Fall 2012

STATIC TYPE CHECKER TOOLS FOR DART

Snigdha Mokkapati
San Jose State University

STATIC TYPE CHECKER TOOLS FOR DART

A Project Report

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

by

Snigdha Mokkapati

December 2012

The Designated Project Committee Approves the Project Titled

STATIC TYPE CHECKER TOOLS FOR DART

by

Snigdha Mokkapati

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2012

Dr. Cay Horstmann	Department of Computer Science
Prof. Ronald Mak	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science

ABSTRACT

STATIC TYPE CHECKER TOOLS FOR DART

by Snigdha Mokkapati

This project presents the static type checkers that I developed for the optional type system of the Dart programming language. Dart is an optionally typed language and as a result has an unsound type system. In this project I have created the static type checker tools for dart. The first static type checker tool ensures mandatory typing of Dart code. This checker can be invoked by giving a new compiler option that I have added to the compiler configuration. This checker will help in catching any type errors early at compile time rather than at run time. The second static type checker improves the Dart's support for covariant generics. This static checker issues warnings at compile time if the covariant use of generics is followed by a modification of the collection passed covariantly. I have also introduced three annotations that will add more type safety to the Dart programming language. The `@notnull` annotation is to ensure that null values are not passed as arguments to method parameters. This nullness checker ensures that a running program will never throw a null pointer exception. The `@modifies` annotation supports the covariance check. The `@linear` annotation is used to prevent unexpected modification of objects by aliasing. The `@linear` annotation can be used in conjunction with Dart isolates for concurrent programming.

ACKNOWLEDGEMENTS

This project is the result of the support of many individuals. First, great appreciation is due to Dr. Cay Horstmann who encouraged, guided and supported me through all the levels of the project. Second, thanks to Prof. Ronald Mak and Dr. Chris Pollett who gave me constant feedback, guidance and support. Lastly, thanks to my friends and project mates Jesus Rocha and Purna Chatterjee Patra for helping me in the setup of Dart tools and SDK.

Contents

List of Figures

List of Tables

1. Introduction

Unlike most programming languages like Java, Scala, C++, C#, Haskell, etc., which are statically typed, Dart is a dynamically typed language. The most popular feature of Dart is the use of optional types. Programs can be written and run without using any type annotations, or optionally type annotations can be added to the programs. However, adding type annotations to the program will not prevent it from compiling or running, even if the annotations are incomplete or wrong. Currently type annotations mainly serve the purpose of documentation and early error detection in Dart.

For early error detection, Dart provides a static checker which warns about the potential problems at compile time. Most of the checks provided by this static checker are related to types. However, these warnings don't prevent from running the program. There is also a checked mode in Dart where certain type checks are performed during the development stage or in developer mode (run time). However, neither of the static checker or the checked mode provide the same level of static checking performed in a statically typed language.

The focus of this project is to create static type checker tools which will interpret the type annotations in Dart as in a statically typed language. First static type checker is a mandatory types checker to ensure typing in Dart code. The project also involves improving the static type checks performed by the compiler as strictly as in a statically typed language. The main purpose of this type checker is to provide the possibility of having more type checking for beginning users such as students. The second type checker is a relatively complex type checker concentrating on the variance of generics in Dart. Dart generics are covariant and pass both static and dynamic checks. The contravariant generics in Dart pass static checks but fail dynamic checks. The goal is to create a static type checker for these generic type inferences in Dart. The project also involves implementation of annotations in Dart in support of static type checkers. The nullness checker is to ensure that null values are not passed to method parameters. The implementation of `@notnull` annotation parsing is added to the dartc compiler to support `@notnull` annotation for method parameters and identifiers. The linear checker is to prevent unexpected modifications of objects through aliasing. The implementation of `@linear` annotation parsing is added to the dartc compiler to support `@linear` annotation for method parameters and method local identifiers. These type checker plugins enhance the optional types feature of Dart by providing a static type environment for those who are interested in it.

2. Optional Type Systems

Type checking is the process of mapping and verifying the type constraints in a program. The type constraints can be explicitly stated or implicitly referenced. The process of type checking can be performed at compile-time or at run-time. The programming languages which support

compile-time type checking are called statically typed languages and the languages which support run-time type checking are called dynamically typed languages. Statically typed languages include Ada, C, Haskell, Java, Objective-C and Scala. Dynamically typed languages include Erlang, JavaScript, Lisp, PHP, Python, Ruby and Smalltalk.

Statically typed languages are traditionally considered as languages with well-established advantages. A language that supports static checking provides advantages such as early error detection, opportunity to do code optimization based on type information and a form of machine-checkable documentation [**3**]. Dynamically typed languages on the other hand can be more expressive and are better suited for the changing requirements [**7**]. While a good and strong mandatory type system adds completeness to a programming language, it is difficult to formulate a perfect real world mandatory type system. Formulation might involve assumptions which in turn results in bugs in the implementation.

To bridge the gap between statically typed languages and dynamically typed languages, Gilad Bracha proposed *pluggable type systems* [**3**]. A pluggable type system is a language environment where several *optional type systems* can exist together. He defines an optional type system to be one that:

1. has no effect on the runtime semantics of the programming language, and
2. does not mandate type annotations in the syntax.

If a language implements the concept of optional type systems, then it is not mandatory that a programmer provide type annotations in the source code. Also, providing type annotations or skipping them does not affect the run time behavior of the program. If the run time behavior of the program is independent of the type system, then type systems can be used as plug-ins [**3**]. While developing a framework for pluggable type system for dynamically typed languages, challenges might arise with respect to both optional typing and dynamic typing. Since a programmer is not expected to provide the type annotations in an optionally typed language, the pluggable type system should be able to handle both partially typed and untyped programs. Also, since there is no basis for static type checking in a dynamically typed language, it might be a challenge to do the static type analysis for constructs such as dynamic binding or reflection.

Dart is a dynamically typed language with support for optional type system. It is inspired by Strongtalk, a type checker for Smalltalk. Smalltalk is inherently a dynamically typed language and many attempts have been made to introduce a static type system into Smalltalk [**4**]. It was difficult to introduce a complete static type system into Smalltalk with the requirement to type check existing pieces of non-typed Smalltalk code. Instead, Bracha and Griswold developed Strongtalk, a type checker for Smalltalk in a production environment [**4**]. Strongtalk is not designed with the idea to type check existing Smalltalk code without modification [**4**]. It is designed with the strong support for *downward compatibility* meaning that Strongtalk code can be easily converted to Smalltalk code anytime by removing the type annotations. Following are some of the key features of the Strongtalk type system. Strongtalk separates types from classes. Instead, types are included in the Smalltalk *protocols*. In Smalltalk, everything is an object and

the objects communicate with each other by sending and receiving messages. The set of messages which an object understands and responds to is known as message protocol or simply protocol. For example, a string's protocol contains `indexOf, size, asUppercase` and many other string operations related messages. Smalltalk groups objects sharing the same protocol into classes. So every object is an instance of some class and each class is associated with a protocol.

Strongtalk refined the notion of protocol in Smalltalk by introducing types into the message signatures of a protocol. Along with the names of the messages, the types of the arguments and the types of the objects they return are included by giving type annotations in the angle brackets. Following is an example of Strongtalk protocol and a class taken from [**4**].

```
protocol PlanarPoint
      x ^<Integer>.
      x: <Integer> ^<Integer>.
      y ^<Integer>.
      y: <Integer> ^<Integer>.
      + <Self> ^<Self>.

class BasicPlanarPoint
      instance var x <Integer>.
      instance var y <Integer>.
      class methods
            new ^<Instance>
                  ^super new init.
      instance methods
            init
                  x := 0 y:= 0.
            x ^<Integer>
                  ^x.
            x: xval <Integer>
                  x := xval.
            y ^<Integer>
                  ^y.
            y: yval <Integer>
                  y := yval.
            + p <Self> ^<Self>
                  ^(self class new x: self x + p x)
                                          y: self y + p y.
```

In the above example, `PlanarPoint` is a protocol for points in a plane. Class `BasicPlanarPoint` implements the `PlanarPoint` protocol. The protocol provides four messages to access the instance variables and one to take the sum of two `PlanarPoint` objects**.** The caret is used to represent the return type of the message. The `Self` keyword in a protocol refers to the protocol of the receiver. The code above is in a form of pseudo-code, not the concrete syntax of Strongtalk. Strongtalk also introduced typing of variable declarations which include blocks and method arguments, instance, class, pool and global variables. Strongtalk also supports generic type definitions. Below is an example of a generic `List` protocol in Strongtalk [**4**].

```
generic protocol List[T]
      add:<T>.
```

```
head ^<T | Nil>.
tail ^<Self | Nil>.
map:<Block[T, ^S]> ^<List[S]>
       where S :: (actual arg:1) returnType
```

The generics in Strongtalk work in a similar way to generics in Java. A generic can be invoked by passing the type of actual parameter, e.g., `List[Integer]`. The type of the actual parameter replaces the formal parameter `T` in the generic for that invocation. The other interesting feature supported by Strongtalk is parametric polymorphism. In some cases, the return type of a method may depend on the actual parameter types passed into the method. Parametric polymorphism is used to express the signature of such methods. In the above code example of `List protocol`, the `map` message is an example of parametric polymorphism. The `map` message takes a block as argument and returns a list. The type of the list being returned is dependent on the return type of the block `S`.

In 2007, Niklaus Haldimann created TypePlug, a framework for pluggable type systems for Smalltalk [**7**]. In TypePlug, Haldimann introduced typing of elements in Smalltalk syntax. He introduced types for basic constructs of Smalltalk such as literals, global variables, blocks, arrays and primitives by introducing the `typeFor*` family of methods. For example, according to his syntax,

`typeForLiteral: aValue` method provides a type for literals such as integers, strings, symbols, constants etc. `aValue` represents the literal value being typed.
`typeForArray: anArray` method provides a type for an array literal.

Haldimann also added the support for subtyping and unification in TypePlug. He supported subtyping by implementing a `is:subtypeOf:` method which takes two types as arguments and returns a boolean value. Following is the syntax of the method,

```
is: aType subtypeOf: anotherType
```

This method returns true if `aType` is the subtype of `anotherType`.
The unification operation creates a type that represents the union of two types [**7**]. Following is the syntax of the method used for unification.

```
unifyType: aType with: anotherType
```

The implementation of all these above methods is similar to the way I have added annotations in the Dart. These additional methods for typing are introduced into Smalltalk protocols and classes and when parsed in the frontend, are used to add additional type information to the syntax tree nodes being constructed. At the backend when the type checker walks through the syntax tree, type safety is ensured by certain type checking rules for assignments, return statements, message sends, and so on.

In 2012, version 1.4.4 of the Checker Framework: Custom pluggable types for Java was developed at MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA, USA [**11**]. The Checker Framework supports adding pluggable type systems to the Java language in a backward-compatible way [**1**]. In this project, pluggable type checkers for Java have been developed which can be invoked as plug-ins to the javac compiler. Checkers include nullness checker to check for null pointer errors, regex checker to prevent use of syntactically invalid regular expressions and a lock checker for concurrency and lock errors. Following is an example of running a checker,

```
javac -processor checkers.nullness.NullnessChecker Source.java
```

`-processor` is the command line option to give the name of the checker to run. In the above example, the nullness checker is being invoked on the source file `Source.java`.

In this project, I have added a static type checker to the Dart programming language which can be invoked by giving a new compiler option while running the dartc compiler. The purpose of this static checker is to ensure that static types are provided in the Dart source code. I have also implemented the checkers for Dart covariant generics, nullness checks and linear checks. For covariance, nullness and linear checks, I have given an ad-hoc implementation of the checkers rather than creating a framework. I have supported these checks by adding annotation support in the Dart source code and in the dartc compiler.

3. Dart Type Concepts

3.1 Dart Optional Types

One of the Dart programming language's features is the use of optional types [**2**]. Essentially Dart is a dynamically typed language. Unlike most other dynamically typed languages, Dart also supports optional typing. However, adding types to Dart code does not make it equivalent to the code written in a strictly statically typed language. The reason behind this is that static checks performed by Dart are unsound. The static checker of the Dart warns about the possible type errors at the compile time but only about the ones that are likely to be real problems. It does not warn about every possible type error because, of course, Dart is not a statically typed language. The static checker just issues warnings. The code can still be compiled and can be run. One example of unsoundness of Dart static typing is *down assignments*. For example, according to Dart, a type T may be assigned to a type S if T is a subtype of S *or* if S is a subtype of T [**2**]. Clearly this rule might fail for some down assignments if not assigned properly.

```
class Person {}

class Employee extends Person {
```

```
        //some method or field specific to Employee
}

class Student extends Person {
        //some method or field specific to Student
}

main() {
        Person aPerson = new Person();
        Employee aEmployee = new Employee();
        Employee anotherEmployee = new Employee();
        Student aStudent = new Student();

        aPerson = aEmployee; //Perfect, always valid
        anotherEmployee = aPerson; //valid in this case
}
```

In the code above, the last statement `anotherEmployee = aPerson;` is valid here because `aPerson` is also pointing to an `Employee`. However, `aPerson` could have been pointing to a `Student` also and then it will be invalid at runtime. However, the Dart static type checker does not differentiate those two cases. It just passes both the checks supporting its optimistic nature.

The Dart's static checker does not complain if no types are given because Dart supports a special type called "Dynamic". When no type is given, the default type is "Dynamic" and so the static checker does not complain about its type.

The other place where type checking is performed is during run time in *checked mode*. Dart programs can be run in a special mode called checked mode which automatically executes certain type checks while running the program. Some of these checks include type checking while passing arguments to methods, returning results and executing assignments. If these checks fail in checked mode, then the execution of the program stops. Following is an example to explain the significance of the checked mode. An assignment statement like

```
int number = new Object();
```

fails in the checked mode, because `Object` is not a subtype of `int`.

```
Object getValue() {
        return 25;
}

int number = getValue();
```

However, the above code passes the checked mode even though the method signature says that the return type is Object. Dart checks the runtime type of the object against the declared type of the variable. The checked mode is only useful if the code is typed. If the typing is completely eliminated from the code, checked mode does not get in your way.

3.2 Dart Covariant Generics

The generic type G is covariant if `A <: B => G<A> <: G<B>` and contravariant if `A <: B => G<B> <: G<A>`. For mutable collections, if A is subtype of B then ReadOnlyReference<A> is substitutable for ReadOnlyReference<B> according to the covariance rule and WriteOnlyReference<B> is substitutable for WriteOnlyReference<A> according to the contravariance rule. Mutable covariant collections are unsound because `A` can be stored in `G<B>`. Please refer to [**10**] for detailed understanding of the concept of variance in Java and Scala.

The Dart generics are covariant similar to Java arrays. It is based on the idea that contravariant use of generics is not as common as covariant use of generics. This is another example of Dart's "optimistic" approach. So, instead of making its generics invariant, Dart chose to support covariant generics following this approach.

Since Dart supports covariant generics, in case of contravariant usage, static checking passes, but during run time, in checked mode, dynamic checking fails. If no checked mode is used, code runs correctly. However, there is a workaround for this problem by removing the generic type of the collection but that completely silences the type checking and allows even incorrect use.

The following is an example of the contravariant use of collections with generics:

```
main() {
	List<Person> plist = new List<Person>(4);
	contFunctionWithGenerics(elist);
	// Contravariant use:
	// * static type checking OK
	// * dynamic type checking fails!
	// * runs correctly if dynamic checking is off

}

contFunctionWithGenerics(List<Employee> elist) {
	elist.add(new Employee());
}
```

The following is an example of the contravariant use of collections without generics:

```
main() {
	List<int> ilist = new List<int>(4);
	contFunctionWithoutGenerics(ilist);
	// Incorrect use, but workaround has silenced type checking:
	// * static type checking OK
	// * dynamic type checking OK
	// * fails at runtime

}
contFunctionWithoutGenerics(List list) {
	list.add(new Employee());
}
```

3.3 Errors and Warnings

Dart supports the following errors and warnings: [**6**]

1. Compile time errors - These are errors issued at the compile time and stop you from executing the program. The error is displayed along with the message “Compilation failed.”
2. Static warnings - These are warnings issued at the compile time. Most of them are related to the types. The Dart’s static checker is invoked during the compilation and issues these warnings. However, the code is still compiled and can be executed.
3. Dynamic errors - These are the type errors that are produced while running the program in the checked mode. Program execution stops if any dynamic errors exist.
4. Run time errors - These are actual run time errors or exceptions. If not handled, program execution stops.

The following code snippets are examples of the occurrence of four types of errors or warnings described above. In the code below, declaring two variables with the same name in a scope gives a compile time error.

```
class Person {

	var firstname = 'Bob';
	var firstname = 'Duncan';

	Person() {}
}

main() {
	Person aPerson = new Person();
}

$dartc Person.dart
Person.dart/Person.dart:4: name clashes with a previously defined member at
Person.dart line 3 column 7
	3:	var firstname = 'Bob';
	4:	var firstname = 'Duncan';
Compilation failed with 1 problem.
```

In the code below, assigning a string to an integer gives a compile type static warning. However, the code will still be compiled and running the program prints a string '455' as output.

```
class Person {

	var firstname = 'Bob';
	String age = '45';

	getUpdatedAge(diff) {
		int finalAge = age + diff;
```

```
                print(finalAge);
        }
}

main() {
        Person aPerson = new Person();
        aPerson.getUpdatedAge('5');
}

Compilation Warning:
$dartc Person.dart
Person.dart/Person.dart:10: String is not assignable to int
        9:     getUpdatedAge(diff) {
        10:           int finalAge = age + diff;


Output:
$dart_bin Person.dart
455
```

The same code above when run with `--enable_type_checks` option gives the following dynamic error.

```
$dart_bin --enable_type_checks Person.dart
Unhandled exception:
Failed type check: type OneByteString is not assignable to  type int of
finalAge
```

A null pointer exception is an example of the run time error.

4. Dart Compilers

Dart code can be executed in two different ways: Either on a native virtual machine or on top of a JavaScript engine by using a compiler that translates Dart code to JavaScript. Dart provides a Chromium based browser called Dartium which includes the Dart Virtual Machine. Chromium is an open source web browser project. Google Chrome's source code is drawn from Chromium's source code. Other browsers based on Chromium include Comodo Dragon, CodeWeavers CrossOver, RockMelt, SRWare Iron. The Dart Virtual Machine is a language based VM as opposed to regular bytecode based VMs [**9**]. Hence, Dartium can execute Dart web apps directly without needing them to be compiled to JavaScript. However, to run the Dart code on other browsers, the code still needs to be compiled into JavaScript using one of its compilers.

Dart Editor can also be used to run the Dart code. It is an open source IDE (Eclipse based), that can be used to edit and run Dart web apps and also to invoke Dart to JavaScript compiler. Similar to Eclipse IDE, Dart Editor provides basic editing functionality along with the special features such as API browsing, code completion and refactoring. When a web app is run in Dart Editor, it brings up a browser window in which the app code is run. The default browser is Dartium but a different browser can also be specified using launch configuration. By default, Dart Editor

compiles the Dart code to JavaScript before executing it in the browser. There is also an option to just compile the code to JavaScript and not execute the app. Dart Editor download includes the Dart Editor, Dart SDK and Dartium. Dart SDK includes the Dart libraries and the command line tools for Dart-to-JavaScript compiler, Dart VM, Dart static analyzer and Dart package manager.

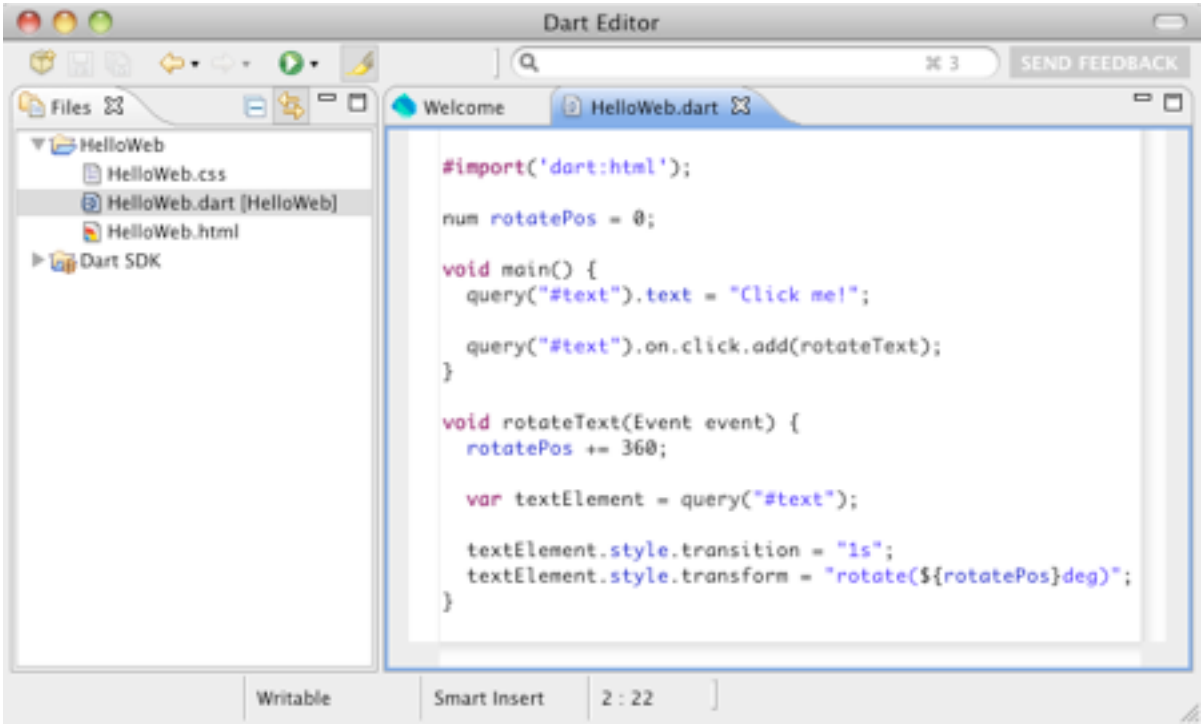

Figure 4.1 Dart Editor

From the time Dart was introduced, the following three compilers have been created:

1. dartc - First Dart compiler that emitted JavaScript code from Dart code. dartc was written in java and was initially the compiler behind the Dart Editor. dartc was released with the initial version of Dart source code in October 2011. Around mid 2012 (exact date not known) dartc was discontinued to generate JavaScript. From that time, it is only used for static analysis of Dart code and is currently renamed to dart_analyzer.
2. Frog - Frog is a dart to Javascript compiler written in Dart. However, Frog didn't implement the full semantics of the language and so did not have all the capabilities of dartc. Frog was released around November 2011.
3. dart2js – This is a new compiler introduced after Frog. dart2js is written in Dart as well. It is intended to implement full semantics of the language and is now the only compiler in Editor and SDK. dart2js was released for testing in May 2012.

Compared to Frog and dart2js, dartc produces long javascript code.

Currently Dart SDK includes dart2js as the Dart-to-JavaScript compiler, dart_analyzer as the Dart static analyzer and dart as Dart VM. All these are included as command line tools in the SDK.

In this project, I worked on the dartc compiler. It was the only compiler with complete language support in practice at the time I started working on this project in December 2011, so I improved on one of the initial versions of dartc. Frog was more focused towards increasing the compilation speed and reducing the size of the resulting javascript. However, the focus of my project was on the language syntax and type checking which was supported by dartc. I added an additional compiler flag, added the support for annotations by modifying the dartc parser and added the type checkers by modifying the syntax tree nodes and the type checker classes of the compiler. For the testing, I implemented the new compiler flag, mandatory types checker and variance checker without annotations in the current version of dart_analyzer.

Compiler Flags:

Below is a table of flags supported by dartc compiler and the dart_analyzer static analyzer along with the new flag I implemented.

Table 4.1 Compiler Flags of dartc Compiler

Flag	Purpose
--fatal-type-errors	If this option is given with the compiler, any type errors are considered fatal and the compilation is failed. Without the option, type errors are just warnings during compilation.
--enable_type_checks	This option is given at the run time with dart_bin to enter the checked mode.
--check-only	With this option, the code is just parsed for the type errors and will not generate any byte code.
--must-have-types	I added this option to enforce static typing in the Dart code.

5. Mandatory Types Checker

I implemented a static checker to add static typing functionality to Dart as an inherently statically typed language would have. With this mandatory types checker, typing is made mandatory in the Dart code. If types are given, Dart's static checker gives most of the warnings at the compile time. The dartc compiler has an option called `--fatal-type-errors`. If the compiler is invoked

using this option, all static warnings are considered fatal and the compilation fails. However, in the absence of that option, the code will still be compiled and can be run.

As first part of the project, I added a compiler flag `--must-have-types` to the compiler configuration. If this flag is given while invoking the dartc compiler, dynamic typing is not allowed. Please refer to Appendix A for the code modifications I did to add the compiler flag.

To support must static typing, following are the changes I made to the code.

I added a new error code in `TypeErrorCode.java` to throw a no dynamic typing error.

```
DYNAMIC_TYPING_NOT_ALLOWED("dynamic typing is not allowed \"%s\""),
```

The setting of the `--must-have-types` option can be obtained from the `CompilerConfiguration` object. It was not available in the `TypeAnalyzer` class and its private class `Analyzer`. So I modified following methods and constructors to pass the `CompilerConfiguration` object around.

In `DartCompilationPhase.java` file, I changed the `exec` method signature by adding `CompilerConfiguration` in the method parameters.

```
public interface DartCompilationPhase {

	/**
	* Execute this phase on a unit.
	*
	* @param unit the program to process
	* @param context context where to report error messages
	* @param config config contains the compiler configuration
	* including options
	*/
	DartUnit exec(DartUnit unit, DartCompilerContext context,
		CoreTypeProvider typeProvider, CompilerConfiguration config);
}
```

Since the `TypeAnalyzer` class implements `DartCompilationPhase`, I changed the `exec` method signature in `TypeAnalyzer.java`. Inside the `exec` method, the `CompilerConfiguration` object is passed to the `Analyzer` object.

```
public DartUnit exec(DartUnit unit, DartCompilerContext context,
                         CoreTypeProvider typeProvider,
                         CompilerConfiguration config) {

	unit.accept(new Analyzer(context, typeProvider,
                    unimplementedElements, diagnosedAbstractClasses, config));
	return unit;
}
```

I added a `CompilerConfiguration` parameter to `analyze` method in `TypeAnalyzer.java`. Because this method creates an object of the `Analyzer` class, a private class in `TypeAnalyzer.java`, `CompilerConfiguration` object is passed to `Analyzer.`

```
public static Type analyze(DartNode node,
            CoreTypeProvider typeProvider,
            DartCompilerContext context,
            InterfaceType currentClass,
            CompilerConfiguration config) {

      ConcurrentHashMap<ClassElement, List<Element>> unimplementedElements =
                        new ConcurrentHashMap<ClassElement, List<Element>>();

      Set<ClassElement> diagnosed = Collections.newSetFromMap(new
                        ConcurrentHashMap<ClassElement, Boolean>());

      Analyzer analyzer = new Analyzer(context, typeProvider,
                        unimplementedElements, diagnosed, config);

      analyzer.setCurrentClass(currentClass);
      return node.accept(analyzer);
}
```

In `TypeAnalyzer.visitVariableStatement` method, I added a check to see if the `--must-have-types` option is given and then a check for type `DYNAMIC`. For variables `TypeKind.of(type)` is `DYNAMIC` (`type` is `<dynamic>`) and for `Lists` with no generic type, type is `<<dynamic>>`.

```
@Override
public Type visitVariableStatement(DartVariableStatement node) {

      Type type = typeOf(node.getTypeNode());
      if (compilerConfiguration.mustHaveTypes()) {

            if(TypeKind.of(type).toString().equals("DYNAMIC")
                  || type.toString().contains("dynamic")) {

            typeError(node,TypeErrorCode.DYNAMIC_TYPING_NOT_ALLOWED, node);
            }
      }
      visit(node.getVariables());
      return type;
}
```

In `DartCompilerMainContext.onError` method, I added an if condition to check if `TypeErrorCode` of the error is `DYNAMIC_TYPING_NOT_ALLOWED` and if yes, it checks if the `--must-have-types` compiler flag is on or not. If yes, then type error count is incremented.

```
@Override
public void onError(DartCompilationError event) {
     if(event.getErrorCode().equals(TypeErrorCode.DYNAMIC_TYPING_NOT_ALLOWED))
     {
            if (compilerConfiguration.mustHaveTypes()) {
```

```
                incrementTypeErrorCount();
            }
        } else if (event.getErrorCode().getSubSystem() == SubSystem.STATIC_TYPE) {
                incrementTypeErrorCount();

        } else if (shouldWarnOnNoSuchType() && event.getErrorCode() ==
                                            ResolverErrorCode.NO_SUCH_TYPE) {
                incrementTypeErrorCount();

        } else if (event.getErrorCode().getErrorSeverity() == ErrorSeverity.ERROR) {
                incrementErrorCount();

        } else if (event.getErrorCode().getErrorSeverity() == ErrorSeverity.WARNING) {
                incrementWarningCount();
        }
        listener.onError(event);
    }
```

The `Resolver.java` file also contains a class `Phase` which implements `DartCompilationPhase`. I changed the `exec` method signature in that class also.

Following is the code to illustrate how `--must-have-types` works:

```
        // This is where the app starts executing.
        main() {
            // String
            var name = 'Bob';

            // using final
            final f_name = "Alice";

            // numbers
            // int
            var count;

            // double
            var d_count = 0.0;

            // boolean
            var value = true;

            // In dart, arrays are list objects
            var list = [1, 2, 3];

            // Maps
            var gifts = {
                "first"  : "partridge",
                "second" : "turtledoves",
                "fifth"  : "golden rings"};

            var map = new Map();

            // Collections without generic type
            List names = new List();
        }
```

On compiling the above code with `--must-have-types` and `--fatal-type-errors` I got the following errors:

```
no_types.dart/no_types.dart:5: dynamic typing is not allowed "var name = "Bob";"
       4:   // String
       5:   var name = 'Bob';
              ~~~~~~~~~~~~~~~~

no_types.dart/no_types.dart:8: dynamic typing is not allowed "var f_name = "Alice";"
       7:   // using final
       8:   final f_name = "Alice";
            ~~~~~~~~~~~~~~~~~~~~~~

no_types.dart/no_types.dart:16: dynamic typing is not allowed "var count;"
              15:   // int
              16:   var count;
                   ~~~~~~~~~~

no_types.dart/no_types.dart:19: dynamic typing is not allowed "var d_count = 0.0;"
              18:   // double
              19:   var d_count = 0.0;
                   ~~~~~~~~~~~~~~~~~~

no_types.dart/no_types.dart:22: dynamic typing is not allowed "var value = true;"
              21:   // boolean
              22:   var value = true;
                   ~~~~~~~~~~~~~~~~~

no_types.dart/no_types.dart:25: dynamic typing is not allowed "var list = [1, 2, 3];"
              24:   // In dart, arrays are list objects
              25:   var list = [1, 2, 3];
                   ~~~~~~~~~~~~~~~~~~~~~~

no_types.dart/no_types.dart:28: dynamic typing is not allowed "var gifts = {"first" :
"partridge", "second" : "turtledoves", "fifth" : "golden rings"};"
              27:   // Maps
              28:   var gifts = {
                    ~~~~~~~~~~~~~

no_types.dart/no_types.dart:33: dynamic typing is not allowed "var map = new Map();"
              32:
              33:   var map = new Map();
                    ~~~~~~~~~~~~~~~~~~~~

no_types.dart/no_types.dart:36: dynamic typing is not allowed "List names = new
List();"
              35:   // Collections without generic type
              36:   List names = new List();
                    ~~~~~~~~~~~~~~~~~~~~~~~~

Compilation failed with 9 problems.
```

6. Type Checker for Dart Generic Type Inferences

6.1 Reified Generics (In Support of Optional Types)

Generics are types with type parameters. Dart supports reified generics. That means the type parameter can be accessed at run time. The generic type given to a collection through its constructor is assigned to the collection at run time. This concept supports Dart's theory of optional types. That means you could just say `new List()` if you don't ever want to use any types in your code.

Suppose I have the following collection where `p1`, `p2` and `p3` are `Person` objects:

```
List<Person> persons = <Person>[p1, p2, p3];
```

Then the type parameter can be checked at runtime:

```
persons is List<Person> // Returns true.
```

Also interestingly for the following collection:

```
List personCodes = [p1, p2, p3];
```

`personCodes is List<Person>` that is `new List() is List<Person>` returns true. This is supported because there might be instances where you have to integrate your untyped code with typed libraries and in that case an object of generic type without the type parameter is considered subtype of any other version of that generic [**8**]. However, this is unsound because `personCodes is List<String>` also returns true. This is another example of unsoundness of Dart's type system.

6.2 Covariance and Contravariance in Dart

In simple terms, if `A` is subtype of `B`, then `G<A> <: G<B>` is called covariance. If `G` is a collection, covariance is sound for reading but unsound for writing. For example, `List<Employee>` can be passed where `List<Person>` is required for reading. If you are expecting to read from a `List` of `Person`, you can read from a `List` of `Employee` because an `Employee` is a `Person`. Immutable collections are an example of covariant generics. On the other hand, if `A` is subtype of `B`, then, `G<B> <: G<A>` is called contravariance. For example, `List<Person>` can be passed where `List<Employee>` is required for writing. If you want to add an element to `List<Employee>`, `List<Person>` can hold it because an Employee is a Person. The type of a function is contravariant on its argument type.

The Dart type system is not able to distinguish between covariance and contravariance rules. Since Dart type annotations are supposed to be simple, lightweight and optional, Dart has decided to support covariance and not support contravariance. The reason for that is, Dart thinks

most of the uses of generics are read-only and they act covariantly [**5**]. For the same reason, Dart has made its generics covariant instead of invariant, supporting its optimistic nature. So Dart generics are covariant. What that means is, if `A` is subtype of `B`, and if you pass a `List<A>` to a function expecting `List<B>` and if that function just reads from the list then the list is used covariantly. Dart's support of covariant generics is similar to Java arrays except that when trying to store into a covariant collection Java throws an ArrayStoreException but Dart does not give any exception. Instead, an exception is thrown when the falsely stored object is used.

The following is an example of covariance:

```
Person getFirst(List<Person> list) {
	return list[0];
}

main() {
	List<Employee> elist = new List<Employee>(10);
	getFirst(elist);
}
```

The above code passes Dart's static check because of assignment rule (it is acceptable if `Person` is subtype of `Employee` or `Employee` is subtype of `Person`), passes Dart's dynamic check (in checked mode) because Dart supports covariant generics and runs correctly.

In case of the below code, list is used contravariantly, so static checking passes (again because assignment is symmetric), dynamic checking fails but runs without error if the dynamic checking is off.

```
void addAnEmployee(List<Employee> list> {
	list.add(new Employee());
}

main() {
	List<Person> plist = new List<Person>(10);
	addAnEmployee(plist);
}
```

A workaround to allow contravariance is, for the method `addAnEmployee,` instead of `List<Employee>` as the argument, `List` with no generic type can be made as argument. That allows any list and passes the dynamic check and code runs without aborting. But that will introduce a scope of error because if it is a `List` what if someone passes a `List<int>`? The program then fails at runtime.

Dart supports covariant generics assuming most of the cases are read only. So what happens if the function writes to the collection? Taking the previous example of covariance, consider there is an add statement like in the code below:

```
Person getAPerson(List<Person> list) {
```

```
        list.add(new Student());
        return list[0];
    }

    main() {
        List<Employee> elist = new List<Employee>(10);
        getAPerson(elist);
    }
```

In the above code add executes fine because `Student` is a `Person`, but now the `Employee` list contains a `Student`. There will be no problem until it encounters another method call which is probably calling a particular method of `Employee` class on each element of `Employee` list which `Student` is not aware of and then program fails.To avoid that problem, I created a variance static checker which issues a warning at the compile time if the collection is being modified inside the method. The next section contains the code changes that I have done for variance static checker.

6.3 Variance Checker without Annotation

`TypeAnalyzer.visitUnqualifiedInvocation` method is called in the backend when a method is invoked in Dart source code. `MethodElementImplementation.java` is the file which contains the called method details. It stores type of the function. It also has modifiers, holder, kind and parameters as final fields. To get the method definition from this class I made the following code modifications to `MethodElementImplementation.java`: I changed the visibility of `MethodElementImplementation.java` to public and added the following code to `MethodElementImplementation` class.

```
    private DartMethodDefinition node;

    public DartMethodDefinition getNode() {
        return node;
    }

    public void setNode(DartMethodDefinition node) {
        this.node = node;
    }

    protected MethodElementImplementation(DartMethodDefinition node,
                                          String name, EnclosingElement holder) {
        super(node, name);

        this.node = node;
        if (node != null) {
        modifiers = node.getModifiers();
        } else {
        modifiers = Modifiers.NONE;
        }
        this.holder = holder;
        this.kind = ElementKind.METHOD;
    }
```

I added two private instance variables to `Analyzer` class in `TypeAnalyzer.java`.

```
private boolean method_definition_modifies_collection;
private final CompilerConfiguration compilerConfiguration;
```

To get the method definition, in `TypeAnalyzer.visitUnqualifiedInvocation` method, in `case METHOD,` I added the below code:

```
public Type visitUnqualifiedInvocation(DartUnqualifiedInvocation node) {
       DartIdentifier target = node.getTarget();
       String name = target.getTargetName();
       Element element = target.getTargetSymbol();
       node.setReferencedElement(element);
       Type type;

       switch (ElementKind.of(element)) {
              case FIELD:
                     type = typeAsMemberOf(element, currentClass);
                     break;

              case METHOD:
                     MethodElementImplementation member_method =
                     (MethodElementImplementation)element;

                     visitMethodNode(member_method.getNode());
                     type = typeAsMemberOf(element, currentClass);
                     break;

              case NONE:
                     return typeError(target, TypeErrorCode.NOT_A_METHOD_IN,
                                          name, currentClass);
              default:
                     type = element.getType();
                     break;
       }
       return checkInvocation(node, target, name, type);
}
```

In the above code I am sending the method definition to the `TypeAnalyzer.visitMethodNode` method which I added. Below is the code of `visitMethodNode` method which basically parses through each statement of the method and if the statement is a method invocation, it pulls out the function name and currently if it is `"add", "addAll"` or `"addLast"` (hardcoded) it sets a boolean variable to true.

```
public void visitMethodNode(DartNode node) {
       if (node != null) {
       DartMethodDefinition methodNode = (DartMethodDefinition)node;

       // Added this to get the param nodes for @notnull checks
       functionParams = methodNode.getFunction().getParameters();

       DartFunction function = methodNode.getFunction();
```

```
	if (function != null) {
		DartBlock block = function.getBody();

		if (block != null) {
		List<DartStatement> methodStatements = block.getStatements();

		for (int i = 0; i < methodStatements.size(); i++) {
			DartNode stmt = methodStatements.get(i);

			if (stmt.getClass().toString().equals("class
				com.google.dart.compiler.ast.DartExprStmt"))
			{
			DartExprStmt exprStmt = (DartExprStmt)stmt;
			DartExpression expression = exprStmt.getExpression();

			if (expression.getClass().toString().equals("class
			com.google.dart.compiler.ast.DartMethodInvocation")) {

			DartMethodInvocation methodExpression =
						(DartMethodInvocation)expression;
			String methodCalledOn = methodExpression.getTarget().toString();

			String functionName = methodExpression.getFunctionNameString();

			for(DartParameter param : functionParams) {

			DartTypeNode type = param.getTypeNode();
			if (type != null) {
			if (param.getParameterName().equals(methodCalledOn)
			&& param.getTypeNode().getIdentifier().toString().equals("List")
			&& (functionName.equals("add") || functionName.equals("addAll")
			|| functionName.equals("addLast"))) {
				method_definition_modifies_collection = true;
			 }
			 }
			 }
			 }
			 }
		}
		}
	}
	}
    }
```

In `TypeAnalyzer.checkArguments` method, I modified the call of the `checkAssignable` method to the `checkMethodInvocationValid` that I added to `TypeAnalyzer.java`.

The following is the code of the `checkArguments` method:

```
private Type checkArguments(DartNode diagnosticNode,
                        List<? extends DartExpression> argumentNodes,
                        Iterator<Type> argumentTypes, FunctionType ftype) {

	int argumentCount = 0;
	List<? extends Type> parameterTypes = ftype.getParameterTypes();
```

```
        for (Type parameterType : parameterTypes) {
               if (argumentTypes.hasNext()) {

               checkMethodInvocationValid(argumentNodes.get(argumentCount),
                                                  parameterType, argumentTypes.next());
               argumentCount++;
               } else {
               typeError(diagnosticNode,TypeErrorCode.MISSING_ARGUMENT,parameterType);
               }
        }
        Map<String, Type> namedParameterTypes = ftype.getNamedParameterTypes();
        Iterator<Type> named = namedParameterTypes.values().iterator();

        while (named.hasNext() && argumentTypes.hasNext()) {
               checkMethodInvocationValid(argumentNodes.get(argumentCount),
                                                  named.next(), argumentTypes.next());
               argumentCount++;
        }

        while (ftype.hasRest() && argumentTypes.hasNext()) {
               checkMethodInvocationValid(argumentNodes.get(argumentCount),
                                                  ftype.getRest(), argumentTypes.next());
               argumentCount++;
        }

        while (argumentTypes.hasNext()) {
               argumentTypes.next();
               typeError(argumentNodes.get(argumentCount),TypeErrorCode.EXTRA_ARGUMENT);
               argumentCount++;
        }

        return ftype.getReturnType();
}
```

The following is the code of the `checkMethodInvocationValid` method:

```
        private void checkMethodInvocationValid(DartNode node, Type t, Type s) {
               t.getClass();
               s.getClass();

               if(!types.isMethodInvocationValid(t, s) &&
                                          method_definition_modifies_collection) {
                      typeError(node, TypeErrorCode.METHOD_INVOCATION_NOT_VALID, s, t);
                      method_definition_modifies_collection = false;
               }
        }
```

The `checkMethodInvocationValid` method checks if argument `s` is a subtype of parameter `t` and if the `method_definition_modifies_collection` is true. If yes then a type error has occurred and a warning will be issued if the `--fatal-type-errors` flag is off or an error will be issued if the `--fatal-type-errors` flag is on .

Below is the code of the `Types.isMethodInvocationValid` method which is called by the `checkMethodInvocationValid` method above.

```
public boolean isMethodInvocationValid(Type t, Type s) {
	return isSubtype(t, s);
}
```

I added a new error in `TypeErrorCode.java` to throw when the method definition modifies the collection.

```
METHOD_INVOCATION_NOT_VALID("Method definition modifies the collection."),
```

Below is the code to illustrate the covariance check I added above:

```
class Person {}
class Student extends Person {}
class Employee extends Person {}

main() {
	List<Employee> elist = new List<Employee>();

	readingFunction(elist);

	writingFunction(elist);
}

readingFunction(List<Person> p) {
	Person p = p[0];
}

writingFunction(List<Person> p) {
	Student s = new Student();
	p.add(s);
}
```

On compiling this code, we get the below warning:

```
variance.dart/variance.dart:10: Method definition modifies the collection.
9:
10:    writingFunction(elist);
```

6.4 Variance Checker with `@modifies` Annotation

The first version of the variance checker I created parses through all the statements of the method definition to check if the method definition contains any statements that call the collection's mutation methods. I introduced an annotation called `@modifies`, which when given with the method can be used as a metadata to the method node to specify that method does modify the

collection passed covariantly. With the addition of this annotation, there is no need to parse through the method definition to check if the method modifies the collection provided the programmer annotates the method with the `@modifies` annotation.

The following are the code modifications done to support `@modifies` annotation with procedures or functions.

I added an enumeration constant as a reserved keyword, `MODIFIES("modifies", 0),` to `Token` enum class in `Token.java`.

In the `DartParser.parseFieldOrMethod` method, I added the check for `@modifies` annotation.

```
private DartNode parseFieldOrMethod(boolean allowStatic) {
       .....
       // parsing @modifies annotation for method.
       boolean modifies = false;
       if (peek(0) == Token.AT_TOKEN) {
              if (expect(Token.AT_TOKEN) && expect(Token.MODIFIES)) {
                     modifies = true;
              }
       }
       .....
       case IDENTIFIER: {
              if (peek(1) == Token.LPAREN ||
                     peek(1) == Token.PERIOD ||
                     peekPseudoKeyword(0, OPERATOR_KEYWORD) ||
                     peekPseudoKeyword(0, GETTER_KEYWORD) ||
                     peekPseudoKeyword(0,SETTER_KEYWORD)) {
                     member = parseMethodOrAccessor(modifiers, null, modifies);
                     break;
              }
              ....
              if (peek(1) == Token.SEMICOLON || peek(1) == Token.COMMA
                                                 || peek(1) == Token.ASSIGN) {
              .....
              } else {
                     member = parseMethodOrAccessor(modifiers, type, modifies);
              }
       }
}
```

I added the boolean parameter to `parseMethodOrAccessor` method in `DartParser.java`.

```
private DartNode parseMethodOrAccessor(Modifiers modifiers,
                                DartTypeNode returnType, boolean modifies) {
       DartMethodDefinition method = done(parseMethod(modifiers, returnType,
                                                                  modifies));
       ...
}
```

I added the boolean parameter to `parseMethod` in `DartParser.java`.

```
private DartMethodDefinition parseMethod(Modifiers modifiers,
                                DartTypeNode returnType, boolean modifies){
       ...
       return DartMethodDefinition.create(name, function, modifiers,
                                                initializers, null, modifies);
}
```

I made the following changes to `DartMethodDefinition.java`. I added a new protected boolean field `modifies` and a new method `doesModify` to `DartMethodDefintion.java`.

```
protected boolean modifies;

public boolean doesModify() {
       return modifies;
}
```

I overloaded the `create` method in `DartMethodDefinition.java` with a new boolean `modifies` parameter.

```
public static DartMethodDefinition create(DartExpression name,
                                        DartFunction function,
                                        Modifiers modifiers,
                                        List<DartInitializer> initializers,
                                        List<DartTypeParameter> typeParameters,
                                        boolean modifies) {
       if (initializers == null) {
              return new DartMethodDefinition(name, function, modifiers,
                                                    typeParameters, modifies);
       } else {
              return new DartMethodWithInitializersDefinition(name, function,
                                        modifiers, initializers, modifies);
       }
}
```

I overloaded the constructor of `DartMethodDefinition` class and the constructor of `DartMethodWithInitializersDefintion` which is a static nested class of `DartMethodDefinition` class.

```
private DartMethodDefinition(DartExpression name,
                                DartFunction function,
                                Modifiers modifiers,
                                List<DartTypeParameter> typeParameters,
                                boolean modifies) {
       super(name, modifiers);
       this.function = becomeParentOf(function);
       this.typeParameters = typeParameters;
       this.modifies = modifies;
}

DartMethodWithInitializersDefinition(DartExpression name,
                                DartFunction function,
                                Modifiers modifiers,
```

```
                                    List<DartInitializer> initializers,
                                    boolean modifies) {
        super(name, function, modifiers, null);
        this.initializers = becomeParentOf(initializers);
        this.modifies = modifies;
    }
```

In `DartParser.parseMethod` method I changed the return statement to call the `create` method of `DartMethodDefinition` with the boolean argument.

```
    private DartMethodDefinition parseMethod(Modifiers modifiers,
                                             DartTypeNode returnType,
                                             boolean modifies){
        ...
        return DartMethodDefinition.create(name, function, modifiers,
                                                 initializers, null, modifies);
    }
```

In `TypeAnalyzer.visitUnqualifiedInvocation` method `case METHOD`, `visitMethodNode` method was called in the variance checker without annotation to parse the method definition. For this variance checker using `@modifies` annotation, just a check for the annotation has been added. Also to ensure the covariance of collections, I added a check to ensure the parameter is a collection (List).

```
    public Type visitUnqualifiedInvocation(DartUnqualifiedInvocation node) {
    ...
    case METHOD:
    // Have to get the list of parameters for notnull check
        MethodElementImplementation member_method =
                                             (MethodElementImplementation)element;
        DartNode mNode = member_method.getNode();

        if (mNode != null) {
            DartMethodDefinition methodNode = (DartMethodDefinition)mNode;
            functionParams = methodNode.getFunction().getParams();
            boolean isParamList = false;

            // For now we will put the @modifies only for lists in collection.
            for (DartParameter p : functionParams) {
            if (p.getTypeNode().getIdentifier().toString().equals("List")) {
                isParamList = true;
            }
            }
        if (methodNode.doesModify() && isParamList)
            method_definition_modifies_collection = true;
        }
        type = typeAsMemberOf(element, currentClass);
        break;
    ....
    }
```

In `TypeAnalyzer.checkMethodInvocationValid` method, I separated the check for parameter and argument assignment and `@modifies` annotation.

```
private void checkMethodInvocationValid(DartNode node, Type t, Type s) {

	if(!types.isMethodInvocationValid(s, t)) {
	typeError(node, TypeErrorCode.TYPE_NOT_ASSIGNMENT_COMPATIBLE, s, t);
	}
	if (method_definition_modifies_collection) {
		typeError(node, TypeErrorCode.METHOD_INVOCATION_NOT_VALID, s, t);
		method_definition_modifies_collection = false;
	}
}
```

I also ensured that collections can be passed only covariantly by modifying the `isMethodInvocationValid` method in `Types.java`.

```
public boolean isMethodInvocationValid(Type t, Type s) {
	return isSubtype(t, s);
}
```

Below is the source file on which the variance checker has been checked.

```
class Person {}
class Student extends Person {}
class Employee extends Person {}

main() {
	List<Employee> elist = new List<Employee>();

	readingFunction(elist);

	writingFunction(elist);
	stringFunction("a");
	contravariantFunction(new List<Person>());
}

readingFunction(List<Person> p) {
	Person p = p[0];
}

@modifies writingFunction(List<Person> p) {
	Student s = new Student();
	p.add(s);
}

@modifies stringFunction(String s){
}

@modifies contravariantFunction(List<Student> p){
}
```

Below is the output for the variance checker on the above source code.

```
variance.dart/variance.dart:10: Method definition modifies the collection.
	9:
```

```
      10:    writingFunction(elist);
                             ~~~~~
variance.dart/variance.dart:12: List<Person> is not assignable to List<Student>
      11:    stringFunction("a");
      12:    contravariantFunction(new List<Person>());
                                   ~~~~~~~~~~~~~~~~~~
variance.dart/variance.dart:12: Method definition modifies the collection.
      11:    stringFunction("a");
      12:    contravariantFunction(new List<Person>());
                                   ~~~~~~~~~~~~~~~~~~
```

7. Nullness Checker with `@notnull` Annotation

The `@notnull` annotation can be used to ensure that no null values can be passed around in the source program. Dart program that passed the Nullness Checker never throws a null pointer exception. For this version of Dart language supported by the dartc compiler, I have added the support for the `@notnull` annotation for method parameters and identifiers. If a method parameter is annotated as `@notnull`, only a `@notnull` annotated identifier can be passed to the method without any warning.

The following are the changes made to the code to support `@notnull` annotation for method parameters and identifiers.

I added enumeration constants to `Token` enum class in `Token.java`.

```
public enum Token {
...

/* Punctuators. */
AT_TOKEN("@", 0),
...

/* Keywords. */
NOTNULL("notnull", 0),
...
}
```

In the Dart language, '@' token is already used to annotate raw strings. But the second character after the '@' is either a single quote or a double quote for raw strings. I changed the `scanToken` method in `DartScanner.java` to support the annotations which are basically '@' followed by a keyword.

```
case '@':
       // Raw strings.
advance();

       if (is('\'') || is('"')) {
              boolean isRaw = true;
```

```
            return scanString(isRaw);
        }

        // changing from ILLEGAL to AT_TOKEN
        return Token.AT_TOKEN;
```

The `notnull` keyword is scanned as identifier, then figured as keyword token. The same process has been followed for `@modifies` annotation for variance checker.

In `DartParser.parseFormalParameter` method, I added a check for `@notnull` annotation. In the method, I added a boolean local variable `notNull`. At the end of the method while returning a `DartParameter` object, I changed the call to new overloaded constructor of `DartParameter` class which takes the `notNull` boolean.

```
    private DartParameter parseFormalParameter(boolean isNamed) {
        boolean notNull = false;
        ...
        if (peek(0) == Token.AT_TOKEN) {
            if (peek(1) == Token.NOTNULL) {
              if (expect(Token.AT_TOKEN) && expect(Token.NOTNULL)) {
                  notNull = true;
              }
            }
        }
        ...
        return done(new DartParameter(paramName, type, functionParams,
                                      defaultExpr, modifiers, notNull));

    }
```

I overloaded the `DartParameter` constructor to take a boolean parameter.

```
    public DartParameter(DartExpression name,
                         DartTypeNode typeNode,
                         List<DartParameter> functionParameters,
                         DartExpression defaultExpr,
                         Modifiers modifiers,
                         boolean notNull) {
        super(name);
        ...
        this.notNull = notNull;
    }
```

In `DartNode.java`, I added a protected instance field `notNull` because `DartNode.java` is the abstract class which is extended by other abstract syntax tree nodes.

```
    public abstract class DartNode extends AbstractNode implements DartVisitable {
        ...
        protected boolean notNull = false;
        ...

        public boolean isNotNull() {
```

```
                return notNull;
        }
        ...
    }
```

I added a private instance variable `functionParams` to `Analyzer` class in `TypeAnalyzer.java`.

```
    private List<DartParameter> functionParams;
```

In the constructor of `Analyzer`, I added the statement `this.functionParams = null;`.

In `visitUnqualifiedInvocation` method `case METHOD` in `TypeAnalyzer.java`, `functionParams` is assigned from `methodNode`.

```
    public Type visitUnqualifiedInvocation(DartUnqualifiedInvocation node) {
    ...
    case METHOD:
        // Have to get the list of parameters for notnull check
        MethodElementImplementation member_method =
                                        (MethodElementImplementation)element;
        DartNode mNode = member_method.getNode();

        if (mNode != null) {
            DartMethodDefinition methodNode = (DartMethodDefinition)mNode;
            functionParams = methodNode.getFunction().getParams();
            boolean isParamList = false;

            // For now we will put the @modifies only for lists in collection.
            for (DartParameter p : functionParams) {
            if (p.getTypeNode().getIdentifier().toString().equals("List")) {
                isParamList = true;
            }
            }

        if (methodNode.doesModify() && isParamList)
            method_definition_modifies_collection = true;
        }
        type = typeAsMemberOf(element, currentClass);
        break;
    ....
}
```

I added an extra `DartParameter` parameter to `checkMethodInvocationValid` method in `TypeAnalyzer.java` and added a check for `notNull` for both parameter and argument.

```
private void checkMethodInvocationValid(DartParameter param, DartNode node,
                                                        Type t, Type s) {
    t.getClass();  // Null Check
    s.getClass();   // Null Check

    if (param.isNotNull() && !node.isNotNull()) {
        typeError(node, TypeErrorCode.NULL_ASSIGNED_TO_NOTNULL_PARAMETER, s, t);
    }
```

```
        if(!types.isMethodInvocationValid(s, t)) {
                typeError(node, TypeErrorCode.TYPE_NOT_ASSIGNMENT_COMPATIBLE, s, t);
        }

        if (method_definition_modifies_collection) {
                typeError(node, TypeErrorCode.METHOD_INVOCATION_NOT_VALID, s, t);
                method_definition_modifies_collection = false;
        }
}
```

In the `TypeAnalyzer.checkArguments` method, I modified the call to `checkMethodInvocation` by passing an extra `DartParameter` argument.

```
private Type checkArguments(DartNode diagnosticNode,
                                List<? extends DartExpression> argumentNodes,
                                Iterator<Type> argumentTypes,
                                FunctionType ftype) {
        int argumentCount = 0;
        List<? extends Type> parameterTypes = ftype.getParameterTypes();

        for (Type parameterType : parameterTypes) {
                if (argumentTypes.hasNext()) {
                checkMethodInvocationValid(functionParams.get(argumentCount),
                                                argumentNodes.get(argumentCount),
                                                parameterType,
                                                argumentTypes.next());
                argumentCount++;
                } else {
                typeError(diagnosticNode, TypeErrorCode.MISSING_ARGUMENT, parameterType);
                }
       }
        Map<String, Type> namedParameterTypes =       ftype.getNamedParameterTypes();
        Iterator<Type> named = namedParameterTypes.values().iterator();

        while (named.hasNext() && argumentTypes.hasNext()) {
                checkMethodInvocationValid(functionParams.get(argumentCount),
                                                argumentNodes.get(argumentCount),
                                                named.next(),
                                                argumentTypes.next());
                argumentCount++;
        }
        while (ftype.hasRest() && argumentTypes.hasNext()) {
                checkMethodInvocationValid(functionParams.get(argumentCount),
                                                argumentNodes.get(argumentCount),
                                                ftype.getRest(),
                                                argumentTypes.next());
                argumentCount++;
        }

        while (argumentTypes.hasNext()) {
                argumentTypes.next();
                typeError(argumentNodes.get(argumentCount),TypeErrorCode.EXTRA_ARGUMENT);
                argumentCount++;
       }
       return ftype.getReturnType();
```

```
}
```

I added a new error code to `TypeErrorCode.java` to support `@notnull` annotation.

```
NULL_ASSIGNED_TO_NOTNULL_PARAMETER("Null value might be assigned to not null
parameter"),
```

To support `@notnull` annotation to identifiers, following are the changes made to the code. In the `DartParser`, below is the flow of code for `DartVariable` which includes parsing an identifier.

```
parseStatement -> parseNonLabelledStatement ->
parseInitializedVariableList -> parseIdentifier
```

In the `parseNonLabelledStatement` method `case IDENTIFIER`, I added a check for `@notnull` annotation, added a local boolean variable `notNull` and added an extra parameter to `parseInitializedVariableList` to pass `notNull` value.

```
private DartStatement parseNonLabelledStatement() {
       ...
       case IDENTIFIER:
              boolean notNull = false;
              if (peek(1) == Token.LT || peek(1) == Token.IDENTIFIER
                     || (peek(1) == Token.PERIOD && peek(2) == Token.IDENTIFIER)
                     || (peek(0) == Token.AT_TOKEN)) {

              if (peek(0) == Token.AT_TOKEN) {
                     if (peek(1) == Token.NOTNULL) {
                     if (expect(Token.AT_TOKEN) && expect(Token.NOTNULL)) {
                            notNull = true;
                     }
                     }
              }

              if (peek(1) == Token.LT || peek(1) == Token.IDENTIFIER
              || (peek(1) == Token.PERIOD && peek(2) == Token.IDENTIFIER)) {

              beginTypeFunctionOrVariable();
              DartTypeNode type = tryTypeAnnotation();

              if (type != null && peek(0) == Token.IDENTIFIER) {
              List<DartVariable> vars = parseInitializedVariableList(notNull);
              expect(Token.SEMICOLON);
              return done(new DartVariableStatement(vars, type));
              } else {
              rollback();
              }
              }
       }
       }
...
}
```

I added a boolean parameter to `parseInitializedVariableList` method in `DartParser.java` and passed it to `parseIdentifier` method.

```
private List<DartVariable> parseInitializedVariableList(boolean notNull) {
	List<DartVariable> idents = new ArrayList<DartVariable>();
	do {
		beginVariableDeclaration();
		DartIdentifier name = parseIdentifier(notNull);
		DartExpression value = null;
		if (isParsingInterface) {
			expect(Token.ASSIGN);
			value = parseExpression();
		} else if (optional(Token.ASSIGN)) {
			value = parseExpression();
		}

		idents.add(done(new DartVariable(name, value)));
	} while (optional(Token.COMMA));
	return idents;
}
```

I added an overloaded `parseIdentifier` method with a boolean `notNull` parameter to `DartParser.java`.

```
	private DartIdentifier parseIdentifier(boolean notNull) {
		beginIdentifier();
		expect(Token.IDENTIFIER);
		String token = ctx.getTokenString();

		if (notNull || notNullIdentifiers.contains(token)) {
			if (notNull) {
				notNullIdentifiers.add(token);
			}
		return done(new DartIdentifier(token != null ? token : "", notNull));
		}
		return done(new DartIdentifier(token != null ? token : ""));
	}
```

The `notNullIdentifiers` in the above method is a hash set that I added to `DartParser.java` which can be used as a local symbol table to store the identifiers annotated with `@notnull`. After a `DartVariable` is parsed in the variable declaration statement, when it is used in method call as an argument, it is again parsed as a `DartIdentifier` so it does not remember the `@notnull` annotation used before while declaring. To simplify the symbol table access, I added a hash set to the `DartParser` to store the `@notnull` annotated identifiers and added a check in the `parseIdentifier` to check if the current identifier is annotated `@notnull` or if it has been declared `@notnull` before hand.

In `DartParser.java`, added a new HashSet.

```
private Set<String> notNullIdentifiers = new HashSet<String>();
```

The overloaded `parseIdentifier` method is called only when parsing an initialized variable list. Other places where the identifier is being used, like as an argument to a method, the regular `parseIdentifier` method is called which does not take any arguments. So in that method, it is sufficient to check if the identifier is in `notNullIdentifiers` collection or not.

```
private DartIdentifier parseIdentifier() {
	beginIdentifier();
	expect(Token.IDENTIFIER);
	String token = ctx.getTokenString();
	boolean notNull = notNullIdentifiers.contains(token);
	if (notNull) {
	return done(new DartIdentifier(token != null ? token : "", notNull));
	}
	return done(new DartIdentifier(token != null ? token : ""));
}
```

I added an overloaded constructor to `DartIdentifier` class to take a boolean parameter to set the `notNull` variable declared in its superclass `DartNode`.

```
public DartIdentifier(String targetName, boolean notNull) {
	assert targetName != null;
	this.targetName = targetName;
	this.notNull = notNull;
}
```

Below is the Dart source code on which the nullness check has been checked.

```
main() {
	String hello = "Hello";
	@notnull String world = "World";
	printWord(hello);
	printWord(world);
}

printWord(@notnull String pWord) {
}
```

For the above code, below is the output of the compiler with a warning.

```
notnull.dart/notnull.dart:4: Null value might be assigned to not null parameter
	3:	@notnull String world = "World";
	4:	 printWord(hello);
	                  ~~~~~
```

8. Linear Checker with `@linear` Annotation

The linear checker implements type-checking for a linear type system [**11**]. A linear type system prevents the unexpected modification of objects by restricting only one usable reference to an

object at any point of time. After a reference to an object appears on the right hand side of an assignment, it is used up and cannot be used again. The same rule applies for pseudo-assignments such as procedure-argument passing (including as the receiver) or return [**11**]. The linear checker property can also be used in support of Dart Isolates.

Dart supports concurrent programming by means of isolates. Isolates communicate with each other by passing messages. Isolates send messages using `SendPort` and receive messages using `ReceivePort`. `SendPort` and `ReceivePort` are classes in the dart:isolate library. The content of the message could be a primitive value, an instance of `SendPort`, a list or map whose elements are either primitive values or instances of `SendPort`. In special circumstances, objects of any type can be passed as content of the message. The important concept of the Dart isolates is that it does not support shared-memory threads. That means no two isolates can ever run the same thread at the same time. All the memory used by an isolate is only available to the isolate. No isolate can see or manipulate the memory owned by another isolate. This is accomplished by copying the message before it is received ensuring that two isolates have different instances of the object in the message. The following are the examples of spawning an isolate, sending and receiving messages.

```
import 'dart:isolate';

codeToRunInIsolate() {
	print('This code is running in an isolate.');
}

// main function runs in the first isolate
main() {

	// spawnFunction method creates and spawns an isolate.
	spawnFunction(codeToRunInIsolate);

}
```

In the above code, `main` function itself is running in the first isolate. Using the `spawnFunction`, another isolate is being created and started. In the above code `codeToRunInIsolate` is known as the entry point of the second isolate. As per the Dart rules, the entry point should not expect arguments and should return void.

```
import 'dart:isolate';

codeToRunInIsolate() {
	// Receive messages here.
	print('This code is running in an isolate.');
}

// main function runs in the first isolate
main() {

	// spawnFunction method creates and spawns an isolate.
	var sendPort = spawnFunction(codeToRunInIsolate);
```

```
        // sending a message to the new isolate.
        sendPort.send('This is a message from the main.');

}
```

In the above code, the `main` function grabs the `sendPort` object returned by `spawnFunction` and uses it to send a message to the newly created isolate using `send` method. The message will be received in the `codeToRunInIsolate` function.

```
import 'dart:isolate';

codeToRunInIsolate() {
        print('This code is running in an isolate.');

        port.receive((msg, reply) {
                print('Message received: $msg');
        });
}

// main function runs in the first isolate
main() {

        // spawnFunction method creates and spawns an isolate.
        var sendPort = spawnFunction(codeToRunInIsolate);

        // sending a message to the new isolate.
        sendPort.send('This is a message from the main.');

}
```

In the above code, the message sent by the `main` function is being received by the new isolate using the default `ReceivePort` object `port`. The message received is handled by the callback function passed to the `receive` method. It is also possible to receive a reply after sending a message as in below example.

```
import 'dart:isolate';

codeToRunInIsolate() {
        print('This code is running in an isolate.');

        port.receive((msg, reply) {
                reply.send('Message received: $msg');
        });
}

// main function runs in the first isolate
main() {

   // spawnFunction method creates and spawns an isolate.
   var sendPort = spawnFunction(codeToRunInIsolate);

   // sending a message to the new isolate and handling
  // the reply received
```

```
    sendPort.call('This is a message from the main.').then((reply {
        print('Reply received from new isolate: $reply');
    });
}
```

In the above code, the `main` function is sending a message to the new isolate and has given a handler for the reply message sent back by the new isolate. To send and receive reply, use `call` and `then` methods instead of just `send` method.

In all the above examples, the message content was a `String` primitive. When using `spawnFunction()` method inside the Dart VM, an object of any type can be passed as message content to an isolate. However, this is not yet supported when compiling to JavaScript.

Dart isolates is based on the concept of no-shared memory and hence message is copied before receiving. When an object is passed via messages to multiple isolates, copying of the messages will create multiple instances of that object. Considering the scenario, when a message contains an object of large size and at any point of time if that object can have only one reference, it is wasteful to copy that message and create another instance of that object. A linear type system could be helpful to avoid this scenario by not copying the messages with linear objects as content.

In this project, I have supported the `@linear` annotation for only method parameters as the first step. Following are the `@linear` annotation checks I supported:

- A parameter annotated with `@linear` can be assigned to a variable annotated with `@linear` only and vice versa.
- A parameter annotated with `@linear` can be assigned to a `@linear` variable only once. After it has been appeared on the right hand side of the assignment it can't be reassigned.

Code changes done to support `@linear` annotation are similar to the code changes done to support `@notnull` annotation. Please refer to Appendix B for specific code modification details of `@linear` annotation.

Below is the Dart source code on which the linear annotation checks have been tested.

```
class Pair {
       Object a;
       Object b;
}

linearCheckMethod(Pair o, @linear Pair lp) {
       Object o1 = o;    // both non-linear no error
       Object lo1 = lp;  // linear param not assignable to non-linear variable.
       @linear Object o2 = o;  // non-linear param not assignable to linear
                               // variable (it might have other references)
       @linear Pair lp2 = lp;  // both linear - no error first time
       @linear Pair lp3 = lp;  // already used linear param not reassignable
```

```
}

main() {
	Pair lp1 = new Pair();
	Pair o = new Pair();
	linearCheckMethod(o, lp1);
}
```

Below are the warnings issued by the dartc compiler supporting `@linear` annotation on the above code.

```
linear.dart/linear.dart:8: linear parameter "lp" is not assignable to non-linear
variable "lo1"
	7:	Object o1 = o;
	8:	Object lo1 = lp;
			~~~~~~~~
linear.dart/linear.dart:9: non linear parameter "o" is not assignable to linear
variable "o2"
	8:	Object lo1 = lp;
	9:	@linear Object o2 = o;
				~~~~~~
linear.dart/linear.dart:11: linear parameter "lp" has already been assigned to a
linear variable
	10:	@linear Pair lp2 = lp;
	11:	@linear Pair lp3 = lp;
				~~~~~~~~
```

9. Applicability

In this project, I have added these four type checkers to the initial version of dartc compiler which was released in October 2011. The current Dart SDK includes dart2js compiler written in Dart and a separate executable called dart_analyzer which is the enhanced version of dartc. It is no longer called a compiler because it does not generate any JavaScript code. dart_analyzer is a tool used for static analysis of code and hence the type checkers need to be added to dart_analyzer.

For the applicability testing of these type checkers, I have downloaded the current Dart source code (on December 17, 2012) and incorporated the mandatory types checker and the variance checker without annotation into the dart_analyzer. I ran the current modified version of dart_analyzer on the code samples provided in Dart source code download. The following is a sample run of mandatory types checker on the source code of solar application provided in Dart samples. Please refer to Appendix C for complete source code of original solar application before my modifications.

The following are the warnings given on running dart_analyzer on solar.dart with `--must-have-types` command line option like below

```
./dart_analyzer --must-have-types solar.dart
```

Output:

```
solar_b/solar.dart:18:3: dynamic typing is not allowed "var solarSystem = new
SolarSystem(query("#container"));"
       17: void main() {
       18:   var solarSystem = new SolarSystem(query("#container"));

solar_b/solar.dart:93:5: dynamic typing is not allowed "var earth = new
PlanetaryBody(this, "Earth", "#33f", 1.0, 1.0, 1.0);"
       92:
       93:     var earth = new PlanetaryBody(this, "Earth", "#33f", 1.0, 1.0, 1.0);

solar_b/solar.dart:101:5: dynamic typing is not allowed "var f = 0.1;"
       100:
       101:     final f = 0.1;

solar_b/solar.dart:102:5: dynamic typing is not allowed "var h = 1 /1500.0;"
       101:     final f = 0.1;
       102:     final h = 1 / 1500.0;

solar_b/solar.dart:103:5: dynamic typing is not allowed "var g = 1 / 72.0; "
       102:     final h = 1 / 1500.0;
       103:     final g = 1 / 72.0;

solar_b/solar.dart:105:5: dynamic typing is not allowed "var jupiter = new
PlanetaryBody(this, "Jupiter", "gray", 4.0, 5.203, 11.86);"
       104:
       105:     var jupiter = new PlanetaryBody(

solar_b/solar.dart:134:5: dynamic typing is not allowed "var context =
canvas.context2d;"
       133:
       134:     var context = canvas.context2d;

solar_b/solar.dart:161:7: dynamic typing is not allowed "var radius = 2.06 +
random.nextDouble() * (3.27 - 2.06);"
       160:     for (int i = 0; i < count; i++) {
       161:       var radius = 2.06 + random.nextDouble() * (3.27 - 2.06);

solar_b/solar.dart:255:10: dynamic typing is not allowed "var planet;"
       254:   void drawChildren(CanvasRenderingContext2D context, num x, num y) {
       255:     for (var planet in planets) {
```

The applicability of the mandatory types checker is completely dependent on whether the user wants to write a statically typed program or a dynamically typed program. In the above example, when I ran the dart_analyzer on an existing dynamically typed Dart program, it did give many warnings but these warnings are helpful when you want to incorporate types into an existing dynamically typed Dart program. After the program is statically typed, many type warnings and errors can be caught using the dart_analyzer as well as running other type checkers on a statically typed program will produce better error output. Also since I have added the mandatory types

checker using a command line option, it is easy to switch between static typing and dynamic typing for beginning learners of Dart programming language.

I have incorporated the implementation of the variance checker without annotation into the dart_analyzer. The variance checker issues a warning if a collection (`List`) is passed covariantly to a method and if it gets modified inside the method using `add`, `addAll` or `addLast` `List` methods. Currently, the existing Dart code samples does not do any modification of covariantly passed generics inside method body. Therefore, I was not able to get any warnings when I ran the variance checker on the code samples using collections. However, the existing dart_analyzer completely accepts the modification of covariantly passed generics but a runtime exception occurs when the program is run using Dart or launched using Dartium.
I made some modifications to the solar.dart code to demonstrate this situation.

In the `PlanetaryBody` class I added a method `getPlanets()` to obtain the list of planets under a `PlanetaryBody`. In the `SolarSystem` class I use the `getPlanets` method to get the list of planets under `Sun` and then I pass this list to the method `modifyPlanets` covariantly and do the modification of the list by adding a `String` object into the list of `PlanetaryBody` objects. After calling the `modifyPlanets` method, in a for loop I call the `_calculateSpeed` method of `PlanetaryBody` on each object in the list. Since the list now contains a `String` object, a run time exception is thrown as below. Please refer to Appendix D for the complete source code of solar.dart with my modifications.

```
Exception: NoSuchMethodError : method not found: '_calculateSpeed@0x1522cb60'
Receiver: "DummyPlanetString"
Arguments: [0.0]
Stack Trace: #0      Object._noSuchMethod (dart:core-patch:1260:3)
#1      Object.noSuchMethod (dart:core-patch:1263:25)
#2      SolarSystem._start (http://127.0.0.1:3030/Users/SnigdhaChaitanya/dart/solar/
bin/solar.dart:104:36)
#3      SolarSystem.start.<anonymous closure> (http://127.0.0.1:3030/Users/
SnigdhaChaitanya/dart/solar/bin/solar.dart:79:13)
#4      _completeMeasurementFutures._completeMeasurementFutures (/Volumes/data/b/
build/slave/dartium-mac-full-trunk/build/src/dart/sdk/lib/html/src/Measurement.dart:
122:14)
#5      _MeasurementScheduler._onCallback (/Volumes/data/b/build/slave/dartium-mac-
full-trunk/build/src/dart/sdk/lib/html/src/Measurement.dart:35:19)
#6      _MutationObserverScheduler._handleMutation._handleMutation (/Volumes/data/b/
build/slave/dartium-mac-full-trunk/build/src/dart/sdk/lib/html/src/Measurement.dart:
64:21)
```

The modified version of dart_analyzer which includes the variance checker gives a warning if a method modifies the covariantly passed collection. Such warnings are helpful to avoid the run time exceptions as shown above. Even though the existing code samples do not modify a covariantly passed generics, it is likely that the programs written by users in the future will contain such code. The variance checker is extremely useful to the users to catch these errors at the static analysis stage.

10. Conclusions and Future Scope

In this project I presented the static type checkers I have developed for the optional type system of the Dart programming language. Dart is a dynamically typed language with support for optional type system. A programming language that implements optional type system can be type checked by using pluggable type systems. A dynamically typed programming language can incorporate the advantages of both dynamically typed languages and statically typed languages when combined with pluggable type systems. Since the language itself is inherently dynamically typed, it gains the advantage of being expressive and can adapt to changing requirements. With the support for pluggable type systems, the language can incorporate all the benefits that an inherently statically typed language provides.

Many attempts have been made by individuals to create frameworks for pluggable type systems for various languages and some of them have been successful in introducing type checking into dynamically and optionally typed languages. The Strongtalk type checker for Smalltalk programming language developed by Gilad Bracha and David Griswold is an example of a successful type checker for a dynamically typed language [**4**]. The Checker Framework being developed at MIT is a framework for custom pluggable types for Java [**11**]. While these are the frameworks developed for pluggable type systems, my implementation of the four static checkers for the Dart programming language is an ad-hoc implementation. I have added the type checkers to the dartc compiler of the Dart language.

The first type checker is to enforce the static typing in Dart programs. For this static type checker, I have added a new compiler option for dartc compiler that can be given while invoking the compiler on the Dart programs. This checker checks that the Dart programs are completely statically typed and issues static warnings in case the programs contain any dynamically typed code.

The second type checker I have developed is to enhance type checking of Dart's support for covariant generics. Since Dart type annotations are supposed to be simple, lightweight and optional, Dart has decided to support only covariance. Dart has made its generics covariant instead of invariant, supporting its optimistic nature. However, the whole support for covariant generics is based on the assumption that generics are used for read-only purposes. The variance type checker I have developed checks that a generic collection passed covariantly as a method argument is not subjected to any modification inside the method. I have presented two implementations of the variance type checker. In the first implementation, during parsing in the backend of the compiler, I check whether the method code modifies a covariantly passed collection. For the second implementation, I have introduced annotations in Dart and supported the variance checker by using a `@modifies` annotation to annotate a modifying method. I have added the support for Lists as covariant generics for method arguments. For the future work, the support of covariant generics can be extended to other collection classes in Dart.

The third type checker I have developed is a nullness checker. The nullness checker is useful to ensure that null values are never passed around. Dart program that passed the Nullness Checker never throws a null pointer exception. In this project, I have implemented the Nullness Checker by adding an annotation called `@notnull` for method parameters and identifiers in Dart language. For the future work, the `@notnull` annotation can be supported for method return values as well.

The fourth type checker I have developed for Dart is called linear checker. The linear checker is useful to ensure that no unexpected modifications happen to objects because of multiple references. Dart supports concurrent programming by message passing between isolates. Since Dart does not allow shared memory threads, it does not implement any locking system for concurrent programming. Instead, a message is copied before receiving so that no two isolates share same instance of the object. This might be wasteful in cases where certain objects are usually used by only one reference at a time. By using the Linear Checker, the problem can be avoided by not copying the message objects which are annotated as linear objects. I have implemented the Linear Checker by adding an annotation called `@linear` for method parameters and method local identifiers. The future work could be to support `@linear` annotation for return values and field of classes. The support of `@linear` annotation for fields of classes might require significantly complex parsing through class definitions and methods.

In this project, I have given an ad-hoc implementation of the four type checkers rather than creating a framework for them. Even for the ad-hoc implementation, it was challenging to understand the dartc compiler structure and its support for Dart programming language. Since Dart is an evolving language, there have been many changes in the language constructs and the Dart compilers from the time I started working on this project. Therefore, the testing of the type checkers I implemented on the existing Dart code was another challenging aspect of this project. I initially implemented the type checkers in the dartc compiler which was the first compiler of the Dart written in Java. Currently, Dart supports dart2js compiler written in Dart. The Dart SDK also includes a static analyzer called dart_analyzer which draws its source code from dartc. To test the applicability of the project, I implemented the mandatory types checker and the variance checker without annotations in the dart_analyzer. I ran the modified dart_analyzer on the code samples provided in Dart SDK. In the future, Nullness Checker and the Linear Checker can also be incorporated into the dart_analyzer using annotations or Dart metadata. With the addition of these type checkers, more type safety can be added to the Dart's optional type system. For programmers who intend to skip the types by using the optional type system, the type checkers might be impractical to use but for programmers who like to program in Dart as a statically typed language, these type checkers can provide additional type checking.

Appendix A Code to add a new Compiler Flag

Adding a new compiler flag:

==========

The following are the additions made to code to add a `--must-have-types` flag to dartc compiler.

In `CommandLineOptions.java`, added an option for `--must-have-types` and the corresponding `mustHaveTypes()` method.

```
      @Option(name = "--must-have-types", aliases = { "-must-have-types" }, usage =
"Types must be given")
      private boolean mustHaveTypes = false;
```

In `CompilerConfiguration.java` (interface), added a new method.

```
      boolean mustHaveTypes();
```

In `DefaultCompilerConfiguration.java`, override method `mustHaveTypes()`.

```
      @Override
        public boolean mustHaveTypes() {
             return compilerOptions.mustHaveTypes();
        }
```

In `DelegatingCompilerConfiguration.java`, override the method `mustHaveTypes()`.

```
      @Override
        public boolean mustHaveTypes() {
             return delegate.mustHaveTypes();
        }
```

In `TestCompilerConfiguration.java`, override the method `mustHaveTypes()`.

```
      @Override
        public boolean mustHaveTypes() {
             return false;
        }
```

==========

Appendix B Code to add `@linear` Annotation

Code modifications done to support `@linear` annotation:

==========

I added a `linear` keyword to the `Token` enum class in `Token.java`.

```
LINEAR("linear", 0),
```

I added a local boolean variable `isLinear` to `parseFormalParameter` method in `DartParser.java` and added a check for `@linear` annotation.

```
private DartParameter parseFormalParameter(boolean isNamed) {
       boolean notNull = false;
       boolean isLinear = false;
       ...
       if (peek(0) == Token.AT_TOKEN) {
           if (peek(1) == Token.NOTNULL) {
                if (expect(Token.AT_TOKEN) && expect(Token.NOTNULL)) {
                     notNull = true;
                }
           } else if (peek(1) == Token.LINEAR) {
                if (expect(Token.AT_TOKEN) && expect(Token.LINEAR)) {
                     isLinear = true;
                }
           }
       }
       ...
       return done(new DartParameter(paramName, type, functionParams,
                                     defaultExpr, modifiers,notNull, isLinear));
}
```

I added a new protected instance field `isLinear` to `DartNode` class like `notNull` instance field.

```
public abstract class DartNode extends AbstractNode implements DartVisitable {
       ...
       protected boolean notNull = false;
       protected boolean isLinear = false;
       ...

       public boolean isNotNull() {
              return notNull;
       }

       public boolean isLinear() {
              return isLinear;
       }
       ...
}
```

I overloaded the `DartParameter` constructor to take a boolean parameter for `isLinear` along with the boolean parameter for `notNull`.

```
public DartParameter(DartExpression name,
                           DartTypeNode typeNode,
                           List<DartParameter> functionParameters,
                           DartExpression defaultExpr,
                           Modifiers modifiers,
                           boolean notNull,
                           boolean isLinear) {
      super(name);
      ...
      this.notNull = notNull;
      this.isLinear = isLinear;
}
```

In `parseNonLabelledStatement` method in `DartParser`, added a check for `@linear` annotation for initialized variables.

```
private DartStatement parseNonLabelledStatement() {
...
case IDENTIFIER:
      boolean notNull = false;
      boolean isLinear = false;
      if (peek(1) == Token.LT
            || peek(1) == Token.IDENTIFIER
            || (peek(1) == Token.PERIOD && peek(2) == Token.IDENTIFIER)
            || (peek(0) == Token.AT_TOKEN)) {

            if (peek(0) == Token.AT_TOKEN) {
                  if (peek(1) == Token.NOTNULL) {
                  if (expect(Token.AT_TOKEN) && expect(Token.NOTNULL)) {
                        notNull = true;
                  }
                  } else if (peek(1) == Token.LINEAR) {
                        if (expect(Token.AT_TOKEN) && expect(Token.LINEAR)) {
                              isLinear = true;
                        }
                  }
            }

            if (peek(1) == Token.LT
            || peek(1) == Token.IDENTIFIER
            || (peek(1) == Token.PERIOD && peek(2) == Token.IDENTIFIER)){
                  beginTypeFunctionOrVariable();
                  DartTypeNode type = tryTypeAnnotation();

                  if (type != null && peek(0) == Token.IDENTIFIER) {
                  List<DartVariable> vars =
                        parseInitializedVariableList(notNull, isLinear);
                  expect(Token.SEMICOLON);
                  return done(new     DartVariableStatement(vars, type));
            } else {
                  rollback();
            }
```

```
                }
            }
            }
    ...
}
```

I added a boolean `isLinear` parameter to `parseInitializedVariableList` method in `DartParser` and passed it to overloaded `parseIdentifier` method.

```
    private List<DartVariable> parseInitializedVariableList(boolean notNull,
                                                             boolean isLinear) {
        List<DartVariable> idents = new ArrayList<DartVariable>();
        do {
            beginVariableDeclaration();
            DartIdentifier name = parseIdentifier(notNull,isLinear);
            DartExpression value = null;
            if (isParsingInterface) {
                expect(Token.ASSIGN);
                value = parseExpression();
            } else if (optional(Token.ASSIGN)) {
                value = parseExpression();
            }

            idents.add(done(new DartVariable(name, value)));
        } while (optional(Token.COMMA));
        return idents;
    }
```

I added another boolean parameter for `isLinear` to already overloaded `parseIdentifier` method which was to support `@notnull` annotation.

```
    private DartIdentifier parseIdentifier(boolean notNull, boolean isLinear){
        beginIdentifier();
        expect(Token.IDENTIFIER);
        String token = ctx.getTokenString();
        if (notNull || notNullIdentifiers.contains(token) || isLinear
        || linearIdentifiers.contains(token)) {
            if (notNull) {
                notNullIdentifiers.add(token);
            }
            if (isLinear) {
                linearIdentifiers.add(token);
            }
            return done(new DartIdentifier(token != null ?
                        token : "", notNull, isLinear));
        }
        return done(new DartIdentifier(token != null ? token : ""));
    }
```

Similar to the `notNullIdentifiers` hash set created to support `@notnull` annotation, added a `linearIdentifiers` hash set to support `@linear` annotation. As shown above, in `parseIdentifier` method, added a check to see if the variable has been annotated while declaration. Similarly, added a check in regular `parseIdentifier` method to check if the variable has been annotated while declaration.

```
private DartIdentifier parseIdentifier() {
	beginIdentifier();
	expect(Token.IDENTIFIER);
	String token = ctx.getTokenString();
	boolean notNull = notNullIdentifiers.contains(token);
	boolean isLinear = linearIdentifiers.contains(token);
	if (notNull || isLinear) {
		return done(new DartIdentifier(token != null ?
		token : "", notNull, isLinear));
	}
	return done(new DartIdentifier(token != null ? token : ""));
}
```

I also added another boolean parameter to the overloaded constructor of `DartIdentifier`.

```
public DartIdentifier(String targetName, boolean notNull, boolean isLinear) {
		assert targetName != null;
		this.targetName = targetName;
		this.notNull = notNull;
		this.isLinear = isLinear;
}
```

In `TypeAnalyzer.java`, following is the method call flow. `visitMethodDefinition` method calls `visitFunction` method. In the `DartParser`, since the `isLinear` value of `DartParameter` is set to true if `@linear` annotation is given, that value can be obtained in `TypeAnalyzer.java` while visiting parameter nodes. In `TypeAnalyzer.java`, added a map to store parameter names with their `isLinear` values.

```
private Map<String, Boolean> linearParams;
```

In the `Analyzer` constructor, the `linearParams` map is initialized.

```
this.linearParams = new HashMap<String, Boolean>();
```

In `visitFunction` method in `TypeAnalyzer.java`, added a loop to parse through the function parameters and add them to `linearParams` map.

```
public Type visitFunction(DartFunction node) {
	Type previous = expected;

	for (DartParameter p : node.getParams()) {
		linearParams.put(p.getParameterName(), p.isLinear());
	}

	visit(node.getParams());
	expected = typeOf(node.getReturnTypeNode());
	typeOf(node.getBody());
	expected = previous;
	return voidType;
}
```

After visiting function parameters, `visitFunction` calls visit on function body by calling `visitBlock` method. `visitBlock` method calls `visitVariableStatement` on variable statements. `visitVariableStatement` method calls `visitVariable` method which in turn calls `checkInitializedDeclaration` method. In `checkInitializedDeclaration` method, after calling `checkAssignable`, added a call to `checkLinearAssignable` method which I added.

```
	private Type checkInitializedDeclaration(DartDeclaration<?> node,
	DartExpression value) {
	if (value != null) {
			checkAssignable(node.getSymbol().getType(), value);
			checkLinearAssignable(node, value);
	}
	return voidType;
	}
```

I also added a set to store the used linear params in `TypeAnalyzer.java` and initialized it in the `Analyzer` constructor.

```
	private Set<String> usedLinearParams;
	this.usedLinearParams = new HashSet<String>();
```

Below is the `checkLinearAssignable` method I added to `TypeAnalyzer.java` to support `@linear` annotation checks.

```
private void checkLinearAssignable(DartDeclaration<?> node, DartExpression value) {

	DartIdentifier id = (DartIdentifier) node.getName();
	boolean isLinearIdentifier = id.isLinear();

	boolean isLinearParam = false;
	if (linearParams.get(value.toSource()) != null) {
		isLinearParam = linearParams.get(value.toSource());
	}

	if (isLinearIdentifier) {
		if (isLinearParam) {
			if (usedLinearParams.contains(value.toSource())) {
			typeError(node, TypeErrorCode.LINEAR_PARAM_NOT_REUSABLE, value);
			} else {
			usedLinearParams.add(value.toSource());
			}
		} else {
			typeError(node,
	TypeErrorCode.NON_LINEAR_PARAM_NOT_ASSIGNABLE_TO_LINEAR_VARIABLE, value, id);
		}
	} else {
		if (isLinearParam) {
		typeError(node,TypeErrorCode.LINEAR_PARAM_NOT_ASSIGNMENT_COMPATIBLE,
		value, id);
	}
	}
 }
```

The following are the three error codes I added to `TypeErrorCode.java` to support `@linear` annotation checks.

```
      LINEAR_PARAM_NOT_ASSIGNMENT_COMPATIBLE("linear parameter \"%s\" is not
assignable to non-linear variable \"%s\""),

      LINEAR_PARAM_NOT_REUSABLE("linear parameter \"%s\" has already been assigned to
a linear variable"),

      NON_LINEAR_PARAM_NOT_ASSIGNABLE_TO_LINEAR_VARIABLE("non linear parameter \"%s\"
is not assignable to linear variable \"%s\""),
```

==========

Appendix C Source Code of Solar Application without Modifications

Source code of solar.dart without my modifications:

==========

```
// Copyright (c) 2012, the Dart project authors. Please see the AUTHORS file for details.
// All rights reserved. Use of this source code is governed by a BSD-style license that
// can be found in the LICENSE file.

/**
 * A solar system visualization.
 */

library solar;

import 'dart:html';
import 'dart:math';

/**
 * The entry point to the application.
 */
void main() {
  var solarSystem = new SolarSystem(query("#container"));

  solarSystem.start();
}

double fpsAverage;

/**
 * Display the animation's FPS in a div.
 */
void showFps(num fps) {
  if (fpsAverage == null) {
    fpsAverage = fps;
  }

  fpsAverage = fps * 0.05 + fpsAverage * 0.95;

  query("#notes").text = "${fpsAverage.round().toInt()} fps";
}

// TODO: remove this once dart:html Point works cross-platform
class Point {
  num x, y;

  Point(this.x, this.y);
}

/**
 * A representation of the solar system.
 *
 * This class maintains a list of planetary bodies, knows how to draw its
```

```
 * background and the planets, and requests that it be redraw at appropriate
 * intervals using the [Window.requestAnimationFrame] method.
 */
class SolarSystem {
  CanvasElement canvas;

  num _width;
  num _height;

  PlanetaryBody sun;

  num renderTime;

  SolarSystem(this.canvas) {

  }

  num get width => _width;

  num get height => _height;

  start() {
    // Measure the canvas element.
    window.requestLayoutFrame(() {
      _width = (canvas.parent as Element).clientWidth;
      _height = (canvas.parent as Element).clientHeight;

      canvas.width = _width;

      // Initialize the planets and start the simulation.
      _start();
    });
  }

  _start() {
    // Create the Sun.
    sun = new PlanetaryBody(this, "Sun", "#ff2", 14.0);

    // Add planets.
    sun.addPlanet(
        new PlanetaryBody(this, "Mercury", "orange", 0.382, 0.387, 0.241));
    sun.addPlanet(
        new PlanetaryBody(this, "Venus", "green", 0.949, 0.723, 0.615));

    var earth = new PlanetaryBody(this, "Earth", "#33f", 1.0, 1.0, 1.0);
    sun.addPlanet(earth);
    earth.addPlanet(new PlanetaryBody(this, "Moon", "gray", 0.2, 0.14, 0.075));

    sun.addPlanet(new PlanetaryBody(this, "Mars", "red", 0.532, 1.524, 1.88));

    addAsteroidBelt(sun, 150);

    final f = 0.1;
    final h = 1 / 1500.0;
    final g = 1 / 72.0;

    var jupiter = new PlanetaryBody(
        this, "Jupiter", "gray", 4.0, 5.203, 11.86);
```

```
  sun.addPlanet(jupiter);
  jupiter.addPlanet(new PlanetaryBody(
      this, "Io", "gray", 3.6 * f, 421 * h, 1.769 * g));
  jupiter.addPlanet(new PlanetaryBody(
      this, "Europa", "gray", 3.1 * f, 671 * h, 3.551 * g));
  jupiter.addPlanet(new PlanetaryBody(
      this, "Ganymede", "gray", 5.3 * f, 1070 * h, 7.154 * g));
  jupiter.addPlanet(new PlanetaryBody(
      this, "Callisto", "gray", 4.8 * f, 1882 * h, 16.689 * g));

  // Start the animation loop.
  requestRedraw();
}

void modifyPlanets(List<Object> objects) {
     objects.add("DummyPlanetString");
}

void draw(num _) {
  num time = new Date.now().millisecondsSinceEpoch;

  if (renderTime != null) {
    showFps((1000 / (time - renderTime)).round());
  }

  renderTime = time;

  var context = canvas.context2d;

  drawBackground(context);
  drawPlanets(context);

  requestRedraw();
}

void drawBackground(CanvasRenderingContext2D context) {
  context.fillStyle = "white";
  context.rect(0, 0, width, height);
  context.fill();
}

void drawPlanets(CanvasRenderingContext2D context) {
  sun.draw(context, width / 2, height / 2);
}

void requestRedraw() {
  window.requestAnimationFrame(draw);
}

void addAsteroidBelt(PlanetaryBody body, int count) {
  Random random = new Random();

  // Asteroids are generally between 2.06 and 3.27 AUs.
  for (int i = 0; i < count; i++) {
    var radius = 2.06 + random.nextDouble() * (3.27 - 2.06);

    body.addPlanet(
        new PlanetaryBody(this, "asteroid", "#777",
```

```
              0.1 * random.nextDouble(),
              radius,
              radius * 2));
    }
  }

  num normalizeOrbitRadius(num r) {
    return r * (width / 10.0);
  }

  num normalizePlanetSize(num r) {
    return log(r + 1) * (width / 100.0);
  }
}

/**
 * A representation of a plantetary body.
 *
 * This class can calculate its position for a given time index, and draw itself
 * and any child planets.
 */
class PlanetaryBody {
  final String name;
  final String color;
  final num orbitPeriod;
  final SolarSystem solarSystem;

  num bodySize;
  num orbitRadius;
  num orbitSpeed;

  List<PlanetaryBody> planets;

  PlanetaryBody(this.solarSystem, this.name, this.color, this.bodySize,
      [this.orbitRadius = 0.0, this.orbitPeriod = 0.0]) {
    planets = [];

    bodySize = solarSystem.normalizePlanetSize(bodySize);
    orbitRadius = solarSystem.normalizeOrbitRadius(orbitRadius);
    orbitSpeed = _calculateSpeed(orbitPeriod);
  }

  void addPlanet(PlanetaryBody planet) {
    planets.add(planet);
  }

  void draw(CanvasRenderingContext2D context, num x, num y) {
    Point pos = _calculatePos(x, y);

    drawSelf(context, pos.x, pos.y);

    drawChildren(context, pos.x, pos.y);
  }

  void drawSelf(CanvasRenderingContext2D context, num x, num y) {
    context.save();

    try {
```

```
      context.lineWidth = 0.5;
      context.fillStyle = color;
      context.strokeStyle = color;

      if (bodySize >= 2.0) {
        context.shadowOffsetX = 2;
        context.shadowOffsetY = 2;
        context.shadowBlur = 2;
        context.shadowColor = "#ddd";
      }

      context.beginPath();
      context.arc(x, y, bodySize, 0, PI * 2, false);
      context.fill();
      context.closePath();
      context.stroke();

      context.shadowOffsetX = 0;
      context.shadowOffsetY = 0;
      context.shadowBlur = 0;

      context.beginPath();
      context.arc(x, y, bodySize, 0, PI * 2, false);
      context.fill();
      context.closePath();
      context.stroke();
    } finally {
      context.restore();
    }
  }

  void drawChildren(CanvasRenderingContext2D context, num x, num y) {
    for (var planet in planets) {
      planet.draw(context, x, y);
    }
  }

  num _calculateSpeed(num period) {
    if (period == 0.0) {
      return 0.0;
    } else {
      return 1 / (60.0 * 24.0 * 2 * period);
    }
  }

  Point _calculatePos(num x, num y) {
    if (orbitSpeed == 0.0) {
      return new Point(x, y);
    } else {
      num angle = solarSystem.renderTime * orbitSpeed;

      return new Point(
        orbitRadius * cos(angle) + x,
        orbitRadius * sin(angle) + y);
    }
  }

}
```

==========

Source code of solar.css:

==========

```
/* Copyright (c) 2012, the Dart project authors.  Please see the AUTHORS file */
/* for details. All rights reserved. Use of this source code is governed by a */
/* BSD-style license that can be found in the LICENSE file. */

body {
  background-color: #F8F8F8;
  font-family: 'Open Sans', sans-serif;
  font-size: 14px;
  font-weight: normal;
  line-height: 1.2em;
  margin: 15px;
}

p {
  color: #333;
}

#container {
  width: 100%;
  height: 400px;
  border: 1px solid #ccc;
  background-color: #fff;
}

#summary {
  float: left;
}

#notes {
  float: right;
  width: 120px;
  text-align: right;
}

.error {
  font-style: italic;
  color: red;
}
```

==========

Source code of solar.html:

==========

```
<!DOCTYPE html>
```

```
<!-- Copyright (c) 2012, the Dart project authors.  Please see the AUTHORS file
     for details. All rights reserved. Use of this source code is governed by a
     BSD-style license that can be found in the LICENSE file. -->

<html>
  <head>
    <meta charset="utf-8">
    <title>Solar System Demo</title>
    <link type="text/css" rel="stylesheet" href="solar.css">
  </head>
  <body>
    <h1>Solar System</h1>

    <p>A solar system visualization using requestAnimationFrame.</p>

    <div>
      <canvas id="container" width="500px" height="400px"></canvas>
    </div>

    <footer>
      <p id="summary"> </p>
      <p id="notes"> </p>
    </footer>

    <script type="application/dart" src="solar.dart"></script>
    <script src="https://dart.googlecode.com/svn/branches/bleeding_edge/dart/client/
dart.js"></script>
  </body>
</html>
```

==========

Appendix D Source Code of Solar Application with Modifications

Source code of solar.dart with modifications:

==========

```
// Copyright (c) 2012, the Dart project authors. Please see the AUTHORS file for details.
// All rights reserved. Use of this source code is governed by a BSD-style license that
// can be found in the LICENSE file.

/**
 * A solar system visualization.
 */

library solar;

import 'dart:html';
import 'dart:math';

/**
 * The entry point to the application.
 */
void main() {
  var solarSystem = new SolarSystem(query("#container"));

  solarSystem.start();
}

double fpsAverage;

/**
 * Display the animation's FPS in a div.
 */
void showFps(num fps) {
  if (fpsAverage == null) {
    fpsAverage = fps;
  }

  fpsAverage = fps * 0.05 + fpsAverage * 0.95;

  query("#notes").text = "${fpsAverage.round().toInt()} fps";
}

// TODO: remove this once dart:html Point works cross-platform
class Point {
  num x, y;

  Point(this.x, this.y);
}

/**
 * A representation of the solar system.
 *
```

```
 * This class maintains a list of planetary bodies, knows how to draw its
 * background and the planets, and requests that it be redraw at appropriate
 * intervals using the [Window.requestAnimationFrame] method.
 */
class SolarSystem {
  CanvasElement canvas;

  num _width;
  num _height;

  PlanetaryBody sun;

  num renderTime;

  SolarSystem(this.canvas) {

  }

  num get width => _width;

  num get height => _height;

  start() {
    // Measure the canvas element.
    window.requestLayoutFrame(() {
      _width = (canvas.parent as Element).clientWidth;
      _height = (canvas.parent as Element).clientHeight;

      canvas.width = _width;

      // Initialize the planets and start the simulation.
      _start();
    });
  }

  _start() {
    // Create the Sun.
    sun = new PlanetaryBody(this, "Sun", "#ff2", 14.0);

    // Add planets.
    sun.addPlanet(
        new PlanetaryBody(this, "Mercury", "orange", 0.382, 0.387, 0.241));
    sun.addPlanet(
        new PlanetaryBody(this, "Venus", "green", 0.949, 0.723, 0.615));

    var earth = new PlanetaryBody(this, "Earth", "#33f", 1.0, 1.0, 1.0);
    sun.addPlanet(earth);
    earth.addPlanet(new PlanetaryBody(this, "Moon", "gray", 0.2, 0.14, 0.075));

    sun.addPlanet(new PlanetaryBody(this, "Mars", "red", 0.532, 1.524, 1.88));

    List<PlanetaryBody> planets = sun.getPlanets();

    modifyPlanets(planets);

    for(int i = 0; i < planets.length; i++) {
          planets[i]._calculateSpeed(0.0);
    }
```

```
  addAsteroidBelt(sun, 150);

  final f = 0.1;
  final h = 1 / 1500.0;
  final g = 1 / 72.0;

  var jupiter = new PlanetaryBody(
      this, "Jupiter", "gray", 4.0, 5.203, 11.86);
  sun.addPlanet(jupiter);
  jupiter.addPlanet(new PlanetaryBody(
      this, "Io", "gray", 3.6 * f, 421 * h, 1.769 * g));
  jupiter.addPlanet(new PlanetaryBody(
      this, "Europa", "gray", 3.1 * f, 671 * h, 3.551 * g));
  jupiter.addPlanet(new PlanetaryBody(
      this, "Ganymede", "gray", 5.3 * f, 1070 * h, 7.154 * g));
  jupiter.addPlanet(new PlanetaryBody(
      this, "Callisto", "gray", 4.8 * f, 1882 * h, 16.689 * g));

  // Start the animation loop.
  requestRedraw();
}

void modifyPlanets(List<Object> objects) {
     objects.add("DummyPlanetString");
}

void draw(num _) {
  num time = new Date.now().millisecondsSinceEpoch;

  if (renderTime != null) {
    showFps((1000 / (time - renderTime)).round());
  }

  renderTime = time;

  var context = canvas.context2d;

  drawBackground(context);
  drawPlanets(context);

  requestRedraw();
}

void drawBackground(CanvasRenderingContext2D context) {
  context.fillStyle = "white";
  context.rect(0, 0, width, height);
  context.fill();
}

void drawPlanets(CanvasRenderingContext2D context) {
  sun.draw(context, width / 2, height / 2);
}

void requestRedraw() {
  window.requestAnimationFrame(draw);
}
```

```
  void addAsteroidBelt(PlanetaryBody body, int count) {
    Random random = new Random();

    // Asteroids are generally between 2.06 and 3.27 AUs.
    for (int i = 0; i < count; i++) {
      var radius = 2.06 + random.nextDouble() * (3.27 - 2.06);

      body.addPlanet(
          new PlanetaryBody(this, "asteroid", "#777",
              0.1 * random.nextDouble(),
              radius,
              radius * 2));
    }
  }

  num normalizeOrbitRadius(num r) {
    return r * (width / 10.0);
  }

  num normalizePlanetSize(num r) {
    return log(r + 1) * (width / 100.0);
  }
}

/**
 * A representation of a plantetary body.
 *
 * This class can calculate its position for a given time index, and draw itself
 * and any child planets.
 */
class PlanetaryBody {
  final String name;
  final String color;
  final num orbitPeriod;
  final SolarSystem solarSystem;

  num bodySize;
  num orbitRadius;
  num orbitSpeed;

  List<PlanetaryBody> planets;

  PlanetaryBody(this.solarSystem, this.name, this.color, this.bodySize,
      [this.orbitRadius = 0.0, this.orbitPeriod = 0.0]) {
    planets = [];

    bodySize = solarSystem.normalizePlanetSize(bodySize);
    orbitRadius = solarSystem.normalizeOrbitRadius(orbitRadius);
    orbitSpeed = _calculateSpeed(orbitPeriod);
  }

  void addPlanet(PlanetaryBody planet) {
    planets.add(planet);
  }

  List<PlanetaryBody> getPlanets() {
       return planets;
```

```
}

void draw(CanvasRenderingContext2D context, num x, num y) {
  Point pos = _calculatePos(x, y);

  drawSelf(context, pos.x, pos.y);

  drawChildren(context, pos.x, pos.y);
}

void drawSelf(CanvasRenderingContext2D context, num x, num y) {
  context.save();

  try {
    context.lineWidth = 0.5;
    context.fillStyle = color;
    context.strokeStyle = color;

    if (bodySize >= 2.0) {
      context.shadowOffsetX = 2;
      context.shadowOffsetY = 2;
      context.shadowBlur = 2;
      context.shadowColor = "#ddd";
    }

    context.beginPath();
    context.arc(x, y, bodySize, 0, PI * 2, false);
    context.fill();
    context.closePath();
    context.stroke();

    context.shadowOffsetX = 0;
    context.shadowOffsetY = 0;
    context.shadowBlur = 0;

    context.beginPath();
    context.arc(x, y, bodySize, 0, PI * 2, false);
    context.fill();
    context.closePath();
    context.stroke();
  } finally {
    context.restore();
  }
}

void drawChildren(CanvasRenderingContext2D context, num x, num y) {
  for (var planet in planets) {
    planet.draw(context, x, y);
  }
}

num _calculateSpeed(num period) {
  if (period == 0.0) {
    return 0.0;
  } else {
    return 1 / (60.0 * 24.0 * 2 * period);
  }
}
```

```
  Point _calculatePos(num x, num y) {
    if (orbitSpeed == 0.0) {
      return new Point(x, y);
    } else {
      num angle = solarSystem.renderTime * orbitSpeed;

      return new Point(
        orbitRadius * cos(angle) + x,
        orbitRadius * sin(angle) + y);
    }
  }

}
```

==========

References

[1] Ali, M., Correa, L. T. Jr., Ernst, D. M., Papi, M. M., & Perkins, H. J. *Practical Pluggable Type for Java*, 2008. Proceedings of the 2008 International Symposium on Software Testing and Analysis, July 20-24, 2008, Seattle, WA, USA.

[2] Bracha, G. *Optional Types in Dart*, 2011. Retrieved January 8, 2013, from http://www.dartlang.org/articles/optional-types/.

[3] Bracha, G. *Pluggable Type Systems*, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[4] Bracha, G. & Griswold, D. *Strongtalk: Typechecking Smalltalk in a Production Environment*, 1993. Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA).

[5] Brandt, E. *Why Dart Types are Optional and Unsound*, 2011. Retrieved January 7, 2013 from http://www.dartlang.org/articles/why-dart-types/#generics-covariant.

[6] Griffith, R. *The Dart Programming Language for Non-Programmers - Errors and Warnings*, 2011. Retrieved January 8, 2013, from http://www.greatandlittle.com/studios/index.php?post/2011/11/06/The-Dart-Programming-Language-for-Non-Programmers-Part-2.

[7] Haldimann, N. *TypePlug, Pluggable Type Systems for Smalltalk*, 2007. Master's thesis, University of Bern (April 2007).

[8] Ladd, S. *Generics in Dart, or, Why a JavaScript Programmer Should Care About Types*, 2011. Retrieved January 8, 2012, from http://blog.sethladd.com/2012/01/generics-in-dart-or-why-javascript.html.

[9] Loitsch, F. & Nystrom, B. *Why Not a Bytecode VM?*, 2011. Retrieved January 7, 2013, from http://www.dartlang.org/articles/why-not-bytecode/.

[10] Malone, Matt. *Variance Basics in Java and Scala*, 2008. Retrieved January 8, 2012, from http://oldfashionedsoftware.com/2008/08/26/variance-basics-in-java-and-scala/.

[11] *The Checker Framework: Custom pluggable types for Java*. Retrieved January 8, 2012, from http://types.cs.washington.edu/checker-framework/current/checkers-manual.html.