The Design of Reflectable Capabilities

This document is intended to document the design choices that we have made regarding the class `ReflectCapability` and its subtypes, which is a set of classes declared in the library `package:reflectable/capability.dart` in package `reflectable`. We use the word *capability* to designate instances of subtypes of `ReflectCapability`. This class and its subtypes are used when specifying the level of support that a client of the `reflectable` library will get for reflective operations in a given context, e.g., for instances of a specific class. We use the word *client* when referring to any software artifact which is importing and using the package `reflectable`, generally assuming that it is a package. Using one or another capability as metadata on a class `C` in client code may determine whether or not it is possible to reflectively `invoke` a method on an instance of `C` via an `InstanceMirror`. Given that the motivation for having the package `reflectable` in the first place is to save space consumed by less frugal kinds of reflection, the ability to restrict reflection support to the actual needs is a core point in the design of the package.

Context and Design Ideas

To understand the topics covered in this document, we need to briefly outline how to understand the package `reflectable` as a whole. Then we proceed to explain how we partition the universe of possible kinds of support for reflection, such that we have a set of kinds of reflection to choose from. Finally we explain how capabilities are used to make a selection among these choices, and how they can be applied to specific parts of the client program.

The Package Reflectable

The package `reflectable` is an example of support for mirror-based introspective reflection in object-oriented languages in general, and it should be understandable as such[1]. More specifically, the reflection API offered by the package `reflectable` has been copied verbatim from the API offered by the package `dart:mirrors`, and then modified in a few ways. As a result, code using `dart:mirrors` should be very similar to corresponding code using `reflectable`. The differences that do exist were introduced for two reasons:

- By design, some operations which are declared as top-level functions in `dart:mirrors` are declared as methods on the class `Reflectable` in the package `reflectable`, because instances of subclasses thereof, known as *reflectors*, are intended to play the role as mirror systems[2], and these operations are mirror system specific. For instance, the top-level function `reflect` in `dart:mirrors` corresponds

[1] Bracha, Gilad, and David Ungar. "Mirrors: design principles for meta-level facilities of object-oriented programming languages." *ACM SIGPLAN Notices* 24 Oct. 2004: 331-344.
[2] Bracha, Gilad, and David Ungar, *op. cit.*, or search ‘mirror system’ later in this section.

to two different methods (with different semantics, so they cannot be merged) for two different mirror systems.

- Some proposals have been made for changes to the `dart:mirrors` API. We took the opportunity to try out an updated API by making modifications in the signatures of certain methods. For instance, `InstanceMirror.invoke` will return the result from the method invocation, not an `InstanceMirror` wrapping it. In general, mirror operations return base level values rather than mirrors thereof in the cases where the mirrors are frequently discarded immediately, and where it is easy to create the mirror if needed. Mirror class method signatures have also been modified in one more way: Where `dart:mirrors` methods accept arguments or return results involving `Symbol`, package reflectable uses `String`. This helps avoiding difficulties associated with minification (which is an automated, pervasive renaming procedure that is applied to programs mainly in order to save space), because `String` values remain unchanged throughout compilation.

In summary, the vast majority of the API offered by the package `reflectable` is identical to the API offered by `dart:mirrors`, and design documents about that API or about reflection in general[3][4] will serve to document the underlying ideas and design choices.

Reflection Capability Design

The obvious novel element in package `reflectable` is that it allows clients to specify the level of support for reflection in a new way, by using capabilities in metadata. This section outlines the semantics of reflection capabilities, i.e., which kinds of criteria they should be able to express.

In general, we maintain the property that the specifications of reflection support with one reflector (that is, inside one mirror-system) are *monotone*, in the sense that any program having a certain amount of reflection support specifications will support at least as many reflective operations if additional specifications are added to that reflector. In other words, reflection support specifications can request additional features, they can never prevent any reflection features from being supported. As a result, we obtain a modularity law: a programmer who browses source code and encounters a reflection support specification `S` somewhere can always trust that the corresponding kind of reflection support will be present in the program. Other parts of the program may still add even more reflection support, but they cannot withdraw the features requested by `S`. Similarly, the specifications are *idempotent*, that is, multiple specifications requesting the same feature or overlapping feature sets are harmless, it makes no difference whether a particular thing has been requested once or several times.

[3] Smith, Brian Cantwell. "Procedural reflection in programming languages." 1982.
[4] Sobel, Jonathan M, and Daniel P Friedman. "An introduction to reflection-oriented programming." *Proceedings of reflection* Apr. 1996.

Mirror API Based Capabilities

The level of support for reflection may in principle be specified in many ways: there is a plethora of ways to use reflection, and ideally the client should be able to request support for exactly that which is needed. In order to drastically simplify this universe of possibilities and still maintain a useful level of expressive power, we have decided to use the following stratification as an overall framework for the design:

- The most basic kind of reflection support specification simply addresses the API of the mirror classes directly, that is, it is concerned with "turning on" support for the use of individual methods or small groups of methods in the mirror classes. For instance, it is possible to turn on support for `InstanceMirror.invoke` using one capability, and another capability will turn on `ClassMirror.invoke`. In case a supported method is called it behaves like the corresponding method in a corresponding mirror class from `dart:mirrors`; in case an unsupported method is called, an exception is thrown.
- As a refinement of the API based specification, we have chosen to focus on the specification of allowable argument values given to specific methods in the API. For instance, it is possible to specify a predicate which is used to filter existing method names such that `InstanceMirror.invoke` is supported for methods whose name satisfies that predicate. An example usage could be testing, where reflective invocation of all methods whose name ends in '…`Test`' might be a convenient feature, as opposed to the purely static approach where someone would have to write a centralized listing of all such methods, which could then be used to call them.

With these mechanisms, it is possible to specify support for reflection in terms of mirrors and the features that they offer, independently of the actual source code in the client program.

Reflectee Based Capabilities

Another dimension in the support for reflection is the selection of which parts of the client program the mirrors will be able to reflect upon, both when a `ClassMirror` reflects upon one of those classes, and when an `InstanceMirror` reflects upon one of its instances. In short, this dimension is concerned with the available selection of reflectees.

The general feature covering this type of specification is *quantification* over source code elements—in particular over classes (future extensions will deal with other entities). In this area we have focused on the mechanisms listed below. Note that `MyReflectable` is assumed to be the name of a subclass of `Reflectable` and `myReflectable` is assumed to be a `const` instance of `MyReflectable`, by canonicalization *the* unique `const` instance thereof. This allows us to refer to the general concept of a reflector in terms of the example, `myReflectable`, along with its class and similar associated declarations.

- Reflection support is initiated by invoking one of the methods `reflect` or `reflectType` on `myReflectable`. We have chosen to omit the capability to do `reflect` (in the sense that this is always possible) because there is little reason for having reflection at all without support for instance mirrors. In contrast, we have chosen to have a capability for obtaining class mirrors and similar source code oriented mirrors, which also controls the ability to perform `reflectType`; this is because having these mirrors may have substantial cost in terms of program size. Finally, we have chosen to omit the method `reflectClass`, because it may be replaced by `reflectType`, followed by `originalDeclaration` when `isOriginalDeclaration` is `false`.
- The basic mechanism to get reflection support for a class `C` is to attach metadata to it, and this metadata must be a reflector such as `myReflectable`. The class `Reflectable` has a constructor which is `const` and takes a single argument of type `List<ReflectCapability>` and another constructor which takes up to ten arguments of type `ReflectCapability` (thus avoiding the boilerplate that explicitly makes it a list). `MyReflectable` must have a single constructor which is `const` and takes zero arguments. It is thus enforced that `MyReflectable` passes the `List<ReflectCapability>` in its constructor via a superinitializer, such that every instance of `MyReflectable` has the same state, “the same capabilities”. In summary, this basic mechanism will request reflection support for one class, at the level specified by the capabilities stored in the metadata.
- The reflection support specification can be non-local, that is, it could be placed in a different location in the program than on the target class itself. This is needed when there is a need to request reflection support for a class in a library that cannot be edited (it could be predefined, it could be provided by a third party such that modifications incur repeated maintenance after updates, etc.). This feature has been known as *side tags* since the beginnings of the package `reflectable`. Currently they are attached as metadata to an import directive for the library `package:reflectable/reflectable.dart`, but they could in principle be attached to any program element that admits metadata, or they could occur in other `const` contexts, as long as there is a well-defined convention for finding them such that they can have an effect.
- Quantification generalizes the single-class specifications by allowing a single specification to specify that the capabilities given as its arguments should apply to a set of classes or other program elements. It is easy to provide quantification mechanisms, but we do not want to pollute the package with a bewildering richness of quantification mechanisms, so each of the ones we have should be comprehensible and reasonably powerful, and they should not overlap. So far, we have focused on the following variants:
    - It should be possible to specify that one or more specific classes get a specific level of reflection support; this is a simple generalization of side tags where the target is a list of classes rather than a single class.
    - It should be possible to request reflection support for a set of classes chosen in a more abstract manner than by enumeration. Obvious candidate quantification

mechanisms quantify over all superclasses of a given class; over all supertypes of a given class; over all subclasses of a given class; over all subtypes of a given class; and over all classes whose name matches a given pattern.

- Quantification as in the previous bullet is centralized because it is based on one specification which is then used to 'query' the whole program (or certain large parts of it) for matching entities. It is common and useful to supplement this with a decentralized mechanism, where programmers manually enumerate the members of a set, for instance by attaching a certain marker as metadata to those members. This makes it possible to maintain the set precisely and explicitly, even in the cases where the members do not share obvious common traits that makes the centralized approach convenient. A good example is that a set of methods can be given reflective support by annotating them with metadata; for instance, we may wish to be able to reflectively invoke all methods marked with `@businessRule`.

We subscribe to a point of view where reflective operations are divided into (a) operations concerned with the dynamic behavior of class instances, and (b) operations concerned with the structure of the program; let us call the former *behavioral operations* and the latter *introspective operations*. As an example, using `InstanceMirror.invoke` in order to execute a method on the reflectee is a behavioral operation, whereas it is an introspective operation to use `ClassMirror.declarations` in order to investigate the set of members that an instance of the reflectee class would have.

An important consequence of this distinction is that behavioral operations are concerned with the actual behaviors of objects, which means that inherited method implementations have the same status as method implementations declared in the class which is the runtime type of the reflectee. Conversely, introspective operations are concerned with source code entities such as declarations, and hence the `declarations` reported for a given class does *not* include inherited declarations, they must be found by explicitly iterating over the superclass chain.

Finally, we need to introduce the notion of a *mirror system*, that is, a set of features which provides support for mirror based reflection. This is because we may have several of them: With a choice of a level of reflection support (based on the mirror APIs), and a choice of classes for which this level of support should be provided (based on reflectee selection), it is reasonable to say that we have specified a mirror system. Using multiple mirror systems is relevant in cases where some classes (and/or their instances) require very different levels of support. For example, when a few classes require extensive reflection support and a large number of other classes require just a little bit, using a powerful mirror system with the former and a minimalist one with the latter may be worth the effort, due to the globally improved resource economy. Some extra complexity must be expected; e.g., if we can obtain both a "cheap" and a "powerful" mirror for the same object it will happen via something like `myCheapReflectable.reflect(o)` and `myPowerfulReflectable.reflect(o)`, and it is then up to the programmer to avoid asking the cheap one to do powerful things.

Specifying Reflection Capabilities

As mentioned on page 1, reflection capabilities are specified using the subtype hierarchy rooted in the class `ReflectCapability`, in `package:reflectable/capability.dart`. Instances of these classes are used to build something that may well be considered as abstract syntax trees for a domain specific language. This section describes how this setup can be used to specify reflection support.

The subtype hierarchy under `ReflectCapability` is sealed, in the sense that there is a set of subtypes of `ReflectCapability` in that library, and there should never be any other subtypes of that class; the language does not enforce this concept, but it is a convention that should be followed.

Being used as `const` values, instances of these classes obviously cannot have mutable state, but some of them do contain `const` values such as `String`s or other capabilities. They do not have methods, except the ones that they inherit from `Object`. Altogether, this means that instances of these classes cannot "do anything", but they can be used to build immutable trees, and the universe of possible trees is fixed because the set of classes is fixed. This makes the trees similar to abstract syntax trees, and we can ascribe a semantics to these syntax trees from the outside. That semantics may be implemented by an interpreter or a translator. The sealedness of the set of classes involved is required because an unknown subtype of `ReflectCapability` would not have a semantics, and interpreters and translators would not be able to handle them (and we haven't been convinced that a suitable level of extensibility in those interpreters and translators is worth the effort).

In other words, we specify reflection capabilities by building an abstract syntax tree for an expression in a domain specific language; let us call that language the *reflectable capability language*.

It is obviously possible to have multiple representations of expressions in this language, and we have considered introducing a traditional, textual syntax for it.[5] In this document, we will discuss this language in terms of its grammatical structure, along with an informal semantics of each construct.

Specifying Mirror API Based Capabilities

[5] We could have a parser that accepts a `String`, parses it, and yields an abstract syntax tree consisting of instances of subtypes of `ReflectCapability`, or reports a syntax error. A `Reflectable` constructor taking a `String` argument could be provided, and the `String` could be parsed when needed. This would be a convenient (but less safe) way for programmers to specify reflection support, as an alternative to the current approach where the abstract syntax trees must be specified directly.

Figure 1 shows the raw material for the elements in one part of the reflectable capability language grammar. The left side of the figure contains tokens representing abstract concepts for clustering, and the right side contains tokens representing each of the methods in the entire mirror API. A few tokens represent more than one method (for instance, all of `VariableMirror`, `MethodMirror`, and `TypeVariableMirror` have an `isStatic` getter, and `metadata` is also defined in two classes), but they have been merged into one token because those methods play the same role semantically in all contexts where they occur. In other cases where the semantics differ (`invoke`, `invokeGetter`, `invokeSetter`, and `declarations`) there are multiple tokens for each method name, indicating the enclosing mirror class with a prefix ending in '_'.

Concept	**Specialization**
invocation	`instance_invoke | class_invoke | library_invoke | instance_invokeGetter | class_invokeGetter | library_invokeGetter | instance_invokeSetter | class_invokeSetter | library_invokeSetter | delegate | apply | newInstance`
naming	`simpleName | qualifiedName | constructorName`
classification	`isPrivate | isTopLevel | isImport | isExport | isDeferred | isShow | isHide | isOriginalDeclaration | isAbstract | isStatic | isSynthetic | isRegularMethod | isOperator | isGetter | isSetter | isConstructor | isConstConstructor | isGenerativeConstructor | isRedirectingConstructor | isFactoryConstructor | isFinal | isConst | isOptional | isNamed | hasDefaultValue | hasReflectee | hasReflectedType`
annotation	`metadata`
typing	`instance_type | variable_type | parameter_type | typeVariables | typeArguments | originalDeclaration | isSubtypeOf | isAssignableTo | superClass | superInterfaces | mixin | isSubclassOf | returnType | upperBound | referent`
concretization	`reflectee | reflectedType`
introspection	`owner | function | uri | library_declarations | class_declarations | libraryDependencies | sourceLibrary | targetLibrary | prefix | combinators | instanceMembers | staticMembers | parameters | callMethod | defaultValue`
text	`location | source`

Figure 1. Reflectable capability language API raw material.

Figure 2 shows a reduction of this raw material to a set of capabilities that we consider reasonable. It does not allow programmers to select their capabilities with the same degree of detail, but we expect that the complexity reduction is sufficiently valuable to justify the less fine-grained control.

We have added *RegExp* arguments, specifying that each of these capabilities can meaningfully apply a pattern matching constraint to select the methods, getters, etc. which are included. With

the empty *RegExp* as the default value, all entities in the relevant category are included when no *RegExp* is given. Similarly, we have created variants taking a *MetadataClass* argument which expresses that an entity in the relevant category is included iff it has been annotated with metadata whose type is a subtype of the given *MetadataClass*. This provides support for centralized and slightly abstract selection of entities using regular expressions, and it provides support for decentralized selection of entities using metadata to explicitly mark the entities. It is important to note that the *MetadataClass* is potentially unrelated to the `reflectable` package: We expect the use case where some metadata class from a different package happens to fit well, such that, for instance, it is already attached to exactly the relevant set of methods.

Non-terminal	**Expansion**
apiSelection	*invocation* `|` *naming* `|` *classification* `|` *annotation* `|` *typing* `|` *introspection*
invocation	`instanceInvoke([`*RegExp*`])	instanceInvokeMeta(`*MetadataClass*`)	staticInvoke([`*RegExp*`])	staticInvokeMeta(`*MetadataClass*`)	newInstance([`*RegExp*`])	newInstanceMeta(`*MetadataClass*`)`
naming	`name`
classification	`classify`
annotation	`metadata`
typing	`type([`*UpperBound*`])	typeRelations`
introspection	`owner	declarations	uri	libraryDependencies`

Figure 2. Reflectable capability language API grammar tokens.

The category *text* was removed because we do not plan to support reflective access to the source code as a whole at this point; *naming* has been expressed atomically as `name` because we do not want to distinguish among the different kinds of names, and similarly for all the *classification* predicates, and *annotation*. The category *concretization* was removed because it is trivial to support these features, so they are always enabled.

We have omitted `apply` and `function` because we do not have support for `ClosureMirror` and we do not expect to get it anytime soon.

Moreover, `delegate` was made implicit such that the ability to invoke a method implies the ability to delegate to it.

The category *typing* was simplified in several ways: `instance_type` was renamed into `type` because of its prominence. It optionally receives an *UpperBound* argument which puts a limit on the available class mirrors (class mirrors will only be supported for classes which are subclasses of that *UpperBound*). The method `reflectType` on reflectors is only supported when this

capability is present, and only on class mirrors passing the *UpperBound*, if any. The capabilities `variable_type`, `parameter_type`, and `returnType` were unified into `type` because they are concerned with lookups for the same kind of mirrors. To give some control over the level of detail in the type related mirrors, `typeVariables`, `typeArguments`, `originalDeclaration`, `isSubtypeOf`, `isAssignableTo`, `superClass`, `superInterfaces`, `mixin`, `isSubclassOf`, `upperBound`, and `referent` were unified into `typeRelations`; they all address relations among types, type variables, and `typedef`s, and it may be a substantial extra cost to preserve information about these topics if it is not used.

The category *introspection* was also simplified: We unified `class_declarations`, `library_declarations`, `instanceMembers`, `staticMembers`, `callMethod`, `parameters`, and `defaultValue` into `declarations`. Finally we unified import and export properties into `libraryDependencies` such that it subsumes `sourceLibrary`, `targetLibrary`, `prefix`, and `combinators`. We have retained the `owner` capability separately, because we expect the ability to look up the enclosing declaration for a given declaration to be too costly to include implicitly as part of another capability; and we have retained the `uri` capability because the preservation of information about URIs in JavaScript translated code (which is needed in order to implement the method `uri` on a library mirror) has been characterized as a security problem in some contexts.

Note that certain reflective methods are *non-elementary* in the sense that they can be implemented entirely based on other reflective methods, the *elementary* ones. This affects the following capabilities: `isSubtypeOf`, `isAssignableTo`, `isSubclassOf`, `instanceMembers`, and `staticMembers`. These methods can be implemented in a general manner even for transformed programs, so they are provided as part of the `reflectable` package rather than being generated. Hence, they are supported if and only if the methods they rely on are supported. This is what it means when we say that `instanceMembers` is ‘unified into `declarations`’.

Covering Multiple API Based Capabilities Concisely

In order to avoid overly verbose syntax in the cases where relatively broad reflection support is requested, we have chosen to introduce some grouping tokens. They do not contribute anything new, they just offer a more concise notation for certain selections of capabilities that are expected to occur together frequently. Figure 3 shows these grouping tokens. As an aid to remember what this additional syntax means, we have used words ending in ‘ing’ to give a hint about the tiny amount of abstraction involved in grouping several capabilities into a single construct.

Group	Meaning
`invoking([`*RegExp*`])`	`instanceInvoke([`*RegExp*`]),` `staticInvoke([`*RegExp*`]),` `newInstance([`*RegExp*`])`
`invokingMeta(`*MetadataClass*`)`	`instanceInvokeMeta(`*MetadataClass*`),` `staticInvokeMeta(`*MetadataClass*`),` `newInstanceMeta(`*MetadataClass*`)`
`typing([`*UpperBound*`])`	`type([`*UpperBound*`]), name, classify, metadata, typeRelations, owner, declarations, uri, libraryDependencies`

Figure 3. Grouping tokens for the reflectable capability language.

The semantics of including the capability `invoking(`*RegExp*`)` where *RegExp* stands for a given argument is identical to the semantics of including all three capabilities in the same row on the right hand side of the figure, giving all of them the same *RegExp* as argument. Similarly, `invoking()` without an argument requests support for reflective invocation of all instance methods, all static methods, and all constructors. The semantics of including the capability `invokingMeta(`*MetadataClass*`)` is the same as the semantics of including all three capabilities to the right in the same row, with the same argument. Finally, the semantics of including `typing(`*UpperBound*`)` with a given *UpperBound* is to request support for all the capabilities on the right, passing *UpperBound* to the `type()` capability; that is, requesting support for every feature associated with information about the program structure.

Specifying Reflectee Based Capabilities

In the previous section we found a way to specify mirror API based capabilities as a grammar. It is very simple, because it consists of terminals only, apart from the fact that some of these terminals take an argument that is used to restrict the supported arguments to the matching names. As shown in Fig. 2, the non-terminal *apiSelection* covers them all. We shall use them several at a time, so the typical usage is a list, written as *apiSelection**.

In this section we discuss how the reflection support specified by a given *apiSelection** can be requested for a specific set of program elements. Currently the only supported kind of program element is classes, but this will be generalized later. The program elements that receive reflection support are called the *targets* of the specification, and the specification itself is given as a superinitializer in a subclass (call it `MyReflectable`) of class `Reflectable`, with a unique instance (call it `myReflectable`). Now, `myReflectable` is used as metadata somewhere in the program, and each kind of capability is only applicable as an annotation in certain locations, which is discussed below.

Figure 4 shows how capabilities and annotations can be constructed, generally starting from an *apiSelection**. The non-terminals in this part of the grammar have been named after the intended location of the metadata which is or contains a capability of the corresponding kind.

Non-terminals	**Expansions**
reflector	`Reflectable(`*targetMetadata*`)`
targetMetadata	*apiSelection*`*` `|` `subtypeQuantify(`*apiSelection*`*)` `|` `admitSubtype(`*apiSelection*`*)`
globalMetadata	`globalQuantify(`*RegExp*, *reflector*`)` `|` `globalQuantifyMeta(`*MetadataClass, reflector*`)`

Figure 4. Reflectable capability language target selection constructs.

In practice, a *reflector* is an instance of a subclass of class `Reflectable` that is directly attached to a class as metadata, or passed to a global quantifier; in the running example it is the object `myReflectable`. The reflector has one piece of state that we model with *targetMetadata*. In the grammar in Fig. 4 we use the identifier `Reflectable` to stand for all the subclasses, and we model the state by letting it take the corresponding *targetMetadata* as an argument. The semantics of annotating a class with a given *reflector* depends on the *targetMetadata*, as described below.

A *targetMetadata* capability can be a base level set of capabilities, that is, an *apiSelection**, and it can also be a quantifier taking such an *apiSelection** as an argument. The semantics of attaching a *reflector* containing a plain *apiSelection** to a target class `C` is that reflection support at the level specified by the given *apiSelection** is provided for the class `C` and instances thereof. The semantics of attaching a *reflector* containing `subtypeQuantify(`*apiSelection*`*)` to a class `C` is that the reflection support specified by the given *apiSelection** is provided for all classes which are subtypes of the class `C`, including `C` itself, and their instances. The semantics of attaching a *reflector* containing `admitSubtype(`*apiSelection*`*)` to a class `C` is a pragmatic mix of the former two which is subtle enough to warrant a slightly more detailed discussion, given in the next section. The basic idea is that it allows instances of subtypes of the target class to be treated as if they were instances of the target class.

Finally, we support side tags using global quantifiers, `globalQuantify(`*RegExp*, *reflector*`)` and `globalQuantifyMeta(`*MetadataClass*, *reflector*`)`. Currently, we have decided that they must be attached as metadata to an import statement importing `package:reflectable/reflectable.dart`, but we may relax this restriction if other placements turn out to be helpful in practice. Due to the monotone semantics of capabilities it is

not a problem if a given program contains more than one such *globalMetadata*, the provided reflection support will simply be the least one that satisfies all requests.

The semantics of having `globalQuantify(`*RegExp*, *reflector*`)` in a program is ideally identical to the semantics of having the given *reflector* attached directly to each of those classes in the program whose qualified name matches the given *RegExp*. Similarly, the semantics of having `globalQuantifyMeta(`*MetadataClass*, *reflector*`)` in a program is ideally identical to the semantics of having the given *reflector* attached directly to each of those classes whose metadata includes an instance of type *MetadataClass*. At this point, however, we must add an adjustment to the ideal goal that the semantics is identical: Access to private declarations may not be fully supported with a *globalMetadata*. This is discussed in the next section.

Completely or Partially Mirrored Instances?

Traditionally, it is assumed that reflective access to an instance, a class, or some other entity will provide a complete and faithful view of that entity. For instance, it should be possible for reflective code to access features declared as private even when that reflective code is located in a context where non-reflective access to the same features would not be allowed. Moreover, when a reflective lookup is used to learn which class a given object is an instance of, it is expected that the response describes the actual runtime type of the object, and not some superclass such as the statically known type of that object in some context.

In the package `reflectable` there are reasons for violating this completeness assumption, and some of them are built-in consequences of the reasons for having this package in the first place. In other words, these restrictions will not go away entirely. Other restrictions may be lifted in the future, because they were introduced based on certain trade-offs made in the implementation of the package.

The main motivation for providing the package `reflectable` is that the more general support for reflection provided by the `dart:mirrors` package tends to be too costly at runtime in terms of program size. Hence, it is a core point for `reflectable` to specify a restricted version of reflection that fits the purposes of a given program, such that it can be done using a significantly smaller amount of space. Consequently, it will be perfectly normal for such a program to have reflective support for an object without reflective access to, say, some of its methods. There are several other kinds of coverage which is incomplete by design, and they are not a problem: they are part of the reason for using package `reflectable` in the first place.

The following subsections discuss two different situations where some restrictions apply that are not there by design. We first discuss cases where access to private features is incomplete, and then we discuss the consequences of admitting subtypes as specified with `admitSubtype(`*apiSelection*`*)`.

Privacy Related Restrictions

The restrictions discussed in this subsection are motivated by trade-offs in the implementation in package `reflectable`, so we need to mention some implementation details. The package `reflectable` has been designed for program transformation, i.e., it is intended that a source to source transformer shall be able to receive a given program (which is using package `reflectable`, and indirectly `dart:mirrors`) as input, and transform it to an equivalent program that does not use `dart:mirrors`, generally by generating mirror implementation classes containing ordinary, non-reflective code.

Ordinary code cannot violate privacy restrictions. Hence, reflective operations cannot, say, read or write a private field in a different library. The implication is that private access can only be supported for classes declared in a library which can be transformed, because only then can the generated mirror implementation class coexist with the target class. Using a transformer as we currently do (and plan to do in the future), all libraries in the client package can be transformed. However, libraries outside the client package cannot be transformed, which in particular matters for libraries from other packages, and for pre-defined libraries.

For some libraries which cannot be transformed, it would be possible to create a local copy of the library in the client package and modify the program globally such that it uses that local copy, and such that the semantics of the copied library is identical to the semantics of the original. This cannot be done for libraries that contain language primitives which cannot be implemented in the language; for instance, the pre-defined class `int` cannot be replaced by a user-defined class. Moreover, the need to copy and adjust one library could propagate to another library, e.g., if the latter imports the former. Hence, not even this workaround would enable transformation of all libraries. We do not currently have any plans to use this kind of techniques, and hence only libraries in the current package can be transformed.

Given that the main transformation technique for package `reflectable` is to generate a number of mirror classes for each target class, this means that access to private declarations cannot be supported for classes in libraries that cannot be transformed. This applies to private classes as a whole, and it applies to private declarations in public classes.

It should be noted that transformation of libraries imported from other packages might be manageable to implement, but it requires changes to the basic tools used to process Dart programs, e.g., `pub`. Alternatively, it is possible that this restriction can be lifted in general in the future, if the tools (compilers, analyzers, etc.) are modified to support a privacy overriding mechanism.

Considerations around Admitting Subtypes

When a *targetMetadata* on the form *apiSelection*`*` is attached to a given class `C`, the effect is that reflection support is provided for the class `C` and for instances of `C`. However, that support can be extended to give partial reflection support for instances of subtypes of `C` in a way that does not incur further costs in terms of program size: A mirror generated for instances of class `C` can have a `reflectee` (the object being mirrored by that mirror) whose type is a proper subtype of `C`. A *targetMetadata* on the form `admitSubtype(`*apiSelection*`*)` is used to specify exactly this: It enables an instance mirror to hold a reflectee which is an instance of a proper subtype of the type that the mirror was generated for.

The question arises which instance mirror to use for a given object *O* with runtime type `D` which is given as an argument to the operation `reflect` on a reflector, when there is no mirror class which was created for exactly `D`. This is the situation where a subtype reflectee is actually admitted. In general, there may be multiple candidate mirror classes corresponding to classes C_1, C_2, .. C_k which are “least supertypes of `D`” in the sense that no type `E` is a proper supertype of `D` and a proper subtype of C_i for any `i` (this also implies that no two classes C_i and C_j are subtypes of each other). The language specification includes an algorithm which will find a uniquely determined supertype of C_1 .. C_k which is called their *least upper bound*. We cannot use this algorithm directly because we have an arbitrary subset of the types in a type hierarchy rather than all types, and then we need to make a similar decision for this “sparse” subtype hierarchy that only includes classes with reflection support from the given reflector. Nevertheless, we expect that it is possible to create a variant of the least upper bound algorithm which will work for these sparse subtype hierarchies.

It should be noted that a very basic invariant which is commonly assumed for reflection support in various languages is violated: An instance mirror constructed for type `C` can have a reflectee which is an instance of a proper subtype `D`. Of course, not all mirror systems have anything like the notion of a mirror that is constructed for a given type, but the corresponding problem is relevant everywhere: The mirror will not report on the properties of the object as-is, it will report on the properties of instances of a supertype. This is a kind of incompleteness, and it even causes the mirror to give plain *incorrect* descriptions of the object in some cases.

In particular, assume that an object *O* with runtime type `D` is given, and that we have an instance mirror *IM* whose `reflectee` is *O*. Assume that the class of *IM* was generated for instances of a class `C`, which is a proper supertype of `D`. It is only because of `admitSubtype` that it is even possible for *IM* to have a `reflectee` whose runtime type is not `C`. In many situations this discrepancy makes no difference and *IM* works fine with *O*, but it is informative to focus on a case where it really matters:

Let us use a reflective operation on *IM* to get a class mirror for the class of *O*. *IM.*`type` will return an instance *CM* of the class mirror for `C`, not a class mirror for *O*’s actual runtime type `D`. If a programmer uses this approach to look up the name of the class of an object like *O*, the answer will simply be wrong, it says `"C"` and it should have said `"D"`. Similarly, if we traverse the superclasses we will never see the class `D`, nor the intermediate classes between `D` and `C`.

A real-world example is serialization: if we look up the declarations of fields in order to serialize the reflectee then we will silently fail to include the fields declared in the ignored subclasses down to `D`. In general, there are many unpleasant surprises waiting for the naive user of this feature, so it should be considered to be an expert-only option.

Why not just do the “right thing” and return a class mirror for `D`? It is not possible to simply check the `runtimeType` of `reflectee` in the implementation of the method `type`, and then deliver a class mirror of `D` because, typically, there *is* no class mirror for `D`. In fact, the whole point of having the `admitSubtype` quantifier is that it saves space because a potentially large number of subtypes of a given type can be given partial reflection support without the need to generate a correspondingly large number of mirror classes.

To further clarify what it means to get ‘partial’ reflective support, consider some cases:

Reflectively calling instance methods on *O* which are declared in `C` or inherited into `C` will work as expected, and standard object-oriented method invocation will ensure that it is the correct method implementation for *O* which is called, not just the most specific implementation which is available in `C`.

Calling instance methods on *O* which are declared in a proper subtype of `C`, including methods from `D` itself, will not work. This is because the class of *IM* has been generated under the assumption that no such methods exist, it only knows about `C` methods. As mentioned, if we fetch the class of *O* we may get a proper supertype of the actual class of *O*, and hence all the derived operations will be similarly affected. For instance, the declarations from *CM* will be the declarations in `C`, and they have nothing to do with the declarations in `D`. Similarly, if we traverse the superclasses then we will only see a strict suffix of the actual list of superclasses of the class of *O*.

Based on these serious issues, we have decided that when an instance mirror is associated with the `admitSubtype` quantifier, it shall be an error to execute the `type` method in order to obtain a mirror of a class, because it is very unlikely to work as intended when that class is in fact not the class of the reflectee. It would be possible to allow it in the cases where the match happens to be perfect, but this would be difficult for programmers to use, and they may as well use `reflectType` directly if they want to reflect upon a class which is not taken directly from an instance.

In summary, there is a delicate trade-off to make in the case where an entire subtype hierarchy should be equipped with reflection support. The trade off is to either pay the price in terms of program size and get full support (using `subtypeQuantify`); or to save space aggressively and in return tolerate the partial support for reflection (using `admitSubtype`).

Summary

We have described the design of the capabilities used in the package `reflectable` to specify the desired level of support for reflection. The underlying idea is that the capabilities at the base level specify a selection of operations from the API of the mirror classes, along with some simple restrictions on the allowable arguments to those operations. On top of that, the API based capabilities can be associated with specific parts of the target program (though at this point only classes) such that exactly those classes will have the reflection support specified with the API based capabilities. The target classes can be selected individually, by adding a reflector as metadata on each target class. Alternatively, target classes can be selected by quantification: It is possible to quantify over all subtypes, in which case not only the class `C` that holds the metadata receives reflection support, but also all subtypes of `C`. Finally, it is possible to admit instances of subtypes as reflectees of a small set of mirrors, such that partial reflection support is achieved for many classes, without the cost of having many mirror classes.