamic error if the run-time type of o_2 is not `bool`. Otherwise the result of evaluating b is o_2.

Evaluation of a logical boolean expression b of the form $e_1\&\&e_2$ causes the evaluation of e_1 producing an object o_1. It is a dynamic error if the run-time type of o_1 is not `bool`. If o_1 is **false**, the result of evaluating b is **false**, otherwise e_2 is evaluated to an object o_2. It is a dynamic error if the run-time type of o_2 is not `bool`. Otherwise the result of evaluating b is o_2.

A logical boolean expression b of the form $e_1\&\&e_2$ shows that a local variable v has type T if both of the following conditions hold:

- Either e_1 shows that v has type T or e_2 shows that v has type T.
- v is not mutated in e_2 or within a function other than the one where v is declared.

If e_1 shows that a local variable v has type T, then the type of v is known to be T in e_2, unless any of the following are true:

- v is potentially mutated in e_1,
- v is potentially mutated in e_2,
- v is potentially mutated within a function other than the one where v is declared, or
- v is accessed by a function defined in e_2 and v is potentially mutated anywhere in the scope of v.

It is a compile-time error if the static type of e_1 may not be assigned to `bool` or if the static type of e_2 may not be assigned to `bool`. The static type of a logical boolean expression is `bool`.

17.27 Equality

Equality expressions test objects for equality.

⟨*equalityExpression*⟩ ::=
    ⟨*relationalExpression*⟩ (⟨*equalityOperator*⟩ ⟨*relationalExpression*⟩)?
  | **super** ⟨*equalityOperator*⟩ ⟨*relationalExpression*⟩

⟨*equalityOperator*⟩ ::= '`==`'
  | '`!=`'

An *equality expression* is either a relational expression (17.28), or an invocation of an equality operator on either **super** or an expression e_1, with argument e_2. ◇

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.
- The expression e_2 is evaluated to an object o_2.
- If either o_1 or o_2 is the null object (17.4), then ee evaluates to **true** if both o_1 and o_2 are the null object and to **false** otherwise. Otherwise,
- evaluation of ee is equivalent to the method invocation o_1`.==(`o_2`)`.

Evaluation of an equality expression ee of the form **super** `==` e proceeds as follows:

- The expression e is evaluated to an object o.
- If either **this** or o is the null object (17.4), then ee evaluates to evaluates to **true** if both **this** and o are the null object and to **false** otherwise. Otherwise,
- evaluation of ee is equivalent to the method invocation **super**.==(o).

As a result of the above definition, user defined '==' methods can assume that their argument is non-null, and avoid the standard boiler-plate prelude:

```
if (identical(null, arg)) return false;
```

Another implication is that there is never a need to use `identical()` to test against **null**, nor should anyone ever worry about whether to write **null** == e or e == **null**.

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`)`.

The static type of an equality expression is `bool`.

17.28 Relational Expressions

Relational expressions invoke the relational operators on objects.

⟨*relationalExpression*⟩ ::= ⟨*bitwiseOrExpression*⟩
  (⟨*typeTest*⟩ | ⟨*typeCast*⟩ | ⟨*relationalOperator*⟩ ⟨*bitwiseOrExpression*⟩)?
| **super** ⟨*relationalOperator*⟩ ⟨*bitwiseOrExpression*⟩

⟨*relationalOperator*⟩ ::= '>='
| '>'
| '<='
| '<'

A *relational expression* is either a bitwise expression (17.29), 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`)`.

17.29 Bitwise Expressions

Bitwise expressions invoke the bitwise operators on objects.

⟨*bitwiseOrExpression*⟩ ::=
  ⟨*bitwiseXorExpression*⟩ ('|' ⟨*bitwiseXorExpression*⟩)*
| **super** ('|' ⟨*bitwiseXorExpression*⟩)+

⟨*bitwiseXorExpression*⟩ ::=
    ⟨*bitwiseAndExpression*⟩ (‘^’ ⟨*bitwiseAndExpression*⟩)*
  | **super** (‘^’ ⟨*bitwiseAndExpression*⟩)+

⟨*bitwiseAndExpression*⟩ ::= ⟨*shiftExpression*⟩ (‘&’ ⟨*shiftExpression*⟩)*
  | **super** (‘&’ ⟨*shiftExpression*⟩)+

⟨*bitwiseOperator*⟩ ::= ‘&’
  | ‘^’
  | ‘|’

A *bitwise expression* is either a shift expression (17.30), 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.

17.30 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 (17.31), 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.

17.31 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 (17.32), 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`)`.

The static type of an additive expression is usually determined by the signature given in the declaration of the operator used. However, invocations of the operators `+` and `-` of class `int`, `double` and `num` are treated specially by the typechecker.

Let e be an additive expression of the form e_1 *op* e_2, let T be the static type of e_1, and let C be the context type of e.

If $T <:$ `num` and not $T <:$ `Never`, then the context type of e_2 is determined as follows:

- If `int` $<: C$ and not `num` $<: C$, and $T <:$ `int` then the context type of e_2 is `int`.
- If `double` $<: C$ and not `num` $<: C$, and not $T <:$ `double` then the context type of e_2 is `double`.
- Otherwise the context type of e_2 is `num`.

Let further S be the static type of e_2. If $T <:$ `num` and not $T <:$ `Never` and S is assignable to `num`, then the static type of e is determined as follows:

- If $T <:$ `double` then the static type of e is T.
- Otherwise, if $S <:$ `double` and not $S <:$ `Never`, then the static type of e is `double`.
- Otherwise, if $T <:$ `int`, $S <:$ `int` and not $S <:$ `Never`, then the static type of e is `int`.
- Otherwise the static type of e is `num`.

17.32 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 (17.33), 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`)`.

The static type of an multiplicative expression is usually determined by the signature given in the declaration of the operator used. However, invocations of the operators `*` and `%` of class `int`, `double` and `num` are treated specially by the typechecker.

Let e be a multiplicative expression of the form e_1 *op* e_2 where *op* is one of `*` or `%`, let T be the static type of e_1, and let C be the context type of e.

If T <: `num` and not T <: `Never`, then the context type of e_2 is determined as follows:

- If `int` <: C and not `num` <: C, and T <: `int` then the context type of e_2 is `int`.
- If `double` <: C and not `num` <: C, and not T <: `double` then the context type of e_2 is `double`.
- Otherwise the context type of e_2 is `num`.

Let further S be the static type of e_2. If T <: `num` and not T <: `Never` and S is assignable to `num`, then the static type of e is determined as follows:

- If T <: `double` then the static type of e is T.
- Otherwise, if S <: `double` and not S <: `Never`, then the static type of e is `double`.
- Otherwise, if T <: `int`, S <: `int` and not S <: `Never`, then the static type of e is `int`.
- Otherwise the static type of e is `num`.

17.33 Unary Expressions

Unary expressions invoke unary operators on objects.

⟨*unaryExpression*⟩ ::= ⟨*prefixOperator*⟩ ⟨*unaryExpression*⟩
| ⟨*awaitExpression*⟩
| ⟨*postfixExpression*⟩
| (⟨*minusOperator*⟩ | ⟨*tildeOperator*⟩) **super**
| ⟨*incrementOperator*⟩ ⟨*assignableExpression*⟩

⟨*prefixOperator*⟩ ::= ⟨*minusOperator*⟩
| ⟨*negationOperator*⟩
| ⟨*tildeOperator*⟩

⟨*minusOperator*⟩ ::= '`-`'

⟨*negationOperator*⟩ ::= '`!`'

⟨*tildeOperator*⟩ ::= '`~`'

A *unary expression* is either a postfix expression (17.35), an await expression (17.34) or an invocation of a prefix operator on an expression or an invocation of a unary operator on either **super** or an expression e. ◇

The expression !e is treated as (5) `(`e `?` **false** `:` **true**`)`.

An expression of the form `++`e is treated as `(`e `+= 1)`. An expression of the form `--`e is treated as `(`e `-= 1)`.

Let e be an expression of the form `-`l where l is an integer literal (17.5) with numeric integer value i, and with static context type T. If `double` is assignable to T and `int` is not assignable to T, then the static type of e is `double`; otherwise the static type of e is `int`.

If the static type of e is `int` then e evaluates to an instance of the `int` class representing the numeric value $-i$. If i is zero and the `int` class can represent a negative zero value, then the resulting instance instead represents that negative zero value. It is a compile-time error if the integer $-i$ cannot be represented exactly by an instance of `int`.

If the static type of e is `double` then e evaluates to to an instance of the `double` class representing the numeric value $-i$. If i is zero, the resulting instance instead represents the *negative* zero double value, `-0.0`. It is a compile-time error if the integer $-i$ cannot be represented exactly by an instance of `double`. We treat *-l as if* it is a single integer literal with a negative numeric value. We do not evaluate l individually as an expression, or concern ourselves with its static type.

Any other expression of the form *op* e is equivalent to the method invocation e.*op*(). An expression of the form *op* **super** is equivalent to the method invocation (17.21.3) **super**.*op*().

17.34 Await Expressions

An *await expression* allows code to yield control until an asynchronous operation (9) completes. ◇

⟨*awaitExpression*⟩ ::= **await** ⟨*unaryExpression*⟩

Let a be an expression of the form **await** e. Let S be the static type of e. The static type of a is then *flatten*(S) (17.11). a, e, S

Evaluation of a proceeds as follows: First, the expression e is evaluated to an object o. Let T be *flatten*(S). If the run-time type of o is a subtype of T

`Future<`T`>`, then let f be o; otherwise let f be the result of creating a new object using the constructor `Future<`T`>.value()` with o as its argument. f

Next, the stream associated with the innermost enclosing asynchronous **for** loop (18.6.3), if any, is paused. The current invocation of the function body immediately enclosing a is suspended until after f completes. At some time after f is completed, control returns to the current invocation. If f has completed with an error x and stack trace t, a throws x and t (17.1). If f completes with an object v, a evaluates to v.

The use of flatten to find T and hence determine the dynamic type test implies that we await a future in every case where this choice is sound.

An interesting case on the edge of this trade-off is when e has the static type `FutureOr<Object>?`. You could say that the intention behind this type is that the value of e is a `Future<Object>`, or it is an `Object` which is not a future, or it is **null**. So, presumably, we should await the first kind, and we should pass on the second and third kind unchanged. However, the second kind could be a `Future<Object?>`. This object isn't a `Future<Object>`, and it isn't **null**, so it *must* be considered to be in the second group. Nevertheless, *flatten*(`FutureOr<Object>?`) is `Object?`, so we *will* await a `Future<Object?>`. We have chosen this semantics because it was the smallest breaking change relative to the semantics in earlier versions of Dart, and also because it allows for a simple rule: The type of **await** e is used to decide whether or not the future (if any) is awaited, and there are no exceptions—even in cases like this example, where the type seems to imply that a `Future<Object?>` should not be awaited. In summary, we await every future that we can soundly await.

An await expression can only occur in a function which is declared asynchronous. The **await** identifier has has no special meaning in the context of a normal function, so occurrences of **await** in those functions does not introduce an await expression. However, **await**`(`e`)` can be a valid function invocation in non-asynchronous functions.

An await expression could not meaningfully occur in a synchronous function. If such a function were to suspend waiting for a future, it would no longer be synchronous.

It is not a compile-time error if the type of e is not a supertype or subtype of `Future`. Tools may choose to give a hint in such cases.

17.35 Postfix Expressions

Postfix expressions invoke the postfix operators on objects.

⟨*postfixExpression*⟩ ::= ⟨*assignableExpression*⟩ ⟨*postfixOperator*⟩
  | ⟨*primary*⟩ ⟨*selector*⟩*

⟨*postfixOperator*⟩ ::= ⟨*incrementOperator*⟩

⟨*constructorInvocation*⟩ ::=
    ⟨*typeName*⟩ ⟨*typeArguments*⟩ '.' ⟨*identifier*⟩ ⟨*arguments*⟩

⟨*selector*⟩ ::= '`!`'
| ⟨*assignableSelector*⟩
| ⟨*argumentPart*⟩

⟨*argumentPart*⟩ ::= ⟨*typeArguments*⟩? ⟨*arguments*⟩

⟨*incrementOperator*⟩ ::= '`++`'
| '`--`'

A *postfix expression* is either a primary expression; a function, method or getter invocation; an invocation of a named constructor; or an invocation of a postfix operator on an expression e. All but the latter two are specified elsewhere. ◇

Case ⟨Constructor Invocations⟩**.** Consider a ⟨*constructorInvocation*⟩ e of the form n`<`*typeArguments*`>.`id`(`*arguments*`)`. If n does not denote a class C that declares a constructor named $C.id$, a compile-time error occurs.

Otherwise, if e occurs in a constant context (17.3.2) then e is treated as **const** e, and if e does not occur in a constant context then e is treated as **new** e.

Note that e cannot be anything other than an instance creation (constant or not) because e provides actual type arguments to n, which is not supported if n denotes a library prefix, nor if e is a static method invocation. □

Case ⟨v`++`, v`--`⟩**.** Consider a postfix expression e of the form v *op*, where v is an identifier and *op* is either '`++`' or '`--`'. A compile-time error occurs unless v denotes a variable, or v denotes a getter and there is an associated setter v`=`. Let T be the static type of the variable v or the return type of the getter. A compile-time error occurs if T is not **dynamic** and T does not have an operator '`+`' (when *op* is '`++`') or operator '`-`' (when *op* is '`--`'), or if the return type of this operator is not assignable to the variable respectively the argument type of the setter. A compile-time error occurs if `int` is not assignable to the parameter type of said operator. The static type of e is T.

Evaluation of a postfix expression e of the form v`++` respectively v`--`, where v is an identifier, proceeds as follows: Evaluate v to an object r and let y be a fresh variable bound to r. Evaluate v `=` y `+ 1` respectively v `=` y `- 1`. Then e evaluates to r.

The above ensures that if the evaluation involves a getter, it gets called exactly once. Likewise in the cases below. □

Case ⟨C`.`v`++`, C`.`v`--`⟩**.** Consider a postfix expression e of the form C`.`v *op*, where C is a type literal and *op* is either '`++`' or '`--`'. A compile-time error occurs unless C`.`v denotes a static getter and there is an associated static setter v`=` (possibly implicitly induced by a class variable). Let T be the return type of said getter. A compile-time error occurs if T is not **dynamic** and T does not have an operator '`+`' (when *op* is '`++`') or operator '`-`' (when *op* is '`--`'), or if the return type of this operator is not assignable to the argument type of the setter. A compile-time error occurs if `int` is not assignable to the parameter type of said operator. The static type of e is T.

Evaluation of a postfix expression e of the form C`.`v`++` respectively C`.`v`--` where C is a type literal proceeds as follows: Evaluate C`.`v to an object r and let y be a fresh variable bound to r. Evaluate C`.`v `= `y `+ 1` respectively C`.`v `= `y `- 1`. Then e evaluates to r. □

Case $\langle e_1$`.`v`++`, e_1`.`v`--`$\rangle$**.** Consider a postfix expression e of the form e_1`.`v *op* where *op* is either '`++`' or '`--`'. Let S be the static type of e_1. A compile-time error occurs unless S has a getter named v and a setter named v`=` (possibly implicitly induced by an instance variable). Let T be the return type of said getter. A compile-time error occurs if T is not **dynamic** and T does not have an operator '`+`' (when *op* is '`++`') or operator '`-`' (when *op* is '`--`'), or if the return type of this operator is not assignable to the argument type of the setter. A compile-time error occurs if `int` is not assignable to the parameter type of said operator. The static type of e is T.

Evaluation of a postfix expression e of the form e_1`.`v`++` respectively e_1`.`v`--` proceeds as follows: Evaluate e_1 to an object u and let x be a fresh variable bound to u. Evaluate x`.`v to an object r and let y be a fresh variable bound to r. Evaluate x`.`v `= `y `+ 1` respectively x`.`v `= `y `- 1`. Then e evaluates to r. □

Case $\langle e_1$`[`e_2`]++`, e_1`[`e_2`]--`$\rangle$**.** Consider a postfix expression e of the form e_1`[`e_2`]` *op* where *op* is either '`++`' or '`--`'. Let S_1 be the static type of e_1 and S_2 be the static type of e_2. A compile-time error occurs unless S_1 has an operator '`[]`' and an operator '`[]=`'. Let T be the return type of the former. A compile-time error occurs unless S_2 is assignable to the first parameter type of said operator '`[]=`'. A compile-time error occurs if T is not **dynamic** and T does not have an operator '`+`' (when *op* is '`++`') or operator '`-`' (when *op* is '`--`'), or if the return type of this operator is not assignable to the second argument type of said operator '`[]=`'. A compile-time error occurs if passing the integer literal `1` as an argument to said operator '`+`' or '`-`' would be an error. The static type of e is T.

Evaluation of a postfix expression e of the form e_1`[`e_2`]++` respectively e_1`[`e_2`]--` proceeds as follows: Evaluate e_1 to an object u and e_2 to an object v. Let a and i be fresh variables bound to u and v respectively. Evaluate a`[`i`]` to an object r and let y be a fresh variable bound to r. Evaluate a`[`i`] = `y `+ 1` respectively a`[`i`] = `y `- 1`. Then e evaluates to r. □

Case $\langle e_1$`?.`v`++`, e_1`?.`v`--`$\rangle$**.** Consider a postfix expression e of the form e_1`?.`v *op* where *op* is either '`++`' or '`--`'. Exactly the same compile-time errors that would be caused by e_1`.`v *op* are also generated in the case of e_1`?.`v *op*. The static type of e is the static type of e_1`.`v.

Evaluation of a postfix expression e of the form e_1`?.`v`++` respectively e_1`?.`v`--` proceeds as follows: If e_1 is a type literal, evaluation of e is equivalent to evaluation of e_1`.`v`++` respectively e_1`.`v`--`. Otherwise evaluate e_1 to an object u. if u is the null object, e evaluates to the null object (17.4). Otherwise let x be a fresh variable bound to u. Evaluate x`.`v`++` respectively x`.`v`--` to an object o. Then e evaluates to o. □

17.36 Assignable Expressions

Assignable expressions are terms that can appear on the left hand side of an assignment. This section describes how to evaluate subterms of these terms when needed. The semantics of an assignment as a whole is described elsewhere (17.23). ◇

The grammar of assignable expressions includes very general forms like an identifier id or a qualified identifier $id_1\,.\,id_2$. Hence, an assignable expression can have many different meanings, depending on the binding of those identifiers in the context.

For example, the term `x.y.z` is an assignable expression. `x.y` may refer to a getter `y` on an object referenced by a variable `x`, in which case `x.y` will be evaluated to an object before accessing the `z` member of that object. The term `x.y` could also denote a class `y` referenced through an import prefix `x`, with `z` denoting a static variable of that class. In this case `x.y` will not be evaluated to a value.

⟨*assignableExpression*⟩ ::= ⟨*primary*⟩ ⟨*assignableSelectorPart*⟩
| **super** ⟨*unconditionalAssignableSelector*⟩
| ⟨*identifier*⟩

⟨*assignableSelectorPart*⟩ ::= ⟨*selector*⟩* ⟨*assignableSelector*⟩

⟨*unconditionalAssignableSelector*⟩ ::= '`[`' ⟨*expression*⟩ '`]`'
| '`.`' ⟨*identifier*⟩

⟨*assignableSelector*⟩ ::= ⟨*unconditionalAssignableSelector*⟩
| '`?.`' ⟨*identifier*⟩
| '`?`' '`[`' ⟨*expression*⟩ '`]`'

The section about assignments (17.23) specifies the static analysis and dynamic semantics of various forms of assignment. Each of those cases is applicable when the specified subterms satisfy the given side conditions (e.g., one case of the form $e_1\,.\,v$ `??=` e_2 requires e_1 to be an expression, whereas $C\,.\,v$ `??=` e requires C to be a type literal). The cases requiring subterms to be expressions are considered least specific, that is, they are only used if no other case matches (so the case containing C is used if the corresponding term is a type literal).

Syntactically, these expressions are not derived from ⟨*expression*⟩, but they are derivable from ⟨*expression*⟩, e.g., because an ⟨*assignableExpression*⟩ may contain a ⟨*primary*⟩ which is derivable from ⟨*expression*⟩. We use the following rule to find such expressions:

Let e be an ⟨*assignableExpression*⟩. Assume that t is a term such that e can be derived from t ⟨*assignableSelector*⟩. In this case we say that t is the *receiver term* of e. When t is an expression, we say that t is the *receiver expression* of e. ◇ e, t ◇

In short, we obtain t by cutting off an assignable selector from the end of e. It is easy to see that only some ⟨*assignableExpression*⟩s have a receiver term. For instance, a plain ⟨*identifier*⟩ does not.

Let e be an assignable expression. Assume that e has a receiver expression t. Evaluation of t proceeds in the same way as evaluation of any other expression.

17.37 Lexical Lookup

This section specifies how to look up a name based on the enclosing lexical scopes. This is known as a *lexical lookup*. When id is an identifier, it may look up a name n of the form id as well as of the form id=. ◇

A lexical lookup can yield a declaration or an import prefix, and it can yield nothing.

It is not a compile-time error when the lexical lookup yields nothing. In this situation the given name n will be transformed into **this**.n, and the static analysis of the resulting expression may or may not have any compile-time errors.

A lexical lookup may incur a compile-time error, as specified below. However, that is different from a result yielded by the lexical lookup, because this specification never specifies the propagation of errors.

In other words, when other parts of this specification indicate that a lexical lookup is performed, they need to consider the further steps taken when the lookup yields a declaration, when it yields an import prefix, and when it yields nothing. But they do not mention that, e.g., id`.m` is an error because the lexical lookup for id incurred an error.

A lexical lookup differs from a lookup of an instance member (17.18) because that operation searches through a sequence of superclasses, whereas a lexical lookup searches through a sequence of enclosing scopes. A lexical lookup differs from a straightforward lookup in the enclosing scopes because the lexical lookup "bundles" getters and setters, as detailed below.

Consider the situation where a name n has basename id (10.11) where id is an identifier, and a lexical lookup of n is performed from a given location ℓ. n, id ℓ

We specify a name and a location from where a lexical lookup is performed. The location is not always redundant: In some situations we perform a lookup for a setter named id=, but the token id= does not occur in the program. To handle such situations we must specify both the name which is being looked up, and the location that determines which scopes are the enclosing ones.

When we say that a lexical lookup of the identifier id is performed, it is understood that the lookup is performed from the location of the given occurrence of id.

Let S be the innermost lexical scope containing ℓ which has a declaration with basename id. In the case where S has a declaration named id as well as a declaration named id=, let D be the declaration named n. In the situation where S has exactly one declaration with basename id, let D be that declaration. D

A non-local variable declaration named id will implicitly induce a getter id and possibly a setter id= into the current scope. This means that D may denote an implicitly induced getter or setter rather than the underlying variable declaration. That is significant in the case where an error must arise because the lookup was for one kind, but only the other kind exists.

If we are looking up a name n with basename id, we stop searching if we find any declaration named id or id=. If, in that scope, there are declarations for both id and id=, we return the one which has the requested name n. In the case where only one declaration is present, we return it, even though it may have the name id

when n is id=, or vice versa. That situation may cause an error, as specified below.

In the first step, we check for several potential errors.

Case $\langle D$ exists$\rangle$**.** In this case, at least one declaration with basename id is in scope at the location ℓ. It is a compile-time error if the name of D is not n, unless D is an instance member or a local variable (which may be a formal parameter).

That is, it is an error if we look for a setter and find a getter, or vice versa, but not an error if we look for a setter and find a local variable. If we look for a setter and find an instance getter, or vice versa, it is not an error, because the setter could be inherited. That is checked after yielding nothing (which implies that **this** will be prepended).

If D is an instance member, it is a compile-time error if ℓ does not have access to **this**. □

Case $\langle D$ does not exist$\rangle$**.** It is a compile-time error if ℓ does not have access to **this** (10). □

We are always looking up both *id and id=, no matter whether n is id or id=. This approach creates a tighter connection between a pair of declarations where one is a getter named id and the other is a setter named id=. This allows developers to think about a getter and setter that are declared together as a single entity, rather than two independent declarations.*

For example, if a term refers to id and needs a setter, and the innermost declaration named id or id= is a getter g and there is no corresponding setter, it is a compile-time error. This error occurs even in the case where a more remote enclosing scope has a declaration of a setter s named id=, because we already committed to using g (so that's actually "a setter/getter pair where the setter is missing"), and we could say that this "pair" shadows s:

```
set id(int value) {} // This is s

class A {
  int get id => 42; // This is g
  user() {
    id = 0; // Compile-time error
  }
}
```

In the second and last step, if no error occurred, proceed as described in the first applicable case from the following list:

- When D does not exist, the lexical lookup yields nothing. In this case it is guaranteed that ℓ has access to **this**, and n will be treated as **this**.n. But errors may still occur, e.g., because there is no member of the interface of **this** named n, and also no accessible and applicable extension method.

- Consider the case where D is a formal type parameter declaration of a class or a mixin. It is a compile-time error if ℓ occurs inside a static

method, static getter, or static setter, or inside a class variable initializer. Otherwise, the lexical lookup yields D.

- Consider the case where D is an instance member declaration in a class or mixin A. The lexical lookup then yields nothing. In this case it is guaranteed that ℓ has access to **this**.
- Otherwise, the lexical lookup yields D.

Note that a lexical lookup will never yield a declaration of an instance member. In each case where it is determined that there is no error, and an instance member is the result of the lookup, the lexical lookup yields nothing.

*The reason for this is that there may not be a declaration in scope, but the interface of the class could have the required member, or an extension method could be applicable. So, for uniformity, in these situations we always report that nothing was found, which implies that **this** should be added.*

17.38 Identifier Reference

An *identifier expression* consists of a single identifier; it provides access to an object via an unqualified name. A ⟨*typeIdentifier*⟩ is an identifier which can be used as the name of a type declaration. ◇

A ⟨*qualifiedName*⟩ is not an identifier expression, but we specify its syntax here because it is used in several different contexts, and it is more closely related to the plain identifier than it is to any single one of those grammar rules where it is used.

⟨*identifier*⟩ ::= ⟨*IDENTIFIER*⟩
  | ⟨*BUILT_IN_IDENTIFIER*⟩
  | ⟨*OTHER_IDENTIFIER*⟩

⟨*typeIdentifier*⟩ ::= ⟨*IDENTIFIER*⟩
  | ⟨*OTHER_IDENTIFIER*⟩
  | **dynamic**

⟨*qualifiedName*⟩ ::= ⟨*typeIdentifier*⟩ '.' ⟨*identifier*⟩
  | ⟨*typeIdentifier*⟩ '.' ⟨*typeIdentifier*⟩ '.' ⟨*identifier*⟩

⟨*BUILT_IN_IDENTIFIER*⟩ ::= **abstract** | **as** | **covariant** | **deferred**
  | **dynamic** | **export** | **external** | **extension** | **factory** | **Function** | **get**
  | **implements** | **import** | **interface** | **late** | **library** | **mixin** | **operator**
  | **part** | **required** | **set** | **static** | **typedef**

⟨*OTHER_IDENTIFIER*⟩ ::=
    async | **hide** | **of** | **on** | **show** | **sync** | **await** | **yield**

⟨*IDENTIFIER_NO_DOLLAR*⟩ ::= ⟨*IDENTIFIER_START_NO_DOLLAR*⟩
    ⟨*IDENTIFIER_PART_NO_DOLLAR*⟩*

⟨*IDENTIFIER_START_NO_DOLLAR*⟩ ::= ⟨*LETTER*⟩ | '_'

⟨*IDENTIFIER_PART_NO_DOLLAR*⟩ ::=
⟨*IDENTIFIER_START_NO_DOLLAR*⟩ | ⟨*DIGIT*⟩

⟨*IDENTIFIER*⟩ ::= ⟨*IDENTIFIER_START*⟩ ⟨*IDENTIFIER_PART*⟩*

⟨*IDENTIFIER_START*⟩ ::= ⟨*IDENTIFIER_START_NO_DOLLAR*⟩ | '$'

⟨*IDENTIFIER_PART*⟩ ::= ⟨*IDENTIFIER_START*⟩ | ⟨*DIGIT*⟩

⟨*LETTER*⟩ ::= 'a' .. 'z' | 'A' .. 'Z'

⟨*DIGIT*⟩ ::= '0' .. '9'

⟨*WHITESPACE*⟩ ::= ('\t' | ' ' | ⟨*LINE_BREAK*⟩)+

The ordering of the lexical rules above ensure that ⟨*IDENTIFIER*⟩ and ⟨*IDENTIFIER_NO_DOLLAR*⟩ do not derive any built-in identifiers. Similarly, the lexical rule for reserved words (21.1.1) must be considered to come before the rule for ⟨*BUILT_IN_IDENTIFIER*⟩, such that ⟨*IDENTIFIER*⟩ and ⟨*IDENTIFIER_NO_DOLLAR*⟩ also do not derive any reserved words.

A *built-in identifier* is one of the identifiers produced by the production ⟨*BUILT_IN_IDENTIFIER*⟩. It is a compile-time error if a built-in identifier is used as the declared name of a prefix, class, mixin, type parameter, or type alias. It is a compile-time error to use a built-in identifier other than **dynamic** or **Function** as an identifier in a type annotation or a type parameter bound. ◇

Built-in identifiers are identifiers that are used as keywords in Dart, but are not reserved words. A built-in identifier may not be used to name a class or type. In other words, they are treated as reserved words when used as types. This eliminates many confusing situations, both for human readers and during parsing.

It is a compile-time error if either of the identifiers **await** or **yield** is used as an ⟨*identifier*⟩ in a function body marked with either **async**, **async***, or **sync***.

*This makes the identifiers **await** and **yield** behave like reserved words in a limited context. This approach was chosen because it was less breaking than it would have been to make **await** and **yield** reserved words or built-in identifiers, at the time where these features were added to the language.*

A *qualified name* is two or three identifiers separated by '.'. All but the last one must be a ⟨*typeIdentifier*⟩. It is used to denote a declaration which is imported with a prefix, or a **static** declaration in a class, mixin, or extension, or both. ◇

The static type of an identifier expression e which is an identifier id is determined as follows. Perform a lexical lookup of id (17.37) from the location of e.

Case ⟨Lexical lookup yields a declaration⟩**.** Let D be the declaration yielded by the lexical lookup of id.

- If D declares a class, mixin, type alias, an enumerated type, or a type parameter, the static type of e is `Type`.
- If D is the declaration of a library getter (which may be implicitly induced by a library variable), the static type of e is the static type of the library getter invocation *id* (17.19).
- If D is a static method, library function, or local function, the static type of e is the function type of D.

  Note that e may subsequently be subjected to generic function instantiation (17.17).
- If D is the declaration of a static getter (which may be implicitly induced by a class variable) and D occurs in the class C, the static type of e is the return type of the getter $C\,.\,id$.
- If D is a local variable declaration (which can be a formal parameter) the static type of e is the type of the variable v declared by D, unless v is known to have some type T, where T is a subtype of any other type S such that v is known to have type S, in which case the static type of e is T.
- If D is a declaration of an instance getter in an extension declaration E, then the static type of e is the return type of D.
- If D is a declaration of an instance method in an extension declaration E, then the static type of e is the function type of D.
- A lexical lookup will never yield a declaration which is an instance member of a class.

□

Case ⟨Lexical lookup yields an import prefix⟩**.** In this case the lexical lookup (17.37) for *id* yields an import prefix p. In this case a compile-time error occurs, unless the token immediately following e is '`.`'. No static type is associated with e in this case.

No such type is needed, because every construct where an import prefix p is used and followed by '`.`' is specified in such a way that the type of p is not used. □

Case ⟨Lexical lookup yields nothing⟩**.** When the lexical lookup (17.37) for *id* yields nothing, e is treated as (5) **this**$\,.\,id$.

In this case it is known that e has access to **this** (10). Both the static analysis and evaluation proceeds with **this**$\,.\,id$, so there is no need to further specify the treatment of e. □

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

Case ⟨Lexical lookup yields a declaration⟩**.** In this case the lexical lookup (17.37) for *id* yields a declaration D. The evaluation of e proceeds as follows:

- If D is a class, mixin, or type alias, the value of e is an object implementing the class `Type` which reifies the corresponding type.

- If D is a type parameter X then the value of e is the value of the actual type argument corresponding to X that was passed to the generative constructor that created the current binding of **this**.
- If D is the declaration of a library getter (which may be implicitly induced by a library variable), evaluation of e is equivalent to evaluation of an invocation of the library getter *id* (17.19).
- If D is a library, class, or local constant variable of one of the forms **const** v `=` e'`;` or **const** T v `=` e'`;` then the value of e is the value of the constant expression e'.
- If D is a declaration of a top-level function, static method, or local function, then e evaluates to the function object obtained by closurization (17.16) of D.
- If D is an instance getter declaration in an extension declaration E, then e evaluates to the result of invoking said getter with the current binding of **this**, and the current bindings of the type parameters declared by D.
- If D is an instance method declaration in an extension declaration E with type parameters $X_1, \ldots, X_s$, then e evaluates to the result of the extension method closurization E`<`$X_1, \ldots, X_s$`>(`**this**`).`*id* (13.4).
- If D is a local variable v (which can be a formal parameter) then e evaluates to the current binding of v.

Note that D cannot be the declaration of a class variable, static getter or static setter declared in a class C, because in that case e is treated as (5) the property extraction (17.22) $C.id$, which also determines the evaluation of e. □

Case ⟨Lexical lookup yields an import prefix⟩**.** This situation cannot arise, because it is a compile-time error to evaluate an import prefix as an expression, and no constructs involving an import prefix (e.g., such as a property extraction $p.m$) will evaluate the import prefix. □

Case ⟨Lexical lookup yields nothing⟩**.** This situation cannot arise, because this only occurs when e is treated as **this**`.`*id*, whose evaluation is specified elsewhere (17.22). □

17.39 Type Test

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

⟨*typeTest*⟩ ::= ⟨*isOperator*⟩ ⟨*typeNotVoid*⟩

⟨*isOperator*⟩ ::= **is** '`!`'?

Evaluation of the is-expression e **is** T proceeds as follows:
The expression e is evaluated to an object v. If the dynamic type 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`, $T =$ **dynamic** or $T =$ `Null`. The former two are useless, as is anything of the form e **is** `Object` or e **is dynamic**. Users should test for the null object (17.4) directly rather than via type tests.

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

Let v be a local variable (which can be a formal parameter). An is-expression of the form v **is** T shows that v has type T if T is a subtype of the type of the expression v. Otherwise, if the declared type of v is the type variable X, and T is a subtype of the bound of X, and $X\&T$ is a subtype of the type of the expression v, then e shows that v has type $X\&T$. Otherwise e does not show that v has type T for any T.

The motivation for the "shows that v has type T" relation is to reduce spurious errors thereby enabling a more natural coding style. The rules in the current specification are deliberately kept simple. It would be upwardly compatible to refine these rules in the future; such a refinement would accept more code without errors, but not reject any code now error-free.

The rule only applies to locals and parameters, as instance and static variables could be modified via side-effecting functions or methods that are not accessible to a local analysis.

It is pointless to deduce a weaker type than what is already known. Furthermore, this would lead to a situation where multiple types are associated with a variable at a given point, which complicates the specification. Hence the requirement that the promoted type is a subtype of the current type.

In any case, it is not an error when a type test does not show that a given variable does not have a "better" type than previously known, but tools may choose to give a hint in such cases, if suitable heuristics indicate that a promotion is likely to be intended.

The static type of an is-expression is `bool`.

17.40 Type Cast

The *cast expression* ensures that an object is a member of a type. ◇

⟨*typeCast*⟩ ::= ⟨*asOperator*⟩ ⟨*typeNotVoid*⟩

⟨*asOperator*⟩ ::= **as**

Evaluation of the cast expression e **as** T proceeds as follows:

The expression e is evaluated to an object v. It is a dynamic type error if o is not the null object (17.4), and the dynamic type of o is not a subtype of T. Otherwise e evaluates to v.

The static type of a cast expression e **as** T is T.

18 Statements

A *statement* is a fragment of Dart code that can be executed at run time. ◇
Statements, unlike expressions, do not evaluate to an object, but are instead executed for their effect on the program state and control flow.

⟨*statements*⟩ ::= ⟨*statement*⟩*

⟨*statement*⟩ ::= ⟨*label*⟩* ⟨*nonLabelledStatement*⟩

⟨*nonLabelledStatement*⟩ ::= ⟨*block*⟩
| ⟨*localVariableDeclaration*⟩
| ⟨*forStatement*⟩
| ⟨*whileStatement*⟩
| ⟨*doStatement*⟩
| ⟨*switchStatement*⟩
| ⟨*ifStatement*⟩
| ⟨*rethrowStatement*⟩
| ⟨*tryStatement*⟩
| ⟨*breakStatement*⟩
| ⟨*continueStatement*⟩
| ⟨*returnStatement*⟩
| ⟨*yieldStatement*⟩
| ⟨*yieldEachStatement*⟩
| ⟨*expressionStatement*⟩
| ⟨*assertStatement*⟩
| ⟨*localFunctionDeclaration*⟩

18.0.1 Statement Completion

Execution of a statement *completes* in one of five ways: either it *completes* ◇
normally, it *breaks* or it *continues* (either to a label or without a label), it ◇
returns (with or without an object), or it *throws* an exception object and an ◇
associated stack trace. ◇

In descriptions of statement execution the default is that the execution com- ◇
pletes normally unless otherwise stated.

If the execution of a statement, s, is defined in terms of executing another statement, and the execution of that other statement does not complete normally, then, unless otherwise stated, the execution of s stops at that point and completes in the same way. For example, if execution of the body of a **do** loop returns an object, so does execution of the **do** loop statement itself.

If the execution of a statement is defined in terms of evaluating an expression and the evaluation of that expression throws, then, unless otherwise stated, the execution of the statement stops at that point and throws the same exception object and stack trace. For example, if evaluation of the condition expression of an **if** statement throws, then so does execution of the **if** statement. Likewise, if

evaluation of the expression of a **return** statement throws, so does execution of the **return** statement.

18.1 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.

A block statement introduces a new scope, whose enclosing scope is the current scope of the block statement.

18.2 Expression Statements

An *expression statement* consists of an expression that does not begin with a '`{`' character. ◇

⟨*expressionStatement*⟩ ::= ⟨*expression*⟩? '`;`'

The expression of an expression statement is not allowed to begin with a '`{`'. This means that if some source text could otherwise be parsed as an expression followed by a '`;`', then this grammar production does not apply when the expression starts with a '`{`'. *The restriction resolves an ambiguity while parsing where a '`{`' can start either a block (18.1) or a map literal (17.9.8). By disallowing the latter from starting an expression statement, the parser does not need to look further ahead before deciding that it is parsing a block statement.*

Execution of an expression statement e`;` proceeds by evaluating e. If the expression evaluates to an object, then the object is ignored and the execution completes normally.

18.3 Local Variable Declaration

A *variable declaration statement*, also known as a *local variable declaration*, has the following form: ◇ ◇

⟨*localVariableDeclaration*⟩ ::= ⟨*metadata*⟩ ⟨*initializedVariableDeclaration*⟩ '`;`'

Each local variable declaration introduces a *local variable* into the current scope. ◇

Local variables do not induce getters and setters. Note that a formal parameter declaration also introduces a local variable into the associated formal parameter scope (9.2).

The properties of being *initialized*, *constant*, *final*, and *mutable* apply to local variables with the same definitions as for other variables (8). ◇ ◇

We say that a local variable v is *potentially mutated* in some scope s if v is mutable, and an assignment to v occurs in s. ◇ ◇

A local variable declaration of the form **var** v`;` is equivalent to **var** v `=`

null;. A local variable declaration of the form T v; is equivalent to T v = **null**;.

This holds regardless of the type T. E.g., `int i;` is equivalent to `int i = null;`.

The type of a local variable with a declaration of one of the forms T v = e; **const** T v = e; **final** T v = e; is T. The type of a local variable with a declaration of one of the forms **var** v = e; **const** v = e; **final** v = e; is **dynamic**.

Let v be a local variable declared by an initializing variable declaration, and let e be the associated initializing expression. It is a compile-time error if the static type of e is not assignable to the type of v. It is a compile-time error if a local variable v is final, and the declaration of v is not an initializing variable declaration.

It is also a compile-time error to assign to a final local variable (17.23).

It is a compile-time error if a local variable is referenced at a source code location that is before the end of its initializing expression, if any, and otherwise before the declaring occurrence of the identifier which names the variable.

The example below illustrates the expected behavior. A variable 'x' is declared at the library level, and another 'x' is declared inside the function 'f'.

```
var x = 0;

f(y) {
  var z = x; // compile-time error
  if (y) {
    x = x + 1; // two compile-time errors
    print(x); // compile-time error
  }
  var x = x++; // compile-time error
  print(x);
}
```

The declaration inside 'f' hides the enclosing one. So all references to 'x' inside 'f' refer to the inner declaration of 'x'. However, many of these references are illegal, because they appear before the declaration. The assignment to 'z' is one such case. The assignment to 'x' in the **if** statement suffers from multiple problems. The right hand side reads 'x' before its declaration, and the left hand side assigns to 'x' before its declaration. Each of these are, independently, compile-time errors. The print statement inside the **if** is also illegal.

The inner declaration of 'x' is itself erroneous because its right hand side attempts to read 'x' before the declaration has terminated. The occurrence of 'x' that declares and names the variable (that is, the one to the left of '=' in the inner declaration) is not a reference, and so is legal. The last print statement is perfectly legal as well.

As another example **var** `x = 3, y = x;` is legal, because `x` is referenced after its initializer.

A particularly perverse example involves a local variable name shadowing a type. This is possible because Dart has a single namespace for types, functions and variables.

```
class C {}
perverse() {
  var v = new C(); // compile-time error
  C aC; // compile-time error
  var C = 10;
}
```

Inside `perverse()`, 'C' denotes a local variable. The type 'C' is hidden by the variable of the same name. The attempt to instantiate 'C' causes a compile-time error because it references a local variable prior to its declaration. Similarly, for the declaration of 'aC'.

Execution of a variable declaration statement of one of the forms **var** v = e; T v = e; **const** v = e; **const** T v = e; **final** v = e; or **final** T v = e; proceeds as follows:

The expression e is evaluated to an object o. Then, the variable v is set to o. A dynamic type error occurs if the dynamic type of o is not a subtype of the actual type (20.10.1) of v.

18.4 Local Function Declaration

A function declaration statement declares a new local function (9.1).

⟨*localFunctionDeclaration*⟩ ::= ⟨*metadata*⟩ ⟨*functionSignature*⟩ ⟨*functionBody*⟩

A function declaration statement of one of the forms *id signature* `{` *statements* `}` or T *id signature* `{` *statements* `}` causes a new function named *id* to be added to the current scope. It is a compile-time error to reference a local function before its declaration.

This implies that local functions can be directly recursive, but not mutually recursive. Consider these examples:

```
f(x) => x++; // a top level function

top() { // another top level function
  f(3); // illegal
  f(x) => x > 0? x*f(x-1): 1; // recursion is legal
  g1(x) => h(x, 1); // error: h is not declared yet
  h(x, n) => x > 1? h(x-1, n*x): n; // again, recursion is fine
  g2(x) => h(x, 1); // legal

  p1(x) => q(x,x); // illegal
  q1(a, b) ⇒ a > 0 ? p1(a-1): b; // fine
```

```
  q2(a, b) => a > 0 ? p2(a-1): b; // illegal
  p1(x) => q2(x,x); // fine
}
```

There is no way to write a pair of mutually recursive local functions, because one always has to come before the other is declared. These cases are quite rare, and can always be managed by defining a pair of variables first, then assigning them appropriate function literals:

```
top2() { // a top level function
  var p, q;
  p = (x) => q(x,x);
  q = (a, b) => a > 0 ? p(a-1): b;
}
```

The rules for local functions differ slightly from those for local variables in that a function can be accessed within its declaration but a variable can only be accessed after its declaration. This is because recursive functions are useful whereas recursively defined variables are almost always errors. It therefore makes sense to harmonize the rules for local functions with those for functions in general rather than with the rules for local variables.

18.5 If

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

⟨*ifStatement*⟩ ::= **if** '(' ⟨*expression*⟩ ')' ⟨*statement*⟩ (**else** ⟨*statement*⟩)?

An if statement of the form **if** (*e*) s_1 **else** s_2 where s_1 is not a block statement is equivalent to the statement **if** (*e*) {s_1} **else** s_2. An if statement of the form **if** (*e*) s_1 **else** s_2 where s_2 is not a block statement is equivalent to the statement **if** (*e*) s_1 **else** {s_2}.

The reason for this equivalence is to catch errors such as

```
void main() {
  if (somePredicate)
    var v = 2;
  print(v);
}
```

Under reasonable scope rules such code is problematic. If we assume that `v` *is declared in the scope of the method* `main()`*, then when* `somePredicate` *is false,* `v` *will be uninitialized when accessed. The cleanest approach would be to require a block following the test, rather than an arbitrary statement. However, this goes against long standing custom, undermining Dart's goal of familiarity. Instead, we choose to insert a block, introducing a scope, around the statement following the predicate (and similarly for* **else** *and loops). This will cause a*

compile-time error in the case above. Of course, if there is a declaration of `v` *in the surrounding scope, programmers might still be surprised. We expect tools to highlight cases of shadowing to help avoid such situations.*

Execution of an if statement of the form **if** `(`b`)` s_1 **else** s_2 where s_1 and s_2 are block statements, proceeds as follows:

First, the expression b is evaluated to an object o. It is a dynamic error if the run-time type of o is not `bool`. If o is **true**, then the block statement s_1 is executed, otherwise the block statement s_2 is executed.

It is a compile-time error if the type of the expression b may not be assigned to `bool`.

If b shows that a local variable v has type T, then the type of v is known to be T in s_1, unless any of the following are true

- v is potentially mutated in s_1,
- v is potentially mutated within a function other than the one where v is declared, or
- v is accessed by a function defined in s_1 and v is potentially mutated anywhere in the scope of v.

An if statement of the form **if** `(`e`)` s is equivalent to the if statement **if** `(`e`)` s **else** `{}`.

18.6 For

The *for statement* supports iteration. ◇

⟨*forStatement*⟩ ::= **await**? **for** '(' ⟨*forLoopParts*⟩ ')' ⟨*statement*⟩

⟨*forLoopParts*⟩ ::= ⟨*forInitializerStatement*⟩ ⟨*expression*⟩? ';' ⟨*expressionList*⟩?
| ⟨*metadata*⟩ ⟨*declaredIdentifier*⟩ **in** ⟨*expression*⟩
| ⟨*identifier*⟩ **in** ⟨*expression*⟩

⟨*forInitializerStatement*⟩ ::= ⟨*localVariableDeclaration*⟩
| ⟨*expression*⟩? ';'

The for statement has three forms - the traditional for loop and two forms of the for-in statement - synchronous and asynchronous.

18.6.1 For Loop

Execution of a for statement of the form **for** `(`**var** v `=` e_0`;` c`;` e`)` s proceeds as follows:

If c is empty then let c' be **true** otherwise let c' be c.

First the variable declaration statement **var** $v = e_0$ is executed. Then:

1. If this is the first iteration of the for loop, let v' be v. Otherwise, let v' be the variable v'' created in the previous execution of step 3.

2. The expression $[v'/v]c$ is evaluated to an object o. It is a dynamic error if the run-time type of o is not `bool`. If o is **false**, the for loop completes normally. Otherwise, execution continues at step 3.

3. The statement $[v'/v]\{s\}$ is executed.

   If this execution completes normally, continues without a label, or continues to a label (18.13) that prefixes this **for** statement (18.0.1), then execution of the statement is treated as if it had completed normally.

   Let v'' be a fresh variable. v'' is bound to the value of v'.

4. The expression $[v''/v]e$ is evaluated, and the process recurses at step 1.

The definition above is intended to prevent the common error where users create a function object inside a for loop, intending to close over the current binding of the loop variable, and find (usually after a painful process of debugging and learning) that all the created function objects have captured the same value—the one current in the last iteration executed.

Instead, each iteration has its own distinct variable. The first iteration uses the variable created by the initial declaration. The expression executed at the end of each iteration uses a fresh variable v'', bound to the value of the current iteration variable, and then modifies v'' as required for the next iteration.

It is a compile-time error if the static type of c may not be assigned to `bool`.

18.6.2 For-in

Let D be derived from ⟨*finalConstVarOrType*⟩?. A for statement of the form **for** `(`D id **in** e`)` S is then treated as the following code, where id_1 and id_2 are fresh identifiers:

```
T id1 = e;
var id2 = id1.iterator;
while (id2.moveNext()) {
  D id = id2.current;
  { S }
}
```

If the static type of e is a top type (15.2) then T is `Iterable<dynamic>`, otherwise T is the static type of e. It is a compile-time error if T is not assignable to `Iterable<dynamic>`.

It follows that it is a compile-time error if D is empty and id is a final variable; and it is a dynamic error if e has a top type, but e evaluates to an instance of a type which is not a subtype of `Iterable<dynamic>`.

18.6.3 Asynchronous For-in

A for-in statement may be asynchronous. The asynchronous form is de-

signed to iterate over streams. An asynchronous for loop is distinguished by the keyword **await** immediately preceding the keyword **for**.

Let D be derived from ⟨*finalConstVarOrType*⟩?. Execution of a for-in statement, f, of the form **await for** `(`D id **in** e`)` s proceeds as follows:

The expression e is evaluated to an object o. It is a dynamic type error if o is not an instance of a class that implements `Stream`. It is a compile-time error if D is empty and id is a final variable.

The stream associated with the innermost enclosing asynchronous for loop, if any, is paused. The stream o is listened to, producing a stream subscription u, and execution of the asynchronous for-in loop is suspended until a stream event is available. This allows other asynchronous events to execute while this loop is waiting for stream events.

Pausing an asynchronous for loop means pausing the associated stream subscription. A stream subscription is paused by calling its `pause` method. If the subscription is already paused, an implementation may omit further calls to `pause`.

The pause call can throw, although that should never happen for a correctly implemented stream.

For each *data event* from u, the statement s is executed with id bound to the value of the current data event. ◇

Either execution of s is completely synchronous, or it contains an asynchronous construct (**await**, **await for**, **yield**, or **yield***) which will pause the stream subscription of its surrounding asynchronous loop. This ensures that no other event of u occurs before execution of s is complete, if o is a correctly implemented stream. If o doesn't act as a valid stream, for example by not respecting pause requests, the behavior of the asynchronous loop may become unpredictable.

If execution of s continues without a label, or to a label (18.13) that prefixes the asynchronous for statement (18.0.1), then the execution of s is treated as if it had completed normally.

If execution of s otherwise does not complete normally, the subscription u is canceled by evaluating **await** v`.cancel()` where v is a fresh variable referencing the stream subscription u. If that evaluation throws, execution of f throws the same exception and stack trace. Otherwise execution of f completes in the same way as the execution of s. Otherwise the execution of f is suspended again, waiting for the next stream subscription event, and u is resumed if it has been paused. The `resume` call can throw, in which case the asynchronous for loop also throws. That should never happen for a correctly implemented stream.

On an *error event* from u, with error object e and stack trace st, the subscription u is canceled by evaluating **await** `v.cancel()` where v is a fresh variable referencing the stream subscription u. If that evaluation throws, execution of f throws the same exception object and stack trace. Otherwise execution of f throws with e as exception object and st as stack trace. ◇

When u is done, execution of f completes normally.

It is a compile-time error if an asynchronous for-in statement appears inside a synchronous function (9). It is a compile-time error if a traditional for loop (18.6.1) is prefixed by the **await** keyword.

An asynchronous loop would make no sense within a synchronous function, for the same reasons that an await expression makes no sense in a synchronous function.

18.7 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. It is a dynamic error if the run-time type of o is not `bool`.

If o is **false**, then execution of the while statement completes normally (18.0.1).

Otherwise o is **true** and then the statement $\{s\}$ is executed. If that execution completes normally or it continues with no label or to a label (18.13) that prefixes the **while** statement (18.0.1), then the while statement is re-executed. If the execution breaks without a label, execution of the while statement completes normally. If the execution breaks with a label that prefixes the **while** statement, it does end execution of the loop, but the break itself is handled by the surrounding labeled statement (18.13).

It is a compile-time error if the static type of e may not be assigned to `bool`.

18.8 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. If that execution continues with no label, or to a label (18.13) that prefixes the do statement (18.0.1), then the execution of s is treated as if it had completed normally.

Then, the expression e is evaluated to an object o. It is a dynamic error if the run-time type of o is not `bool`. If o is **false**, execution of the do statement completes normally (18.0.1). If o is **true**, then the do statement is re-executed.

It is a compile-time error if the static type of e may not be assigned to `bool`.

18.9 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*⟩* **default** ':' ⟨*statements*⟩

Consider a switch statement of the form

switch (*e*) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
  $label_{(n+1)1}$ ... $label_{(n+1)j_{n+1}}$ **default** : s_{n+1}
}

or the form

switch (*e*) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
}

Note that each expression $e_j, j \in 1..n$ occurs in a constant context (17.3.2), which means that **const** modifiers need not be specified explicitly.

It is a compile-time error unless each expression $e_j, j \in 1..n$ is constant. It is a compile-time error if the value of the expressions $e_j, j \in 1..n$ are not either:

- instances of the same class C, for all $j \in 1..n$, or
- instances of a class that implements `int`, for all $j \in 1..n$, or
- instances of a class that implements `String`, for all $j \in 1..n$.

In other words, all the expressions in the cases evaluate to constants of the exact same user defined class or are of certain known types. Note that the values of the expressions are known at compile time, and are independent of any type annotations.

It is a compile-time error if the class C does not have primitive equality (10.2.3).

The prohibition on user defined equality allows us to implement the switch efficiently for user defined types. We could formulate matching in terms of identity instead, with the same efficiency. However, if a type defines an equality operator, programmers would presumably find it quite surprising if equal objects did not match.

The **switch** statement should only be used in very limited situations (e.g., interpreters or scanners).

Execution of a switch statement of the form

switch (*e*) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
  $label_{(n+1)1}$ ... $label_{(n+1)j_{n+1}}$ **default**: s_{n+1}
}

or the form

switch (*e*) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
}

proceeds as follows:

The statement **var** *id* = *e*; is evaluated, where *id* is a fresh variable. It is a dynamic error if the value of e is not an instance of the same class as the constants $e_1, \ldots, e_n$.

Note that if there are no case clauses ($n = 0$), the type of e does not matter.

Next, the case clause **case** e_1: s_1 is matched against *id*, if $n > 0$. Otherwise if there is a **default** clause, the case statements s_{n+1} are executed (18.9.1).

Matching of a **case** clause **case** $e_k : s_k$ of a switch statement

switch (*e*) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
  $label_{(n+1)1}$ ... $label_{(n+1)j_{n+1}}$ **default**: s_{n+1}
}

against the value of a variable *id* proceeds as follows:

The expression e_k == *id* is evaluated to an object o. It is a dynamic error if the run-time type of o is not `bool`. If o is **false** the following case, **case** $e_{k+1} : s_{k+1}$ is matched against *id* if $k < n$, and if $k = n$, then the **default** clause's statements are executed (18.9.1). If o is **true**, let h be the smallest number such that $h \geq k$ and s_h is non-empty. If no such h exists, let $h = n + 1$. The case statements s_h are then executed (18.9.1).

Matching of a **case** clause **case** $e_k : s_k$ of a switch statement

switch (*e*) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n

```
}
```

against the value of a variable *id* proceeds as follows:

The expression e_k `==` *id* is evaluated to an object o. It is a dynamic error if the run-time type of o is not `bool`. If o is **false** the following case, **case** $e_{k+1} : s_{k+1}$ is matched against *id* if $k < n$. If o is **true**, let h be the smallest integer such that $h \geq k$ and s_h is non-empty. If such a h exists, the case statements s_h are executed (18.9.1). Otherwise the switch statement completes normally (18.0.1).

It is a compile-time error if the type of e may not be assigned to the type of e_k. Let s be the last statement of the statement sequence s_k. If s is a non-empty block statement, let s instead be the last statement of the block statement. It is a compile-time error if s is not a **break**, **continue**, **rethrow**, or **return** statement, or an expression statement where the expression is a **throw** expression.

The behavior of switch cases intentionally differs from the C tradition. Implicit fall through is a known cause of programming errors and therefore disallowed. Why not simply break the flow implicitly at the end of every case, rather than requiring explicit code to do so? This would indeed be cleaner. It would also be cleaner to insist that each case have a single (possibly compound) statement. We have chosen not to do so in order to facilitate porting of switch statements from other languages. Implicitly breaking the control flow at the end of a case would silently alter the meaning of ported code that relied on fall-through, potentially forcing the programmer to deal with subtle bugs. Our design ensures that the difference is immediately brought to the coder's attention. The programmer will be notified at compile time if they forget to end a case with a statement that terminates the straight-line control flow.

The sophistication of the analysis of fall-through is another issue. For now, we have opted for a very straightforward syntactic requirement. There are obviously situations where code does not fall through, and yet does not conform to these simple rules, e.g.:

```
switch (x) {
  case 1: try { ... return; } finally { ... return; }
}
```

Very elaborate code in a case clause is probably bad style in any case, and such code can always be refactored.

It is a static warning if all of the following conditions hold:

- The switch statement does not have a default clause.
- The static type of e is an enumerated type with elements $id_1, \ldots, id_n$.
- The sets $\{e_1, \ldots, e_k\}$ and $\{id_1, \ldots, id_n\}$ are not the same.

In other words, a static warning will be emitted if a switch statement over an enum is not exhaustive.

18.9.1 Switch case statements

Execution of the case statements s_h of a switch statement

switch (*e*) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
}

or a switch statement

switch (*e*) {
  $label_{11}$ ... $label_{1j_1}$ **case** e_1 : s_1
  ...
  $label_{n1}$ ... $label_{nj_n}$ **case** e_n : s_n
  $label_{(n+1)1}$ ... $label_{(n+1)j_{n+1}}$ **default**: s_{n+1}
}

proceeds as follows:

Execute $\{s_h\}$. If this execution completes normally, and if s_h is not the statements of the last case of the switch ($h = n$ if there is no **default** clause, $h = n + 1$ if there is a **default** clause), then the execution of the switch case throws an error. Otherwise s_h are the last statements of the switch case, and execution of the switch case completes normally.

In other words, there is no implicit fall-through between non-empty cases. The last case in a switch (default or otherwise) can 'fall-through' to the end of the statement.

If execution of $\{s_h\}$ breaks with no label (18.0.1), then the execution of the switch statement completes normally.

If execution of $\{s_h\}$ continues to a label (18.0.1), and the label is $label_{ij}$, where $1 \leq i \leq n+1$ if the **switch** statement has a **default**, or $1 \leq i \leq n$ if there is no **default**, and where $1 \leq j \leq j_i$, then let h be the smallest number such that $h \geq i$ and s_h is non-empty. If no such h exists, let $h = n + 1$ if the **switch** statement has a **default**, otherwise let $h = n$. The case statements s_h are then executed (18.9.1).

If execution of $\{s_h\}$ completes in any other way, execution of the **switch** statement completes in the same way.

18.10 Rethrow

The *rethrow statement* is used to re-throw an exception and its associated stack trace. ◇

⟨*rethrowStatement*⟩ ::= **rethrow** ';'

Consider a **rethrow** statement S. Let f be the immediately enclosing func-

tion of S. A compile-time error occurs unless S is located in an **on-catch** clause whose immediately enclosing function is f.

Execution of a **rethrow** statement proceeds as follows:

Let f be the immediately enclosing function, and let **on** T **catch** `(`p_1`,` p_2`)` be the immediately enclosing catch clause (18.11).

*A **rethrow** statement always appears inside a **catch** clause, and every **catch** clause is treated as some **catch** clause of the form **on** T **catch** `(p1, p2)`. So we can consider the **rethrow** statement to be enclosed in a **catch** clause of that form.*

The **rethrow** statement then throws (18.0.1) with the value of p_1 as the exception object, and the value of p_2 as the stack trace.

18.11 Try

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

⟨*tryStatement*⟩ ::= **try** ⟨*block*⟩ (⟨*onPart*⟩+ ⟨*finallyPart*⟩? | ⟨*finallyPart*⟩)

⟨*onPart*⟩ ::= ⟨*catchPart*⟩ ⟨*block*⟩
| **on** ⟨*typeNotVoid*⟩ ⟨*catchPart*⟩? ⟨*block*⟩

⟨*catchPart*⟩ ::= **catch** '`(`' ⟨*identifier*⟩ ('`,`' ⟨*identifier*⟩)? '`)`'

⟨*finallyPart*⟩ ::= **finally** ⟨*block*⟩

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

1. A set of **on-catch** clauses, each of which specifies (either explicitly or implicitly) the type of exception object to be handled, one or two exception parameters, and a block statement.

2. A **finally** clause, which consists of a block statement.

*The syntax is designed to be upward compatible with existing Javascript programs. The **on** clause can be omitted, leaving what looks like a Javascript catch clause.*

A try statement of the form **try** s_1 $on-catch_1 \ldots on-catch_n$`;` is equivalent to the statement **try** s_1 $on-catch_1 \ldots on-catch_n$ **finally** {}.

An **on-catch** clause of the form **on** T **catch** `(`p_1`)` s is equivalent to an **on-catch** clause **on** T **catch** `(`p_1`,` p_2`)` s where p_2 is a fresh identifier.

An **on-catch** clause of the form **on** T s is equivalent to an **on-catch** clause **on** T **catch** `(`p_1`,` p_2`)` s where p_1 and p_2 are fresh identifiers.

An **on-catch** clause of the form **catch** `(`p`)` s is equivalent to an **on-catch** clause **on dynamic catch** `(`p`,` p_2`)` s where p_2 is a fresh identifier.

An **on-catch** clause of the form **catch** `(`p_1`,` p_2`)` s is equivalent to an **on-catch** clause **on dynamic catch** `(`p_1`,` p_2`)` s.

An **on-catch** clause of the form **on** T **catch** `(`p_1`,` p_2`)` s introduces a new

scope CS in which final local variables specified by p_1 and p_2 are defined. The statement s is enclosed within CS. The static type of p_1 is T and the static type of p_2 is `StackTrace`.

Execution of a **try** statement s of the form:

try b
on T_1 **catch** `(`e_1`,` t_1`)` c_1
...
on T_n **catch** `(`e_n`,` t_n`)` c_n
finally f

proceeds as follows:

First b is executed. If execution of b throws (18.0.1) with exception object e and stack trace t, then e and t are matched against the **on-catch** clauses to yield a new completion (18.11.1).

Then, even if execution of b did not complete normally or matching against the **on-catch** clauses did not complete normally, the f block is executed.

If execution of f does not complete normally, execution of the **try** statement completes in the same way. Otherwise if execution of b threw (18.0.1), the **try** statement completes in the same way as the matching against the **on-catch** clauses. Otherwise the **try** statement completes in the same way as the execution of b.

It is a compile-time error if T_i, $1 \leq i \leq n$ is a deferred type.

18.11.1 on-catch clauses

Matching an exception object e and stack trace t against a (potentially empty) sequence of **on-catch** clauses of the form

on T_1 **catch** `(`e_1`,` st_1`)` `{` s_1 `}`
...
on T_n **catch** `(`e_n`,` st_n`)` `{` s_n `}`

proceeds as follows:

If there are no **on-catch** clauses ($n = 0$), matching throws the exception object e and stack trace t (18.0.1).

Otherwise the exception is matched against the first clause.

Otherwise, if the type of e is a subtype of T_1, then the first clause matches, and then e_1 is bound to the exception object e and t_1 is bound to the stack trace t, and s_1 is executed in this scope. The matching completes in the same way as this execution.

Otherwise, if the first clause did not match e, e and t are recursively matched against the remaining **on-catch** clauses:

on T_2 **catch** `(`e_2`,` t_2`)` `{` s_2 `}`
...

on T_n **catch** `(`e_n`,` t_n`)` `{` s_n `}`

18.12 Return

The *return statement* returns a result to the caller of a synchronous function, completes the future associated with an asynchronous function, or terminates the stream or iterable associated with a generator (9). ◇

⟨*returnStatement*⟩ ::= **return** ⟨*expression*⟩? ';'

Consider a return statement s of the form **return** e`?;`. Let S be the static type of e, if e is present, let f be the immediately enclosing function, and let T be the declared return type of f.

Case ⟨Synchronous non-generator functions⟩**.** Consider the case where f is a synchronous non-generator function (9). It is a compile-time error if s is **return;**, unless T is **void**, **dynamic**, or `Null`. It is a compile-time error if s is **return** e`;`, T is **void**, and S is neither **void**, **dynamic**, nor `Null`. It is a compile-time error if s is **return** e`;`, T is neither **void**, **dynamic**, nor `Null`, and S is **void**. It is a compile-time error if s is **return** e`;`, S is not **void**, and S is not assignable to T.

Note that T cannot be **void**, **dynamic**, or `Null` in the last case, because all types are assignable to those types. An error will not be raised if f has no declared return type, since the return type would be **dynamic**, to which every type is assignable. However, a synchronous non-generator function that declares a return type which is not "voidy" must return an expression explicitly. *This helps catch situations where users forget to return an object in a return statement.* □

Case ⟨Asynchronous non-generator functions⟩**.** Consider the case where f is an asynchronous non-generator function (9). It is a compile-time error if s is **return;**, unless *flatten*(T) (17.11) is **void**, **dynamic**, or `Null`. *An asynchronous non-generator always returns a future of some sort. If no expression is given, the future will be completed with the null object (17.4) which motivates this rule.* It is a compile-time error if s is **return** e`;`, *flatten*(T) is **void**, and *flatten*(S) is neither **void**, **dynamic**, nor `Null`. It is a compile-time error if s is **return** e`;`, *flatten*(T) is neither **void**, **dynamic**, nor `Null`, and *flatten*(S) is **void**. It is a compile-time error if s is **return** e`;`, *flatten*(S) is not **void**, and `Future<`*flatten*(S)`>` is not assignable to T.

Note that *flatten*(T) cannot be **void**, **dynamic**, or `Null` in the last case, because then `Future<`U`>` is assignable to T for *all* U. In particular, when T is `FutureOr<Null>` (which is equivalent to `Future<Null>`), `Future<`*flatten*(S)`>` is assignable to T for all S. This means that no compile-time error is raised, but *only* the null object (17.4) or an instance of `Future<Null>` can successfully be returned at run time. This is not an anomaly, it corresponds to the treatment of a synchronous function with return type `Null`; but tools may choose to give a hint that a downcast is unlikely to succeed.

An error will not be raised if f has no declared return type, since the return type would be **dynamic**, and `Future<`*flatten*(S)`>` is assignable to **dynamic** for all S.

However, an asynchronous non-generator function that declares a return type which is not "voidy" must return an expression explicitly. *This helps catch situations where users forget to return an object in a return statement of an asynchronous function.* □

Case ⟨Generator functions⟩**.** It is a compile-time error if a return statement of the form **return** *e*; appears in a generator function.

In the case of a generator function, the object returned by the function is the iterable or stream associated with it, and individual elements are added to that iterable using yield statements, and so returning an object makes no sense. □

Case ⟨Generative constructors⟩**.** It is a compile-time error if a return statement of the form **return** *e*; appears in a generative constructor (10.7.1).

It is quite easy to forget to add the ***factory*** *modifier 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 an object from a generative constructor is meaningless.* □

Executing a return statement **return** *e*; proceeds as follows:

First the expression e is evaluated, producing an object o. Let S be the run-time type of o and let T be the actual return type of f (20.10.1). If the body of f is marked **async** (9) and S is a subtype of `Future<`*flatten*(T)`>` then let r be the result of evaluating `await` v where v is a fresh variable bound to o. Otherwise let r be o. Then the return statement returns the object r (18.0.1).

Let U be the run-time type of r.

- If the body of f is marked **async** (9) it is a dynamic type error if `Future<`U`>` is not a subtype of T.
- Otherwise, it is a dynamic type error if U is not a subtype of T.

Executing a return statement with no expression, **return**; returns without an object (18.0.1).

18.13 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 (18.9) prefixed by a label L. ◇ ◇ ◇

The sole role of labels is to provide targets for the break (18.14) and continue (18.15) statements.

⟨*label*⟩ ::= ⟨*identifier*⟩ ':'

Execution a labeled statement s, *label* : s_l, consists of executing s_l. If execution of s_l breaks to the label *label* (18.0.1), then execution of s completes normally, otherwise execution of s completes in the same ways as the execution of s_l.

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.

18.14 Break

The *break statement* consists of the reserved word **break** and an optional label (18.13). ◇

⟨*breakStatement*⟩ ::= **break** ⟨*identifier*⟩? ';'

Let s_b be a **break** statement. If s_b is of the form **break** L**;**, then it is a compile-time error if s_b is not enclosed in a labeled statement with the label L within the innermost function in which s_b occurs. If s_b is of the form **break;**, then it is a compile-time error if s_b is not enclosed in an **await for** (18.6.3), **do** (18.8), **for** (18.6), **switch** (18.9) or **while** (18.7) statement within the innermost function in which s_b occurs.

Execution of a **break** statement **break** L**;** breaks to the label L (18.0.1). Execution of a **break** statement **break;** breaks without a label (18.0.1).

18.15 Continue

The *continue statement* consists of the reserved word **continue** and an optional label (18.13). ◇

⟨*continueStatement*⟩ ::= **continue** ⟨*identifier*⟩? ';'

Let s_c be a **continue** statement. If s_c is of the form **continue** L**;**, then it is a compile-time error if s_c is not enclosed in either an **await for** (18.6.3), **do** (18.8), **for** (18.6), or **while** (18.7) statement labeled with L, or in a **switch** statement with a case clause labeled with L, within the innermost function in which s_c occurs. If s_c is of the form **continue;** then it is a compile-time error if s_c is not enclosed in an **await for** (18.6.3) **do** (18.8), **for** (18.6), or **while** (18.7) statement within the innermost function in which s_c occurs.

Execution of a **continue** statement **continue** L**;** continues to the label L (18.0.1). Execution of a **continue** statement **continue;** continues without a label (18.0.1).

18.16 Yield

The *yield statement* adds an object to the result of a generator function (9). ◇

⟨*yieldStatement*⟩ ::= **yield** ⟨*expression*⟩ ';'

Let s be a yield statement of the form **yield** e**;**. Let f be the immediately enclosing function of s. It is a compile-time error if there is no such function, or it is not a generator. It is a compile-time error if the static type of e may not be assigned to the element type of f (9).

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

First, the expression e is evaluated to an object o. If the enclosing function m is marked **async*** (9) and the stream u associated with m has been paused, then the nearest enclosing asynchronous for loop (18.6.3), if any, is paused and execution of m is suspended until u is resumed or canceled.

Next, o is added to the iterable or stream associated with the immediately enclosing function.

Note that a dynamic error occurs if the dynamic type of o is not a subtype of the element type of said iterable or stream.

If the enclosing function m is marked **async*** and the stream u associated with m has been canceled, then the **yield** statement returns without an object (18.0.1), otherwise it completes normally.

*The stream associated with an asynchronous generator could be canceled by any code with a reference to that stream at any point where the generator was passivated. Such a cancellation constitutes an irretrievable error for the generator. At this point, the only plausible action for the generator is to clean up after itself via its **finally** clauses.*

Otherwise, if the enclosing function m is marked **async*** (9) then the enclosing function may suspend, in which case the nearest enclosing asynchronous for loop (18.6.3), if any, is paused first.

*If a **yield** occurred inside an infinite loop and the enclosing function never suspended, there might not be an opportunity for consumers of the enclosing stream to run and access the data in the stream. The stream might then accumulate an unbounded number of elements. Such a situation is untenable. Therefore, we allow the enclosing function to be suspended when a new object is added to its associated stream. However, it is not essential (and in fact, can be quite costly) to suspend the function on every **yield**. The implementation is free to decide how often to suspend the enclosing function. The only requirement is that consumers are not blocked indefinitely.*

If the enclosing function m is marked **sync*** (9) then:

- Execution of the function m immediately enclosing s is suspended until the nullary method `moveNext()` is invoked upon the iterator used to initiate the current invocation of m.
- The current call to `moveNext()` returns **true**.

18.17 Yield-Each

The *yield-each statement* adds a series of objects to the result of a generator function (9). ◇

⟨*yieldEachStatement*⟩ ::= **yield** '*' ⟨*expression*⟩ ';'

Let s be a yield-each statement of the form '**yield*** e;'. Let f be the immediately enclosing function of s. It is a compile-time error if there is no such function, or it is not a generator.

Let T_f be the element type of f (9), and let T be the static type of e. If f is a synchronous generator, it is a compile-time error if T may not be assigned to `Iterable<`T_f`>`. Otherwise f is an asynchronous generator, and it is a compile-time error if T may not be assigned to `Stream<`T_f`>`.

Execution of a statement s of the form '**yield*** e;' proceeds as follows:

First, the expression e is evaluated to an object o.

If the immediately enclosing function m is marked **sync*** (9), then:

1. It is a dynamic type error if the class of o is not a subtype of `Iterable<`T_f`>`. Otherwise
2. The method `iterator` is invoked upon o returning an object i.
3. The `moveNext` method of i is invoked on it with no arguments. If `moveNext` returns **false** execution of s is complete. Otherwise
4. The getter `current` is invoked on i. If the invocation throws (17.1), execution of s throws the same exception object and stack trace (18.0.1). Otherwise, the result x of the getter invocation is added to the iterable associated with m. Note that a dynamic error occurs if the dynamic type of x is not a subtype of the element type of said iterable. Execution of the function m immediately enclosing s is suspended until the nullary method `moveNext()` is invoked upon the iterator used to initiate the current invocation of m, at which point execution of s continues at 3.
5. The current call to `moveNext()` returns **true**.

If m is marked **async*** (9), then:

- It is a dynamic type error if the class of o is not a subtype of `Stream<`T_f`>`. Otherwise
- The nearest enclosing asynchronous for loop (18.6.3), if any, is paused.
- The o stream is listened to, creating a subscription s, and for each event x, or error e with stack trace t, of s:
  - If the stream u associated with m has been paused, then execution of m is suspended until u is resumed or canceled.
  - If the stream u associated with m has been canceled, then s is canceled by evaluating **await** `v.cancel()` where v is a fresh variable referencing the stream subscription s. Then, if the cancel completed normally, the stream execution of s returns without an object (18.0.1).

- Otherwise, x, or e with t, are added to the stream associated with m in the order they appear in o. Note that a dynamic error occurs if x is added and the dynamic type of x is not a subtype of the element type of said stream. The function m may suspend.

- If the stream o is done, execution of s completes normally.

18.18 Assert

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

⟨*assertStatement*⟩ ::= ⟨*assertion*⟩ '`;`'

⟨*assertion*⟩ ::= **assert** '`(`' ⟨*expression*⟩ ('`,`' ⟨*expression*⟩)? '`,`'? '`)`'

The grammar allows a trailing comma before the closing parenthesis, similarly to an argument list. That comma, if present, has no effect. An assertion with a trailing comma is equivalent to one with that comma removed.

An assertion of the form **assert**`(`e`)` is equivalent to an assertion of the form **assert**`(`e`, null)`.

Execution of an assert statement executes the assertion as described below and completes in the same way as the assertion.

When assertions are not enabled, execution of an assertion immediately completes normally (18.0.1). That is, no subexpressions of the assertion are evaluated. When assertions are enabled, execution of an assertion **assert**`(`c`,` e`)` proceeds as follows:

The expression c is evaluated to an object r. It is a dynamic type error if r is not of type `bool`. Hence it is a compile-time error if that situation arises during evaluation of an assertion in a **const** constructor invocation. If r is **true** then execution of the assert statement completes normally (18.0.1). Otherwise, e is evaluated to an object m and then the execution of the assert statement throws (18.0.1) an `AssertionError` containing m and with a stack trace corresponding to the current execution state at the assertion.

It is a compile-time error if the type of c may not be assigned to `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 when assertions are disabled. Also, in the absence of final methods, one could not prevent it being overridden (though there is no real harm in that). It cannot be viewed as a function call that is being optimized away because the arguments might have side effects.

19 Libraries and Scripts

A Dart program consists of one or more libraries, and may be built out of one or more *compilation units*. A compilation unit may be a library or a part (19.5). ◇

A library consists of (a possibly empty) set of imports, a set of exports, and a set of top-level declarations. A top-level declaration is either a class (10), a type alias declaration (20.3), a function (9) or a variable declaration (8). The members of a library L are those top level declarations given within L.

⟨*topLevelDeclaration*⟩ ::= ⟨*classDeclaration*⟩
  | ⟨*mixinDeclaration*⟩
  | ⟨*extensionDeclaration*⟩
  | ⟨*enumType*⟩
  | ⟨*typeAlias*⟩
  | **external** ⟨*functionSignature*⟩ ';'
  | **external** ⟨*getterSignature*⟩ ';'
  | **external** ⟨*setterSignature*⟩ ';'
  | ⟨*functionSignature*⟩ ⟨*functionBody*⟩
  | ⟨*getterSignature*⟩ ⟨*functionBody*⟩
  | ⟨*setterSignature*⟩ ⟨*functionBody*⟩
  | (**final** | **const**) ⟨*type*⟩? ⟨*staticFinalDeclarationList*⟩ ';'
  | **late final** ⟨*type*⟩? ⟨*initializedIdentifierList*⟩ ';'
  | **late**? ⟨*varOrType*⟩ ⟨*initializedIdentifierList*⟩ ';'

⟨*libraryDeclaration*⟩ ::=
    ⟨*scriptTag*⟩? ⟨*libraryName*⟩? ⟨*importOrExport*⟩* ⟨*partDirective*⟩*
    (⟨*metadata*⟩ ⟨*topLevelDeclaration*⟩)* ⟨*EOF*⟩

⟨*scriptTag*⟩ ::= '`#!`' (~('`\r`' | '`\n`'))* ⟨*LINE_BREAK*⟩

⟨*libraryName*⟩ ::= ⟨*metadata*⟩ **library** ⟨*dottedIdentifierList*⟩ ';'

⟨*importOrExport*⟩ ::= ⟨*libraryImport*⟩
  | ⟨*libraryExport*⟩

⟨*dottedIdentifierList*⟩ ::= ⟨*identifier*⟩ ('.' ⟨*identifier*⟩)*

A library contains a string which is derived from ⟨*libraryDeclaration*⟩.

We could say that ⟨*libraryDeclaration*⟩ is a *start symbol* of the grammar, because the syntactic derivation of any library starts from there. Unlike a traditional context free grammar, the Dart grammar does not have exactly one start symbol. In particular, ⟨*partDeclaration*⟩ (19.5) is used in the same manner to derive the contents of a part. There could be more, e.g., a hypothetical Dart REPL (read-eval-print loop) could use ⟨*expression*⟩ or ⟨*statement*⟩ as a start symbol. So there is no grammar for Dart programs as such, only for some building blocks that are used to construct Dart programs. ◇

Libraries may be explicitly named or implicitly named. An *explicitly named library* begins with the word **library** (possibly prefaced with any applicable metadata annotations), followed by a qualified identifier that gives the name of the library. ◇

Technically, each dot and identifier is a separate token and so spaces between them are acceptable. However, the actual library name is the concatenation of the simple identifiers and dots and contains no spaces.

An implicitly named library has the empty string as its name.

The name of a library is used to tie it to separately compiled parts of the library (called parts) and can be used for printing and, more generally, reflection. The name may be relevant for further language evolution.

Libraries intended for widespread use should avoid name collisions. Dart's pub package management system provides a mechanism for doing so. Each pub package is guaranteed a unique name, effectively enforcing a global namespace.

A library may optionally begin with a *script tag*. Script tags are intended for use with scripts (19.6). A script tag can be used to identify the interpreter of the script to whatever computing environment the script is embedded in. The script tag must appear before any whitespace or comments. A script tag begins with '`#!`' and ends at the end of the line. Any characters that follow '`#!`' in the script tag 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.

Since top level privates are not imported, using the top level privates of another library is never possible.

The *public namespace* of library L is the namespace that maps the name of each public top-level member declaration m of L to m. The *local namespace* of library L is the namespace that maps the names introduced by each top-level declaration of L to the corresponding declaration. The *library scope* of library L is the outermost scope in L, and its namespace is the library namespace of L (19.1.1). ◇ ◇ ◇

It is a compile-time error if the local namespace of library L has two declarations with the same basename, except when they are a getter and a setter.

Two distinct names n_1 and n_2 can only have the same basename when they are of the form id and $id=$, so this kind of conflict *always* involves a setter. But the other declaration could be a function, a class, etc.

19.1 Imports

An *import* specifies a library whose exported namespace (or a subset of its mappings) is made available in the current library. ◇

⟨*libraryImport*⟩ ::= ⟨*metadata*⟩ ⟨*importSpecification*⟩

⟨*importSpecification*⟩ ::=
  import ⟨*configurableUri*⟩ (**deferred**? **as** ⟨*identifier*⟩)? ⟨*combinator*⟩* '**;**'

The interpretation of configurable URIs is described elsewhere (19.7). An import specifies a URI s where the declaration of an imported library is to be found. It is a compile-time error if the specified URI of an import does not refer to a library declaration.

The *current library* is the library currently being compiled. The import ◇
modifies the namespace of the current library in a manner that is determined by the imported library and by the optional elements of the import.

Imports may be deferred or immediate. A *deferred import* is distinguished ◇
by the occurrence of the built-in identifier **deferred** after the URI. An *immediate* ◇
import is an import that is not deferred.

An immediate import directive I may optionally include a *prefix clause* of ◇
the form '**as** *id*' used to prefix names imported by I. In this case we say that *id* is an *import prefix*, or simply a *prefix*. ◇

Note that the grammar enforces that a deferred import includes a prefix clause, ◇
so we can refer to *the* prefix clause of a deferred import.

It is a compile-time error if the prefix used in a deferred import is also used as the prefix of another import clause. It is a compile-time error if *id* is an import prefix, and the current library declares a top-level member with basename *id*.

An import directive I may optionally include namespace combinator clauses used to restrict the set of names imported by I. Their syntax, usage, and effect on namespaces is described elsewhere (19.3, 19.1.1, 19.2).

The dart core library `dart:core` is implicitly imported into every dart library other than itself via an import clause of the form **import** `'dart:core';` unless the importing library explicitly imports `dart:core`. Any import of `dart:core`, even if restricted via **show**, **hide**, or **as**, preempts the automatic import.

It would be nice if there was nothing special about `dart:core`. However, its use is pervasive, which leads to the decision to import it automatically. On the other hand, some library L may wish to define entities with names used by `dart:core` (which it can easily do, as the names declared by a library take precedence). Other libraries may wish to use L, and may want to use members of L that conflict with the core library without having to use a prefix and without encountering errors. The above rule makes this possible, essentially canceling `dart:core`'s special treatment by means of yet another special rule.

19.1.1 The Imported Namespace

In the following, we specify the imported namespace of a library L, NS_{import}, and use that to define the namespace that defines the library scope of L.

We need to introduce system libraries because they have special rules. A *system library* is a library that is part of the Dart implementation. Any other ◇
library is a *non-system library*. ◇

A system library can generally be recognized by having a URI that starts with '`dart:`'.

The special rules for system libraries exist for the following reason. Normal conflicts are resolved at deployment time, but the functionality of a system library is injected into an application at run time, and may vary over time as the platform is upgraded. Thus, conflicts with a system library can arise outside the developer's control. To avoid breaking deployed applications in this way, conflicts with the system libraries are treated specially.

Let I be an import directive and let L be the library imported by I. The

namespace provided by by the import directive I is the namespace obtained from applying the namespace combinators of I to the exported namespace of L (19.3). ◇

Note that the namespace provided by an import directive I is not the same as the namespace imported from I. The latter includes conflict resolution, and is defined later in this section.

Let NS_{local} be the local namespace of L (19). Let $I_1, \ldots, I_m$ be the import directives of L, and let $L_1, \ldots, L_m$ be libraries such that I_i refers to L_i for all $i \in 1..m$. It is not an error to have multiple imports of the same library.

Let $i \in 1..m$.

Step one. In the first step we compute the namespace obtained from each imported library, respectively by each group of libraries imported with the same prefix, $NS_{one,i}$.

Case ⟨Step one for imports without a prefix⟩**.** When I_i has no prefix, $NS_{one,i}$ is obtained from the namespace provided by the import directive I_i by eliminating every binding for a name whose basename is the same as the basename of a top-level declaration in L, or whose basename is the prefix of an import directive in L.

This step ensures that **show** and **hide** directives are taken into account, and that an import prefix as well as a local declaration will shadow an imported name. □

Case ⟨Step one for prefixed imports⟩**.** In this step we resolve name conflicts among names imported with the same prefix. When I_i has prefix p_i, let $I'_1, \ldots, I'_k$ be the sublist of $I_1, \ldots, I_m$ that have prefix p_i, and let $L'_1, \ldots, L'_k$ be the corresponding libraries (which is a sublist of $L_1, \ldots, L_m$). Let $NS_{exported,j}$, $j \in 1..k$, be the namespace provided by the import directive I'_j (which takes **show** and **hide** into account).

When I_i is a non-deferred import: Let $NS_{prefix,i}$ be the namespace obtained from applying conflict merging to $NS_{exported,1}, \ldots, NS_{exported,k}$ (19.4).

When I_i is a deferred import: In this case k is 1 (otherwise a compile-time error would occur), and $NS_{prefix,i}$ is the namespace obtained by adding a binding of the name `loadLibrary` to an implicitly induced declaration of a function with signature `Future<void> loadLibrary()` to $NS_{exported,1}$.

Then $NS_{one,i}$ is a namespace that has a single binding that maps p_i to $NS_{prefix,i}$.

In this situation we say that $NS_{one,i}$ is a *prefix namespace*, because it maps a library prefix to a namespace. ◇

A prefix namespace is not a Dart object, it is merely a device which is used to manage expressions of the form ⟨*qualifiedName*⟩ and what they refer to during static analysis. Consequently, any attempt to use a prefix namespace as an object is a compile-time error, e.g., it cannot be the result of an expression evaluation.

Note that if l and q are such that I_l and I_q both have the same prefix p then $NS_{one,l} = NS_{one,q} = NS_{one,l} \cup NS_{one,q}$. □

Step two. In the second step we resolve top level conflicts among the namespaces obtained in the first step.

The *imported namespace* of L, NS_{import}, is then the result of applying conflict ◇

merging to the namespaces $NS_{one,1}, \ldots, NS_{one,m}$.

Let $\mathcal{E}$ be a set of extension declarations (13) with the following members: An extension declaration E is a member of $\mathcal{E}$ if there is an $i \in 1..m$ such that E is in the namespace provided by the import directive I_i (note that this takes **show** and **hide** into account, and it includes extensions imported both without and with a prefix), and E is not declared in the current library (which could be the case if it imports itself). Let $NS_{extensions}$ be a namespace that for each extension E in $\mathcal{E}$ maps a fresh name to E.

$NS_{extensions}$ provides a fresh name allowing implicit access to each extension exported by an imported library and not removed by **hide** or **show**, even the ones that cannot be accessed using their declared name, because of a name clash.

The *library namespace* of L is then $NS_{local} \cup NS_{import} \cup NS_{extensions}$. ◇

Let $i \in 1..m$. If L_i is imported without a prefix, the *namespace imported from* L_i is the conflict narrowed namespace (19.4) of $NS_{one,i}$. Otherwise L_i is imported with a prefix p, in which case the *namespace imported from* L_i is the conflict narrowed namespace of $NS_{exported,i}$. ◇ ◇

So the namespace imported by L_i contains the bindings exported by L_i, except the ones removed by namespace combinators, and except the ones removed by conflict merging.

Let L be a library with imported namespace $NS_{.}$. We say that a name is *imported by* L if the name is a key of $NS_{.}$. We say that a declaration is *imported by* L if the declaration is a value of $NS_{.}$. We say that a name is *imported by* L *with prefix* p if the name is a key of $NS(p)$. We say that a declaration is *imported by* L *with prefix* p if the declaration is a value of $NS(p)$. ◇ ◇ ◇ ◇

19.1.2 Semantics of Imports

Let I_i be an import directive that refers to a URI via the string s_i. The semantics of I_i is specified as follows:

Case ⟨Semantics of deferred imports⟩**.** If I_i is a deferred import with prefix p, a binding of p to a *deferred prefix run-time namespace* $NS_{deferred}$ is present in the library namespace of the current library L. Let $NS_{import,i}$ be the namespace imported from the library specified by I_i, as defined previously (19.1.1). $NS_{deferred}$ then has the following bindings: ◇

- The name `loadLibrary` is bound to a function with signature `Future<void> loadLibrary()`. This function returns a future f. When called, the function causes an immediate import I' to be executed at some future time, where I' is derived from I_i by eliding the word **deferred** and adding a **hide** `loadLibrary` combinator clause. The execution of the immediate import may fail for implementation specific reasons. For instance, I' imports a different library than the one that the specified URI referred to at compile-time; or an OS level file read error occurs; etc. We say that the invocation of `loadLibrary` *succeeds* if f completes with a value, and that the invocation *fails* if f completes with an error. ◇ ◇

- For every top level function f named id in $NS_{import,i}$, a corresponding function named id with the same signature as f. Calling the function results in a dynamic error, and so does closurizing it (17.16).
- For every top level getter g named id in $NS_{import,i}$, a corresponding getter named id with the same signature as g. Calling the getter results in a dynamic error.
- For every top level setter s named id= in $NS_{import,i}$, a corresponding setter named id= with the same signature as s. Calling the setter results in a dynamic error.
- For every class, mixin and type alias declaration named id in $NS_{import,i}$, a corresponding getter named id with return type `Type`. Calling the getter results in a dynamic error.

The purpose of having members of the imported library in $NS_{deferred}$ is to ensure that usages of members that have not yet been loaded can be resolved normally and has a well-defined behavior, which will raise errors.

When an invocation of p`.loadLibrary()` succeeds, the name p is mapped to a non-deferred prefix run-time namespace NS_{loaded}, with bindings as described below for immediate imports. In addition, NS_{loaded} maps `loadLibrary` to a function with the same signature as before, and so it is possible to invoke p`.loadLibrary()` again, which will always succeed. If a call fails, the library has not been loaded, and one has the option to invoke p`.loadLibrary()` again. Whether a repeated call to p`.loadLibrary()` succeeds will vary, as described below.

Note that it is a compile-time error for a deferred prefix to be used in more than one import, which means that the update of the binding of p does not interfere with other imports.

The effect of a repeated invocation of p`.loadLibrary()` is as follows:

- If another invocation of p`.loadLibrary()` has already succeeded, the repeated invocation also succeeds. Otherwise,
- If another invocation of p`.loadLibrary()` has failed:
  - If the failure is due to a compilation error, the repeated invocation fails for the same reason.
  - If the failure is due to other causes, the repeated invocation behaves as if no previous call had been made.

In other words, a successful `loadLibrary()` guarantees that the import can be successfully accessed through the import prefix. If an invocation of `loadLibrary()` is initiated after another invocation has completed successfully, it is guaranteed to also complete successfully (success is idempotent). We do not specify which object the returned future resolves to. □

Case ⟨Semantics of immediate imports⟩**.** When I_i is an immediate import

with a library URI, u_i, represented by the string s_i: Let L_i be the library obtained from the source code denoted by s_i. We then say that the URI u_i denotes the library L_i. All imports and exports of the same URI in a Dart program denotes the same library, and imports or exports of different URIs denote distinct libraries.

Let $NS_{import,i}$ be the namespace imported from L_i. The run-time namespace NS_i will then have the following bindings:

- For every top level function, getter, or setter m named n in $NS_{import,i}$, a binding from n to said function, getter, or setter.
- For every name *id* in $NS_{import,i}$ that is bound to a class, mixin, or type alias declaration introducing a type T, a binding from *id* to the compiled representation of T.

If I_i has prefix p, the run-time namespace of the current library maps p to a non-deferred prefix run-time namespace NS_p containing the mappings of NS_i.

NS_p may have additional mappings because there can be several imports with the same prefix as long as they are all immediate.

Otherwise, when I_i does not have a prefix, the run-time library namespace of the current library contains each mapping in NS_i. □

19.2 Exports

A library L exports a namespace (6.1), meaning that the declarations in the namespace are made available to other libraries if they choose to import L (19.1). The namespace that L exports is known as its *exported namespace*. ◇

A library always exports all names and all declarations in its public namespace. In addition, a library may choose to re-export additional libraries via *export directives*, often referred to simply as *exports*: ◇ ◇

⟨*libraryExport*⟩ ::= ⟨*metadata*⟩ **export** ⟨*configurableUri*⟩ ⟨*combinator*⟩* ';'

The interpretation of configurable URIs is described elsewhere (19.7). An export specifies a URI s where the declaration of an exported library is to be found. It is a compile-time error if the specified URI does not refer to a library declaration.

The *exported namespace* of a library is determined as follows, in two steps. ◇ Let L be a library, let $E_1, \ldots, E_m$ be the export directives of L, and let $L_1, \ldots, L_m$ be libraries such that E_i refers to L_i for all $i \in 1..m$. Let NS_{public} be the public namespace of L (19).

Note that private names and import prefixes are not present in NS_{public}.

In the first step we compute the namespace provided by each exported library: For each $i \in 1..m$, $NS_{exported,i}$ is the namespace obtained from applying the namespace combinators of E_i to the exported namespace of L_i (19.3), and removing each binding of a name n such that NS_{public} is defined at n', and n and n' have the same basename. Because local declarations will shadow re-exported ones.

The *namespace re-exported from* L_i is $NS_{exported,i}$. ◇

In the second step we compute the exported namespace of L. Let NS_{merged} be the result of applying conflict merging (19.4) to $NS_{exported,1}$, ..., $NS_{exported,m}$. A compile-time error occurs if any name in NS_{merged} is conflicted (19.4).

This rule is more strict than the corresponding rule for imports: When two imported declarations have a name clash, it is only an error to use *the conflicted name, it is not an error that the name clash exists. With exported names, it is an error that the name clash exists. The reason for this difference is that the conflict could silently break importers of the current library L, if we were to use the same approach for exports as for imports: If a library L' imports L and uses a name n which is re-exported by L from L_n then the addition of a declaration named n to some other re-exported library will make the use of n in L' an error, and the maintainers of L' may not be in a position to change L.*

The *exported namespace* of L is $NS_{public} \cup NS_{merged}$. ◇

We say that a name is *exported by a library* if the name is in the library's ◇ exported namespace. We say that a declaration *is exported by a library* if the ◇ declaration is in the library's exported namespace.

For a given i, we say that L *re-exports library* L_i, and also that L *re-exports* ◇ *namespace* $NS_{exported,i}$. When no confusion can arise, we may simply state that ◇ L *re-exports* L_i, or that L *re-exports* $NS_{exported,i}$. ◇ ◇

19.3 Namespace Combinators

Imports (19.1) and exports (19.2) rely on *namespace combinators* in order to ◇ adjust namespaces (6.1) and manage name clashes. The supported namespace combinators are **show** and **hide**.

⟨*combinator*⟩ ::= **show** ⟨*identifierList*⟩ | **hide** ⟨*identifierList*⟩

⟨*identifierList*⟩ ::= ⟨*identifier*⟩ (‘**,**’ ⟨*identifier*⟩)*

We define several operations that compute namespaces. The *union of two* ◇ *namespaces*, $NS_a \cup NS_b$, is defined when every key where both are defined is ◇ mapped to the same value. This union is the namespace that maps each key n of NS_a to the corresponding value $NS_a(n)$, and each key n of NS_b to $NS_b(n)$.

Note that this *is* a namespace because the union is only defined when the mapping of any given key is unambiguous.

The function $hide(l, NS_a)$ takes a list of identifiers l and a namespace NS_a, ◇ and produces a namespace NS_b that is identical to NS_a except that for each identifier *id* in l, NS_b is undefined at *id* and at *id*=.

The function $show(l, NS_a)$ takes a list of identifiers l and a namespace NS_a, ◇ and produces a namespace NS_b that maps each identifier *id* in l where NS_a is defined to $NS_a(n)$. Furthermore, for each identifier *id* in l where NS_a is defined at *id*=, NS_b maps *id*= to $NS_a(id=)$. Finally, NS_b is undefined at all other names.

Let C_1, ..., C_n be a sequence of combinator clauses. They would come from an import or export directive. The result of *applying the combinator clauses* to a ◇

given namespace NS_{start} is a namespace NS_{end}, which is computed as follows.

Let NS_0 be NS_{start}. For each combinator clause C_j, $j \in 1..n$: If C_j is of the form **show** $id_1, \ldots, id_k$ then let $NS_j = show([id_1, \ldots, id_k], NS_{j-1})$. If C_i is of the form **hide** $id_1, \ldots, id_k$ then let $NS_j = hide([id_1, \ldots, id_k], NS_{j-1})$. Then NS_{end} is NS_n.

NS_{end} will always agree with NS_{start} wherever both are defined, and the set of names where NS_{end} is defined is always a subset of that of NS_{start}. In that sense, this is a *narrowing* procedure.

Note that it is possible to use **hide** or **show** on an identifier which is not in the given namespace. *Allowing this prevents situations where, e.g., removing a declaration from a library L would cause breakage in a library that imports L.*

19.4 Conflict Merging of Namespaces

In this section we define an operation on namespaces which eliminates certain bindings of names in case of name clashes, and keeps track of such name clashes using a special value, *NAME_CONFLICT*. ◇

When a name n is mapped to NAME_CONFLICT by a namespace, we say that n is *conflicted*. A lookup for a conflicted name will succeed according to ◇ the normal rules for namespaces and lexical scoping, but it is a compile-time error at the location where the name is used that this name is conflicted.

Let $NS_1, \ldots, NS_m$ be a list of namespaces (the list may or may not contain duplicates). The *conflict merging* of $NS_1, \ldots, NS_m$ is then $NS_{conflict} \cup$ ◇ $NS_{narrowed,1} \cup \ldots NS_{narrowed,m}$, where the namespaces $NS_{conflict}$ and $NS_{narrowed,i}$, $i \in 1..m$, are specified as follows:

$NS_{conflict}$ is empty except for the bindings mentioned below. For each $i \in 1..m$, $NS_{narrowed,i}$ is identical to NS_i, except that the former is undefined at each name n where the latter is defined, and one of the following conditions holds:

- There exists a $j \in 1..m$ and name n' such that NS_j is defined at n', n and n' have the same basename, $NS_i(n)$ is a declaration in a system library, and $NS_j(n')$ is a declaration in a non-system library. So a declaration from a non-system library shadows declarations from system libraries.

- Otherwise, there exists a $j \in 1..m$ and name n' such that NS_j is defined at n', n and n' have the same basename, $NS_i(n)$ and $NS_j(n')$ are not the same declaration and not the same namespace and not a getter and a setter declared in the same library, and either none or both of $NS_i(n)$ and $NS_j(n')$ are declarations in a system library. So with two distinct declarations with the same basename, both are eliminated, except when it is a getter and a setter from the same library. Note that $NS_i(n)$ and $NS_j(n')$ must both be declarations or must both be namespaces, because conflict merging is applied to namespaces where imported declarations that conflict with an import prefix have already been eliminated. When they are both namespaces there is no conflict, because it is then the same namespace.

In this situation n is mapped to NAME_CONFLICT by $NS_{conflict}$. We can swap all names and use the rule again, so n' is also conflicted.

In short, conflict merging takes a list of namespaces and produces a namespace that is the union of several disjoint namespaces (because all name clashes have been eliminated): A conflict namespace that records all unresolved name clashes, and a list of namespaces where clashing names have been removed. Some name clashes have been resolved by preferring declarations from non-system libraries over declarations from system libraries.

It is useful to be able to refer to the result of narrowing. Let $NS_1, \ldots, NS_m$ be a list of namespaces, $i \in 1..m$, and consider the conflict merging as specified above. The *conflict narrowed namespace* of NS_i is then $NS_{narrowed,i}$. ◇

19.5 Parts

A library may be divided into *parts*, each of which can be stored in a separate location. A library identifies its parts by listing them via **part** directives. ◇

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

⟨*partDirective*⟩ ::= ⟨*metadata*⟩ **part** ⟨*uri*⟩ '**;**'

⟨*partHeader*⟩ ::= ⟨*metadata*⟩ **part of** (⟨*dottedIdentifierList*⟩ | ⟨*uri*⟩) '**;**'

⟨*partDeclaration*⟩ ::= ⟨*partHeader*⟩ (⟨*metadata*⟩ ⟨*topLevelDeclaration*⟩)* ⟨*EOF*⟩

A part contains a string which is derived from ⟨*partDeclaration*⟩.

So we could say that ⟨*partDeclaration*⟩ is a start symbol of the grammar, as discussed in Sect. 19.

A *part header* begins with **part of** followed by the name of the library the part belongs to, or a ⟨*uri*⟩ denoting said library. A part declaration consists of a part header followed by a sequence of top-level declarations. ◇

Compiling a part directive of the form **part** 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 part declaration. It is a compile-time error if the referenced part declaration p names a library other than the current library as the library to which p belongs.

It is a compile-time error if a library contains two part directives with the same URI.

We say that a library L_1 is *reachable from* a library L if any of the following is true (19.1, 19.2): ◇

- L and L_1 is the same library.
- L imports or exports a library L_2, and L_1 is reachable from L_2.

Let L be a library, let u be a URI, and let L_1 and L_2 be distinct libraries which are reachable from L. It is a compile-time error if L_1 and L_2 both contain a part directive with URI u.

In particular, it is an error to use the same part twice in the same program (19.6). Note that a relative URI is interpreted as relative to the location of the enclosing library (19.7), which means that L_1 and L_2 may both have a part identified by `'myPart.dart'`, but they are not the same URI unless L_1 and L_2 have the same location.

19.6 Scripts

A *script* is a library whose exported namespace (19.2) includes a top-level function declaration named `main` that has either zero, one or two required arguments. ◇

A script S is executed as follows:

First, S is compiled as a library as specified above. Then, the top-level function defined by `main` in the exported namespace of S is invoked (17.15) as follows: If `main` can be called with with two positional arguments, it is invoked with the following two actual arguments:

1. An object whose run-time type implements `List<String>`.
2. An object specified when the current isolate i was created, for example through the invocation of `Isolate.spawnUri` that spawned i, or the null object (17.4) if no such object was supplied.

If `main` cannot be called with two positional arguments, but it can be called with one positional argument, it is invoked with an object whose run-time type implements `List<String>` as the only argument. If `main` cannot be called with one or two positional arguments, it is invoked with no arguments.

Note that if `main` requires more than two positional arguments, the library is not considered a script.

A Dart program will typically be executed by executing a script.

It is a compile-time error if a library's export scope contains a declaration named `main`, and the library is not a script. This restriction ensures that all top-level `main` declarations introduce a script main-function, so there cannot be a top-level getter or field named `main`, nor can it be a function that requires more than two arguments. The restriction allows tools to fail early on invalid `main` methods, without needing to know whether a library will be used as the entry point of a Dart program. It is possible that this restriction will be removed in the future.

19.7 URIs

URIs are specified by means of string literals:

⟨*uri*⟩ ::= ⟨*stringLiteral*⟩

⟨*configurableUri*⟩ ::= ⟨*uri*⟩ ⟨*configurationUri*⟩*

⟨*configurationUri*⟩ ::= **if** '(' ⟨*uriTest*⟩ ')' ⟨*uri*⟩

⟨*uriTest*⟩ ::= ⟨*dottedIdentifierList*⟩ ('==' ⟨*stringLiteral*⟩)?

It is a compile-time error if a string literal that describes a URI or a string literal that is used in a ⟨*uriTest*⟩ contains a string interpolation.

A *configurable URI* c of the form $uri\ \ configurationUri_1\ \ldots configurationUri_n$ ◇ *specifies a URI* as follows: ◇

- Let u be uri.
- For each of the following configuration URIs of the form **if** $(test_i)\ uri_i$, in source order, do the following.
  - If $test_i$ is *ids* with no '==' clause, it is equivalent to *ids* `== "true"`.
  - If $test_i$ is *ids* `==` *string*, then create a string, *key*, from *ids* by concatenating the identfiers and dots, omitting any spaces between them that may occur in the source.
  - Look up *key* in the available *compilation environment*. ◇ The compilation environment is provided by the platform. It maps some string keys to string values, and can be accessed programmatically using the `const String.fromEnvironment` constructor. Tools may choose to only make some parts of the compilation environment available for choosing configuration URIs.
  - If the environment contains an entry for *key* and the associated value is equal, as a constant string value, to the value of the string literal *string*, then let u be uri_i and stop iterating the configuration URIs.
  - Otherwise proceed to the next configuration URI.
- The URI specified by c is u.

This specification does not discuss the interpretation of URIs, with the following exceptions.

The interpretation of URIs is mostly left to the surrounding computing environment. For example, if Dart is running in a web browser, that browser will likely interpret some URIs. While it might seem attractive to specify, say, that URIs are interpreted with respect to a standard such as IETF RFC 3986, in practice this will usually depend on the browser and cannot be relied upon.

A URI of the form `dart:`s is interpreted as a reference to a system library (19.1) s.

A URI of the form `package:`s is interpreted in an implementation specific manner.

The intent is that, during development, Dart programmers can rely on a package manager to find elements of their program.

Otherwise, any relative URI is interpreted as relative to the location of the current library. All further interpretation of URIs is implementation dependent.

This means it is dependent on the runtime.

20 Types

Dart supports static typing based on interface types.

The type system is sound in the sense that if a variable of type T refers to an object of type S at run time, then S is a subtype of T. In other words, the contents of the heap satisfies the expectations expressed by static typing.

However, type parameters are covariant (e.g., `List<int>` <: `List<num>`) and this implies that certain operations are subject to dynamic type checks (such as `myList.add(1.5)`, which will throw at run time if `myList` has declared type `List<num>`, but it is actually a `List<int>`). Hence, a program can be free of compile-time errors, and it may still incur a type error at run time.

This choice was made deliberately during the early days of Dart (and it is undoubtedly controversial). It represents a trade-off where the potential for run-time type errors is the cost, and the benefit is simpler programs. In recent years, Dart has evolved to have more and more static type safety, e.g., null safety, and this trend is likely to continue.

20.1 Static Types

Type annotations can occur in variable declarations (8), including formal parameters (9.2), in the return types of functions (9), and in the bounds of type variables (15). Type annotations are used during static checking and when running programs. Types are specified using the following grammar rules.

A ⟨*typeIdentifier*⟩ is an identifier which can be the name of a type, that is, it denotes an ⟨*IDENTIFIER*⟩ which is not a ⟨*BUILT_IN_IDENTIFIER*⟩ (17.38).

Non-terminals with names of the form ⟨*...NotFunction*⟩ derive terms which are types that are not function types. Note that it *does* derive the type **Function**, which is not itself a function type, but it is the least upper bound of all function types.

⟨*type*⟩ ::= ⟨*functionType*⟩ '?'?
  | ⟨*typeNotFunction*⟩

⟨*typeNotVoid*⟩ ::= ⟨*functionType*⟩ '?'?
  | ⟨*typeNotVoidNotFunction*⟩

⟨*typeNotFunction*⟩ ::= **void**
  | ⟨*typeNotVoidNotFunction*⟩

⟨*typeNotVoidNotFunction*⟩ ::= ⟨*typeName*⟩ ⟨*typeArguments*⟩? '?'?
  | **Function** '?'?

⟨*typeName*⟩ ::= ⟨*typeIdentifier*⟩ ('.' ⟨*typeIdentifier*⟩)?

⟨*typeArguments*⟩ ::= '<' ⟨*typeList*⟩ '>'

⟨*typeList*⟩ ::= ⟨*type*⟩ (‘`,`’ ⟨*type*⟩)*

⟨*typeNotVoidNotFunctionList*⟩ ::=
⟨*typeNotVoidNotFunction*⟩ (‘`,`’ ⟨*typeNotVoidNotFunction*⟩)*

⟨*functionType*⟩ ::= ⟨*functionTypeTails*⟩
| ⟨*typeNotFunction*⟩ ⟨*functionTypeTails*⟩

⟨*functionTypeTails*⟩ ::= ⟨*functionTypeTail*⟩ ‘`?`’? ⟨*functionTypeTails*⟩
| ⟨*functionTypeTail*⟩

⟨*functionTypeTail*⟩ ::= **Function** ⟨*typeParameters*⟩? ⟨*parameterTypeList*⟩

⟨*parameterTypeList*⟩ ::= ‘`(`’ ‘`)`’
| ‘`(`’ ⟨*normalParameterTypes*⟩ ‘`,`’ ⟨*optionalParameterTypes*⟩ ‘`)`’
| ‘`(`’ ⟨*normalParameterTypes*⟩ ‘`,`’? ‘`)`’
| ‘`(`’ ⟨*optionalParameterTypes*⟩ ‘`)`’

⟨*normalParameterTypes*⟩ ::=
⟨*normalParameterType*⟩ (‘`,`’ ⟨*normalParameterType*⟩)*

⟨*normalParameterType*⟩ ::= ⟨*metadata*⟩ ⟨*typedIdentifier*⟩
| ⟨*metadata*⟩ ⟨*type*⟩

⟨*optionalParameterTypes*⟩ ::= ⟨*optionalPositionalParameterTypes*⟩
| ⟨*namedParameterTypes*⟩

⟨*optionalPositionalParameterTypes*⟩ ::= ‘`[`’ ⟨*normalParameterTypes*⟩ ‘`,`’? ‘`]`’

⟨*namedParameterTypes*⟩ ::= ‘`{`’ ⟨*namedParameterType*⟩ (‘`,`’ ⟨*namedParameterType*⟩)*
‘`,`’? ‘`}`’

⟨*namedParameterType*⟩ ::= ⟨*metadata*⟩ **required**? ⟨*typedIdentifier*⟩

⟨*typedIdentifier*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩

A Dart implementation must provide a static checker that detects and reports exactly those situations this specification identifies as compile-time errors, and only those situations. Similarly, the static checker must emit static warnings for at least the situations specified as such in this specification.

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 must not preclude successful compilation and execution of Dart code.

A type T is *malformed* iff: ◇

- T has the form *id* or the form *prefix*.*id*, and it does not denote a declaration of a type.
- T denotes a type variable, but it occurs in the signature or body of a static member.
- T is a parameterized type of the form G<$S_1, \ldots, \ S_n$>, and G is malformed, or G is not a generic type, or G is a generic type, but it declares n' type parameters and $n' \neq n$, or S_j is malformed for some $j \in 1..n$.
- T is a function type of the form

  T_0 **Function**<X_1 **extends** $B_1, \ldots, \ X_m$ **extends** B_m>
  (`(`$T_1 \ x_1, \ldots, \ T_k \ x_k$, `[`$T_{k+1} \ x_{k+1}, \ldots, \ T_n \ x_n$`])`

  or of the form

  T_0 **Function**<X_1 **extends** $B_1, \ldots, \ X_m$ **extends** B_m>
  (`(`$T_1 \ x_1, \ldots, \ T_k \ x_k$, `{`$T_{k+1} \ x_{k+1}, \ldots, \ T_n \ x_n$`})`

  where each x_j which is not a named parameter may be omitted, and T_j is malformed for some $j \in 0..n$, or B_j is malformed for some $j \in 1..m$.
- T denotes declarations that were imported from multiple imports clauses.

Any occurrence of a malformed type in a library is a compile-time error.

A type T is *deferred* iff it is of the form $p.T$ where p is a deferred prefix. It is a compile-time error to use a deferred type in a type annotation, type test, type cast or as a type parameter. However, all other compile-time errors must be issued under the assumption that all deferred libraries have successfully been loaded. ◇

20.1.1 Type Promotion

The static type system ascribes a static type to every expression. In some cases, the type of a local variable (which can be a formal parameter) may be promoted from the declared type, based on control flow.

We say that a variable v is known to have type T whenever we allow the type of v to be promoted. The exact circumstances when type promotion is allowed are given in the relevant sections of the specification (17.26, 17.24 and 18.5).

Type promotion for a variable v is allowed only when we can deduce that such promotion is valid based on an analysis of certain boolean expressions. In such cases, we say that the boolean expression b shows that v has type T. As a rule, for all variables v and types T, a boolean expression does not show that v has type T. Those situations where an expression does show that a variable has a type are mentioned explicitly in the relevant sections of this specification (17.39 and 17.26).

20.2 Dynamic Type System

Let o be an instance. The *dynamic type* of o is the class which is specified for the situation where o was obtained as a fresh instance (10.7.2, 17.9.4, 17.9.8, 17.13.1, 17.15). ◇

In particular, the dynamic type of an instance never changes. It is at times only specified that the given class implements a certain type, e.g., for a list literal. In these cases the dynamic type is implementation dependent, except of course that said superinterface constraint must be satisfied.

The dynamic types of a running Dart program are equivalent to the static types with regard to subtyping.

Certain dynamic type checks are performed during execution (8, 10.7.1, 10.7.1, 10.7.2, 10.7.2, 17.9.4, 17.13.1, 17.15.3, 17.21.1, 17.23, 17.40, 18.3, 18.6.3, 18.12, 18.17, 18.18). As specified in those locations, these dynamic checks are based on the dynamic types of instances, and the actual types of declarations (20.10.1).

When types are reified as instances of the built-in class `Type`, those objects override the '`==`' operator inherited from the `Object` class, so that two `Type` objects are equal according to operator '`==`' iff the corresponding types are subtypes of each other.

For example, the `Type` objects for the types **dynamic** and `Object` are equal to each other and hence `dynamic == Object` must evaluate to **true**. No constraints are imposed on the built-in function `identical`, so `identical(dynamic, Object)` may be **true** or **false**.

Similarly, `Type` instances for distinct type alias declarations declaring a name for the same function type are equal:

```
typedef F = void Function<X>(X);
typedef G = void Function<Y>(Y);

void main() {
  assert(F == G);
}
```

Instances of `Type` can be obtained in various ways, for example by using reflection, by reading the `runtimeType` of an object, or by evaluating a *type literal* expression.

An expression is a *type literal* if it is an identifier, or a qualified identifier, which denotes a class, mixin or type alias declaration, or it is an identifier denoting a type parameter of a generic class or function. It is a *constant type literal* if it does not denote a type parameter, and it is not qualified by a deferred prefix. A constant type literal is a constant expression (17.3).

20.3 Type Aliases

A *type alias* declares a name for a type, or for a mapping from type arguments to types. ◇

It is common to use the phrase "a typedef" for such a declaration, because of the prominent occurrence of the token **typedef**.

⟨*typeAlias*⟩ ::=
    typedef ⟨*typeIdentifier*⟩ ⟨*typeParameters*⟩? '**=**' ⟨*type*⟩ '**;**'
  | **typedef** ⟨*functionTypeAlias*⟩

⟨*functionTypeAlias*⟩ ::= ⟨*functionPrefix*⟩ ⟨*formalParameterPart*⟩ '**;**'

⟨*functionPrefix*⟩ ::= ⟨*type*⟩? ⟨*identifier*⟩

Consider a type alias declaration D of the form
typedef id**<**X_1 **extends** B_1, ..., X_s **extends** B_s**> = ** T**;**
declared in a library L. The effect of D is to introduce id into the library scope of L. When $s = 0$ (where the type alias is non-generic) id is bound to T. When $s > 0$ (where the type alias is generic) id is bound to a mapping from a type argument list $U_1, \ldots, U_s$ to the type $[U_1/X_1, \ldots, U_s/X_s]T$.

Under the assumption that $X_1, \ldots, X_s$ are types such that $X_j <: B_j$, for all $j \in 1..s$, it is a compile-time error if T is not regular-bounded, and it is a compile-time error if any type occurring in T is not well-bounded.

This means that the bounds declared for the formal type parameters of a generic type alias must be such that when they are satisfied, the bounds that pertain to the body are also satisfied, and a type occurring as a subterm of the body can violate its bounds, but only if it is a correct super-bounded type.

Moreover, let $T_1, \ldots, T_l$ be types and let U be the parameterized type id**<**$T_1, \ldots, T_l$**>** in a location where id denotes D. It is a compile-time error if $l \neq s$. It is a compile-time error if U is not well-bounded (15.2).

For historic reasons, a type alias can have two more forms, derived using ⟨*functionTypeAlias*⟩. Let S? be a term which is empty or derived from ⟨*type*⟩, and similarly for T_j? for any j. The two older forms are then defined as follows: S? T_j?

A type alias of the form
typedef S? id**<**X_1 **extends** B_1, ..., X_s **extends** B_s**>(**
  T_1? $p_1, \ldots,$ T_n? $p_n,$ $[T_{n+1}$? $p_{n+1}, \ldots,$ T_{n+k}? $p_{n+k}]$**);**
is treated as
typedef id**<**X_1 **extends** B_1, ..., X_s **extends** B_s**> =**
  S? **Function(**T_1 $p_1, \ldots,$ T_n $p_n,$ $[T_{n+1}$ $p_{n+1}, \ldots,$ T_{n+k} $p_{n+k}]$**);**

A type alias of the form
typedef S? id**<**X_1 **extends** B_1, ..., X_s **extends** B_s**>(**
  T_1? $p_1, \ldots,$ T_n? $p_n,$ $\{T_{n+1}$? $p_{n+1}, \ldots,$ T_{n+k}? $p_{n+k}\}$**);**
is treated as
typedef id**<**X_1 **extends** B_1, ..., X_s **extends** B_s**> =**
  S? **Function(**T_1 $p_1, \ldots,$ T_n $p_n,$ $\{T_{n+1}$ $p_{n+1}, \ldots,$ T_{n+k} $p_{n+k}\}$**);**

In these rules, for each j, if T_j? is empty then T_j is **dynamic**, otherwise T_j is T_j?. T_j

This means that the older forms allow for a parameter type to be omitted, in which case it is taken to be **dynamic**, but the parameter name cannot be omitted.

This behavior is error prone, and hence the newer form (with '=') is recommended. For instance, the declaration **typedef** F`(int);` specifies that F denotes the type **dynamic Function**(**dynamic**), and it is documented, but technically ignored, that the parameter has the name `int`. It is extremely likely that a reader will misread this, and assume that `int` is the parameter type.

It is a compile-time error if a default value is specified for a formal parameter in these older forms, or if a formal parameter has the modifier **covariant**.

Note that the old forms can only define function types, and they cannot denote the type of a generic function. When such a type alias has type parameters, it always expresses a family of non-generic function types. These restrictions exist because that syntax was defined before generic functions were added to Dart.

Let D_F be a type alias declaration of the form
typedef F`<`X_1 **extends** B_1, ..., X_s **extends** B_s`> = `T`;`
If T or B_j for some $j \in 1..s$ is or contains a $\langle$*typeName*$\rangle$ G denoting a type alias declaration D_G, then we say that D_F *depends on* D_G. ◇

Let D be a type alias declaration, and let M be the transitive closure of the type alias declarations that D depends on. A compile-time error occurs if $D \in M$.

In other words, it is an error for a type alias declaration to depend on itself, directly or indirectly.

This kind of error may also arise when type arguments have been omitted in the program, but are added during static analysis via instantiation to bound (15.3) or via type inference, which will be specified later (6).

When D is a type alias declaration of the form
typedef F`<`X_1 **extends** B_1, ..., X_s **extends** B_s`> = `T`;`
we say that the parameterized type U of the form F`<`U_1, ..., U_s`>` in a scope where F resolves to D *alias expands in one step to* $[U_1/X_1, \ldots, U_s/X_s]T$. ◇

Note that s can be zero, in which case F is non-generic, and we are just replacing the type alias name by its body.

If U is a type we may repeatedly replace each subterm of U which is a parameterized type applying a type alias to some type arguments by its alias expansion in one step (including the non-generic case where there are no type arguments). When no further steps are possible we say that the resulting type is the *transitive alias expansion* of U. ◇

Note that the transitive alias expansion exists, because it is an error for a type alias declaration to depend on itself. However, it may be large. For instance, **typedef** F`<`X`> = Map<`X`, `X`>;` yields an exponential size increase for F`<`F`<...`F`<int>...>>`.

Let D be a type alias declaration of the form — D, F, T
typedef F`<`X_1 **extends** B_1, ..., X_s **extends** B_s`> = `T`;`
and let U be a type of the form F or p`.`F in a scope where this term denotes D. — U
Assume that the transitive alias expansion of F`<`X_1, ..., X_s`>` is a type of the form C or q`.`C, optionally followed by $\langle$*typeArguments*$\rangle$, where q is an identifier denoting an import prefix, and C respectively q`.`C denotes a class or mixin (in particular, C can not be a type variable). Assume that U occurs in an expression e of the form 'U`.`*id args*' where *args* is derived from $\langle$*argumentPart*$\rangle$?, such that — C, q

id is the name of a static member of C respectively $q.C$. The expression e is then treated as 'C.*id args*' respectively '$q.C$.*id args*'.

This means that it is possible to use a type alias to invoke a static member of a class or a mixin. For example:

```
class C<X> {
  static void staticMethod() {}
}

typedef F = C<int>;

void main() {
  F.staticMethod(); // OK.
}
```

Note that the type arguments passed to C respectively $p.C$ are erased, such that the resulting expression can be a correct static member invocation. If a future version of Dart allows type arguments to be passed via the class in an invocation of a static member, it may be preferable to preserve these type arguments. At that time, existing invocations where type arguments are passed in this manner will not break, because it is currently an error for a static member to depend on the value of a formal type parameter of the enclosing class.

Let D be a type alias declaration of the form D, F, X_j, B_j
typedef F<X_1 **extends** B_1, ..., X_s **extends** B_s> = T;
where $s > 0$. Let $Y_1, \ldots, Y_s$ be fresh type variables, assumed to satisfy the Y_j
bounds of D. We say that D *expands to a type variable* if the transitive alias ◇
expansion of F<Y_1, ..., Y_s> is Y_j for some $j \in 1..s$.

Let T be a parameterized type F<T_1, ..., T_s> (where s can be zero) in a T, F, D
scope where F denotes a type alias declaration D. We say that T *uses D as a* ◇
class when T occurs in one of the following ways:

- T occurs in an instance creation expression of the form *nc* T`(...)` or the form *nc* $T.id$`(...)`, where *nc* is either **new** or **const** (17.13). Note that, e.g., T`(42)` may be an instance creation, but it is then treated as **new** T`(42)` or **const** T`(42)`, which means that it is included here (17.15.4).
- T or $T.id$ is the redirectee in a redirecting factory constructor, (10.7.2).
- T is used as a superclass, a mixin, or a superinterface in a class declaration (10.9, 12.3, 10.10), or as an **on** type or a superinterface in a mixin declaration (12.2).
- T is used to invoke a static member of a class or mixin, as described above.

A compile-time error occurs if D expands to a type variable, and T uses D as a class. A compile-time error occurs if T uses D as a class, and T is not regular-bounded. For example:

```
class C<X extends num> {}
typedef F<Y extends int> = C<Y>;
typedef G<Z> = Z;

void main() {
  F<int>(); // OK.
  C<num>(); // OK.
  F<num>(); // Error.
  G<C<int>>(); // Error.
}
```

When we use phrases like 'let S be a class' or 'assume that S is a mixin', it is understood that this includes the case where S is a $\langle$*typeName*$\rangle$ denoting a type alias, or S is a parameterized type of the form $\langle$*typeName*$\rangle$ $\langle$*typeArguments*$\rangle$ where the type name denotes a type alias, and the transitive alias expansion of S denotes a class respectively a mixin.

20.4 Subtypes

This section defines when a type is a *subtype* of another type. The core of this section is the set of rules defined in Figure 4, but we will need to introduce a few concepts first, in order to clarify what those rules mean. ◇

A reader who has read many research papers about object-oriented type systems may find the meaning of the given notation obvious, but we still need to clarify a few details about how to handle syntactically different denotations of the same type, and how to choose the right initial environment, Γ. For a reader who is not familiar with the notation used in this section, the explanations given here should suffice to clarify what it means, with reference to the natural language explanations given at the end of the section for obtaining an intuition about the meaning.

This section is concerned with subtype relationships between types during static analysis as well as subtype relationships as queried in dynamic checks, type tests (17.39), and type casts (17.40).

A variant of the rules described here is shown in an appendix (21.2), demonstrating that Dart subtyping can be decided efficiently.

Types of the form $X\&S$ arise during static analysis due to type promotion (20.1.1). They never occur during execution, they are never a type argument of another type, nor a return type or a formal parameter type, and it is always the case that S is a subtype of the bound of X. The motivation for $X\&S$ is that it represents the type of a local variable v whose type is declared to be the type variable X, and which is known to have type S due to promotion. Similarly, $X\&S$ may be seen as an intersection type, which is a subtype of X and also a subtype of S. Intersection types are *not* supported in general, only in this special case. Every other form of type may occur during static analysis as well as during execution, and the subtype relationship is always determined in the same way. ◇

$$1\frac{}{\Gamma \vdash S <: S} \qquad 2\frac{T \in \{\texttt{Object}, \textbf{dynamic}, \textbf{void}\}}{\Gamma \vdash S <: T}$$

$$3\frac{}{\Gamma \vdash \bot <: T} \qquad 4\frac{T \neq \bot}{\Gamma \vdash \texttt{Null} <: T}$$

$$5\frac{\textbf{typedef}\ F\texttt{<}X_1\ \textbf{extends}\ \ldots,\ \ldots,\ X_s\ \textbf{extends}\ \ldots\texttt{>}\ \texttt{=}\ \texttt{U} \quad \Gamma \vdash [S_1/X_1, \ldots, S_s/X_s]U <: T}{\Gamma \vdash F\texttt{<}S_1, \ldots, S_s\texttt{>} <: T}$$

$$6\frac{\textbf{typedef}\ F\texttt{<}X_1\ \textbf{extends}\ \ldots,\ \ldots,\ X_s\ \textbf{extends}\ \ldots\texttt{>}\ \texttt{=}\ \texttt{U} \quad \Gamma \vdash S <: [T_1/X_1, \ldots, T_s/X_s]U}{\Gamma \vdash S <: F\texttt{<}T_1, \ldots, T_s\texttt{>}}$$

$$7\frac{\Gamma \vdash S <: T \quad \Gamma \vdash \texttt{Future<}S\texttt{>} <: T}{\Gamma \vdash \texttt{FutureOr<}S\texttt{>} <: T} \qquad 8\frac{}{\Gamma \vdash X\&S <: X}$$

$$9\frac{\Gamma \vdash S <: X \quad \Gamma \vdash S <: T}{\Gamma \vdash S <: X\&T} \qquad 10\frac{\Gamma \vdash S <: \texttt{Future<}T\texttt{>}}{\Gamma \vdash S <: \texttt{FutureOr<}T\texttt{>}}$$

$$11\frac{\Gamma \vdash S <: T}{\Gamma \vdash S <: \texttt{FutureOr<}T\texttt{>}} \qquad 12\frac{\Gamma \vdash S <: T}{\Gamma \vdash X\&S <: T}$$

$$13\frac{\Gamma \vdash \Gamma(X) <: T}{\Gamma \vdash X <: T} \qquad 14\frac{T \text{ is a function type}}{\Gamma \vdash T <: \textbf{Function}}$$

$$15\frac{\begin{array}{c}\Gamma' = \Gamma \uplus \{X_i \mapsto B_i \,|\, 1 \le i \le s\} \qquad \Gamma' \vdash S_0 <: T_0 \\ n_1 \le n_2 \quad n_1 + k_1 \ge n_2 + k_2 \quad \forall j \in 1..n_2 + k_2 \colon \Gamma' \vdash T_j <: S_j\end{array}}{\begin{array}{c}\Gamma \vdash S_0\ \textbf{Function}\texttt{<}X_1 \triangleleft B_1, \ldots, X_s \triangleleft B_s\texttt{>}(S_1, \ldots, S_{n_1}\texttt{,}\ \texttt{[}S_{n_1+1}, \ldots, S_{n_1+k_1}\texttt{]}) <: \\ T_0\ \textbf{Function}\texttt{<}X_1 \triangleleft B_1, \ldots, X_s \triangleleft B_s\texttt{>}(T_1, \ldots, T_{n_2}\texttt{,}\ \texttt{[}T_{n_2+1}, \ldots, T_{n_2+k_2}\texttt{]})\end{array}}$$

$$16\frac{\begin{array}{c}\Gamma' = \Gamma \uplus \{X_i \mapsto B_i \,|\, 1 \le i \le s\} \quad \Gamma' \vdash S_0 <: T_0 \quad \forall j \in 1..n \colon \Gamma' \vdash T_j <: S_j \\ \{\, y_{n+1}, \ldots, y_{n+k_2} \,\} \subseteq \{\, x_{n+1}, \ldots, x_{n+k_1} \,\} \\ \forall p \in 1..k_2, q \in 1..k_1 : \quad y_{n+p} = x_{n+q} \quad \Rightarrow \quad \Gamma' \vdash T_{n+p} <: S_{n+q}\end{array}}{\begin{array}{c}\Gamma \vdash S_0\ \textbf{Function}\texttt{<}X_1 \triangleleft B_1, \ldots, X_s \triangleleft B_s\texttt{>}(S_1, \ldots, S_n\texttt{,}\ \texttt{\{}S_{n+1}\ x_{n+1}, \ldots, S_{n+k_1}\ x_{n+k_1}\texttt{\}}) <: \\ T_0\ \textbf{Function}\texttt{<}X_1 \triangleleft B_1, \ldots, X_s \triangleleft B_s\texttt{>}(T_1, \ldots, T_n\texttt{,}\ \texttt{\{}T_{n+1}\ y_{n+1}, \ldots, T_{n+k_2}\ y_{n+k_2}\texttt{\}})\end{array}}$$

$$17\frac{\textbf{class}\ C\texttt{<}X_1\ \textbf{extends}\ \ldots,\ \ldots,\ X_s\ \textbf{extends}\ \ldots\texttt{>}\ \ldots\ \texttt{\{\}} \quad \forall j \in 1..s \colon \Gamma \vdash S_j <: T_j}{\Gamma \vdash C\texttt{<}S_1, \ldots, S_s\texttt{>} <: C\texttt{<}T_1, \ldots, T_s\texttt{>}}$$

$$18\frac{\begin{array}{c}\textbf{class}\ C\texttt{<}X_1\ \textbf{extends}\ \ldots,\ \ldots,\ X_s\ \textbf{extends}\ \ldots\texttt{>}\ \ldots\ \texttt{\{\}} \\ D\texttt{<}T_1, \ldots, T_m\texttt{>} \in \mathit{Superinterfaces}(C) \quad \Gamma \vdash [S_1/X_1, \ldots, S_s/X_s]D\texttt{<}T_1, \ldots, T_m\texttt{>} <: T\end{array}}{\Gamma \vdash C\texttt{<}S_1, \ldots, S_s\texttt{>} <: T}$$

Figure 4: Subtype rules.

20.4.1 Meta-Variables

A *meta-variable* is a symbol which stands for a syntactic construct that satisfies some static semantic requirements. ◇

For instance, X is a meta-variable standing for an identifier `W`, but only if `W` denotes a type variable declared in an enclosing scope. In the definitions below, we specify this by saying that 'X ranges over type variables'. Similarly, C is a meta-variable standing for a $\langle typeName \rangle$, for instance, `p.D`, but only if `p.D` denotes a class in the given scope. We specify this as 'C ranges over classes'.

In this section we use the following meta-variables:

- X ranges over type variables.
- C ranges over classes,
- F ranges over type aliases.
- T and S range over types, possibly with an index like T_1 or S_j.
- B ranges over types, again possibly with an index; it is only used as a type variable bound.

20.4.2 Subtype Rules

We define several rules about subtyping in this section. Whenever a rule contains one or more meta-variables, that rule can be used by *instantiating* it, that is, by consistently replacing each occurrence of a given meta-variable by concrete syntax denoting the same type. ◇

In general, this means that two or more occurrences of a given meta-variable in a rule stands for identical pieces of syntax, and the instantiation of the rule proceeds as a simple search-and-replace operation. For instance, rule 1 in Figure 4 can be used to conclude $\emptyset \vdash$ `int` $<:$ `int`, where $\emptyset$ denotes the empty environment (any environment would suffice because no type variables occur).

However, the wording 'denoting the same type' above covers additional situations as well: For instance, we may use rule 1 to show that `p1.C` is a subtype of `p2.C` when `C` is a class declared in a library L which is imported by libraries L_1 and L_2 and used in declarations there, when L_1 and L_2 are imported with prefixes `p1` respectively `p2` by the current library. The important point is that all occurrences of the same meta-variable in a given rule instantiation stands for the same type, even in the case where that type is not denoted by the same syntax in both cases.

Conversely, we can *not* use the same rule to conclude that `C` is a subtype of `C` in the case where the former denotes a class declared in library L_1 and the latter denotes a class declared in L_2, with $L_1 \neq L_2$. This situation can arise without compile-time errors, e.g., if L_1 and L_2 are imported indirectly into the current library and the two "meanings" of `C` are used as type annotations on variables or formal parameters of functions declared in intermediate libraries importing L_1 respectively L_2. The failure to prove "$\emptyset \vdash$ `C` $<:$ `C`" will then occur, e.g., in a situation where

we check whether such a variable can be passed as an actual argument to such a function, because the two occurrences of `C` do not denote the same type.

Every ⟨*typeName*⟩ used in a type mentioned in this section is assumed to have no compile-time error and denote a type.

That is, no subtyping relationship can be proven for a type that is or contains an undefined name or a name that denotes something other than a type. Note that it is not necessary in order to determine a subtyping relationship that every type satisfies the declared bounds, the subtyping relation does not depend on bounds. However, if an attempt is made to prove a subtype relationship and one or more ⟨$typeName$⟩s receives an actual type argument list whose length does not match the declaration (including the case where some type arguments are given to a non-generic class, and the case where a generic class occurs, but no type arguments are given) then the attempt to prove the relationship simply fails.

The rules in Figure 4 use the symbol Γ to denote the given knowledge about the bounds of type variables. Γ is a partial function that maps type variables to types. At a given location where the type variables in scope are X_1 **extends** B_1, ..., X_s **extends** B_s (as declared by enclosing classes and/or functions), we define the environment as follows: $\Gamma = \{ X_1 \mapsto B_1, \ \ldots \ X_s \mapsto B_s \}$. That is, $\Gamma(X_1) = B_1$, and so on, and Γ is undefined when applied to a type variable Y which is not in $\{ X_1, \ldots, X_s \}$. When the rules are used to show that a given subtype relationship exists, this is the initial value of Γ.

If a generic function type is encountered, an extension of Γ is used, as shown in the rules 15 and 16 of Figure 4. Extension of environments uses the operator $\uplus$, which is the operator that produces the union of disjoint sets, and gives priority to the right hand operand in case of conflicts.

So $\{\texttt{X} \mapsto \texttt{int}, \texttt{Y} \mapsto \texttt{double}\} \uplus \{\texttt{Z} \mapsto \texttt{Object}\} = \{\texttt{X} \mapsto \texttt{int}, \texttt{Y} \mapsto \texttt{double}, \texttt{Z} \mapsto \texttt{Object}\}$ and $\{\texttt{X} \mapsto \texttt{int}, \texttt{Y} \mapsto \texttt{FutureOr<List<double>>}\} \uplus \{\texttt{Y} \mapsto \texttt{int}\} = \{\texttt{X} \mapsto \texttt{int}, \texttt{Y} \mapsto \texttt{int}\}$. Note that operator $\uplus$ is concerned with scopes and shadowing, with no connection to, e.g., subtypes or instance method overriding.

In this specification we frequently refer to subtype relationships and assignability without mentioning the environment explicitly, as in $S <: T$. This is only done when a specific location in code is in focus, and it means that the environment is that which is obtained by mapping each type variable in scope at that location to its declared bound.

Each rule in Figure 4 has a horizontal line, to the left of which the *rule number* is indicated; under the horizontal line there is a judgment which is the *conclusion* of the rule, and above the horizontal line there are zero or more *premises* of the rule, which are typically also subtype judgments. When that is not the case for a given premise, we specify the meaning explicitly.

Instantiation of a rule, mentioned above, denotes the consistent replacement of meta-variables by actual syntactic terms denoting types everywhere in the rule, that is, in the premises as well as in the conclusion, simultaneously.

20.4.3 Being a subtype

A type S is shown to be a *subtype* of another type T in an environment Γ by

providing an instantiation of a rule R whose conclusion is $\Gamma \vdash S <: T$, along with rule instantiations showing each of the premises of R, continuing until a rule with no premises is reached. ◇

For rule 4, note that the `Null` type is a subtype of all non-$\bot$ types, even though it doesn't actually extend or implement those types. The other types are effectively treated as if they were nullable, which makes the null object (17.4) assignable to them.

The first premise in the rules 5 and 6 is a type alias declaration. This premise is satisfied in each of the following situations:

- A non-generic type alias named F is declared. In this case s is zero, no assumptions are made about the existence of any formal type parameters, and actual type argument lists are omitted everywhere in the rule.
- We may choose s and $X_1, \ldots, X_s$ such that the following holds: A generic type alias named F is declared, with formal type parameters $X_1, \ldots, X_s$. Each formal type parameter X_j may have a bound, but the bounds are never used in this context, so we do not introduce metavariables for them.

Rule 14 has as a premise that 'T is a function type'. This means that T is a type of one of the forms introduced in section 9.3. This is the same as the forms of type that occur at top level in the conclusions of rule 15 and rule 16.

In rules 17 and 18, the first premise is a class declaration. This premise is satisfied in each of the following situations:

- A non-generic class named C is declared. In this case s is zero, no assumptions are made about the existence of any formal type parameters, and actual type argument lists are omitted everywhere in the rule.
- We may choose s and $X_1, \ldots, X_s$ such that the following holds: A generic class named C is declared, with formal type parameters $X_1, \ldots, X_s$. Each formal type parameter X_j may have a bound, but the bounds are never used in this context, so we do not introduce metavariables for them.

The second premise of rule 18 specifies that a parameterized type D<...> belongs to *Superinterfaces*(C). The semantic function *Superinterfaces*(_) applied to a generic class C yields the set of direct superinterfaces of C (10.10). ◇

Note that one of the direct superinterfaces of C is the interface of the superclass of C, and that may be a mixin application (12.3), in which case D in the rule is the synthetic class which specifies the semantics of that mixin application.

The last premise of rule 18 substitutes the actual type arguments $S_1, \ldots, S_s$ for the formal type parameters $X_1, \ldots, X_s$, because $T_1, \ldots, T_m$ may contain those formal type parameters.

The rules 17 and 18 are applicable to interfaces, but they can be used with classes as well, because a non-generic class C which is used as a type denotes the interface of C, and similarly for a parameterized type C<$T_1, \ldots, T_k$> where C denotes a generic class.

20.4.4 Informal Subtype Rule Descriptions

This section gives an informal and non-normative natural language description of each rule in Figure 4.

The descriptions use the rule numbers to make the connection explicit, and also adds names to the rules that may be helpful in order to understand the role played by each rule.

In the following, many rules contain meta-variables (20.4.1) like S and T, and it is always the case that they can stand for arbitrary types. For example, rule 10 says that "The type S is a ... of `FutureOr<`T`>` ...", and this is taken to mean that for any arbitrary types S and T, showing that S is a subtype of T is sufficient to show that S is a subtype of `FutureOr<`T`>`.

Another example is the wording in rule 1: "... in any environment Γ", which indicates that the rule can be applied no matter which bindings of type variables to bounds there exist in the environment. It should be noted that the environment matters even with rules where it is simply stated as a plain Γ in the conclusion and in one or more premises, because the proof of those premises could, directly or indirectly, include the application of a rule where the environment is used.

1 **Reflexivity:** Every type is a subtype of itself, in any environment Γ. In the following rules except for a few, the rule is also valid in any environment and the environment is never used explicitly, so we will not repeat that.

2 **Top:** Every type is a subtype of `Object`, every type is a subtype of **dynamic**, and every type is a subtype of **void**. Note that this implies that these types are equivalent according to the subtype relation. We denote these types, and others with the same property (such as `FutureOr<Object>`), as top types (15.2).

3 **Bottom:** Every type is a supertype of $\bot$.

4 **Null:** Every type other than $\bot$ is a supertype of `Null`.

5 **Type Alias Left:** An application of a type alias to some actual type arguments is a subtype of another type T if the expansion of the type alias to the type that it denotes is a subtype of T. Note that a non-generic type alias is handled by letting $s = 0$.

6 **Type Alias Right:** A type S is a subtype of an application of a type alias if S is a subtype of the expansion of the type alias to the type that it denotes. Note that a non-generic type alias is handled by letting $s = 0$.

7 **Left FutureOr:** The type `FutureOr<`S`>` is a subtype of a given type T if S is a subtype of T and `Future<`S`>` is a subtype of T, for every type S and T.

8 **Left Promoted Variable:** The type $X\&S$ is a subtype of X.

9 **Right Promoted Variable A:** The type S is a subtype of $X\&T$ if S is a subtype of both X and T.

10 **Right FutureOr A:** The type S is a subtype of `FutureOr<`T`>` if S is a subtype of `Future<`T`>`.

11 **Right FutureOr B:** The type S is a subtype of `FutureOr<`T`>` if S is a subtype of T.

12 **Left Promoted Variable B:** The type $X\&S$ is a subtype of T if S is a subtype of T.

13 **Left Variable Bound:** The type variable X is a subtype of a type T if the bound of X (as specified in the current environment Γ) is a subtype of T.

14 **Right Function:** Every function type is a subtype of the type **Function**.

15 **Positional Function Type:** A function type F_1 with positional optional parameters is a subtype of another function type F_2 with positional optional parameters if the former has at most the same number of required parameters as the latter, and the latter has at least the same total number of parameters as the former; the return type of F_1 is a subtype of that of F_2; and each parameter type of F_1 is a *supertype* of the corresponding parameter type of F_2, if any. Note that the relationship to function types with no optional parameters, and the relationship between function types with no optional parameters, is covered by letting $k_2 = 0$ respectively $k_1 = k_2 = 0$. For every subtype relation considered in this rule, the formal type parameters of F_1 and F_2 must be taken into account (as reflected in the use of the extended environment Γ'). We can assume without loss of generality that the names of type variables are pairwise identical, because we consider types of generic functions to be equivalent under consistent renaming (9.3). In short, "during the proof, we will rename them as needed". Finally, note that the relationship between non-generic function types is covered by letting $s = 0$.

16 **Named Function Type:** A function type F_1 with named optional parameters is a subtype of another function type F_2 with named optional parameters if they have the same number of required parameters, and the set of names of named parameters for the latter is a subset of that for the former; the return type of F_1 is a subtype of that of F_2; and each parameter type of F_1 is a *supertype* of the corresponding parameter type of F_2, if any. Note that the relationship to function types with no optional parameters, and the relationship between function types with no optional parameters, is covered by letting $k_2 = 0$ respectively $k_1 = k_2 = 0$, and also that the latter case is identical to the rule obtained from rule 15 concerning subtyping among function types with no optional parameters. As in rule 15, we can assume without loss of generality that the names of type variables are pairwise identical. Similarly, non-generic functions are covered by letting $s = 0$.

17 **Class Covariance:** A parameterized type based on a generic class C is a subtype of a parameterized type based on the same class C if each actual type argument of the former is a subtype of the corresponding actual type

argument of the latter. This rule may have $s = 0$ and cover a non-generic class as well, but that is redundant because this is already covered by rule 1.

18 **Superinterface:** Considering the case where $s = 0$ and $m = 0$ first, a parameterized type based on a non-generic class C is a subtype of a parameterized type based on a different non-generic class D if D is a direct superinterface of C. When $s > 0$ or $m > 0$, this rule describes a subtype relationship which includes one or more generic classes, in which case we need to give names to the formal type parameters of C, and specify how they are used in the specification of the superinterface based on D. With those pieces in place, we can specify the subtype relationship that exists between two parameterized types based on C and D. The case where the superclass is a mixin application is covered via the equivalence with a declaration of a regular (possibly generic) superclass (12.3), and this means that there may be multiple subtype steps from a given class declaration to the class specified in an **extends** clause.

20.4.5 Additional Subtyping Concepts

S is a *supertype* of T in a given environment Γ, written $\Gamma \vdash S :> T$, iff $\Gamma \vdash T <: S$. ◇

A type T *may be assigned* to a type S in an environment Γ, written $\Gamma \vdash S \Longleftrightarrow T$, iff either $\Gamma \vdash S <: T$ or $\Gamma \vdash T <: S$. In this case we say that the types S and T are *assignable*. ◇

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

For example, assigning an object of static type `Object` to a variable with static type `String`, while not guaranteed to be correct, might be fine if the run-time value happens to be a string.

A static analyzer or compiler may support more strict static checks as an option. ◇

20.5 Function Types

Function types come in two variants: ◇

1. The types of functions that only have positional parameters. These have the general form

   T **Function**<$X_1 \lhd B_1, \ldots, X_s \lhd B_s$>(T1, …, T_n, [T_{n+1}, …, T_{n+k}]).

2. The types of functions with named parameters. These have the general form

   T **Function**<$X_1 \lhd B_1, \ldots, X_s \lhd B_s$>(T1, …, T_n, {T_{n+1} x_{n+1}, …, T_{n+k} x_{n+k}}).

Note that the non-generic case is covered by having $s = 0$, in which case the type parameter declarations are omitted (15). The case with no optional parameters is covered by having $k = 0$; note that all rules involving function types of the two kinds coincide in this case.

Two function types are considered equal if consistent renaming of type parameters can make them identical.

A common way to say this is that we do not distinguish function types which are alpha-equivalent. For the subtyping rule below this means we can assume that a suitable renaming has already taken place. In cases where this is not possible because the number of type parameters in the two types differ or the bounds are different, no subtype relationship exists.

A function object is always an instance of some class C that implements the class **Function** (20.6), and which has a method named `call`, whose signature is the function type C itself. Consequently, all function types are subtypes of **Function** (20.4).

20.6 Type Function

The built-in class **Function** is a supertype of all function types (20.5). It is impossible to extend, implement, or mix in the class **Function**.

If a class declaration or mixin application has **Function** as superclass, it instead uses `Object` as superclass.

If a class or mixin declaration implements **Function**, it has no effect. It is as if the **Function** was removed from the `implements` clause (and if it's the only implemented interface, the entire clause is removed). The resulting class or mixin interface does not have **Function** as a superinterface.

If a mixin application mixes **Function** onto a superclass, it follows the normal rules for mixin-application, but since the result of that mixin application is equivalent to a class with `implements Function`, and that clause has no effect, the resulting class also does not implement **Function**. The **Function** class declares no concrete instance members, so the mixin application creates a sub-class of the superclass with no new members and no new interfaces.

*Since using **Function** in these ways has no effect, it would be reasonable to disallow it completely, like we do extending, implementing or mixing in types like `int` or `String`. For backwards compatibility with Dart 1 programs, the syntax is allowed to remain, even if it has no effect. Tools may choose to warn users that their code has no effect.*

20.7 Type dynamic

The type **dynamic** is a static type which is a supertype of all other types, just like `Object`, but it differs from other types in that the static analysis assumes that every member access has a corresponding member with a signature that admits the given access.

For instance, when the receiver in an ordinary method invocation has type **dynamic**, any method name can be invoked, with any number of type arguments or

none, with any number of positional arguments, and any set of named arguments, of any type, without error. Note that the invocation will still cause a compile-time error if there is an error in one or more arguments or other subterms.

If no static type annotation has been provided, the type system considers declarations to have type **dynamic**. If a generic type is used but type arguments are not provided, the type arguments default to type **dynamic**.

This means that given a generic declaration G<$P_1, \ldots, P_n$>..., where P_i is a formal type parameter declaration, $i \in 1..n$, the type G is equivalent to G<**dynamic**, ..., **dynamic**>.

The built-in type declaration `dynamic`, which is declared in the library `dart:core`, denotes the **dynamic** type. When the name **dynamic** exported by `dart:core` is evaluated as an expression, it evaluates to a `Type` object representing the **dynamic** type, even though **dynamic** is not a class.

This `Type` object must compare equal to the corresponding `Type` objects for `Object` and **void** according to operator '==' (20.2).

To improve the precision of static types, member accesses on a receiver of type **dynamic** that refer to declarations of the built-in class `Object` are given the static type corresponding to those declarations whenever doing so is sound.

- Let e be an expression of the form d`.`id, which is not followed by an argument part, where the static type of d is **dynamic**, and id is the name of a getter declared in `Object`; if the return type of `Object.`id is T then the static type of e is T. For instance, `d.hashCode` has type `int` and `d.runtimeType` has type `Type`.

- Let e be an expression of the form d`.`id, which is not followed by an argument part, where the static type of d is **dynamic**, and id is the name of a method declared in `Object` whose method signature has type F (which is a function type). The static type of e is then F. For instance, d`.toString` has type `String` **Function**`()`.

- Let e be an expression which is of the form d`.`id`(`*arguments*`)` or the form d`.`id`<`*typeArguments*`>(`*arguments*`)`, where the static type of d is **dynamic**, id is the name of a getter declared in `Object` with return type F, *arguments* are derived from ⟨*arguments*⟩, and *typeArguments* are derived from ⟨*typeArguments*⟩, if present. Static analysis will then process e as a function expression invocation where an object of static type F is applied to the given argument part. So this is always a compile-time error. For instance, d`.runtimeType(42)` is a compile-time error, because it is checked as a function expression invocation where an entity of static type `Type` is invoked. Note that it could actually succeed: An overriding implementation of `runtimeType` could return an instance whose dynamic type is a subtype of `Type` that has a `call` method. We decided to make it an error because it is likely to be a mistake, especially in cases like d`.hashCode()` where a developer might have forgotten that `hashCode` is a getter.

- Let e be an expression of the form d`.`id`(`*arguments*`)` where the static type of d is **dynamic**, *arguments* is an actual argument list derived from

⟨*arguments*⟩, and *id* is the name of a method declared in `Object` whose method signature has type F. If the number of positional actual arguments in *arguments* is less than the number of required positional arguments of F or greater than the number of positional arguments in F, or if *arguments* includes any named arguments with a name that is not declared in F, the type of e is **dynamic**. Otherwise, the type of e is the return type in F. So d`.toString(bazzle: 42)` has type **dynamic** whereas d`.toString()` has type `String`. Note that invocations which "do not fit" the statically known declaration are not errors, they just get return type **dynamic**.

- Let e be an expression of the form d`.`*id*`<`*typeArguments*`>(`*arguments*`)` where the static type of d is **dynamic**, *typeArguments* is a list of actual type arguments derived from ⟨*typeArguments*⟩, and *arguments* is an actual argument list derived from ⟨*arguments*⟩. It is a compile-time error if *id* is the name of a non-generic method declared in `Object`. No generic methods are declared in `Object`. Hence, we do not specify that there must be the statically required number of actual type arguments, and they must satisfy the bounds. That would otherwise be the consistent approach, because the invocation is guaranteed to fail when any of those requirements are violated, but generalizations of this mechanism would need to include such rules.

- For an instance method invocation e (including invocations of getters, setters, and operators) where the receiver has static type **dynamic** and e does not match any of the above cases, the static type of e is **dynamic**. When an expression derived from ⟨*cascadeSection*⟩ performs a getter or method invocation that corresponds to one of the cases above, the corresponding static analysis and compile-time errors apply. For instance, d`..foobar(16)..hashCode()` is an error.

Note that only very few forms of instance method invocation with a receiver of type **dynamic** can be a compile-time error. Of course, some expressions like `x[1, 2]` are syntax errors even though they could also be considered "invocations", and subexpressions are checked separately so any given actual argument could be a compile-time error. But almost any given argument list shape could be handled via `noSuchMethod`, and an argument of any type could be accepted because any formal parameter in an overriding declaration could have its type annotation contravariantly changed to `Object`. So it is a natural consequence of the principle of that a **dynamic** receiver admits almost all instance method invocations. The few cases where an instance method invocation with a receiver of type **dynamic** is an error are either guaranteed to fail at run time, or they are very, very likely to be developer mistakes.

20.8 Type FutureOr

The built-in type declaration `FutureOr`, which is exported by the library `dart:async`, defines a generic type with one type parameter (15). The type `FutureOr<`T`>` is a non-class type which is regular-bounded for all T.

The subtype relations involving `FutureOr` are specified elsewhere (20.4.2). Note, however, that they entail certain useful properties:

- $T <:$ `FutureOr<`T`>`.
- `Future<`T`>` $<:$ `FutureOr<`T`>`.
- If $T <: S$ and `Future<`T`>` $<: S$, then `FutureOr<`T`>` $<: S$.

That is, `FutureOr` is in a sense the union of T and the corresponding future type. The last point guarantees that `FutureOr<`T`>` $<:$ `Object`, and also that `FutureOr` is covariant in its type parameter, just like class types: if $S <: T$ then `FutureOr<`S`>` $<:$ `FutureOr<`T`>`.

If the type arguments passed to `FutureOr` would incur compile-time errors if applied to a normal generic class with one type parameter, the same compile-time errors are issued for `FutureOr`. The name `FutureOr` as an expression denotes a `Type` object representing the type `FutureOr<dynamic>`.

The `FutureOr<`T`>` type represents a case where an object can be either an instance of the type T or the type `Future<`T`>`. Such cases occur naturally in asynchronous code. The available alternative would be to use a top type (e.g., ***dynamic****), but `FutureOr` allows some tools to provide a more precise type analysis.*

The type `FutureOr<`T`>` has an interface that is identical to that of `Object`. That is, only members that `Object` has can be invoked on an object with static type `FutureOr<`T`>`.

We only want to allow invocations of members that are inherited from a common supertype of both T and `Future<`T`>`. In most cases the only common supertype is `Object`. The exceptions, like `FutureOr<Future<Object>>` which has `Future<Object>` as common supertype, are few and not practically useful, so for now we choose to only allow invocations of members inherited from `Object`.

We define the auxiliary function $futureOrBase(T)$ as follows: ◇

- If T is `FutureOr<`S`>` for some S then $futureOrBase(T) = futureOrBase(S)$.
- Otherwise $futureOrBase(T) = T$.

20.9 Type Void

The special type **void** is used to indicate that the value of an expression is meaningless and intended to be discarded.

A typical case is that the type **void** is used as the return type of a function that "does not return anything". Technically, there will always be *some* object which is returned (9). But it is perfectly meaningful to have a function whose sole purpose is to create side-effects, such that *any* use of the returned object would be misguided. This does not mean that there is anything wrong with the returned object as such. It could be any object whatsoever. But the developer who chose the return type **void** did that to indicate that it is a misunderstanding to ascribe any meaning to that object, or to use it for any purpose.

The type **void** is a top type (15.2), so **void** and `Object` are subtypes of each other (20.4), which also implies that any object can be the value of an expression of type **void**. Consequently, any instance of type `Type` which reifies the type **void** must compare equal (according to the '==' operator 17.27) to any instance of `Type` which reifies the type `Object` (20.2). It is not guaranteed that `identical(`**void**`, Object)` evaluates to true. In fact, it is not recommended that implementations strive to achieve this, because it may be more important to ensure that diagnostic messages (including stack traces and dynamic error messages) preserve enough information to use the word 'void' when referring to types which are specified as such in source code.

In support of the notion that the value of an expression with static type **void** should be discarded, such objects can only be used in specific situations: The occurrence of an expression of type **void** is a compile-time error unless it is permitted according to one of the following rules.

- In an ⟨*expressionStatement*⟩ e`;`, e may have type **void**. *The value of e is discarded.*
- In the initialization and increment expressions of a for-loop, **for** `(`e_1`;` e_2`;` e_3`)` ..., e_1 may have type **void**, and each of the expressions in the expression list e_3 may have type **void**. *The values of e_1 and e_3 are discarded.*
- In a type cast e `as` T, e may have type **void**. *Developers thus obtain the ability to* override *the constraints on usages of values with static type **void**. This means that it is not enforced that such values are discarded, but they can only be used when the wish to do so has been indicated explicitly.*
- In a parenthesized expression `(`e`)`, e may have type **void**. *Note that (e) itself has type **void**, which implies that it must occur in some context where it is not an error to have it.*
- In a conditional expression e `?` e_1 `:` e_2, e_1 and e_2 may have type **void**. *The static type of the conditional expression is then **void**, even if one of the branches has a different type, which means that the conditional expression must again occur in some context where it is not an error to have it.*
- In a null coalescing expression e_1 `??` e_2, e_2 may have type **void**. *The static type of the null coalescing expression is then **void**, which in turn restricts where it can occur.*
- In an expression of the form **await** e, e may have type **void**. *This rule was adopted because it was a substantial breaking change to turn this situation into an error at the time where the treatment of **void** was changed. Tools may choose to give a hint in such cases.*
- In a return statement **return** e`;`, e may have type **void** in a number of situations (18.12).

- In an arrow function body `=>` e, the returned expression e may have type **void** in a number of situations (9).

- An initializing expression for a variable of type **void** may have type **void**. *Usages of that variable are constrained.*

- An actual parameter expression corresponding to a formal parameter whose statically known type annotation is **void** may have type **void**. *Usages of that parameter in the body of the callee are statically expected to be constrained by having type **void**. See the discussion about soundness below (20.9.1).*

- In an expression of the form e_1 `=` e_2 where e_1 is an ⟨*assignableExpression*⟩ denoting a variable or formal parameter of type **void**, e_2 may have type **void**. *Usages of that variable or formal parameter are statically expected to be constrained by having type **void**. See the discussion about soundness below (20.9.1).*

- Let e be an expression ending in a ⟨*cascadeSection*⟩ of the form `..` S s `=` e_1, where S is of the form

  (⟨*cascadeSelector*⟩ ⟨*argumentPart*⟩*) (⟨*assignableSelector*⟩ ⟨*argumentPart*⟩*)*

  and e_1 is of the form ⟨*expressionWithoutCascade*⟩.

  If s is an ⟨*assignableSelector*⟩ of the form `.`*id* or `?.`*id* where the static type of the identifier *id* is **void**, e_1 may have type **void**. Otherwise, if s is an ⟨*assignableSelector*⟩ of the form `[`e_0`]` where the static type of the first formal parameter in the statically known declaration of operator `[]=` is **void**, e_0 may have type **void**. Also, if the static type of the second formal parameter is **void**, e_1 may have type **void**.

Finally, we need to address situations involving implicit usage of an object whose static type can be **void**. It is a compile-time error for a for-in statement to have an iterator expression of type T such that `Iterator<void>` is the most specific instantiation of `Iterator` that is a superinterface of T, unless the iteration variable has type **void**. It is a compile-time error for an asynchronous for-in statement to have a stream expression of type T such that `Stream<void>` is the most specific instantiation of `Stream` that is a superinterface of T, unless the iteration variable has type **void**.

Here are some examples:

```
for (Object x in <void>[]) {} // Error.
await for (int x in new Stream<void>.empty()) {} // Error.
for (void x in <void>[]) {...} // OK.
for (var x in <void>[]) {...} // OK, type of x inferred.
```

However, in the examples that are not errors the usage of `x` in the loop body is constrained, because it has type **void**.

20.9.1 Void Soundness

The constraints on usage of an object obtained from the evaluation of an expression with static type **void** are not strictly enforced.

The usage of a "void value" is not a soundness issue, that is, no invariants needed for correct execution of a Dart program can be violated because of such a usage.

*It could be said that the type **void** is used to help developers maintain a certain self-imposed discipline about the fact that certain objects are not* intended *to be used. Because of the fact that enforcement is not necessary, and because of the treatment of **void** in earlier versions of Dart, the language uses a* best effort *approach to ensure that the value of an expression of type **void** will not be used.*

In fact, there are numerous ways in addition to the type cast in which a developer can get access to such an object:

```
abstract class A<X> {
  final X x;
  A(this.x);
  Object foo(X x);
}

class B<X> extends A<X> {
  B(X x): super(x);
  Object foo(Object x) => x;
}

Object f<X>(X x) => x;

void main() {
  void x = 42;
  print(f(x)); // (1)

  A<void> a = B<void>(x);
  A<Object> aObject = a;
  print(aObject.x); // (2)
  print(a.foo(x)); // (3)
}
```

At (1), a variable `x` of type **void** is passed to a generic function `f`, which is allowed because the actual type argument **void** is inferred, and it is allowed to pass an actual argument of type **void** to a formal parameter with the same type. However, no special treatment is given when an expression has a type which is or contains a type variable whose value could be **void**, so we are allowed to return `x` in the body of `f`, even though this means that we indirectly get access to the value of an expression of type **void**, under the static type `Object`.

At (2), we indirectly obtain access to the value of the variable `x` with type **void**, because we use an assignment to get access to the instance of `B` which was created with type argument **void** under the type `A<Object>`. Note that `A<Object>` and `A<`**void**`>` are subtypes of each other, that is, they are equivalent according to the subtype rules, so neither static nor dynamic type checks will fail.

At (3), we indirectly obtain access to the value of the variable `x` with type **void** under the static type `Object`, because the statically known method signature of `foo` has parameter type **void**, but the actual implementation of `foo` which is invoked is an override whose parameter type is `Object`, which is allowed because `Object` and **void** are both top types.

Obviously, the support for preventing developers from using objects obtained from expressions of type ***void*** *is far from sound, in the sense that there are many ways to circumvent the rule that such an object should be discarded.*

However, we have chosen to focus on the simple, first-order usage (where an expression has type ***void****, and the value is used), and we have left higher-order cases largely unchecked, relying on additional tools such as linters to perform an analysis which covers indirect data flows.*

It would certainly have been possible to define sound rules, such that the value of an expression of type ***void*** *would be guaranteed to be discarded after some number of transfers from one variable or parameter to the next one, all with type* ***void****, explicitly, or as the value of a type parameter. In particular, we could require that method overrides should never override return type* `Object` *by return type* ***void****, or parameter types in the opposite direction; parameterized types with type argument* ***void*** *could not be assigned to variables where the corresponding type argument is anything other than* ***void****, etc. etc.*

But this would be quite impractical. In particular, the need to either prevent a large number of type variables from ever having the value ***void****, or preventing certain usages of values whose type is such a type variable, or whose type contains such a type variable, that would be severely constraining on a very large part of all Dart code.*

So we have chosen to help developers maintain this self-imposed discipline in simple and direct cases, and leave it to ad-hoc reasoning or separate tools to ensure that the indirect cases are covered as closely as needed in practice.

20.10 Parameterized Types

A *parameterized type* is a syntactic construct where the name of a generic type declaration is applied to a list of actual type arguments. A *generic instantiation* is the operation where a generic type is applied to actual type arguments.

So a parameterized type is the syntactic concept that corresponds to the semantic concept of a generic instantiation. When using the former, we will often leave the latter implicit.

Let T be a parameterized type G<$S_1, \ldots, S_n$>.

It is a compile-time error if G is not a generic type, or G is a generic type, but the number of formal type parameters in the declaration of G is not n. Oth-

erwise, let $X_1, \ldots, X_n$ be the formal type parameters of G, and let $B_1, \ldots, B_n$ be the corresponding upper bounds, using **dynamic** when no bound is declared.

T is *malbounded* iff either S_i is malbounded for one or more $i \in 1..n$, or T is not well-bounded (15.2). ◇

It is a compile-time error if T is malbounded.

T is evaluated as follows. Let t_i be the result of evaluating S_i, for $i \in 1..n$. T then evaluates to the generic instantiation where G is applied to $t_1, \ldots, t_n$.

Let T be a parameterized type of the form G<$A_1, \ldots, A_n$> and assume that T is not malformed and not malbounded. If S is the static type of a member m of G, then the static type of the member m of an expression of type G<$A_1, \ldots, A_n$> is $[A_1/X_1, \ldots, A_n/X_n]S$, where $X_1, \ldots, X_n$ are the formal type parameters of G.

20.10.1 Actual Types

Let T be a term derived from $\langle type \rangle$. Let $X_1, \ldots, X_s$ be the formal type parameters in scope at the location where T occurs. In a context where the actual type arguments corresponding to $X_1, \ldots, X_s$ are $t_1, \ldots, t_s$, the *actual value of the type* T is then $[t_1/X_1, \ldots, t_s/X_s]T$. ◇

Let D be a declaration with name n and type annotation T. The *actual type* of D and of n in a given context is then the actual value of T in that context. ◇

In the non-generic case where $s = 0$, the actual type is equal to the declared type, in the sense that we use simple terms like `int` to denote both. Note that $X_1, \ldots, X_s$ may be declared by multiple entities, e.g., one or more enclosing generic functions and an enclosing generic class.

Let X **extends** B be a formal type parameter declaration. The *actual bound* of X in a given context is the actual value of B in that context. ◇

Note that even though X may occur in B it does not occur in the actual value of B, because no type has an actual value that includes a type variable.

20.10.2 Least Upper Bounds

Given two interfaces I and J, let S_I be the set of superinterfaces of I, let S_J be the set of superinterfaces of J and let $S = (\{I\} \cup S_I) \cap (\{J\} \cup S_J)$. Furthermore, we define $S_n = \{T | T \in S \wedge depth(T) = n\}$ for any finite n where $depth(T)$ is the number of steps in the longest inheritance path from T to `Object`. Let q be the largest number such that S_q has cardinality one, which must exist because S_0 is {`Object`}. The least upper bound of I and J is the sole element of S_q.

The least upper bound of **dynamic** and any type T is **dynamic**. The least upper bound of **void** and any type $T \neq$ **dynamic** is **void**. The least upper bound of $\bot$ and any type T is T. Let U be a type variable with upper bound B. The least upper bound of U and a type $T \neq \bot$ is the least upper bound of B and T.

The least upper bound operation is commutative and idempotent, but it is not associative.

The least upper bound of a function type and an interface type T is the least

upper bound of **Function** and T. Let F and G be function types. If F and G differ in their number of required parameters, then the least upper bound of F and G is **Function**. Otherwise:

- If

  $F = \texttt{<}X_1\ B_1, \ldots,\ X_s\ B_s\texttt{>(}T_1, \ldots,\ T_r,\ \texttt{[}T_{r+1}, \ldots,\ T_n\texttt{])} \rightarrow T_0$ and

  $G = \texttt{<}X_1\ B_1, \ldots,\ X_s\ B_s\texttt{>(}S_1, \ldots,\ S_r,\ \texttt{[}S_{r+1}, \ldots,\ S_k\texttt{])} \rightarrow S_0$

  where $k \leq n$ then the least upper bound of F and G is

  $\texttt{<}X_1\ B_1, \ldots,\ X_s\ B_s\texttt{>(}L_1, \ldots,\ L_r,\ \texttt{[}L_{r+1}, \ldots,\ L_k\texttt{])} \rightarrow L_0$

  where L_i is the least upper bound of T_i and $S_i, i \in 0..k$.

- If

  $F = \texttt{<}X_1\ B_1, \ldots,\ X_s\ B_s\texttt{>(}T_1, \ldots,\ T_r,\ \texttt{[}T_{r+1}, \ldots,\ T_n\texttt{])} \rightarrow T_0$,

  $G = \texttt{<}X_1\ B_1, \ldots,\ X_s\ B_s\texttt{>(}S_1, \ldots,\ S_r,\ \texttt{\{ ... \})} \rightarrow S_0$

  then the least upper bound of F and G is

  $\texttt{<}X_1\ B_1, \ldots,\ X_s\ B_s\texttt{>(}L_1, \ldots,\ L_r\texttt{)} \rightarrow L_0$

  where L_i is the least upper bound of T_i and $S_i, i \in 0..r$.

- If

  $F = \texttt{<}X_1\ B_1, \ldots,\ X_s\ B_s\texttt{>(}T_1, \ldots,\ T_r,\ \texttt{\{}T_{r+1}\ p_{r+1}, \ldots,\ T_f\ p_f\texttt{\})} \rightarrow T_0$,

  $G = \texttt{<}X_1\ B_1, \ldots,\ X_s\ B_s\texttt{>(}S_1, \ldots,\ S_r,\ \texttt{\{}S_{r+1}\ q_{r+1}, \ldots,\ S_g\ q_g\texttt{\})} \rightarrow S_0$

  then let $\{x_m, \ldots, x_n\} = \{p_{r+1}, \ldots, p_f\} \cap \{q_{r+1}, \ldots, q_g\}$ and let X_j be the least upper bound of the types of x_j in F and $G, j \in m..n$. Then the least upper bound of F and G is

  $\texttt{<}X_1\ B_1, \ldots,\ X_s\ B_s\texttt{>(}L_1, \ldots,\ L_r,\ \texttt{\{}X_m\ x_m, \ldots,\ X_n\ x_n\texttt{\})} \rightarrow L_0$

  where L_i is the least upper bound of T_i and $S_i, i \in 0..r$

Note that the non-generic case is covered by using $s = 0$, in which case the type parameter declarations are omitted (15).

21 Reference

21.1 Lexical Rules

Dart source text is represented as a sequence of Unicode code points. This sequence is first converted into a sequence of tokens according to the lexical rules given in this specification. At any point in the tokenization process, the longest possible token is recognized.

21.1.1 Reserved Words

A *reserved word* can only be used in the syntactic positions specified by the grammar. In particular, a compile-time error occurs if a reserved word is used where an identifier is expected. ◇

Note that reserved words occur bold and unquoted in grammar rules (e.g., **assert**) even though the consistent notation would use quotes (e.g., 'assert'). This notational abuse occurs because we believe it makes the grammar rules more readable.

⟨*RESERVED_WORD*⟩ ::= **assert** | **break** | **case** | **catch** | **class** | **const**
| **continue** | **default** | **do** | **else** | **enum** | **extends** | **false** | **final** | **finally** | **for**
| **if** | **in** | **is** | **new** | **null** | **rethrow** | **return** | **super** | **switch** | **this** | **throw**
| **true** | **try** | **var** | **void** | **while** | **with**

In the grammar, the rule for reserved words above must occur before the rule for ⟨*BUILT_IN_IDENTIFIER*⟩ (17.38).

This ensures that ⟨*IDENTIFIER*⟩ and ⟨*IDENTIFIER_NO_DOLLAR*⟩ do not derive any reserved words, and they do not derive any built-in identifiers.

21.1.2 Comments

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

⟨*SINGLE_LINE_COMMENT*⟩ ::=
'//' ~(⟨*LINE_BREAK*⟩)* (⟨*LINE_BREAK*⟩)?

⟨*MULTI_LINE_COMMENT*⟩ ::=
'/*' (⟨*MULTI_LINE_COMMENT*⟩ | ~ '*/')* '*/'

Dart supports both single-line and multi-line comments. A *single line comment* begins with the token //. Everything between // and the end of line must be ignored by the Dart compiler unless the comment is a documentation comment. ◇

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 comments that begin with the tokens /// or /**. Documentation comments are intended to be processed by a tool that produces human readable documentation. ◇

The current scope for a documentation comment immediately preceding the declaration of a class C is the *body scope* of C.

The current scope for a documentation comment immediately preceding the declaration of a function f is the formal parameter scope of f.

21.2 Operator Precedence

Operator precedence is given implicitly by the grammar.
The following non-normative table may be helpful

Description	Operator	Associativity	Precedence
Unary postfix	e., e?., e++, e--, $e1[e2]$, $e()$	None	16
Unary prefix	-e, !e, ~e, ++e, --e, **await** e	None	15
Multiplicative	*, /, ~/, %	Left	14
Additive	+, -	Left	13
Shift	<<, >>, >>>	Left	12
Bitwise AND	&	Left	11
Bitwise XOR	^	Left	10
Bitwise Or	|	Left	9
Relational	<, >, <=, >=, **as**, **is**, **is**!	None	8
Equality	==, !=	None	7
Logical AND	&&	Left	6
Logical Or	||	Left	5
If-null	??	Left	4
Conditional	$e1$? $e2$: $e3$	Right	3
Cascade	..	Left	2
Assignment	=, *=, /=, +=, -=, &=, ^=, etc.	Right	1

Appendix: Algorithmic Subtyping

$$1_{\text{algo}}\frac{S \text{ not composite}}{\Gamma \vdash S <: S} \qquad 8.1\frac{}{\Gamma \vdash X <: X}$$

$$8.2\frac{\Gamma \vdash X <: T}{\Gamma \vdash X <: X\&T} \qquad 8.3\frac{\Gamma \vdash X\&S <: T}{\Gamma \vdash X\&S <: X\&T}$$

$$11.1\frac{\Gamma \vdash \Gamma(X) <: \texttt{FutureOr<}T\texttt{>}}{\Gamma \vdash X <: \texttt{FutureOr<}T\texttt{>}} \qquad 11.2\frac{\Gamma \vdash S <: \texttt{FutureOr<}T\texttt{>}}{\Gamma \vdash X\&S <: \texttt{FutureOr<}T\texttt{>}}$$

Figure 5: Algorithmic subtype rules. Rules 2–18 are unchanged and hence omitted here.

The text in this appendix is not part of the specification of the Dart language. However, we still use the notation where precise information uses the style associated with normative text in the specification (this style), whereas examples and explanations use commentary style (like this).

This appendix presents a variant of the subtype rules given in Figure 4 on page 229.

The rules will prove the same set of subtype relationships, but the rules given here show that there is an efficient implementation that will determine whether $\Gamma \vdash S <: T$ holds, for any given types S and T. It is easy to see that the

algorithmic rules will prove at most the same subtype relationships, because all rules given here can be proven by means of rules in Figure 4. It is also relatively straightforward to sketch out proofs that the algorithmic rules can prove at least the same subtype relationships, also when the following ordering and termination constraints are observed.

The only rule which is modified is number 1, which is modified to 1_{algo}. This only changes the applicability of the rule: This rule is only used for types which are not atomic. An *atomic type* is a type which is not a type variable, not a promoted type variable, not a function type, and not a parameterized type. ◇

In other words, rule 1_{algo} is used for special types like **dynamic**, **void**, and **Function**, and it is used for non-generic classes, but it is not used for any type where it is an operation that takes more than one comparison to detect whether it is the same as some other type. The point is that the remaining rules will force a structural traversal anyway, as far as needed, and we may hence just as well omit the initial structural traversal which might take many steps only to report that two large type terms are not quite identical.

The rules are ordered by means of their rule numbers: A rule given here numbered $N.1$ is inserted immediately after rule N, followed by rule $N.2$, and so on, followed by the rule whose number is $N + 1$. So the order is 1_{algo}, 2–8, 8.1, 8.2, 8.3, 9, and so on.

We now specify the procedure which is used to determine whether $\Gamma \vdash S <: T$ holds, for some specific types S and T: Select the first rule R whose syntactic constraints are satisfied by the given types S and T, and proceed to show that its premises hold. If so, we terminate and conclude that the subtype relationship holds. Otherwise we terminate and conclude that the subtype relationship does not hold, except if R is 10, 11, 11.1, or 11.2. In particular, for the original query $\Gamma \vdash S <: T$, we do not backtrack into trying to use a rule that has a higher rule number than that of R, except that we may try all of the rules with `FutureOr<`T`>` to the right.

Apart from the fact that the full complexity of subtyping is potentially incurred each time it is checked whether a premise holds, the checks applied for each rule is associated with an amount of work which is constant for all rules except the following: First, the group of rules 10, 11, 11.1, and 11.2 may cause backtracking to take place. Next, rules 15–17 require work proportional to the size of S and T, due to the number of premises that must be checked. Finally, rule 18 requires work proportional to the size of S, and it may also incur the cost of searching up to the entire set of direct and indirect superinterfaces of the candidate subtype S, until the corresponding premise for one of them is shown to hold, if any.

Additional optimizations are applicable. For instance, we can immediately conclude that the subtype relationship does not hold when we are about to check rule 18 if T is a type variable or a function type. For several other forms of type, e.g., a promoted type variable, `Object`, **dynamic**, **void**, `FutureOr<`T`>` for any T, or **Function**, it is known that it will never occur as T for rule 18, which means that this seemingly expensive step can be confined to some extent.

Appendix: Integer Implementations

The `int` type represents integers. The specification is written with 64-bit two's complement integers as the intended implementation. But when Dart is compiled to JavaScript, the implementation of `int` will instead use the JavaScript number type and the corresponding JavaScript operations, except for bit operations as explained below.

This introduces a number of differences:

- Valid values of JavaScript `int` are any IEEE-754 64-bit floating point number with no fractional part. This includes positive and negative *infinity*, which can be reached by overflowing (integer division by zero is still a dynamic error). Otherwise valid integer literals (including any leading minus sign) that represent invalid JavaScript `int` values are compile-time errors when compiling to JavaScript. Operations on integers may lose precision, because the operands and the result are represented as 64-bit floating point numbers that are limited to 53 significant bits. ◇

- JavaScript `int` instances also implement `double`, and integer-valued `double` instances also implement `int`. The `int` and `double` class are still separate subclasses of the class `num`, but *instances* of either class that represent an integer act as if they are actually instances of a common subclass implementing both `int` and `double`. Fractional numbers only implement `double`.

- Bitwise operations on integers ('`&`', '`|`', '`^`', '`~`', '`<<`', and '`>>`') all truncate the operands to 32-bit two's complement integers, perform 32-bit operations on those, and the resulting 32 bits are interpreted as a 32-bit unsigned integer. For example, `-1 << 1` evaluates to 4294967294 (also known as `(-2).toUnsigned(32)`). The right shift operator, '`>>`', performs a signed right shift on the 32 bits when the original number is negative, and an unsigned right shift when the original number is non-negative. Both kinds of shift writes bit $k+1$ into position k, $0 \leq k < 31$; but the signed shift leaves bit 31 unchanged, and the unsigned shift writes a 0 as bit 31. For example: `0x80000002 >> 1 == 0x40000001`, but `-0x7FFFFFFE >> 1 == 0xC0000001`. In this example we note that both `0x80000002` and `-0x7FFFFFFE` yield the 32-bit two's complement representation 0x80000002, but they have different values for the IEEE 754 sign bit.

- The `identical` method cannot distinguish the values 0.0 and -0.0, and it cannot recognize any *NaN* value as identical to itself. For efficiency, the `identical` operation uses the JavaScript `===` operator. ◇

Index