Dart DEP #30: Non-null Types and Non-null By Default (NNBD)

Patrice Chalin, chalin@dsrg.org

2015-07-01 (0.6.4) - revision history

Contents

DEP #30: Non-null Types and Non-null By Default (NNBD)

Contact information

- Patrice Chalin, @chalin, chalin@dsrg.org
- **DEP #30 home**: github.com/chalin/DEP-non-null.
- **Additional stakeholders**:
  - Leaf Petersen, @leafpetersen, Dart Dev Compiler team.

1 Introduction

In this DEP we propose 4 *core* updates and additions to the language allowing us to naturally recover a **non-null-by-default** (NNBD) interpretation of Dart *class types*. That is, *generally speaking*, an unadorned class type T will represent the *non-null* type consisting (strictly) of instances of T. When needed, the *meta type annotation* `?` can be used; $?T$ denotes the *nullable* type derived from T, consisting of values from T or `null`.

Careful consideration has been given to language design goals (Section 5) during the writing of this proposal. For example, this proposal fully preserves the optional nature of static type annotations.

The scope of this proposal includes: generics (Part C); dealing with `dynamic` and missing static type annotations (Part D); miscellaneous features including syntactic sugar (Part E); preliminary impact on the Dart SDK libraries (Part F), and finally a migration strategy (Part G).

TL;DR: the *executive summary* of updates and additions (Section 8.1) fits in 3/4 of a page.

2 Motivation

2.1 Dart: improving compilation to JavaScript

Being able to **compile Dart into compact and efficient JavaScript** (JS) is paramount, possibly even more so given the Dart news *Dart for the Entire Web*, March 25, 2015 (emphasis mine):

> In order to do what's best for our users and the web, and not just Google Chrome, we will **focus our web efforts on compiling Dart to JavaScript**. We have decided **not to integrate the Dart VM into Chrome**.

NNBD allows compilers and VMs to perform important optimizations. For example, under the current *nullable-by-default* semantics, the Dart Dev Compiler (DDC) translates this Dart fragment:

```
return (a + b) * a * b;  // a and b are declared as int
```

into this JS:

```
return dart.notNull(dart.notNull((dart.notNull(a) + dart.notNull(b)))
     * dart.notNull(a)) * dart.notNull(b);
```

as is pointed out in DDC issue #64. Under NNBD, the `dart.notNull()` wrappers become unnecessary, so that the Dart and corresponding JS would be identical. (We ignore here the issue of finite vs. arbitrary precision numerics, as it is both orthogonal to, and outside the scope of this proposal.)

Interest in non-null types for Dart and related projects has been manifested in:

- Dart issue #22, starred by 198 individuals, is a request for *support for non-nullable types*.
- Bob Nystrom's 2011 strawman *proposal for Null-safety in Dart* suggested the use of non-null types and NNBD. Unfortunately, at the time, the language committee was busy with other issues that took precedence.
- Chrome V8 team is giving thought to NNBD for JS; see Experiments with Strengthening JavaScript, especially p. 39 (the second to last page) of JSExperimentalDirections.
- Finally, non-null types are on Lasse R.H. Nielsen's 2014 wish list, and so far, almost half of his wishes have either come true or are the subject of a DEP ;).

2.2 Precedent: modern web/mobile languages with non-null types and NNBD

Programming languages, many recently released, that are relevant to web applications (either dialects of JS or that compile to JS) and/or mobile, and that support *non-null types* and NNBD include:

Language	About	v1.0?	Nullable via	Reference
Ceylon (Red Hat)	Compiles to JS, Java Bytecode (**JB**)	2013Q4	*T*?	Ceylon optional types
Fantom	Compiles to JS, JB, .Net CLR	2005	*T*?	Fantom nullable types
Flow (Facebook)	JS superset and static checker	2014Q4	*T*?	Flow maybe types
Kotlin (JetBrains)	Compiles to JS and JB	2011Q3	*T*?	Kotlin null safety
Haste	Haskell to JS compiler	@0.4.4	option type	Haskell maybe type
Swift (Apple)	iOS/OS X Objective-C successor	2014Q4	option type	Swift optional type

(Note: 2014Q4, for example, refers to the 4th quarter of 2014). There is even discussion of introducing non-null types to TypeScript, the Dart peer that brings optional types and static type checking to JavaScript:

- TS issue #185, *Suggestion: non-nullable type*.
- TS issue #1265, *Comparison with Facebook Flow Type System*.
- TS issue #3003, *Compile / edit time pluggable analyzers like C#* as a possible mechanism for introducing support for nullity checks.

3 Normative reference, terms and notation

The main normative reference for this proposal is the *ECMA Dart Specification Standard*, 2nd Edition (December 2014, Dart v1.6) which we abbreviate as DSS. When proposing changes to the DSS, text that is removed will be ~~marked like this~~ and, new or updated text will be [[marked like this]].

Throughout this proposal, qualified section numbers, sometimes in parentheses, e.g. (DSS 16.19), will refer to the corresponding section of the named resource.

We will refer to current Dart, with nullable-by-default semantics, as *classic Dart* (**DartC**) and we will use **DartNNBD** to denote Dart as adapted to conform to this proposal. In cases where it is pertinent, we will mark code samples as being DartC or DartNNBD code.

We adhere to the following terminology (DSS 7, "Errors and Warnings") a: *static warning* is a problem reported by the static checker; *dynamic type error* is a type error reported in checked mode; *run-time error* is an exception raised during execution.

4 Impact, examples and benefits: a first look

Among other important language design goals (5), this proposal has been designed so as to *minimize* the effort involved in migrating existing code, so that DartC code will require *no* or *few textual changes* to run with the *same behavior* in DartNNBD.

Consider the following program, slightly adapted from an article on Dart types (Bracha, 2012):

```
class Point {
  final num x, y;
  Point(this.x, this.y);
  Point operator +(Point other) => new Point(x+other.x, y+other.y);
  String toString() => "x: $x, y: $y";
}

void main() {
  Point p1 = new Point(0, 0);
  Point p2 = new Point(10, 10);
  print("p1 + p2 = ${p1 + p2}");
}
```

This code has the same behavior in DartC as in DartNNBD. But, in DartNNBD, an expression like `new Point(0,`*some nullable expression*`)` would cause a static warning to be issued and a dynamic type error to be raised, whereas no problems would be reported in DartC.

Dart SDK libraries. What would be the impact on the Dart SDK libraries? The `int` API would look textually the same in DartNNBD except for the addition of 3 instances of the `?` meta type annotation, out of 44 possible places where such an addition could be made (F.1.1). The `Iterable<E>` interface would be unchanged (F.1.2).

The **benefits** of this proposal include:

- An increased *potential* for the static detection of unanticipated use of `null`.
- A base semantics which enables compilers (and VMs) to be much more effective at generating efficient code (e.g., code with fewer runtime checks).
- Annotated code contributes to improving API documentation.

5 Language design goals

Language design is a process of *balancing tensions* among goals under given constraints. It is difficult to assess a language, or a language proposal, when its design principles and goals are not clearly identified.

In this section, we present some of the broad language design goals for this proposal. Other goals will be presented, as relevant, in their respective parts.

The group of goals presented next will be collectively referred to as **G0**; subgoals will be referred to by name, such as G0, compatibility.

- **Goal G0, optional types**. Specifically these two aspects, which follow from the fundamental property of Dart that *static type annotations are* **optional** (see "Overview", Bracha, 2012):

  (a) Static type annotations, whether they include nullity meta type annotations or not, shall have *no impact on* **production mode execution**.

  (b) Static type checking rules shall never prevent code from executing, and hence never coerce a developer into *adding or changing* static (nullity) type annotations.

  Comment. This is why problems reported by the static checker are warnings and not errors. The basic premise is that "the developer knows best" since he or she can see beyond the inherent limitations of the static type system.

- *Maximize* each of the following properties:

  – **Goal G0, utility** (of which more is said in the paragraph below).
  – **Goal G0, usability**: the ease with which DartNNBD can be learned and used.
  – **Goal G0, compatibility** with DartC; i.e., mainly *backwards compatibility*, though the intent is also to be respectful of Dart's language design philosophy.

- **Goal G0, ease migration**. *Minimize* the effort associated with the: (a) migration of DartC code to DartNNBD; (b) reengineering of tooling.

The main purpose of this proposal (**G0, utility**) is to enable static and dynamic checks to report situations where a possibly null expression is being used but a non-null value is expected. Even in the presence of non-null types, developers could choose to declare all types as nullable (and hence be back in the realm of DartC). Consequently, to the extent possible, in this proposal we will give preference to new language features that will interpret unadorned types as being non-null *by default*.

6 Proposal details

For ease of comprehension, this proposal has been divided into parts. Each part treats a self-contained topic and generally builds upon its predecessor parts, if any. Most parts start with a brief introduction or motivation, followed by proposed feature details, then ending with a discussion and/or presentation of alternatives.

- A. Recovering non-null types.
- B. Non-null by default—also introduces the type operators `?` and `!`.
- C. Generics.
- D. Dealing with `dynamic` and missing static type annotations.
- E. Miscellaneous, syntactic sugar and other conveniences.
- F. Impact on core libraries.
- G. Migration strategy.

7 Alternatives and deliverables

Alternatives, as well as implications and limitations have been addressed throughout the proposal. Appendix I is a review (survey) of nullity in programming languages: from languages without `null` to strategies for coping with `null`. It establishes a broad and partly historical context for this proposal and some of its presented alternatives.

Once a "critical mass" of this proposal's features have gained approval, a fully updated version of the *Dart Language Specification* will be produced. Tooling reengineering and a preliminary experience report can be found in Appendix II.

8 Executive summary

8.1 Language updates and additions

Core language design decisions:

- A.2. *Drop semantic rules giving special treatment to* `null`. In particular, the static type of `null` is taken to be `Null`, not $\bot$ (while still allowing `null` to be returned for `void` functions). As a consequence, all non-`Null` class types (except `Object`, which is addressed next) lose assignment compatibility with `null`, and hence *naturally recover* their status as *non-null types*.
- B.2. Create a *new class hierarchy root* named `_Anything` with only two immediate subclasses: `Object` and `Null`. This new root is internal and hence inaccessible to users. Thus, `Object` *remains the implicit upper bound* of classes.
- B.2. Introduce *type operators*:
  - $?T$ defines the *nullable* variant of type T;
  - $!T$, an inverse of `?`, is useful in the context of type (C.3) and optional parameters (E.1.1).
- C.3. Redefine the *default type parameter upper bound* as `?Object`, i.e., nullable-by-default (C.3.4). Non-null type parameters extend `Object` (C.3.3). Support for generics requires no additional features.

Subordinate language design decisions:

- B.2.4. Resolution of negated type test (`is!`) syntactic ambiguity.
- B.2.5. Syntax for nullable factory constructors.
- B.2.6. Syntax for nullable parameters declared using function signature syntax.
- B.3.1. Union type interoperability.
- B.3.3. Runtime representation of type operators and other shared semantics.
- B.3.5. Adjusted semantics for "assignment compatible" ($\Longleftrightarrow$).
- B.3.6. Static semantics of members of ?T.
- B.3.7. Type promotion.
- B.3.8. Type least upper bound.
- B.3.9. Null-aware operators.
- D.2.1. `!dynamic` is the unknown non-null type, and `?dynamic` is `dynamic`.
- D.2.2. Defining `!dynamic <:` S.
- E.1.1. Optional parameters are nullable-by-default in function bodies only.
- E.1.2. Normalization of type expressions.
- E.2. Syntactic sugar and other conveniences.

8.2 What is unchanged?

Other than the changes listed above, the semantics of DartNNBD match DartC, most notably:

- B.3.4. *Default variable initialization* semantics are untouched; i.e., `null` is the value of a variable when it is not explicitly initialized. Given that `null` is only assignment compatible with `Null` in DartNNBD, this will result in static warnings and dynamic type errors for uninitialized variables declared to have a non-null type.
- D.2. The role and semantics of `dynamic` are untouched. Thus, `dynamic` (and `?dynamic`) denote the "unknown type", supertype of all types. Also, e.g., in the absence of static type annotations or type arguments, `dynamic` is still assumed.

8.3 Summary of alternatives and discussions

Discussions / clarifications:

- B.4.1. Precedent: Ceylon's root is `Object | Null`.
- C.5.5. Generics and nullity in other languages or frameworks.
- D.3.1. Clarification of the semantics of `T extends !dynamic`.
- E.3.1. Scope of NNBD in DartNNBD.
- E.3.4. Subtype relation over function types unaffected by nullity.
- E.3.5. Catch target types and meta type annotations.
- E.3.7. Dart Style Guide on `Object` vs. `dynamic`.

Points of variance / proposal part alternatives:

- A.3.1. Why non-null *types*?
- A.3.2. Embracing non-null types but preserving nullable-by-default?
- B.4.2. Default initialization of non-null variables.
- B.4.3. Factory constructors.
- B.4.4. Dealing with `!Null`.
- B.4.5. Resolution of negated type test (`is!`) syntactic ambiguity.
- B.4.6. Encoding `?` and `!` as metadata.
- B.4.7. Ensuring `Object` is non-null: making `Null` a root too.
- C.5.1. Loss of expressivity due to union type interoperability.
- C.5.2. Lower bounds to distinguish nullable/maybe-nullable parameters.
- C.5.3. Statically constraining a type parameter to be nullable but *not* `Null`.
- C.5.4. Parametric nullity abstraction.
- D.3.2. Semantics for `dynamic`.
- D.3.3. Defining `!dynamic <:` *S*.
- E.3.2. Scope of NNBD in other languages or frameworks.
- E.3.3. Optional parameters are always nullable-by-default.
- E.3.6. Reducing the annotation burden for local variables.

8.4 Assessment of goals

This proposal has strictly upheld G0, optional types, in particular, in the choices made to preserve the DartC semantics:

- Regarding default (non-null) variable initialization (B.3.4 vs. B.4.2), and by
- Leaving `dynamic`, the unknown type, as nullable (D.2 vs. D.3.2).

Consequently, these features also support G0, compatibility, and hence G0, usability—since fewer differences relative to DartC, and fewer special cases in the semantic rules, make DartNNBD easier to learn and use—as well as G0, ease migration.

Unavoidably, recovery of non-null types (A.2), induces two **breaking changes** that may impact the production mode execution of existing programs that:

(a) Use *reflection* to query: the direct members of, or the supertype of, `Object` or `Null` (B.2.1); or, the upper bound of a type parameter (C.3.4).
(b) Perform *type tests* of the form *e* `is Object` since this will now return false for `null`. It seems unlikely though, that fielded code would actually contain such a type test given that it is always true in DartC.

We have noted that breaking changes of similar magnitude are sometimes incorporated in Dart minor releases—see the Dart CHANGELOG.

There are **no other backwards incompatible** changes impacting *production mode execution* (G0, compatibility).

Trending seems to indicate that there is value (G0, utility) in having non-null types and NNBD supported by languages with static type systems. This proposal helps Dart recover its non-null types and proposes adoption of NNBD. The latter is the principle feature in support of G0, ease migration; another key decision in support of ease of migration is leaving optional parameters (E.1.1) outside the scope of NNBD (E.3.1).

In fact, most of the core language design decisions adopted for this proposal relate back to the choices made concerning the **scope of NNBD** (E.3.1). During the creation of this proposal we constantly revisited potential impacts on the scope of NNBD to ensure that the proposal stayed true to Dart's overall language design philosophy. Our main point of comparison, detailing the many ways in which this proposal could have been differently crafted, is the section on the *scope of NNBD in other languages or frameworks* (E.3.2).

Overall, we are hopeful that this proposal has found a suitable balance between the G0 goals of *utility*, *usability*, *compatibility* and *ease of migration*.

Part A: Recovering non-null types

The purpose of this part is to "recover" Dart's non-null types, in the sense that we describe next.

A.1 Non-null types in DartC

In Dart, *everything* is an object. In contrast to other mainstream languages, the term `null` refers to the null *object*, not the null *reference*. That is, `null` denotes the singleton of the `Null` class. Although built-in, `Null` is like a regular Dart class and so it is a subtype of `Object`, etc.

Given that everything is an object in Dart, and in particular that `null` is an object of type `Null` as opposed to a null *reference* then, in a sense, **DartC types are already non-null**. To illustrate this, consider the following DartC code:

```
1  const Null $null = null;
2
3  void main() {
4    int i = null,
5        j = $null,
6        k = "a-string";
7    print("i = $i, j = $j, k = $k");
8    print("i is ${i.runtimeType}, j is ${j.runtimeType}");
9  }
```

Running the Dart Analyzer results in

```
Analyzing [null.dart]...
[warning] A value of type 'Null' cannot be assigned to a variable of type 'int' (line 5, col 11)
[warning] A value of type 'String' cannot be assigned to a variable of type 'int' (line 6, col 11)
2 warnings found.
```

A.1.1 Static checking

As is illustrated above, the `Null` type is unrelated to the type `int`. In fact, as a direct subtype of `Object`, `Null` is only related to `Object` and itself. Hence, the assignment of `$null` to `j` results in a static warning just as it would for an instance of any other type (such as `String`) unrelated to `int` (DSS 16.19, "Assignment", referring to an assignment $v = e$):

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

While the static type of `$null` is `Null`, the language specification has a special rule used to establish the static type of `null`. This rule makes `null` assignment compatible with any type T, including `void` (DSS 16.2, "Null"):

> The static type of `null` is $\bot$ (bottom). *(Rationale) The decision to use $\bot$ instead of `Null` allows `null` to be assigned everywhere without complaint by the static checker.*

Because bottom is a subtype of every type (DSS 19.7, "Type Void"), `null` can be assigned to or used as an initializer for a variable of any type, without a static warning or dynamic type error (DSS 16.19; 19.4, "Interface Types").

A.1.2 Checked mode execution

Execution in checked mode of the program given above results in an exception being reported only for the assignment to `k`:

```
> dart -c null.dart
 Unhandled exception: type 'String' is not a subtype of type 'int' of 'k'.
 #0      main (~/example/null.dart:6:11)
```

The assignment to `j` raises no exception because of this clause (DSS 16.19, "Assignment", where o is the result of evaluating e in $v = e$):

> In checked mode, it is a dynamic type error if o is not `null` and the interface of the class of o is not a subtype of the actual type (19.8.1) of v.

A.1.3 Production mode execution

Production mode execution of our sample code results in successful termination and the following output is generated:

```
i = null, j = null, k = a-string
i is Null, j is Null
```

Note that `Null` is the `runtimeType` of both `null` and `$null`; bottom is not a runtime type.

A.1.4 Relations over types: <<, <:, and $\Longleftrightarrow$

We reproduce here the definitions of essential binary relations over Dart types found in DSS 19.4, "Interface Types". We will appeal to these definitions throughout the proposal. Let S and T be types.

- T may be *assigned to* S, written $T \Longleftrightarrow S$, iff either $T <: S$ or $S <: T$. (Let T be the static type of e. We will sometimes write "e may be assigned to S" when we mean that "T may be assigned to S". Given that this relation is symmetric, we will sometimes write that S and T are **assignment compatible**.)
- T is a *subtype* of S, written $T <: S$, iff $[\bot/\textbf{dynamic}]T << S$.
- T is *more specific than* S, written $T << S$, if one of the following conditions is met:
  - T is S.

- T is $\bot$.
- S is **dynamic**.
- S is a direct supertype of T.
- T is a type parameter and S is the upper bound of T.
- T is a type parameter and S is Object.
- T is of the form $I < T_1, \ldots, T_n >$ and S is of the form $I < S_1, \ldots, S_n >$ and: $T_i << S_i, 1 \leq i \leq n$
- T and S are both function types, and $T << S$ under the rules of DSS 19.5.
- T is a function type and S is Function.
- $T << U$ and $U << S$.

A.2 Feature details: recovering non-null types

To recover the general interpretation of a class type T as non-null, we propose the following changes.

A.2.1 `Null` is the static type of `null`

We drop the rule that attributes a special static type to `null`, and derive the static type of `null` normally as it would be done for any constant declared of type `Null` (DSS 16.2, "Null"): "~~The static type of `null` is $\bot$ (bottom). *(Rationale) The decision to use $\bot$ … checker.*~~".

A.2.2 `Null` may be assigned to `void`

As explained in DSS 17.12, "Return", functions declared `void` must return *some* value. (In fact, in production mode, where static type annotations like `void` are irrelevant, a `void` function can return *any* value.)

> Comment. Interestingly, a `void` function in Ceylon is considered to have the return type `Anything`, though such functions always return `null`. Identification with `Anything` is to permit reasonable function subtyping ([Ceylon functions).

In DartC checked mode, `void` functions can either implicitly or explicitly return `null` without a static warning or dynamic type error. As was mentioned, this is because the static type of `null` is taken as $\bot$ in DartC. In DartNNBD, we make explicit that `Null` can be *assigned to* `void`, by establishing that `Null` is more specific than `void` (A.1.4): `Null << void`.

> Comment. In a sense, this makes explicit the fact that `Null` is being treated as a "carrier type" for `void` in Dart. `Null` is a unit type, and hence returning `null` conveys no information. The above also fixes the slight irregularity noted in A.1.1: in DartNNBD, no static warning will result from a statement like `return $null;` used inside a `void` function (where `$null` is declared as a `const Null`).

A.2.3 Drop other special semantic provisions for `null`

Special provisions made for `null` in the DartC semantics are dropped in DartNNBD, such as:

- DSS 16.19, "Assignment": In checked mode, it is a dynamic type error if ~~*o* is not null and~~ the interface of the class of o is not a subtype of the actual type (19.8.1) of v.
- DSS 17.12, "Return", e.g., for a synchronous function: it is a dynamic type error if ~~*o* is not null and~~ the runtime type of o is not a subtype of the actual return type of f.

We will address other similar ancillary changes to the semantics once a "critical mass" of this proposal's features have gained approval (7).

A.3 Discussion

As we do at the end of most parts, we discuss here topics relevant to the changes proposed in this part.

A.3.1 Why non-null *types*?

Of course, one can appeal to programmer discipline and encourage the use of coding idioms and design patterns as a means of avoiding problems related to `null`. For one thing, an option type can be realized in most languages (including Dart), as can the *Null Object* pattern (Fowler, C&C). Interestingly, Java 8's new `java.util.Optional<T>` type is being promoted as a way of avoiding `null` pointer exceptions (**NPE**s) in this Oracle Technology Network article entitled, "Tired of Null Pointer Exceptions? Consider Using Java SE 8's Optional!".

Coding discipline can only go so far. Avoiding problems with `null` is best achieved with proper language support that enables mechanized tooling diagnostics (vs. manual code reviews). Thus, while the use of option types (or any other discipline/strategy for avoiding `null` described in the survey) could be applicable to Dart, we do not give serious consideration to any language feature less expressive than non-null types. Given that there is generally *some* effort involved on the part of developers who wish nullable and non-null types to be distinguished in their code, support for non-null *types* offer the **highest return on investment** (ROI), especially in the presence of generics. Hence, we have chosen to base this proposal on non-null types rather than, e.g., non-null *declarator* annotations (I.3.1(b), Dart issue #5545), which would not impact the type system. Languages like JML, for example, which previously only supported nullity assertion constraints and nullity declaration modifiers, evolved to support non-null types and NNBD (I.3.1).

It is interesting to note a similar evolution in tool support for *potential "null dereference" errors* in modern (and popular) IDEs like in IntelliJ and the Eclipse JDT. Following conventional terminology, we will refer to such errors as NPEs. As Stephan Herrmann (Eclipse JDT committer) points out (Herrmann ECE 2014, page 3), NPEs remain the most frequent kind of exception in Eclipse. This high rate of occurrence of NPEs is not particular to the Eclipse code base or even to Java.

Slide 5 of Stephan Herrmann's *Advanced Null Type Annotations* talk summarizes the evolution of support for nullity analysis in the Eclipse JDT. While initial analysis was ad hoc, the advent of Java 5 metadata allowed for the introduction of nullity annotations like `@NonNull` and `@Nullable`. Such annotations were used early on by the Eclipse JDT and the popular Java linter Findbugs to perform intraprocedural analysis. As of Eclipse Luna (4.4), support is provided for non-null *types* (and interprocedural analysis), and options exist for enabling NNBD at various levels of granularity. Such an evolution (from ad hoc, to nullity declarator annotations, to non-null types), seems to be part of a general trend that we are witnessing in programming language evolution (I.4), towards features that enable efficient and effective static checking, so as to help uncover coding errors earlier—in particular through the use of non-null types, and in many cases, NNBD.

A.3.2 Embracing non-null types but preserving nullable-by-default?

As an alternative to the changes proposed in this part, the nullable-by-default semantics of DartC could be preserved in favor of the introduction of a *non-null* meta type annotation `!`. Reasons for not doing this are given in the *Motivation* section of the next part.

Part B: Non-null by default (NNBD)

B.1 Motivation: nullable-by-default increases migration effort

Several languages (see the survey) with nullable-by-default semantics that have been subsequently retrofitted with support for non-null types have achieved this through the introduction of meta type annotations like `?` and `!`, used to indicate the nullable and non-null variants of a type, respectively.

The simplest adaptation to a language with a nullable-by-default semantics like DartC, is to leave the default untouched and require developers to explicitly mark types as non-null using the `!` meta type annotation.

```
// DartC extended with the meta type annotation `!'
int i = null;     // ok
!String s = null; // error
```

Unfortunately, this would unnecessarily burden developers and negatively impact G0, ease migration as we explain next. An empirical study of Java code established that 80% of declarations (having a reference type) are meant to be non-null, *by design*. An independent study reports 20 nullity annotations per KLOC (Dietl, 2014; Dietl et al., 2011).

We expect the proportion of non-null vs. nullable declarations in Dart to be similarly high; a claim supported by anecdotal evidence—e.g., Nystrom, 2011, and our preliminary experiments in translating the Dart SDK libraries (Part F). For example, under a variant of DartC extended with `!`, `int.dart` would have to be updated with 38 `!` annotations (that's 86%) against 6 declarations left undecorated.

B.2 Feature details: non-null by default

A consequence of dropping the special semantic rules for `null` (A.2) is that all non-`Null` classes except `Object` lose assignment compatibility with `Null`, and hence *naturally recover* their status as *non-null types*. In DartC, `Null` directly extends `Object` and so `Null <: Object`. This means that `Null` may still be assigned to `Object`, effectively making `Object` nullable. We ensure that `Object` is non-null as follows.

B.2.1 Ensuring `Object` is non-null: elect `_Anything` as a new root

We define the internal class `_Anything` as the **new root** of the class hierarchy. Being internal, it cannot be subclassed or instantiated by users. `Object` and `Null` are immediate subclasses of `_Anything`, redeclared as:

```
abstract class _Anything { const _Anything(); }

abstract class _Basic extends _Anything {
  bool operator ==(other) => identical(this, other);
```

```
  int get hashCode;
  String toString();
  dynamic noSuchMethod(Invocation invocation);
  Type get runtimeType;
}

class Object extends _Anything implements _Basic {
  const Object();
  ... // Methods of _Basic are all declared external
}
```

The definition of `Null` is the same as in DartC except that the class extends `_Anything` and implements `_Basic`. The latter declares all members of DartC's `Object`. Note that the declaration of equality allows a `null` operand (such a definition is needed, e.g., by the Dart Analyzer).

> Comment. Declaring `_Anything` as a class without methods allows us to provide a conventional definition for `void` as an empty interface, realized only by `Null`:
>
> ```
> abstract class void extends _Anything {}
> class Null extends _Anything implements _Basic, void { /* Same as in DartC */ }
> ```

The changes proposed in this subsection impact various sections of the language specification, including (DSS 10, "Classes"): "Every class has a single superclass except class ~~`Object`~~[[`_Anything`]] which has no superclass".

As is discussed below (B.4.1), Ceylon has a class hierarchy like the one proposed here for Dart.

> Comments:
>
> - `Object` remains the implicit upper bound of classes (i.e., `extends` clause argument).
> - Under this new hierarchy, `Null` is only assignable to the new root, `void` and itself.

B.2.2 Nullable type operator `?`

The **nullable type operator**, `?`T, is used to introduce a nullable variant of a type T.

> Comment. Like other metadata annotations in Dart, `?` is applied as a prefix.

B.2.3 Non-null type operator `!`

The **`!`** (bang) **non-null type operator**, can be thought of as an inverse of the nullable type operator `?`. It also acts as an identity function when applied to non-null types.

B.2.4 Resolution of negated type test (`is!`) syntactic ambiguity

Unfortunately, the choice of `!` syntax introduces an ambiguity into the grammar relative to negated type tests, such as: `o is !`T. The ambiguity shall be resolved in favor of the original negated type test production, requiring parentheses for a type test against a non-null type, as in `o is (!`T`)`. See B.4.5 for further discussion and an alternative.

B.2.5 Syntax for nullable factory constructors

It may seem unnecessary to qualify that factory constructors are non-null, but in DartC, a factory constructor for a class T is permitted to return an instance of *any* subtype of T, including `null` (DSS 10.6.2, "Factories"):

> In checked mode, it is a dynamic type error if a factory returns a non-`null` object whose type is not a subtype of its actual return type. *(Rationale) It seems useless to allow a factory to return `null`. But it is more uniform to allow it, as the rules currently do.*

In support of G0, compatibility, we propose to extended the syntax of factory constructors so that they can be marked nullable, as is illustrated next. For further discussion and an alternative see B.4.3.

```
// DartNNBD - part of dart.core;
abstract class int extends num {
  external const factory ?int.fromEnvironment(String name, {int defaultValue});
  ...
}
```

B.2.6 Syntax for nullable parameters declared using function signature syntax

A formal parameter can be declared by means of a function signature (DSS 9.2.1, "Required Formals") as is done for `f` in: `int applyTo1(int f(int)) => f(1)`.

This, in effect, declares an anonymous class type (DSS 19.5, "Function Types") "that implements the class `Function` and implements a `call` method with the same signature as the function". The NNBD rule also applies to such anonymous class types, so special syntax must be introduced to allow them to be marked as nullable. To declare a such a parameter as nullable, the parameter name can be *suffixed* with `?` as in:

```
int applyTo1(int f?(int)) => f == null ? 1 : f(1);
```

This can be thought of as equivalent to:

```
typedef int _ANON(int);
int applyTo1(?_ANON f) => f == null ? 1 : f(1);
```

This syntactic extension to function signatures can only be used in formal parameter declarations, not in other places in which function signatures are permitted by the grammar (e.g., class member declarations and type aliases).

> Comment. We avoid suggesting the use of `?` as a *prefix* to the function name since that could be interpreted as an implicitly (nullable) `dynamic` return type when no return type is provided.

B.3 Semantics

B.3.1 Semantics of ?

(a) Union type interoperability

While *union types* are not yet a part of Dart, they have been discussed by the Dart standards committee, and a proposal is anticipated. Once introduced, union types and the language features suggested by this proposal—especially the `?` type operator—will need to "interoperate" smoothly. This can be achieved by defining the nullable type operator as:

$$\texttt{?}T = T \mid \texttt{Null}$$

The semantics of `?` then follow naturally from this definition. While the Dart union type proposal has yet to be published, it can be safe to assume that its semantics will be *similar* to that of union types in other languages such as:

- TypeScript, see Section 3.4 of the TypeScript language specification; or,
- Ceylon, see the language specification Section on Ceylon union types.

From such a semantics it follows that, e.g., `Null <: ?T` and `T <: ?T` for any T.

(b) Core properties of `?`

This proposal does not *require* union types. In the absence of union types we characterize `?` by its core properties. For any type T

- `Null` and T are *more specific* than $?T$ (A.1.4):
  - $\texttt{Null} << ?T$,
  - $T << ?T$;
- $??T = ?T$ (idempotence),
- $?\texttt{Null} = \texttt{Null}$ (fixed point),
- $?\texttt{dynamic} = \texttt{dynamic}$ (fixed point, D.2.1).

These last three equations are part of the rewrite rules for the **normalization** of $?T$ expressions (B.3.3). When $?V$ and $?U$ are in normal form, then:

- $?V << S$ **iff** $\texttt{Null} << S \wedge V << S$.

It is a compile-time error if `?` is applied to `void`. It is a static warning if an occurrence of $?T$ is not in normal form.

B.3.2 Semantics of `!`

When regarding $?T$ as the union type $T \mid \texttt{Null}$, then `!` can be seen as a projection operator that yields the non-`Null` union member T. For all non-null class types T `<: Object`

- $!?T = T$ (inverse of `?`)
- $!T = T$ (identity over non-null types)

These equations are part of the rewrite rules for the **normalization** of $!T$ expressions (B.3.3).

It is a compile-time error if `!` is applied to `void`. Application of `!` to an element outside its domain is considered a *malformed* type (DSS 19.1, "Static Types") and "any use of a malformed type gives rise to a static warning. A malformed type is then interpreted as `dynamic` by the static type checker and the runtime unless explicitly specified otherwise". Alternatives are presented in B.4.4.

> Comment. Currently in DartNNBD, the only user expressible type outside of the domain of `!` is `Null` since `_Anything` is not accessible to users (B.2.1).

B.3.3 Runtime representation of type operators and other shared semantics

Besides the semantic rules presented in the previous two subsections for their respective type operators, all other checked mode semantics (static warnings or dynamic type errors) for both `?` and `!` follow from those of DartC and the semantics of DartNNBD presented thus far.

Type expressions involving type operators shall be represented at runtime, in normalized form (E.1.2, for use in:

- Reflection.
- Reification (C.4).
- Structural type tests of function types (E.3.4).

B.3.4 Default initialization of non-null variables is like DartC

We make no changes to the rules regarding default variable initialization, even if a variable is statically declared as non-null. In particular, the following rule still applies (DSS 8, "Variables"): "A variable that has not been initialized has the initial value `null`".

> Comment. The term *variable* refers to a "storage location in memory", and encompasses local variables, library variables, instance variables, etc. (DSS 8).

Explicit initialization checks are extended to also address cases of implicit initialization with `null`. Thus, generally speaking, explicit or implicit initialization of a variable with a value whose static type cannot be assigned to the variable, will result in:

- Static warning.
- Dynamic type error.
- No effect on production mode execution.

Rule details are given next.

(a) Instance variables

An instance variable v that is:

1. `final`, or
2. declared in a non-`abstract` class and for which: `null` cannot be assigned to the (actual) type of v;

then v be explicitly initialized (either from a declarator initializer, a field formal parameter, or a constructor field initialization).

> Comment. Conforming to DartC, the above holds true for nullable `final` instance variables even if this is not strictly necessary. In (2) we disregard abstract classes since we cannot easily and soundly determine if all of its uses (e.g. as an interface, mixin or extends clause target) will result in all non-null instance variables being explicitly initialized).

(b) Class (static) and library variables

A class or library variable that is (1) `const` or `final`, or (2) declared non-null, must be explicitly initialized.

> Comment. Conforming to DartC, the above holds true for nullable `const` or `final` variables even if this is not strictly necessary.

(c) Local variables

(1) A `const` or `final` local variable must be explicitly initialized.
(2) For a non-null local variable, a static warning (and a dynamic type error) will result if there is a path from its declaration to an occurrence of the variable where its value is being read. If a local variable read in inside a closure, then it is assumed to be read at the point of declaration of the closure. Also see E.3.6.

B.3.5 Adjusted semantics for "assignment compatible" ($\Longleftrightarrow$)

Consider the following DartNNBD code:

```
?int i = 1; // ok
class C1<T1 extends int> { T1 i1 = 1; } // ok
class C2<T2 extends int> { ?T2 i2 = 1; } // should be ok
```

According to the DartC definition of assignment compatible described in A.1.4, a static warning should be reported for the initialization of `i2`. To understand why, let us examine the general case of

```
class C<T extends B> { T o = s; }
```

where `s` is some expression of type S. Let us write T^B to represent that the type parameter T has upper bound B. The assignment to `o` is valid if S is assignment compatible with T^B, written $S \Longleftrightarrow T^B$. But T^B is incomparable when it is not instantiated. The best we can do is compare S to B and try to establish that $B <: S$. Thus, $S \Longleftrightarrow T^B$

$= S <: T^B \lor T^B <: S$ (by definition of $\Longleftrightarrow$)
$\Longleftarrow S <: T^B \lor T^B <: B \land B <: S$
$= S <: T^B \lor B <: S$ (simplified because B is the upper bound of T^B).

where $\Longleftarrow$ is reverse implication. In the case of class `C2` above, the field `i2` is of type ?`T2`, hence we are dealing with the general case: $S \Longleftrightarrow {}^?T^B$

$= S <: {}^?T^B \lor {}^?T^B <: S$ (by definition of $\Longleftrightarrow$)
$= S <: \text{Null} \lor S <: T^B \lor {}^?T^B <: S$ (property of ?)
$= S <: \text{Null} \lor S <: T^B \lor (\text{Null} <: S \land T^B <: S)$ (property of ?)
$\Longleftarrow S <: \text{Null} \lor S <: T^B \lor (\text{Null} <: S \land T^B <: B \land B <: S)$
$= S <: \text{Null} \lor S <: T^B \lor (\text{Null} <: S \land B <: S)$. (*)

If we substitute the type of `i2` and the bound of `T2` for S and B in (*) and we get:

$\text{int} <: \text{Null} \lor \text{int} <: T^{\text{int}} \lor (\text{Null} <: \text{int} \land \text{int} <: \text{int})$
$= \text{false} \lor \text{int} <: T^{\text{int}} \lor (\text{false} \land \text{true})$
$= \text{int} <: T^{\text{int}} \lor \text{false}$
$= \text{false}$.

This seems counter intuitive: if `i2` is (at least) a nullable `int`, then it should be valid to assign an `int` to it. The problem is that the definition of assignment compatible is too strong in the presence of union types. Before proposing a relaxed definition we repeat the definition of assignability given in A.1.4, along with the associated commentary from (DSS 19.4):

> An interface type T may be assigned to a type S, written $T \Longleftrightarrow S$, iff either $T <: S$ or $S <: T$. *This rule may surprise readers accustomed to conventional type checking. The intent of the $\Longleftrightarrow$ relation is not to ensure that an assignment is correct. Instead, it aims to only flag assignments that are almost certain to be erroneous, without precluding assignments that may work.*

In the spirit of the commentary, we refine the definition of "assignment compatible" as follows: let T, S, V and U be any types such that $^{?}V$ and $^{?}U$ are in normal form, then we define $\Longleftrightarrow$ by cases:

- $T \Longleftrightarrow {}^{?}U$ **iff** $T \Longleftrightarrow \text{Null} \vee T \Longleftrightarrow U$, when T is *not* of the form $^{?}V$
- Otherwise the DartC definition holds; i.e., $T \Longleftrightarrow S$ iff $T <: S \vee S <: T$.

  Comment. It follows that $^{?}V \Longleftrightarrow {}^{?}U$ iff $V \Longleftrightarrow U$. An equivalent redefinition is:
  $T \Longleftrightarrow S$ **iff** $T <: S \vee S <: T \vee S = {}^{?}U \wedge U <: T$ (for some U).

If we expand this new definition for arguments $^{?}V$ and S, we end up with the formula (*) as above, except that the last logical operator is a disjunction rather than a conjunction. Under this new relaxed definition of assignment compatible, `i2` can be initialized with an `int` in DartNNBD.

B.3.6 Static semantics of members of ?T

We define the static semantics of the members of $?T$ as if it were an anonymous class with `Null` and T as superinterfaces. Then the rules of member inheritance and type overrides as defined in (DSS 11.1.1) apply.

B.3.7 Type promotion

In the context of `if` statements, conditional expressions, and conjunction and disjunction expressions, the following type promotions shall be performed for any expression e of type $?T$:

Condition	True context	False context
e == null	e is `Null`	e is T
e != null	e is T	e is `Null`
e is T	e is T	-
e is! T	-	e is T

This applies to function types as well.

B.3.8 Type least upper bound

The least upper bound of `Null` and any non-`void` type T is $?T$.

B.3.9 Null-aware operators

Comment. TODO.

B.4 Discussion

B.4.1 Precedent: Ceylon's root is `Object | Null`

The Ceylon language essentially has the nullity semantics established so far in this proposal but without `!`, i.e.: types are non-null by default, `?` is a (postfix) nullable meta type annotation, and the top of the Ceylon type hierarchy is defined with a structure identical to that proposed in B.2.1 for DartNNBD, namely:

```
abstract class Anything of Object | Null
class Null of null extends Anything
class Object extends Anything
```

Thus, `Anything` is defined as the *union type* of `Object` and `Null`.

B.4.2 Default initialization of non-null variables, alternative approaches

(a) Preserving DartC semantics is consistent with JavaScript & TypeScript

Our main proposal (B.3.4) preserves the DartC semantics, i.e., a variable not explicitly initialized is set to `null`. In JavaScript, such variables are set to `undefined` (ES5 8.1), and TypeScript conforms to this behavior as well (TSLS 3.2.6).

For variables statically declared as non-null, some might prefer to see this proposal *mandate* (i.e., issue a compile-time error) if the variable is not explicitly initialized (with a value assignable to its statically declared type, and hence not `null`) but this would go against G0, optional types.

In our opinion, preserving the default variable initialization semantics of DartC is the only approach that is consistent with G0, optional types. Also see I.3.2 for a discussion of issues related to soundness. Although Dart's static type system is already unsound by design (Brandt, 2011), this proposal does not contribute to (increase) the unsoundness because of non-null types. NNBD scope and local variables are also discussed in E.3.2(a).

Also see E.3.2(a) and E.3.6.

(b) Implicit type-specific initialization of non-null variables

In some other languages (especially in the presence of primitive types), it is conventional to have type-specific default initialization rules—e.g., integers and booleans are initialized to 0 and false, respectively. Due to our desired conformance to G0, optional types, it is not possible to infer such type-specific default initialization from a static type annotation *alone*. On the other hand, special declarator syntax, such as (where `T` is a class type and `<U,...>` represents zero or more type arguments):

```
!T<U,...> v;
```

could be treated as syntactic sugar for

```
T<U,...> v = T<U,...>.DEFAULT_INIT();
```

In production mode this would be interpreted as:

```
var v = T<U,...>.DEFAULT_INIT();
```

Any class type `T`, for which this form of initialization is desired, would provide `DEFAULT_INIT()` as a factory constructor, e.g.:

```
abstract class int extends num {
  factory int.DEFAULT_INIT() => 0;
  ...
}
```

Although what we are proposing here effectively overloads the meaning of meta type annotation `!`, there is no ambiguity since, in an NNBD context, a class type T is already non-null, and hence $!T$—which is not in normal form (B.3.3)—can be interpreted as a request for an implicit type-specific initialization. This even extends nicely to handle `!T` optional parameter declarations (E.1.1).

B.4.3 Factory constructors, an alternative

In B.3.2 we extended the syntax of factory constructors so that they could be marked as nullable. Allowing a factory constructor to return `null` renders *all* `new`/`const` expressions *potentially nullable*. This is an unfortunate complication in the semantics of Dart (and hence goes against G0, usability).

As was mentioned earlier, in DartC, a factory constructor for a class T is permitted to return an instance of *any* subtype of T, including `null` (DSS 10.6.2, "Factories"): "In checked mode, it is a dynamic type error if a factory returns a non-`null` object whose type is not a subtype of its actual return type. *(Rationale) It seems useless to allow a factory to return* `null`*. But it is more uniform to allow it, as the rules currently do*". From the statement of rationale, it seems that factory constructors have been permitted to return `null` out of a desired uniformity in the application of the semantic constraint on factory results (which is based on subtyping).

Given that `Null` is no longer a subtype of every type in DartNNBD, we could also choose to (strictly) uphold the uniformity of the subtype constraint, thus *disallowing* a factory *constructor* from returning `null`—of course, factory *methods* could be nullable. Unfortunately, this would be a breaking change impacting features of the Dart core library, in particular `const factory` constructors like `int.fromEnvironment()` and `String.fromEnvironment()`. Because of the `const` nature of these factories, they have proven useful in "*compile-time dead code elimination*" (Ladd, 2013). We suspect that few other factory constructors return `null` other than in the context of this idiom, and those that do, could provide a non-null default return value.

There has been some discussions of the possible elimination of `new` and/or `const` as constructor qualifiers (e.g., Nielsen, 2015), in which case the attempted distinction made here of factory constructors vs. factory methods would be moot.

B.4.4 Dealing with `!Null`, alternatives

In the absence of generics, `!Null` could simply be reported as a compile-time error. With generics, the issue is more challenging since we must deal with type expressions like `!T` possibly when type parameter `T` is instantiated with `Null` (Part C).

While we proposed, in B.3.2, to define $!T$ as malformed when T is `Null`, alternatives include treating it as (i) $\bot$, or (ii) a distinct empty (error) type that is assignment compatible with no other type. The latter would introduce a new way of handling type errors to Dart, in contrast to the current uniform treatment of such "errored types" as malformed instead. Use of $\bot$ would also be a new feature since, to our knowledge, no type expression can be $\bot$ in DartC. Hence both of these alternatives introduce extra complexity, thus decreasing G0, usability and increasing retooling costs (G0, ease migration).

B.4.5 Resolution of negated type test (`is!`) syntactic ambiguity, an alternative

Syntactic ambiguity between a negated type test and a type test against a non-null type (B.2.4) could be avoided by adopting a different symbol, such as `~`, for the non-null type operator, but `!` is conventional. It helps somewhat that there is a lexical convention (enforced by the Dart Code Formatter) of writing the tokens `is` and `!` immediately adjacent to each other. It might further help if the analyzer reported a hint when the tokens `is` and `!` are separated by whitespace, inquiring (something like): "did you intend to write `o is (!`T`)`?".

Note that there is no *class name* T that can be written in a non-null type test `o is (!`T`)` because `!Null` is malformed and $!T$ will not be in normal form otherwise (B.3.2). But as we shall see in Part C, it is legal to write $!T$ when T is a type parameter name.

B.4.6 Encoding `?` and `!` as metadata

Use of specialized syntax for meta type annotations `?` and `!` requires changes to Dart tooling front ends, impacting G0, ease migration. We can *almost* do away with such front-end changes by encoding the meta type annotations as metadata such as `@NonNull` and `@Nullable`. We write "almost" because Dart metadata annotations would first need to be (fully) extended to types through an equivalent of JSR-308 which extended Java's metadata facility to types. Broadened support for type metadata (which was mentioned in the DEP 2015/03/18 meeting) could be generally beneficial since nullity type annotations are only one among a variety of useful kinds of type annotation. E.g., the Checker Framework, created jointly with JSR itself by the team that realized JSR-308, offers 20 checkers as examples, not the least of which is the Nullness Checker. It might also make sense to consider *internally* representing `?` and `!` as type metadata. But then again, special status may make processing of this core feature more efficient in both tooling and runtimes.

Regardless, the use of the single character meta type annotations `?` and `!` seems to have become quite common: it is certainly much shorter to type and it makes for a less noisy syntax.

B.4.7 Ensuring `Object` is non-null: making `Null` a root too

An alternative to creating a new class hierarchy root (B.2.1) is to create a class hierarchy *forest* with two roots `Object` and `Null`. This has the advantage of being a less significant change to the class hierarchy, benefiting G0, ease migration, though it is less conventional.

```
  class Object {
    const Object();
    bool operator ==(other) => identical(this, other);
    external int get hashCode;
    external String toString();
    external dynamic noSuchMethod(Invocation invocation);
    external Type get runtimeType;
    }

- class Null {
+ class Null /*no supertype*/ {
    factory Null._uninstantiable() {
      throw new UnsupportedError('class Null cannot be instantiated');
    }
+   external int get hashCode;
    String toString() => "null";
+   external dynamic noSuchMethod(Invocation invocation);
+   external Type get runtimeType;
  }
```

Note that `dynamic` remains the top of the subtype relation.

Part C: Generics

C.1 Motivation: enhanced generics through non-null types

One of the main benefits of a non-null type system is its potential interplay with generics. It is quite useful, for example, to be able to declare a `List` of non-null elements, and know that list element access will yield non-null instances.

C.2 Design goals for this part

G1: Support three kinds of formal type parameter

Support three kinds of formal type parameter: i.e., formal type parameters that constrain arguments to be

1. Non-null.
2. Nullable.
3. Either non-null or nullable.

(We address whether the last two cases should be distinguished in C.3.2 and C.5.2.)

G2: Support three kinds of type parameter expression in a class body

Within the body of a generic class, we wish to be able to represent three kinds of type parameter expression for any given formal type parameter: i.e., use of a type parameter name as part of a type expression, occurring in the class body, that is

1. Non-null.
2. Nullable.
3. Matching the nullity of the argument.

Running example

Defining and assessing suitable DartNNBD language features in support of Goals G1 and G2 has been one of the most challenging aspects of this proposal. To help us understand the choices we face, we will use the following Dart code as a running example. Note that this code uses `/*(...)*/` comments to mark those places where we want to come up with appropriate syntax. Each of the three cases of Goal G2 is represented in the class body.

```
class Box< /*(...)*/ T /*extends (...) Object*/ > {
  final /*(non-null)*/ T _default; // non-null (G2.1)
  /*(matching)*/ T value; // match nullity of type parameter T (G2.3)

  Box(this._default, this.value);

  /*(nullable)*/ T maybeNull() => // nullable (G2.2)
      value == _default ? null : value;

  /*(non-null)*/ T neverNull() => value == null ? _default : value;
}
```

Thus, `Box<`*U*`>.value` would have the same nullity as *U*. For example, `Box<?int>.value` would be of type `?int` and `Box<String>.value` of type `String`. As defined above, `Box<`*U*`>.maybeNull()` returns `null` when `value` matches `_default`, even if *U* is non-null. Finally, `Box<`*U*`>.neverNull()` always returns a non-null value regardless of the nullity of *U*.

C.3 Feature details: generics

We now work through the three cases of Goal G1 in reverse order.

C.3.1 Maybe-nullable formal type parameter, case G1.3

Here is an illustration of the base syntax (without any syntactic sugar or abbreviations) for the maybe-nullable formal type parameter case (code inessential to presentation has been elided, "`...`"):

```
// DartNNBD
class Box<T extends ?Object> {
  final !T _default;     // non-null (G2.1)
  T value;               // nullity matching parameter (G2.3)
  ?T maybeNull() => ...; // nullable (G2.2)
  ...
}
```

C.3.2 Nullable formal type parameter, case G1.2

Given that Dart generics are covariant and that `T <: ?T`, it would be a significant departure from the current Dart semantics if we were to define static checking rules *requiring* that a type argument be nullable while rejecting non-null arguments. Thus, we propose that cases G1.2 and G1.3 be indistinguishable in DartNNBD. For an alternative, see C.5.2.

C.3.3 Non-null formal type parameter, case G1.1

For a non-null formal type parameter `T` we simply have `T` extend `Object`; again, here is the syntax without any sugar or abbreviations:

```
// DartNNBD
class Box<T extends Object> {
  final !T _default;     // non-null (G2.1)
```

```
  T value;               // nullity matching parameter (G2.3)
  ?T maybeNull() => ...; // nullable (G2.2)
  ...
}
```

Comment. Given that `T` is non-null, the use of `!` could be dropped in the body.

C.3.4 Default type parameter upper bound is `?Object`

When no explicit upper bound is provided for a type parameter it is assumed to be `?Object`, thus providing clients of a generic type the most flexibility in instantiating parameters with either a nullable or non-null type (cf. E.3.2). The following are equivalent:

```
// DartNNBD
class Box<T extends ?Object> {...}
class Box<T> {...}                 // Implicit upper bound of ?Object.
```

C.4 Semantics

While the static and dynamic semantics of generics follow from those of DartC and the semantics of DartNNBD introduced in the previous parts, there are quite a few alternative ways of dealing with certain aspects of generics. These are presented in the next section.

C.5 Discussion

C.5.1 Loss of expressivity due to union type interoperability, an alternative

One caveat of "future proofing" the nullable type operator $?T$, so that its semantics are compatible with the union type T | `Null` (B.3.1), is that we lose the ability to statically constrain a generic type parameter to be nullable but *not* `Null`—we discuss *why* we might want to do this in C.5.3. We lose this ability because $?T$ is not a type *constructor*, which would yield a unique (tagged) type, but rather just a type *operator* mapping T to the equivalent of the (untagged) union type T | `Null`. Thus, e.g., no distinction is made between `Null` and `?Null`.

We could alternatively define $?T$ as a type constructor (as if it were introducing a new type like `_$Nullable<`T`>`), orthogonal to union types, but there seems to be little to justify this complexity—future interoperability with union types seems more important and would be much more supportive of G0, usability and G0, ease migration.

C.5.2 Lower bounds to distinguish nullable/maybe-nullable parameters

The Checker Framework supports case G1.2 (nullable type parameter) distinctly from G1.3 (maybe-nullable type parameter) by allowing a type parameter lower bound to be defined (Checker Framework Manual, 23.1.2) in addition to an upper bound (via `extends`). This is a natural fit for Java since the language already has some support for lower bounds through lower bounded wildcards.

Without introducing general support for lower bounds, such an approach could be adopted for DartNNBD as well. In our notation, it would look like this: `class Box<?T extends ?Object>`, which would require an argument U to satisfy `?T <:` U `<: ?Object`, which is only possible if U is nullable.

C.5.3 Statically constraining a type parameter to be nullable but *not* `Null`

Consider the following code:

```
// DartNNBD
class C<T extends ?Object> { List<!T> list; ... }
var c = new C<Null>();
```

In the current form of the proposal, when a type parameter `T` is instantiated with `Null` then `!T` is considered malformed (B.3.2), as is the case for the type of `c.list` from the code sample above. Ideally, we would like to statically constrain `T` so that it cannot be `Null`. This would inform the clients of such a generic class that `T` should not be instantiated with `Null` and if it is, then a static warning could be reported at the earliest point possible, i.e., instantiation expressions like `new C<Null>()`.

It is possible to statically avoid malformed types that arise from such `!T` type expressions. One way is to adopt a completely different semantics for $?T$ as was presented in C.5.1. Another approach is to make use of type parameter lower bounds using syntax similar to what was presented in C.5.2: e.g., `class Box<!T extends ?Object>` would constrain an argument U to satisfy `T <:` U `<: ?Object`. The absence of an explicit lower-bound qualifier would be interpreted as `!`.

C.5.4 Parametric nullity abstraction, an alternative approach to generics

There are a few alternatives to the proposal of C.3 for handling generics. We mention only one here. It consists of broadening the scope of the NNBD rule to encompass type parameter occurrences inside the body of a generic class; i.e., an *undecorated* occurrence of a type parameter would *always* represent a non-null type. Such an alternative is best introduced by an example covering cases G1.2 and G1.3:

```
// DartNNBD
class Box<&T extends ?Object> {
  final T _default;      // non-null (G2.1)
  &T value;              // nullity matching parameter (G2.3)
  ?T maybeNull() => ...; // nullable (G2.2)
  ...
}
```

One can think of the type parameter decorator `&` as a symbol acting as a "formal parameter" for the nullity of the corresponding type argument—i.e., as a form of *parametric nullity abstraction*—which will be instantiated as either `?` or `!`. (This is similar in spirit to the Checker Framework qualifier parameters.) Thus, `Box` could be instantiated as `Box<?int>` or `Box<int>`, with `&` denoting `?` and (an implicit) `!`, respectively.

Case G1.1, for a non-null type parameter, could be written as `class Box<&T extends Object> {...}` or more simply as `class Box<T extends Object> {...}`.

The **main advantage** of this approach is that it upholds *nullity notational consistency* (NNC). That is, just like for class names,

- An *undecorated* type parameter name T represents a non-null type (G2.1),
- $?T$ is its nullable variant (G2.2), and
- $\&T$ matches the nullity of the corresponding type argument (G2.3).

The **main disadvantage** of this alternative is that it introduces a new concept (parametric nullity abstraction) which increases the complexity of the language, impacting G0, usability as well as G0, ease migration. Code migration effort is especially impacted because, in practice, case G2.3 is most frequent; hence, in porting DartC code to DartNNBD, most type parameter uses would need to be annotated with `&` vs. no annotation for our main alternative (C.3).

C.5.5 Generics and nullity in other languages or frameworks

(a) Default type parameter upper bound

As we have done here, the Nullness Checker of the Checker Framework has `@Nullable Object` as the implicit upper bound for type parameters, following its general CLIMB-to-top principle (which is further discussed in E.3.2). Ceylon's implicit type parameter upper bound is `Anything`, i.e., `Object | Null`, which is also nullable.

(b) Nullity polymorphism

Because Java generics are invariant, the Checker Framework Nullness Checker originally resorted to defining a special annotation to handle some common cases of polymorphism in type parameter nullities. E.g.,

```
@PolyNull T m(@PolyNull Object o) { ... }
```

The above constrains the return type of `m` to have a nullity that matches that of `o`. Since February 2015, a new form of polymorphism was introduced into the Checker Framework, namely the qualifier parameters mentioned in C.5.3.

(c) Ceylon cannot represent G2.1

It is interesting to note that case G2.1 cannot be represented in Ceylon due to the absence of a non-null type operator `!`:

```
// Ceylon
class Box<T> {
  final T _default;      // can't enforce non-null; fall back to nullity matching param.
  T value;               // nullity matching parameter (G2.3)
  ?T maybeNull() => ...; // nullable (G2.2)
  ...
}
```

Part D: Dealing with `dynamic` and missing static type annotations

D.1 Type `dynamic` in DartC

In DartC, `dynamic`

- "denotes the *unknown type*" (DSS 19.6, "Type dynamic"), and
- is a supertype of all types (DSS 19.7, "Type Void").

The type `dynamic` is used/assumed when, e.g.:

- A type is malformed (DSS 19.1, "Static Types").
- No static type annotation is provided, or type arguments are missing (DSS 19.6, "Type dynamic").
- An incorrect number of type arguments are provided for a generic class (DSS 19.8, "Parameterized Types").

D.2 Feature details: dynamic

The DartC role and static and dynamic semantics of `dynamic` are preserved in DartNNBD.

D.2.1 `!dynamic` is the unknown non-null type, and `?dynamic` is `dynamic`

The authors of Ceylon suggest that its `Anything` type can be interpreted as a union of all possible types. Such an interpretation leads to a natural understanding of the meaning of `dynamic` possibly decorated with the type operators `?` and `!`:

- `dynamic`, *the* unknown type, can be interpreted as the union of all types, and hence the supertype of all types.
- `!dynamic` can be interpreted as the union of all *non-null* types, and hence a supertype of all non-null types.
- `?dynamic` = `dynamic` | `Null` = `dynamic`.

Thus, `T << !dynamic` precisely when `T << Object` (A.1.4). It follows that `T <: !dynamic` for any class type T other than `Null` and `_Anything`.

> Comment. From another perspective, we can say that `!dynamic` represents an unknown non-null type rooted at `Object`, and `?dynamic` represents an unknown type rooted at `_Anything`.

D.2.2 Defining `!dynamic <:` S

Let T and S be normalized types (E.1.2). We introduce, $\bot_{\mathsf{Object}}$ to represent the bottom element of the non-null type subhierarchy and add the following as one of the conditions to be met for $T << S$ to hold (A.1.4):

> T is $\bot_{\mathsf{Object}}$ and $S << \mathsf{Object}$.

We refine `<:` in the following backwards compatible manner: $T <: S$ iff

> $[\bot/\mathbf{dynamic}]U << S$ where $U = [\bot_{\mathsf{Object}}/!\mathbf{dynamic}]T$.

See D.3.3 for a discussion and alternative.

D.3 Discussion

D.3.1 Clarification of the semantics of `T extends !dynamic`

As a point of clarification, we note that a generic class declared with a type parameter `T extends !dynamic`:

- is equivalent to `T extends Object`, except that;
- for the purpose of static checking, T is treated as an unknown type.

This is semantically consistent with the manner in which `T extends dynamic` is treated in DartC.

D.3.2 Semantics for `dynamic`, an alternative

The main alternative relevant to this part, consists of interpreting an undecorated occurrence of `dynamic` as `!dynamic`. This would broaden the scope of the NNBD rule to encompass `dynamic`.

This corresponds to the choice made in the Kotlin language which has types `Any` and `Any?` as representative of "any non-null type", and "any type", respectively. Notice how the unadorned type `Any` is non-null.

The main disadvantage of this alternative is that static warnings could be reported for programs without any static type annotations—such as for the statement `var o = null`, because the static type of `o` would be `!dynamic`. This goes contrary to G0, optional types.

D.3.3 Defining `!dynamic <:` S, an alternative

The DartC definition of the subtype relation (A.1.4) states that S `<:` T iff

> $[\bot/\mathbf{dynamic}]S << T$.

Replacing `dynamic` by $\bot$ ensures that expressions having the static type `dynamic` can "be assigned everywhere without complaint by the static checker" (DSS 16.2, "Null"), and that `dynamic` is a valid type argument for any type parameter.

The refined definitions of `<<` and `<:` given in D.2.2 allows `!dynamic` to be:

- Assigned everywhere a non-`Null` type is expected without complaint by the static checker, and;

- Used as a valid type argument for any non-`Null` type parameter.

Introducing a new bottom element for the `Object` subhierarchy most accurately captures our needs thought it renders the semantics more complex, decreasing G0, usability and increasing tool reengineering costs.

An alternative, allowing us to avoid this extra complexity, is to treat `!dynamic` simply as $\bot$. What we lose, are static warnings and/or dynamic type errors when: an expression of the static type `!dynamic` is assigned to variable declared as `Null` and, when `!dynamic` is used as a type argument for a `Null` type parameter. But such uses of `Null` are likely to be rare.

Part E: Miscellaneous, syntactic sugar and other conveniences

E.1 Feature details: miscellaneous

In this section we cover some features, and present features summaries, that require concepts from all of the previous parts.

E.1.1 Optional parameters are nullable-by-default in function bodies only

Dart supports positional and named optional parameters, as illustrated here:

```
int f([int i = 0]) => i; // i is an optional positional parameter
int g({int j : 0}) => j; // j is an optional named parameter
```

Within a function's body, its optional parameters are naturally nullable, since they are initialized to `null` when no default value is provided and corresponding optional arguments are omitted at a point of call. I.e., `null` is used as a default mechanism by which missing optional arguments can be *detected*.

We adopt a *dual view* for the types of optional parameters as is explained next. Suppose that an optional parameter `p` is declared to be of the normalized type T (E.1.2):

(a) **Within the scope of the function's body**, `p` will have static type:

- T if `p`:
  - is *explicitly* declared non-null—i.e., T is $!U$ for some U;
  - has no meta type annotation, and has a non-null default value (see E.1.1.1);
  - is a field parameter (see E.1.1.2).
- $?T$ otherwise. (Note that if T has type arguments, then the interpretation of the nullity of these type arguments is not affected.)

(b) **In any other context**, the type of `p` is T.

This helps enforce the following **guideline**: from a caller's perspective, an optional parameter can either be *omitted*, or given a value matching its declared type.

Comments:

- E.g., one can invoke `f`, defined above, as either `f()` or `f(1)`, but `f(null)` would result in a static warning and dynamic type error.

- Just like for any other declaration, an optional parameter can be marked as nullable. So `f([?int j])` would permit `f(null)` without warnings or errors.
- Explicitly marking an optional parameter as non-null, e.g., `int h([!int i = 0]) => i`, makes it non-null in both views. But, if a non-null default value is not provided, then a static warning and dynamic type error will be reported.
- *T*, the type of `p`, might implicitly be `dynamic` if no static type annotation is given (D.2). By the rules above, `p` has type `?dynamic`, i.e., `dynamic` (D.2.1), in the context of the declaring function's body. Hence, a caveat is that we cannot declare `p` to have type `dynamic` in the function body scope and type `!dynamic` otherwise.
- The dual view presented here is an example of an application of G0, utility. This is further discussed, and an alternative is presented, in E.3.3.
- Also see E.3.4 for a discussion of function subtype tests.

E.1.1.1 Optional parameters with non-null initializers are non-null

In Dart, the initializer of an optional parameter must be a compile time constant (DSS 9.2.2). Thus, in support of G0, ease migration, an optional parameter with a non-null default value is considered non-null.

E.1.1.2 Default field parameters are single view

Dart field constructor parameters can also be optional, e.g.:

```
class C {
  num n;
  C([this.n]);
  C.fromInt([int this.n]);
}
```

While `this.n` may have a type annotation (as is illustrated for the named constructor `C.fromInt()`), the notion of dual view does not apply to optional field parameters since they do not introduce a new variable into the constructor body scope.

E.1.2 Normalization of type expressions

A *normalized* type expression has no *superfluous* applications of a type operator (B.3.1, B.3.2).

Let P be a type parameter name and N a non-null class type, N `<: Object`. In all contexts where NNBD applies (E.3.1), the following type expressions, used as static type annotations or type arguments, are in *normal form*:

- N, and $?N$
- P, $?P$, and $!P$
- `dynamic` and `!dynamic`
- `Null`

In the context of an optional function parameter `p` as viewed from within the scope of the declaring function body (E.1.1(a)), the following is also a normal form (in addition to the cases listed above): $!N$.

> Comment. Excluded are `void`, to which type operators cannot be applied (B.3.1, B.3.2), `?dynamic`, `?Null` and various repeated and/or canceling applications of `?` and `!` (B.3).

E.2 Feature details: syntactic sugar and other conveniences

We define various syntactic sugars and other syntactic conveniences in this section. Being conveniences, they are **not essential to the proposal** and their eventual adoption may be subject to an "applicability survey", in particular through analysis of existing code.

E.2.1 Non-null `var`

While `var x` introduces `x` with static type `dynamic`, we propose that `var !x` be a shorthand for `!dynamic x`. Note that this shorthand is applicable to all kinds of variable declaration as well as function parameters.

E.2.2 Formal type parameters

In C.3.4 we defined the default type parameter upper bound as `?Object`; i.e., `class Box<T>` is equivalent to `class Box<T extends ?Object>`. We define `class Box<T!>` as a shorthand for `class Box<T extends Object>`. Note that `!` is used as a *suffix* to `T`; though it is a meta type annotation *prefix* to the implicit `Object` type upper bound.

> Comment. We avoid suggesting `class Box<!T>` as a sugar because it opens the door to `class Box<?T>` and `class Box<?T extends Object>`. The latter is obviously be an error, and for novices the former might lead to confusion about the meaning of an undecorated type parameter `class Box<T>` (which could quite reasonably arise if there is a lack of understanding of the scope of the NNBD rule). Also, `class Box<!T>` would conflict with the use of the same notation for the purpose of excluding `Null` type arguments (C.5.3).

E.2.3 Non-null type arguments

We define `!` as a shorthand for `!dynamic` when used as a type argument as in

```
List listOfNullableAny = ...
List<!> listOfNonnullAny = ...
```

E.2.4 Non-null type cast

The following extension of type casts (DSS 16.34, "Type Cast") allows an expression to be projected into its non-null type variant, if it exists. Let e have the static type T, then e `as! Null` has static type $!T$.

> Comments:
>
> - If T is outside the domain of `!`, then $!T$ is malformed (B.3.2).
> - Syntactic ambiguity, between `as!` and a cast to a non-null type $!T$, is addressed as it was for type tests (B.2.4).
> - In the presence of union types, `as!` might be generalized as follows. If the static type of e is the (normalized) union type $U \mid T$, then the static type of e `as!` U could be defined as T.

E.3 Discussion

E.3.1 Scope of NNBD in DartNNBD

We clarify here the scope of NNBD as defined in this proposal. This will be contrasted with the scope of NNBD in other languages or frameworks in (E.3.2).

(a) The NNBD rule states that for *all* class types T `<: Object`, it is false that `Null` can be *assigned to* T (A.1.4). This includes class types introduced via function signatures in the context of a

- Formal parameter declaration—these are anonymous class types (B.2.6).
- `typedef`—these are named, possibly generic, class types (DSS 19.3, "Type Declarations").

Thus T, unadorned with any type operator, (strictly) represents instances of T (excluding `null`).

(b) The NNBD rule applies to **class types** *only*. In particular, it does **not** apply to:

- Type parameters (Part C).
- Implicit or explicit occurrences of `dynamic` (D.2).

(c) The NNBD rule applies in *all* contexts where a class type is *explicitly* given, *except one*: static type annotations of optional function parameters as viewed from within the scope of the declaring function's body (E.1.1).

E.3.2 Scope of NNBD in other languages or frameworks

In contrast to this proposal, the scope of the NNBD rule in other languages or frameworks often has more exceptions. This is the case for Spec# (Fahndrich and Leino, 2003), JML (Chalin et al., 2008) and Java enhanced with nullity annotations from the Checker Framework. Next, we compare and contrast DartNNBD with the latter, partly with the purpose of justifying the language design decisions made in this proposal, and implicitly for the purpose of presenting potential alternatives for DartNNBD.

The Java Checker Framework has a principle named CLIMB-to-top which, in the case of the Nullness Checker, means that types are interpreted as *nullable-by-default* in the following contexts:

- Casts,
- Locals,
- Instanceof, and
- iMplicit (type parameter) Bounds

(CLIMB). We adhere to this principle for *implicit* type parameter bounds (C.3.4) and discuss other cases next.

(a) Local variables

When retrofitting a strongly (mandatorily) typed nullable-by-default language (like Java) with NNBD it is common to relax NNBD for local variables since standard flow analysis can determine if a local variable is potentially `null` or not, and to do otherwise would result in the need to annotate many local variables as nullable. Unfortunately, excluding local variables from the scope of NNBD is at the cost of loss of a form of *referential transparency*: consider the following declaration

```
List<String> guestList;
```

Is `guestList` nullable? In the Checker Framework, it is not possible to tell without knowing the context: `guestList` is NNBD if this is a (package) field declaration, but nullable if it is a local variable.

In contrast, static type annotations are optional in Dart, and a common idiom is to omit them for local variables. This idiom is in fact prescribed in the Dart Style Guide section on type annotations:

> PREFER using `var` without a type annotation for local variables.

In light of this idiom, if a developer goes out of his or her way to write an explicit static type annotation, then we believe that the type should be interpreted literally; it is for this reason that we have chosen to include local variable declarations in the scope of NNBD (B.3.4, B.4.2(a)). As a benefit, we retain referential transparency for all (non-optional) variable declaration kinds—in particular instance variables and local variables.

As applied to local variables, the NNBD rule of this proposal may result in extra warnings when DartC code is migrated to DartNNBD, but such warnings will *not* prevent the code from being executed in production mode—in strongly typed languages like Java, such migrated code would simply *not run*, and so our approach would not be a realistic alternative. Also, in the case of Dart code migration, tooling can contribute to the elimination of such warnings by automatically annotating explicitly typed local variables determined to be nullable (G0, ease migration). The strategy proposed in E.3.6 can also help reduce warnings.

(b) Type tests

In DartC, the type test expression e `is` T holds only if the result of evaluating e is a value v that is an instance of T (DSS 16.33, "Type Test"). Hence, in DartNNBD, this naturally excludes `null` for all T `<: Object`.

(c) Type casts

Out of the 150K physical Source Lines Of Code (SLOC) of the Dart SDK libraries, there are only 30 or so occurrences of the `as` operator and most clearly assume that their first operand is non-null. Based on such a usage profile, and for reasons similar to those given for local variables (i.e., explicitly declared types interpreted literally), we have chosen to include Dart type casts in the scope of the NNBD rule.

Broad applicability of NNBD rule for DartNNBD

While balancing all G0 language design goals, we have chosen to make the NNBD rule as broadly applicable as possible, thus making the language simpler and hence increasing G0, usability.

E.3.3 Optional parameters are always nullable-by-default, an alternative

The "dual view" semantics proposed above (E.1.1) for optional parameters is an example of a language design feature which is slightly more complex (and hence penalizes G0, usability) but which we believe offers more utility (G0, utility). A simpler alternative is to adopt (a) as the sole view: i.e., optional parameters would be nullable-by-default in all contexts.

E.3.4 Subtype relation over function types unaffected by nullity

In contexts were a function's type might be used to determine if it is a subtype of another type, then optional parameters are treated as NNBD (view E.1.1(b)). But as we explain next, whether optional parameter semantics are based on a "dual" (E.1.1) or "single" (E.3.3) view, this will have no impact on subtype tests.

Subtype tests of function types (DSS 19.5 "Function Types") are structural, in that they depend on the types of parameters and return types (DSS 6, "Overview"). Nullity type operators have no bearing on function subtype tests. This is because the subtype relation over function types is defined in terms of the "assign to" ($\Longleftrightarrow$) relation over the parameter and/or return types. The "assign to" relation (A.1.4), in turn, is unaffected by the nullity: if types S and T differ only in that one is an application of `?` over the other, then either S `<:` T or T `<:` S and hence $S \Longleftrightarrow T$. Similar arguments can be made for `!`.

E.3.5 Catch target types and meta type annotations

The following illustrates a try-catch statement:

```
class C<T> {}
main() {
  try {
    ...
  } on C<?num> catch (e) {
    ...
  }
}
```

Given that `null` cannot be thrown (DSS 16.9), it is meaningless to have a catch target type qualified with `?`; a static warning results if `?` is used in this way. Any such qualification is ignored at runtime. Note that because meta type annotations are reified (C.4), they can be meaningfully applied to catch target type arguments as is illustrated above.

E.3.6 Reducing the annotation burden for local variables, an alternative

This section expands on B.3.4.c.2. We propose as an alternative that standard read-before-write analysis be used for non-null *local variables* without an explicit initializer, to determine if its default initial value of `null` has the potential of being read before the variable is initialized.

Consider the following illustration of a common coding idiom:

```
int v; // local variable left uninitialized
if (...) {
  // possibly nested conditionals, each initializing v
} else {
  // possibly nested conditionals, each initializing v
}
// v is initialized to non-null by this point
```

Without the feature described in this subsection, `v` would need to be declared nullable.

E.3.7 Dart Style Guide on `Object` vs. `dynamic`

The Dart Style Guide recommends DO annotate with `Object` instead of `dynamic` to indicate any object is accepted. Of course, this will need to be adapted to recommend use of `?Object` instead.

Part F: Impact on Dart SDK libraries

The purpose of this part is to illustrate what some of the Dart SDK libraries might look like in DartNNBD and, in some cases, how they might be adapted to be more useful, through stricter type signatures or other enhancements.

F.1 Examples

The examples presented in this section are of types migrated to DartNNBD that *only* require updates through the addition of meta type annotations. Types potentially requiring behavioral changes are addressed in F.2.

F.1.1 `int.dart`

We present here the `int` class with nullity annotations. There are only 3 nullable meta type annotations out of 44 places were such annotations could be placed ($3/44 = 7\%$ are nullable).

```
// DartNNBD - part of dart.core;
abstract class int extends num {
  external const factory ?int.fromEnvironment(String name, {int defaultValue});
  int operator &(int other);
  int operator |(int other);
  int operator ^(int other);
  int operator ~();
  int operator <<(int shiftAmount);
  int operator >>(int shiftAmount);
  int modPow(int exponent, int modulus);
  bool get isEven;
  bool get isOdd;
  int get bitLength;
  int toUnsigned(int width);
  int toSigned(int width);
  int operator -();
  int abs();
  int get sign;
  int round();
  int floor();
  int ceil();
  int truncate();
  double roundToDouble();
  double floorToDouble();
  double ceilToDouble();
```

```
  double truncateToDouble();
  String toString();
  String toRadixString(int radix);
  external static ?int parse(String source,
                             {int radix /* = 10 */,
                              ?int onError(String source) });
}
```

With the eventual added support for generic functions, `parse()` could more usefully redeclared as:

```
  external static I parse<I extends ?int>(..., {..., I onError(String source)});
```

Notes:

- The `source` argument of `parse()` should be non-null, see dart/runtime/lib/integers_patch.dart#L48.
- In conformance to the guideline of E.1.1, the following optional parameters are left as NNBD:
  - `defaultValue` of `factory int.fromEnvironment()`.
  - `radix` and `onError` of `parse()`. Since `radix` has a non-null default value, it could be declared as `!int`, though there is little value in doing so given that `parse()` is `external`.

  (In opposition to the guideline, if we declare `defaultValue` and `onError` as nullable, that would make for 5/44 = 11% of declarators with nullable annotations.)

We have noted that conforming to the guideline for optional parameters of E.1.1 may result in breaking changes for some functions of SDK types. Other SDK type members explicitly document their adherence to the guideline: e.g., the `List([int length])` constructor.

F.1.2 Iterable

The `Iterable<E>` type requires no `?` annotations (thought the optional `separator` parameter of `join()` could be declared as `!String`).

```
// DartNNBD - part of dart.core;
abstract class Iterable<E> {
  const Iterable();
  factory Iterable.generate(int count, [E generator(int index)]);
  Iterator<E> get iterator;
  Iterable map(f(E element));
  Iterable<E> where(bool f(E element));
  Iterable expand(Iterable f(E element));
  bool contains(Object element);
  void forEach(void f(E element));
  E reduce(E combine(E value, E element));
  dynamic fold(var initialValue,
               dynamic combine(var previousValue, E element));
  bool every(bool f(E element));
  String join([String separator = ""]);
  bool any(bool f(E element));
  List<E> toList({ bool growable: true });
  Set<E> toSet();
```

```
  int get length;
  bool get isEmpty;
  bool get isNotEmpty;
  Iterable<E> take(int n);
  Iterable<E> takeWhile(bool test(E value));
  Iterable<E> skip(int n);
  Iterable<E> skipWhile(bool test(E value));
  E get first;
  E get last;
  E get single;
  E firstWhere(bool test(E element), { E orElse() });
  E lastWhere(bool test(E element), {E orElse()});
  E singleWhere(bool test(E element));
  E elementAt(int index);
  String toString();
}
```

F.1.3 `Future<T>`

We mention in passing that the use of `Future<Null>` remains a valid idiom in DartNNBD since the generic class is declared as:

```
abstract class Future<T> {...}
```

Hence `T` is nullable (C.3.4).

F.2 Suggested library improvements

F.2.1 Iterator

DartC

An `Iterator<E>` is "an interface for getting items, one at a time, from an object" via the following API:

```
// DartC - part of dart.core;
abstract class Iterator<E> {
  bool moveNext();
  E get current;
}
```

Here is an example of typical use (excerpt from the API documentation):

```
var it = obj.iterator;
while (it.moveNext()) {
  use(it.current);
}
```

Dart's API documentation for `current` is nonstandard in that it specifies that `current` shall be `null` "*if the iterator has not yet been moved to the first element, or if the iterator has been moved past the last element*". This has the unfortunate consequence of forcing the return type of `current` to be nullable, even if the element type `E` is non-null. Iterators in other languages (such as Java and .Net languages) either raise an exception or document the behavior of `current` as *undefined* under such circumstances—for the latter see, e.g., the .Net IEnumerator.Current Property API.

DartNNBD

We suggest that that Dart Iterator API documentation be updated to state that the behavior of `current` is unspecified when the last call to `moveNext()` returned false (implicit in this statement is that `moveNext()` *must* be called at least once before `current` is used). This would allow us to usefully preserve the interface definition of `Iterator<E>` as:

```
// DartNNBD - part of dart.core;
abstract class Iterator<E> {
  bool moveNext();
  E get current;
}
```

Note that the type and nullity of `current` matches that of the type parameter.

Independent of nullity, the behavior of `current` might be adapted so that it throws an exception if it is invoked in situations where its behavior is undefined. But this would be a potentially breaking change (which, thankfully, would not impact uses of iterators in `for-in` loops).

F.2.2 `List<E>`

We comment on two members of the `List<E>` type.

`factory List<E>([int length])`

In DartNNBD, a dynamic type error will be raised if `length` is positive and `E` is non-null. The error message could suggest using `List<E>.filled(int length, E fill)` instead.

`List<E>.length=`

The `List<E>.length=` setter changes the length of a list. If the new length is greater than the current length, then new entries are initialized to `null`. This will cause a dynamic type error to be issued when `E` is non-null.

Alternatives to growing a list of non-null elements includes:

- Define a mechanism by which an "filler field" could be associated with a list. The filler field could then be used by the length setter when growing a list of non-null elements. E.g.,
  - Add a `List<E>.setFiller(E filler)` method, or;
  - Reuse the filler provided, say, as argument to `List<E>.filled(int length, E fill)`.
- Add a new mutator, `setLength(int newLength, E filler)`.

F.3 Other classes

Object

The `Object` class requires no textual modifications:

```
class Object {
  const Object();
  bool operator ==(other) => identical(this, other);
  external int get hashCode;
  external String toString();
  external dynamic noSuchMethod(Invocation invocation);
  external Type get runtimeType;
}
```

Part G: Migration strategy (sketch)

Comment. An effective migration plan depends on several factors including, for example, whether union types will soon added to Dart or not. Regardless, this part sketches some initial ideas.

G.1 Precedent

As is mentioned in the survey (I.3), both commercial and research languages have successfully migrated from a nullable-by-default to a NNBD semantics. To our knowledge, Eiffel, in 2005, was the first commercial language to have successfully made this transition. JML, a Java BISL, made the transition a few years later (Chalin et al., 2008). More recently, the Eclipse JDT has been allowing developers to enable NNBD at various levels of granularity, including at the level of an entire project or workspace, and work is underway to provide nullity annotations for the types in the SDK.

G.2 Migration aids

It is interesting to note that Eiffel introduced the `!` meta type annotation solely for purpose of code migration. DartNNBD also has `!` at its disposal, though in our case it is a core feature.

We propose (as has been done in JML and the Eclipse JDT) that the following lexically scoped, non-inherited library, part and class level annotations be made available: `@nullable_by_default` and `@non_null_by_default`. Such annotations establish the default nullity in the scope of the entity thus annotated.

Within the scope of an `@nullable_by_default` annotation, every type name T is taken as implicitly $?T$ except for the following: a type name that names

- a constructor in a constructor declaration
- a type target to a catch clause
- the argument type of a type test (`is` expression)

Despite the exclusions above, if any such type name has type arguments then the nullable-by-default rule applies to the type arguments.

G.3 Impact

Tool impacted include (some common subsystems overlap):

- Dart Analyzer.

- Dart Dev Compiler.
- Dart VM.
- dart2js.
- Dart Code Formatter.
- Dart docgen.

G.4 Migration steps

It seems desirable to target Dart 2.0 as a first release under which NNBD would be the *default* semantics. In Dart 2.0, a command line option could be provided to recover nullable-by-default semantics. Initial steps in preparation of this switch would be accomplished in stages in the remaining releases of the 1.x stream.

Here is a preliminary list of possible steps along this migration path, not necessarily in this order:

- (SDK) Create `@nullable_by_default` and `@non_null_by_default` annotations.
- (Tooling) Add support for:
  - Meta type annotation *syntax* (excluding most sugars).
  - Static checks. This includes processing of `@*_by_default` annotations.
  - Runtime support (B.3.3) for nullity type operators, and dynamic checks.
- (SDK) Re-root the class hierarchy (B.2.1).
- (Tooling) Global option to turn on NNBD.
- …

G.5 Migration plan details

Comment. TODO.

Appendix I. Nullity in programming languages, an abridged survey

Problems arising from the presence of `null` have been well articulated over the years. Programming language designers have proposed various approaches to dealing with these problems, including the elimination of `null` entirely. For a survey of the alternate strategies to `null`, and the use of *nullity annotations* and *non-null types* in programming languages circa 2008 see *Chalin et al., 2008*, Section 4 (IEEE, preprint). Below we summarize the survey and include recent developments.

In `null`-enabled languages, `null` is conveniently used to represent a value of any type T, when there is no T value at hand. This, in particular, allows for simple initialization rules: any variable not explicitly initialized can be set to `null`.

I.1 Languages without null

One way to avoid problems with `null` is to avoid making it part of the language. To address the main use case of `null` as *a substitute for a value of type* T *when you don't have a value of type* T many languages without `null` resort to use of option types. This is the case of:

- Most functional programming languages, like ML and Haskell, as well as
- Some object-oriented languages, like CLU, OCaml and, as was mentioned in the introduction, Apple's recently released Swift language.

I.2 Strategies for dealing with null in null-enabled languages

Most imperative programming languages having reference types also support `null`. This is certainly true for mainstream languages of C descent. Various strategies for dealing with `null` and attempting to detect NPEs are detailed next.

- **Tools**, such as *linters*, have been used to perform nullity analysis (among other checks) in the hope of detecting potential NPEs. A notable mention is Splint, which actually assumes that unannotated reference types are non-null.
- **Special macros/annotations** allowing developers to mark *declarators* as non-null, can guide:
  - *Runtime instrumentation* of code so as to eagerly perform runtime checks—e.g., argument checks on function entry rather than throwing an NPE at some later point in the call chain. There was early support for such macros/annotation in, e.g., GNU's gcc and Microsoft's Source-code Annotation Language (SAL) for C/C++ (Chalin et al., 2008).

  - *Static checking.* E.g., Findbugs which makes use of Java 5 metadata annotations such as `@NonNull` and `@CheckForNull`. Modern IDEs like the Eclipse JDT and IntelliJ have been systematically improving their static NPE detection capabilities using such annotations as well, as we have discussed in A.3.1.

- **Language subsets** (sometimes qualified as "safe" subsets) have been defined so as to allow tool analysis to be more effective at flagging potential NPEs while reducing false positives. Pure subsets are rare. Most often they are combined with some form of extension. Examples include:
  - Spark Ada, which eliminated `null` by eliminating references!
  - Cyclone, a safe dialect of C, which introduces the concept of safe/never-`null` pointers. The design of Rust, which is still in beta (2015Q2), was influenced by Cyclone; it also distinguishes safe from raw pointers.

- **Language extensions** and **language evolution**, which are the topic of the next section.

I.3 Retrofitting a null-enabled language with support for non-null

Retrofitting a fielded language is a challenge due to the presence of legacy code. We have seen two main approaches to tackling this challenge: language extensions and language evolution.

I.3.1 Language extensions

Language extensions are, as the name implies, defined atop (and outside of) a given base language. This means that the base language remains unaffected, and hence extensions have no impact on (standard) language tooling. This also implies that code elements from extensions are often encoded in specially marked comments (e.g., `/*@non_null*/`) or metadata.

(a) Contracts

One example, is the still very active Microsoft Code Contracts project, which provides a language-agnostic (i.e., library-based) way to express contracts (preconditions, postconditions, and object invariants) in programs written in most of the .Net family of languages. Contracts can be used to constrain fields, parameters and function results to be non-null, as is illustrated by the following excerpt of the NonNullStack.cs example taken from Fahndrich and Logozzo, 2010:

```
public class NonNullStack<T> where T : class {
  protected T[] arr;
  private int nextFree;

  [ContractInvariantMethod]
  void ObjectInvariant() {
    Contract.Invariant(arr !=null);
    Contract.Invariant(Contract.ForAll(0, nextFree, i => arr[i] != null));
    ...
  }

  public void Push(T x) {
    Contract.Requires(x != null);
    ...
  }
```

```
  public T Pop() {
    Contract.Requires(!this.IsEmpty);
    Contract.Ensures(Contract.Result<T>() != null);
    ...
  }
  ...
}
```

Notice the predicates constraining the following elements to be non-null:

- `arr` field, as well as elements of the `arr` array, in the object invariant,
- `x` parameter of `Push()` in the requires clause,
- return result of `Pop()` in the ensures clause.

Of course, a contract language like this can be used to do much more than assert that fields, variables and results are non-null.

(b) Non-null declarators

The same approach to nullity as Code Contracts was originally adopted in Java Modeling Language (**JML**), a Behavioral Interface Specification Language (**BISL**) for Java. But nullity constraints were so common that declarator annotations were defined as "syntactic sugar" for corresponding contract clauses. E.g., the `NonNullStack` example from above could have been written as:

```
public class NonNullStack<T> ... {
  protected /*@non_null*/ T /*@non_null*/ [] arr;
  public void Push(/*@non_null*/T x) { ... }
  public /*@non_null*/T Pop() { ... }
  ...
}
```

But such an approach actually arose in JML prior to the advent of Java generics. Nullity declarator constraints cannot be used to qualify type parameters, a feature often requested by developers.

Other languages supporting nullity declarators include the Larch family of BISLs, notably Larch/C (LCL) and Larch/C++. The Splint linter mentioned above (I.2) grew out of Larch/C work.

(c) Non-null types

Evolution of language extensions has sometimes been from nullity annotations applied to *declarators* to support for *non-null types*. This has been the case for JML (Chalin et al., 2008). In fact, the need to fully support non-null types in Java led to the creation of JSR-308, "Java Type Annotations" which extends support for annotations to all places type expressions can appear. There has been hints that a similar extension might be considered useful for Dart as well (B.4.6). JSR-308 was included in the March 2014 release of Java 8.

Other language extensions supporting non-null types include:

- Eclipse JDT. The progression, from support of nullity declarator annotations to non-null types, in the context of the Eclipse JDT, was discussed in A.3.1. Not only does the most recent release of the Eclipse JDT support non-null types, but it also allows developers to enable NNBD at various levels of granularity, including the project and workspace levels.

- Nullness Checker of the Java Checker Framework.
- JastAdd, a "meta-compilation system that supports Reference Attribute Grammars" and one of its instances, the JastAddJ compiler for Java, has an extension supporting non-null types.
- Spec# a BISL for C#.

All of these language extensions, except possibly JastAdd also support NNBD.

I.3.2 Language evolution

As was mentioned earlier, adopting non-null types and/or NNBD can be a challenge for languages with a sufficiently large deployed code base. Having a proper migration strategy is key.

To our knowledge, Eiffel is the first *commercial language* to have made the switch, in 2005, from non-null types with a nullable-by-default semantics to NNBD. Eiffel introduced the `!` meta type annotation solely for easing migration efforts.

Adopting non-null types can lead to unsoundness if one is not careful, particularly with respect to the initialization of fields declared non-null (B.4.2)—e.g., due to possible calls to helper methods from within constructors. Fahndrich and Leino, 2003, detail the challenges they faced in bringing non-null types to Spec#. Swift, in which types are non-null by default, adopts such a position for classes and structures which: "*must set all of their stored properties to an appropriate initial value by the time an instance of that class or structure is created. Stored properties cannot be left in an indeterminate state*".

Some feel that the cost of preserving type soundness is too high with respect to language usability. Eberhardt discusses the challenges in writing proper class/structure initialization code in Swift (Eberhardt, 2014). Similar initialization rules are also know to be one of the difficulties facing language designers attempting to add support for non-null types to C#. In fact, Lippert believes that it is completely impractical to retrofit C# with non-null types and instead proposes use of Code Contracts (Lippert, 2013).

We mention in passing that in 2005, .Net was extended with support for nullable *primitive* types. This was done to achieve (uniform) native support for data coming from datasources in which all types are nullable. But this is creating an extra challenge, in terms of notational consistency, for C# language designers who are considering the introduction of non-null types into C# 7, as is illustrated by the following sample declarations:

```
// C# + nullity proposal
int a; //non-nullable value type
int? a; //nullable value type
string! a; //non-nullable reference type
string a; //nullable reference type
```

I.4 Modern web/mobile languages with non-null types and NNBD

Fletch is an experimental runtime (VM) for Dart "that makes it possible to implement highly concurrent programs in the Dart". Meant to address problems in a similar space, is the Pony language (@0.1.5 2015Q2), a statically typed actor-based language (with concurrent garbage collection). Pony has non-null types and NNBD with nullable types introduced via a union with special unit type `None`.

For sake of completeness, we also reproduce here (from 2.2) the list of programming languages, many recently released, that are relevant to web applications (either dialects of JS or that compile to JS) and/or mobile, and that support *non-null types* and NNBD.

Language	About	v1.0?	Nullable via	Reference
Ceylon (Red Hat)	Compiles to JS, Java Bytecode (**JB**)	2013Q4	*T*?	Ceylon optional types
Fantom	Compiles to JS, JB, .Net CLR	2005	*T*?	Fantom nullable types
Flow (Facebook)	JS superset and static checker	2014Q4	*T*?	Flow maybe types
Kotlin (JetBrains)	Compiles to JS and JB	2011Q3	*T*?	Kotlin null safety
Haste	Haskell to JS compiler	@0.4.4	option type	Haskell maybe type
Swift (Apple)	iOS/OS X Objective-C successor	2014Q4	option type	Swift optional type

As was mentioned earlier, there is also discussion of introducing non-null types to TypeScript.

Appendix II. Tooling and preliminary experience report

II.1 Variant of proposal implemented in the Dart Analyzer

We describe here a *version* of this proposal as implemented in the Dart Analyzer. It is "a version" in that we have adopted all core ideas (8.1) but we have made a particular choice of alternatives (8.3). Choices have been driven by the need to create a solution that is **fully backwards compatible**, as will be explained below.

Core language design decisions (cf. 8.1) and main alternatives:

- A.2. Drop semantic rules giving special treatment to `null`.
- B.2. Ensure `Object` is non-null by making `Null` a root (B.4.7).
- B.2. Support meta type annotations
  - `@nullable` and `@non_null` (B.4.6), and in those places where DartC does not currently support metadata,
  - allow the use of specialized comments `/*?*/` and `/*!*/`.
- C.3. Support for generics matches the proposal.
- G.2. Support `library`, `part` and `class` level `@nullable_by_default` annotations.

By our choice of input syntax, DartNNBD annotated code can be **analyzed and executed in DartC** without any impact on DartC tooling.

II.2 Dart Analyzer

We describe here a realization of this proposal in the Dart Analyzer.

II.2.1 Design outline

The dart analyzer processes compilation units within a collection of libraries. The processing of each library is done in multiple phases on the Abstract Syntax Tree (AST) provided by the parser.

(a) AST

No structural changes have been done to the AST types since nullity annotations are represented by metadata and comments. Also, `NullityElements`, described next, are attached to `TypeNames` via the generic AST property mechanism (a hash map associated with any AST node).

(b) Element model

- We introduce two `DartType` subtypes, one for each of `?` and `!` meta type annotations, named `UnionWithNullType` and `NonNullType`, respectively. These represent normalized types (E.1.2).
- The model `Element` representing a `UnionWithNullType` is `UnionWithNullElement`. Its is a representation of a (synthetic) `ClassElement`-like entity that can be queried for members via methods like `lookUpMethod(methodName)`, etc. When queried for a member with a given name n, a (synthetic) multi-member is returned which represents the collection of members matching declarations of n in `Null` and/or the other type argument of `UnionWithNullType`.
- The dual-view for optional function parameters (E.1.1) is realized by associating to each optional parameter (`DefaultParameterElementImpl`) a synthetic counterpart (`DefaultParameterElementWithCalleeViewImpl`) representing the declared parameter from the function-body/callee view (E.1.1(a)). All identifier occurrences within the function body scope have the callee-view parameter instance as an associated static element.

(c) Resolution

We describe here the *added* / *adapted* analyzer processing (sub-)phases:

1. Nullity annotation processing:
   (a) Nullity annotation resolution (earlier than would normally be done since nullity annotations impact *types* in DartNNBD). Note that we currently match annotation names only, regardless of library of origin so as to facilitate experimentation.
   (b) `NullityElement`s (see (b) below) are computed in a top-down manner, and attached to the AST nodes that they decorate (e.g., `TypeName`, `LibraryDirective`, etc.). The final nullity of a type name depends on: global defaults (whether NNBD is enabled or not), `@nullable_by_default` nullity scope annotations, and individual declarator annotations.
2. Element resolution (via `ElementResolver` and `TypeResolverVisitor`) is enhanced to:
   (a) Adjust the static type associated with a, e.g., a `TypeName` based on its nullities.
   (b) Associate a callee view to each default parameter element and suitably adjust its type.
   (c) Handle problem reporting for operator and method (including getter and setter) invocation over nullable targets.
3. Error verification has been adapted to, e.g., check for invalid implicit initialization of variables with `null`. See B.3.4 for details.

The NNBD analyzer also builds upon existing DartC flow analysis and type propagation facilities.

> Caveat excerpt from a code comment: TODO(scheglov) type propagation for instance/top-level fields was disabled because it depends on the order or visiting. If both field and its client are in the same unit, and we visit the client before the field, then propagated type is not set yet.

II.2.2 Source code and change footprint

The NNBD-enabled analyzer sources are in the author's GitHub Dart SDK fork @chalin/sdk, dep30 branch, under `pkg/analyzer`. This SDK fork also contains updates to the SDK library and sample projects which have been subject to nullity analysis (as documented in II.3). Note that

- All code changes are marked with comments containing the token `DEP30` to facilitate identification (and merging of upstream changes from @dart-lang/sdk).

- Most significant code changes are marked with appropriate references to sections of this proposal for easier feature implementation tracking.

As of the time of writing, the Dart Analyzer code change footprint (presented as a git diff summary) is:

```
Showing  9 changed files  with 245 additions and 35 deletions.
+3   -2   pkg/analyzer/lib/src/generated/ast.dart
+5   -3   pkg/analyzer/lib/src/generated/constant.dart
+40  -5   pkg/analyzer/lib/src/generated/element.dart
+53  -9   pkg/analyzer/lib/src/generated/element_resolver.dart
+16  -0   pkg/analyzer/lib/src/generated/engine.dart
+10  -0   pkg/analyzer/lib/src/generated/error.dart
+20  -7   pkg/analyzer/lib/src/generated/error_verifier.dart
+94  -8   pkg/analyzer/lib/src/generated/resolver.dart
+4   -1   pkg/analyzer/lib/src/generated/static_type_analyzer.dart
```

There is approximately 1K Source Lines Of Code (SLOC) of new code (or 3K LOC including comments and whitespace).

II.2.3 Status

Please see the GitHub DEP #30 Analyzer project page.

II.3 Preliminary experience report

We stress from the outset that this is a **preliminary** report.

Our initial objective has been to test run the new analyzer on sample projects. Our first target has been the SDK library Dart sources. We have also used some sample projects found in the Dart SDK `pkg` directory. So far, results are encouraging in that the nullable annotation burden seems to be low as we quantify in detail below.

II.3.1 Nullity annotation density

Dietl, 2014, reports 20 nullity annotations / KLOC (anno/KLOC). So far, nullable annotation density for the SDK sources have been:

- <1 anno/KLOC for the library core (with <2 line/KLOC of general changes related to nullity);
- 1 anno/KLOC for the samples.

We attribute such a low annotation count to Dart's relaxed definition of assignability (see A.1.4 and B.3.5), and a judicious choice in the scope of NNBD (E.3.1), in particular for optional parameters—namely our dual-view approach and use of compile-time default values to influence the nullability (E.1.1).

We are not claiming that such a low annotation count will be typical (it certainly is not the case for the analyzer code itself, in part due to most AST fields being nullable), but results are encouraging.

II.3.2 Dart SDK library

Our strategy has been to run the NNBD analyzer over the SDK library and address any reported issues. In addition, we added the nullable annotations mentioned in Part F. Here is a summary, to date, of the changes.

- `sdk/lib/core/core.dart` updated to include the definition of nullity annotations `@nullable`, `@non_null`, etc. (19 lines).
- Nullable annotations were added in 70 locations. Most (64) were occurrences of `Object`.
- The remaining updates (10 lines) were necessary to overcome the limitations in the analyzer's flow analysis capabilities. For example, when an optional nullable parameter is initialized to a non-null value when it is null at the point of call. This is a typical code change of this nature:

```
*** 280,287 ****
    static void checkValidIndex(int index, var indexable,
!                               [String name, int length, String message]) {
!     if (length == null) length = indexable.length;
      // Comparing with `0` as receiver produces better dart2js type inference.
--- 280,287 ----
     * otherwise the length is found as `indexable.length`.
     */
    static void checkValidIndex(int index, var indexable,
!                     [String name, int _length, String message]) { //DEP30: renamed to _length
!    int length = _length == null ? indexable.length : _length;          //DEP30: assign non-const defau
      // Comparing with `0` as receiver produces better dart2js type inference.
      if (0 > index || index >= length) {
        if (name == null) name = "index";
```

- There remain two false positives related to limitations in the analyzer's flow analysis.

II.3.3 Sample projects

As a sanity test we have run the NNBD analyzer on itself. As expected, a large number of problems are reported, due the nullable nature of AST class type fields. We have chosen not to tackle the annotation of the full analyzer code itself at the moment. On the other hand, we have annotated the nullity specific code, for which we have a nullity annotation ratio is 10 anno/KLOC.

As for other projects, to date, we have run the NNBD analyzer over the following SDK `pkg` projects totaling 2K LOC:

- `expect`
- `fixnum`

Each projects required only a single nullity annotation. The remaining changes to `expect` were to remove redundant (in DartC) explicit initialization of the optional `String reason` parameter with `null` (16 lines).

Revision History

Major updates are documented here.

2016.02.29 (0.6.2)

Expansion and consolidation of proposal details concerning: B.3.4, Default initialization of non-null variables is like DartC. Updates to Appendix II.

2016.02.26 (0.6.0)

New

- B.3.7. Type promotion.
- B.3.8. Type least upper bound.
- B.3.9. Null-aware operators. (Placeholder, section TBC)

2016.02.24 (0.5.0)

The main change is the addition of Appendix II. Tooling and preliminary experience report. In terms of individuals section changes we have:

New

- B.3.5. Adjusted semantics for "assignment compatible" ($\Longleftrightarrow$).
- B.3.6. Static semantics of members of ?T.
- E.1.1.1. Optional parameters with non-null initializers are non-null.
- E.1.1.2. Default field parameters are single view.
- E.3.5. Catch target types and meta type annotations.
- E.3.6. Reducing the annotation burden for local variables, an alternative.
- E.3.7. Dart Style Guide on `Object` vs. `dynamic`.

Updated

- B.2.1. Ensuring `Object` is non-null: elect `_Anything` as a new root. Updated `_Basic` declaration and associated prose since the analyzer expects the `==` operator to be defined for `Null`.
- E.1.1. Optional parameters are nullable-by-default in function bodies only. Now makes reference to cases E.1.1.1 and E.1.1.2.
- G.2. Adjusted name of nullity-scope annotations. Clarified the extent of the scope of `@nullable_by_default`, and that such annotations can also be applied to `parts`.