The Operational Semantics and Implementation of a Core Dart language

Zhivka Gucevska

School of Computer and Communication Sciences

A thesis submitted for the degree of Master of Computer Science
at École Polytechnique Fédérale de Lausanne

August 2017

Supervisor
Prof. Viktor Kuncak
EPFL / LARA

Supervisor
Kevin Millikin
Google

Abstract

In this report we present the specification of the operational semantics of Dart Kernel and a reference implementation of Dart Kernel in Dart.

We design a CESK-like machine to specify the operational semantics of Dart Kernel and we implement an interpreter that follows closely the small-step semantics of the language. This approach allows us to define the behaviour of the rich features of Dart Kernel, such as exception handling and asynchronous execution. Since Dart Kernel is an evolving language, the specification is expected to evolve with it and the semantics for new features needs to be defined further.

The work done for this project sets the grounds for further formalization of the behaviour of the language with a formal proof management system, as Coq.

Contents

Listings

List of Figures

1 Introduction

Formally proving that an implementation of a programming language specification faithfully implements the intended semantics of the language is hard. There are often many implementation-level details whose relation with the semantics specification is non trivial or does not exist, which further hinders proofs of correctness.

The Dart project has decided to separate the implementation of its front end, that consumes Dart code and the runtime implementations, i.e., the back ends that run Dart programs. For this purpose they have designed a high level intermediate representation (IR) for the Dart language, called Dart Kernel. This IR is the interface between the front end and the different back ends and it is a programming language in its own right. It supports the Dart programming language features that are needed for runtime implementations and allows for simpler back ends that only need to implement this core language. The new generation of Dart tools implemented by the language developers consumes Dart Kernel programs instead of Dart source code.

Dart Kernel is also designed to be a framework for program transformation. Different back ends can define their own transformations that eliminate some features of the language and replace them with others. The Dart Kernel team is interested in correctness of these transformations, as well as in correctness of Dart to Dart Kernel translation.

Dart Kernel is also believed to be more susceptible to formalization. In this project we define the operational semantics of the Dart Kernel language by presenting an abstract machine that implements its semantics. We then implement an interpreter that follows closely the formally described semantics. Since the back end implementations will implement Dart Kernel, having a reference implementation for it can be of great value to the language implementers. The CESK-like machine for Dart Kernel is designed with possibility to be implemented in the proof management system Coq, to formally specify and prove properties of the language.

A reference implementation of Dart Kernel in Dart can also serve as a test-bed for language changes that can not be desugared and require modifications in the back end.

The main contributions in this report are:

- A specification of the operational semantics of Dart Kernel. [21]
- An interpreter for a subset of the language Dart Kernel written in Dart.[16]

We also started the formalization of the operational semantics with the proof assistant Coq[21]. The specification of the operational semantics by design is easily portable to Coq – a formal proof management system[26], but due to time constraints and focus, was not done in the scope of this project.

The rest of the report is organized as follows.

In Sections 2 and 3 we present the Dart programming language and a high level intermediate representation for it, Dart Kernel. In Section 4 we present Dart Kernel's object model. Then, we give an overview of the CESK machine, and the modifications of it to support different features of the Dart Kernel language. We show the transition based system for a subset of Dart Kernel's expressions and statements. We then describe a reference implementation of Dart Kernel as an interpreter in Dart.

2 Dart

Dart[1] is a pure class-based, single-inheritance, object-oriented programming language created by Google[7]. It is designed to be used for large web and mobile applications. One example of such application is Google's next generation of the AdWords[2] front end that is built in Dart[13].

Dart is also the language of the framework for cross-platform mobile applications, Flutter[3], and the framework for web application development, AngularDart, that is used by Google engineers to build sophisticated mission-critical applications.[4]

2.1 Overview of Dart

Dart is statically typed[5][6] with type inference and has a sound type system[11]. In Dart 2.0 valid Dart programs behave as expected, with runtime checks added to invalidate programs with type problems[14].

In Dart everything is an object and every object is an instance of a class. All objects in Dart inherit from the "Object" class, which is at the root of the class hierarchy. The language has an automatically imported library, `dart:core`[7], that defines the built-in types, such as numbers, strings and basic collections.

The Dart programming language supports a wide range of features found in modern programming languages. We present some of them below.

Interfaces and mixins Classes in Dart define implicit interfaces, containing all instance members of a class and all the classes it implements. They are used to support an API without inheriting its implementation and classes can implement multiple interfaces via the keyword `implements`.

Dart also supports mixins derived from class declarations and mixin applications. Mixin applications occur when a mixin is mixed into a class dec-

[1] Dart, `dartlang.org`

[2] AdWords, `www.adwords.google.com`

[3] Flutter, `flutter.io`

[4] AngularDart, `webdev.dartlang.org/angular`

[5] Dart 1.x is optionally typed, but had an experimental type system called Strong Mode. In Dart 2.0, the next release of the language, Strong Mode will be the type system of the language.

[6] In this report we only consider Dart 2.0, since the intermediate language considered will only be used for Dart 2.0.

[7] Library `dart:core`, `www.dartlang.org/guides/libraries/library-tour`

laration via the keyword `with`.

Getters and setters DART provides implicit getters for all instance fields and setters for all non-final instance fields in a class. These implicit getters and setters share the same namespace with all other methods, including user defined getters and setters.

Functions Functions in DART are first class values that are instances of the built-in class "Function". They can be created with function expressions, local function declarations or method tear-offs.

Exports and imports DART supports modularity via libraries. A DART program consists of one or more libraries, combined with imports and exports. An import specifies a library to be used in the scope of another library and an export in a given library specifies a namespace that is made available to other libraries that import the given library. As mentioned before DART has the `dart:core` library implicitly imported to all DART programs.

Explicit control flow DART supports labels, which have a separate namespace, and control flow with break and continue to these labels. An interesting feature is `continue` in `switch` statements, which introduces irreducible control flow in the language. DART also supports structured exception handling with `try/catch/finally`.

2.2 DART SDK

The DART language developers provide the DART SDK, which is an implementation of the DART language and a set of development tools for the users of the language. It contains three different implementations of DART:

DART VM The DART Virtual Machine[8] is implemented in C++. It is a Just-in-Time native-code virtual machine supporting multiple target architectures. It also supports Ahead-of-Time compilation.

Dart Dev Compiler (DDC) The DDC compiler[6], implemented in Dart, is intended to be used for developement. It is a non-optimizing compiler of DART programs to JavaScript with the goal of fast compile times and idiomatic human-readable generated JavaScript code.

dart2js The `dart2js` compiler[9], implemented in DART, is an optimizing compiler. It compiles DART programs to JavaScript for the web and performs aggressive whole-program optimizations. It is used for deployment of DART code.

All three of these implementations are monolithic and they do not share any components with each other.

In an effort to combine the common parts of the back ends, the DART project has begun the implementation of a shared DART front end, implemented in DART, that will be used by all three implementation back ends. All common functionalities of the back ends will be combined in the shared front end. The goal of the shared front end effort is to enable faster language evolution, to reduce differences between the implementations' behaviour, and to reduce complexity in the back ends. The shared front end, Fasta (Fully resolved AST, Accelerated)[10], is a compiler framework for compiling DART sources to a high level intermediate representation, DART KERNEL.

DART KERNEL was designed by the DART team to be the interface between the shared front end and the different back ends.

3 Dart Kernel

Dart Kernel is an intermediate language for Dart, designed by the Dart team, to be consumed by the different back ends.[8] This intermediate language is a simplified representation of the Dart programming language and it is intended to have Dart-like semantics and to fully support all Dart 2.0 features.

The introduction of the intermediate representation Dart Kernel enables moving all the shared components from the back ends to the common front end. Only features that require runtime support are represented in Dart Kernel, allowing for simpler back ends that implement only a core language. Similar approaches have been previously used in other programming systems. A few such examples are Java, with Lightweight Java which was designed for academic purposes[25], LLVM[15], and the GHC Haskell compiler, with a core language designed for compilation[27][1].

Dart Kernel has a binary and in-memory representation, and in this report we generally refer to the in-memory representation of Dart Kernel programs.

3.1 Overview of Dart Kernel

Dart Kernel is explicitly typed and runtime checks that are needed for soundness of the type system are inserted by the front end. It is a simplified version of Dart, where many of Dart's features are desugared by the front end. There are some Dart features that do not appear in Dart Kernel at all, such as postfix and null-aware operators.

In Dart Kernel static parts of the program are fully resolved. Static accessors and invocations point to the node in the Dart Kernel graph. Dart Kernel is also designed for separate compilation, hence it uses canonical names that are linked with the other modules.

We present below how some concepts of Dart are presented in Dart Kernel:

Interfaces and mixins In Dart Kernel a class definition is a record that contains all instance members defined in it and a reference to its superclass. Mixins are generated as synthetic class definition records by the frontend and

[8]The Dart Kernel language did not evolve in the scope of or during this project. However the language may evolve in the future, e.g. for introduction of new language features in Dart that require back end support.

Dart Kernel has syntactic support for them. Implemented interfaces are remembered for runtime type checks in the record representing the Dart Kernel class. The object model of Dart Kernel is defined in detail in Section 4.

Getters and setters Instance field members in Dart Kernel are presented as records containing an identifier and an initializer. Other metadata is also contained in these records. In particular, the metadata indicating whether the field is final encodes whether an implementer should generate an implicit setter for the field.

Functions In Dart Kernel, functions are created with function expressions and function declarations, which have a reference to a function record containing all the necessary information for the function, such as formal parameters and statement body. References to function records are also encountered in constructors, user defined getters and setters, and methods.

Explicit control flow Dart Kernel also supports explicit control flow with labelled statements. Labels in Dart Kernel are resolved by the front end and the target of `break` is a statement in the in-memory representation of Dart Kernel. Note that `continue` statements are translated to `break` statements, except for `continue`s in `switch` which have a different semantics.[9] The target of `continue` statement is resolved by the front end to be the `switch` case corresponding to the label.

Dart Kernel is also designed to be a framework for program transformation. These program transformations can be implemented by back ends to desugar some features of the language to simpler features or to a subset of features that the back end implements.

An example of a transformation is desugaring `for in` loops to `while` loops. After this transformation the Dart Kernel program would not contain any `for in` loops, therefore the back end wont need to support them in its implementation. Note that even though some features can be desugared or translated in terms of other features of Dart Kernel, this is not done and these decisions are left to the back end implementers. [10]

[9] Contrary to `break`s and `continue`s in labelled statements, `continue`s in `switch` can target labels of non-enclosing catch clauses.

[10] For example, some back ends might find the distinction between `for in` and `while` loops useful.

3.2 Abstract syntax

As mentioned above, Dart Kernel has a compact binary representation[11], with de Bruijn's levels[3][18] for indexing of variables and labels, contrary to the more commonly used de Bruijn's indices. Dart Kernel also has an in-memory representation and we will refer to it in this report.

The abstract syntax of Dart Kernel is a graph that contains a unique tree, by designating some edges as "references" which are not part of the tree. Graph-based abstract syntax is used for implementing fast rewriting algorithms[2][19]. We refer to the Dart Kernel graph as an Abstract Syntax Tree, AST.

We consider the expressions and statements from Figures 1 and 2 for Dart Kernel AST in the remainder of the report. Expressions in Dart Kernel are denoted as E. As shown in Figure 1, expressions may contain other expressions or statements from the Dart Kernel AST, and the same holds for statements.

For example, variable declarations, denoted as D, are specific statements and when encountered as such, they introduce a variable x in the current scope. A reference to a variable declaration can also appear in other Dart Kernel nodes, such as access to a variable.[12] However, when we consider it in the context of expressions, its semantics is access of the latest value bound to the variable declaration D.

A program in Dart Kernel is represented as a data structure that contains a `main` method and a list of libraries. The execution of the program starts by executing the body of the `main` method.

[11] Dart Kernel's binary representation `https://github.com/dart-lang/sdk/blob/master/pkg/kernel/binary.md`

[12] This also allows us to have unique representation of variables and ignore concerns such as α-conversion.

$E \in$ **Expr** $::=$	
$\mathscr{L}$	literal
x	local variable access
$x = E$	local variable assignment
$E.X$	property extraction
$E.X = E$	property assignment
`super.`X	super property extraction
`super.`$X = E$	super property assignment
$\{M\}$	static variable access
$\{M\} = E$	static variable assignment
$\{M\}(Es)$	static invocation
$E.X(Es)$	dynamic invocation
E `&&` E	
`new` $C\,(Es)$	instance creation
E `is` T	type test
E `as` T	type cast
`this`	
`rethrow`	
`throw` E	
`await` E	
`let` D `in` E	

Figure 1: Expressions in Dart Kernel

$S \in$ **Stmt** $::=$		
	E	`expression`
	$\{\, Ss \,\}$	`block of statements`
	D	`local variable declaration`
	$T\, x\, (\, As\,)\ S$	`local function declaration`
	L	`labelled statement`
	`break` L	
	`while` $(E)\ S$	
	`switch` (E) `SCs`	
	`continue` `SC`	
	`return` E	
	`return`	
	`try` S `catch` `CCs`	
	`try` S `finally` S	
$L \in$ **Stmt** $::= l\ :\ S$		
$D \in$ **Stmt** $::= T\, x = E$		

Figure 2: Statements in Dart Kernel

4 Object Model

In this section we define the object model for Dart Kernel. The Dart language is a pure object-oriented language and Dart Kernel inherits that property. Objects in Dart Kernel are presented as values with a class component and a data component.

4.1 Classes

Classes in Dart Kernel capture the information needed for dynamic dispatch and runtime type checks.

Before execution of the program, we construct a class table that contains all the classes needed for the execution of a given Dart Kernel program. The class table, denoted as τ, maps a reference of a class, which is the Dart Kernel graph node for the class, to its class definition:

$$\tau = \textbf{ClassReference} \mapsto \textbf{Class}$$

Dart Kernel nodes that operate within a class, such as instance `member` accessors, or Dart Kernel interface types, contain the corresponding class reference.

A class definition is constructed from a Dart Kernel node and is represented as an element of **Class**. An element of **Class** is defined as a triple of superclass, interfaces and members as follows:

$$\begin{aligned}\textbf{Class} =& (\emptyset \cup \textbf{ClassReference}) \\ & \times \text{List}\langle\textbf{ClassReference}\rangle \\ & \times (\textbf{Identifier} \mapsto \textbf{Member})\end{aligned}$$

where the domain of **Member** is defined as:

$$\begin{aligned}\textbf{Member} &= \textbf{Getter} \cup \textbf{Setter} \cup \textbf{Method} \\ \textbf{Getter} &= \mathbb{N} \cup \textbf{Stmt} \\ \textbf{Setter} &= \mathbb{N} \cup \textbf{Formals} \times \textbf{Stmt} \\ \textbf{Method} &= \textbf{Formals} \times \textbf{Stmt}\end{aligned}$$

The domain **Identifier** represents the domain of identifiers.

The components `superclass` and `interfaces` are included in the definition of a class to provide support for runtime type checks, while the component `members` contains information for dynamic dispatch. Runtime type

checks are nominal[13], which is why we need to remember the superclass and the implemented interfaces.

Superclass The first component of elements from the domain **Class** represents the superclass of the given class. We allow this component to be empty. In DART the class "Object" is the top of the class hierarchy, therefore its superclass component is empty, i.e. $\emptyset$, and all other DART classes have a reference to the "Object" class as a superclass component. We define the following function for accessing the superclass component of $c \in \textbf{Class}$:

$$\begin{aligned} \text{superclass} &\in \textbf{Class} \to \textbf{Class}, \\ \text{superclass}(c_1) &= \tau(\pi_1(c_1)) = c_2, \end{aligned}$$

where $c_2 \in \textbf{Class}$ is the class in the class table τ corresponding to the first component of $c_1 \in \textbf{Class}$. [14]

Interfaces A class definition contains a list of references to the interfaces the given class implements, represented as elements of **Class** stored in the class table τ.

Members The third component of a class definition contains the information needed for dynamic dispatch. It contains a complete map of members of a class definition, including inherited members. We use the identifiers from the DART KERNEL nodes to index the map of members. These identifiers are guaranteed to be unique, because DART has the same namespace for instance fields, methods, getters and setters.[15]

We define the following function for access of the `members` component for elements in the domain **Class**:

$$\begin{aligned} \text{members} &\in \textbf{Class} \to (\textbf{Identifier} \mapsto \textbf{Member}), \\ \text{members}(c) &= \pi_3(c), \end{aligned}$$

where $c \in \textbf{Class}$.

[13] Type equivalence and sub-typing are defined in terms of declarations by the programmer.

[14] Here and in the remainder of the report we use π_i to express projection on the i^{th} component in a tuple.

[15] From the perspective of the programmer it may seem that setters and getters can have the same identifier in DART, however identifiers of setters are modified by the front end, which makes them unique and allows co-existence in the same namespace as getters.

We also define the following function for looking up a member, given its identifier:

$$\text{lookupMember} \in \textbf{Class} \times \textbf{Identifier} \to \textbf{Member},$$
$$\text{lookupMember}(c,\ X) = \text{members}(c)(X),$$

where $c \in \textbf{Class}$, $X \in \textbf{Identifier}$.

In Dart Kernel's object model we consider that members for a given class are the instance getters and setters, and the instance methods for that class. This also includes the inherited members.

The accessors from the `member`'s component, i.e. elements of $\textbf{Getter} \cup \textbf{Setter}$, contain both user defined and implicit getters and setters. Dart's language specification explicitly defines the creation of implicit getters and setters for instance fields. In Dart Kernel's object model we represent these implicit getters and setters as elements in $\mathbb{N}$. They represent the position of the field with identifier X in the object's payload that we introduce in the next section.

Dart supports mixins, and Dart Kernel has syntax to support mixins as well. Mixins can be translated to normal classes by the frontend, but this is not done yet because of separate compilation. For example the DDC backend cares about speed of compilation and would suffer in performance if mixin applications are translated to normal classes. The semantics of mixin applications is not specified in this report.

4.2 Objects

In Dart Kernel objects are represented as pairs of a class reference and a payload. We distinguish ordinary and primitive object values.

Ordinary object values Ordinary object values are created with a constructor invocation expression. We represent these values as elements of the domain **ObjectValue**, where **ObjectValue** is defined as follows:

$$\textbf{ObjectValue} = \textbf{ClassReference} \times \text{List}\langle\textbf{Location}\rangle$$

The first component of elements from the domain **ObjectValue** is the class component, which captures the class definition for the given object value. We define the following function for accessing the class component of a value:

$$\text{class} \in \textbf{ObjectValue} \to \textbf{Class},$$
$$\text{class}(v) = \tau(\pi_1(v)) = c,$$

where $c \in$ **Class** is the class corresponding to the first component of $v \in$ **ObjectValue**.

The second component of an element in **ObjectValue** is the fields component. It is represented as a list of final locations for the values of the fields of the object. The size of this list is given by the class component. The layout of locations for the fields for an object value in the fields component is the following:

$$\emptyset \mid \text{fields in Object} \mid \ldots \mid \text{fields in } c$$

The first position of this list is reserved for a fresh location which does not correspond to a field and does not store any value. This implies that all objects have at least one fresh location associated to them and we can use it to define "identical" below. In an implementation, this location would store the object header.

The fields in the object are ordered starting from the fields inherited from the topmost class in the hierarchy. The fields from the immediately enclosing class of the object appear last in this list.

We define the following function for accessing the fields component of an element in **ObjectValue**:

$$\begin{aligned} &\text{fields} \in \textbf{ObjectValue} \to \text{List}\langle\textbf{Location}\rangle, \\ &\text{fields}(v) = \pi_2(v) \end{aligned}$$

We define the following function for accessing a field of an element in **ObjectValue**:

$$\begin{aligned} &\text{field} \in \textbf{ObjectValue} \times \mathbb{N} \to \textbf{Location}, \\ &\text{field}(v,\, i) = \pi_i(\text{fields}(v)) \end{aligned}$$

Primitive object values Dart Kernel supports objects of primitive types, such as bools, numbers and Strings. The semantics of these values depends on raw data and to support it we introduce values that we call primitive object values. Primitive object values are created with dedicated Dart Kernel graph nodes – literals, that have payload of typed data: int, double, String or bool.

We define them as elements of the domain **PrimitiveValue**, where **PrimitiveValue** is defined as follows:

$$\textbf{PrimitiveValue} = \textbf{ClassReference} \times \textbf{Primitive}$$

where $\textbf{Primitive} = \textbf{int} \cup \textbf{bool} \cup \textbf{double} \cup \textbf{String}$.

We define the operation "primitive", that projects the second component of a primitive object value:

$$\begin{aligned} \text{primitive} &\in \textbf{PrimitiveValue} \to \textbf{Primitive} \\ \text{primitive}(v) &= \pi_2(v) = l \end{aligned}$$

where $v \in \textbf{Value}$, $l \in \textbf{Primitive}$.

4.3 Vector values

Dart Kernel supports homogeneous[16] arrays with fixed size via vectors. Vectors are introduced to simplify backend implementations of features such as lists, maps or closures.

Vectors of a given size are created with a vector creation expression. We represent vectors as arrays with fixed size.

$$\begin{aligned} \underline{v} &\in \textbf{Vector} : \textbf{Value}^n \\ &\qquad \text{where } n \in \mathbb{N} \text{ is the size of } \underline{v} \end{aligned}$$

We define the following operations on vector values:

Vector access The i^{th} component of a vector is accessed with the `vectorGet` operator:

$$\texttt{vectorGet}(\underline{v}, i) = \pi_i(\underline{v}),$$

where $i \in \mathbb{N}$.

Vector mutation The i^{th} component of a vector is mutated to store a given value with the `vectorSet` operator:

$$\texttt{vectorSet}(\underline{v}, i, v) = \underline{v}',$$

where

$$\texttt{vectorGet}(\underline{v}', j) = \begin{cases} \texttt{vectorGet}(\underline{v}, j) & \text{if } j \neq i \\ v & \text{otherwise} \end{cases}$$

[16]The Dart Kernel team has not yet seen the need to introduce non-homogeneous structures, but the language may evolve to support structs with typed fields in the future.

The bounds for accessing components of the vector array are not checked and we will say that the transitions where the component that is accessed is out of bounds are not defined.

4.4 Function values

DART KERNEL supports local functions and method tear-offs. For this purpose we introduce the domain of function values, or closures, **FunctionValue**, which is defined as follows:

$$\textbf{FunctionValue} = \textbf{ClassReference} \times \textbf{Formals} \times \textbf{Stmt} \times \textbf{Env}$$
$$\text{FunctionValue}(As,\ S,\ \rho) \in \textbf{FunctionValue}$$

where

$As \in \textbf{Formals}$	formal parameters of the function
$S \in \textbf{Stmt}$	body of the function
$\rho \in \textbf{Env}$	environment in which the function was created

The associated class component for function values is the built-in class "Function".

4.5 Identical

The DART language has a built-in function that can detect object identity. It is defined differently for primitive and ordinary objects.

We define identical to be `false` for all values $v_1 \in \textbf{Dom}_1$ and $v_2 \in \textbf{Dom}_2$ such that $\textbf{Dom}_1 \neq \textbf{Dom}_2$. Otherwise, we define it differently for each of the domains below.

Ordinary object values For non primitive object values, we define "identical" to compare the locations of the objects at position 0 in the field component. We say that two objects are identical if these locations are the same:

$$\text{identical} \in \textbf{ObjectValue} \times \textbf{ObjectValue} \to \textbf{bool}$$
$$\text{identical}(v_1, v_2) = \begin{cases} \texttt{true} & \text{if } \text{field}(v_1,\ 0) == \text{field}(v_2,\ 0) \\ \texttt{false} & \text{otherwise} \end{cases}$$

Primitive object values

$$\text{identical} \in \textbf{PrimitiveValue} \times \textbf{PrimitiveValue} \to \textbf{bool}$$

$$\text{identical}(v_1,\, v_2) = \begin{cases} \texttt{true} & \text{if } \texttt{identical}(\text{primitive}(v_1), \text{primitive}(v_2)) \\ \texttt{false} & \text{otherwise} \end{cases}$$

The function `identical` has the semantics described in Section Object Identity in "Dart Programming Language Specification"[7] for primitive objects in DART.

Vector values Vector values are introduced by transformations to simplify implementations in back end. We do not define `identical` for vector values and calling this built-in function on these values is an error.

Function values Function values are introduced in the semantic domain of values to represent closures at runtime. Similarly to vector values, we do not define `identical` for them.

5 Operational Semantics

Formal specification of a programming language usually is a specification that shows the syntax of the language and describes the exact behaviour of its various features. The syntax of the language represents its structure, while the semantics studies the meaning of grammatically correct programs.

Historically, there have been three main approaches to formalizing and studying the semantics of a programming language: operational, denotational and axiomatic semantics.

In denotational semantics the meanings of programs are modelled by mathematical objects that represent the effects of its constructs, by studying the effect of the program rather then how it was obtained. The axiomatic semantics provides a logical system for proving partial correctness of programs by verifying assertions defined with pre- and post-conditions. The operational semantics is closely related to interpreters and abstract interpretation and it defines how an effect of a computation is obtained by giving an abstraction of how the program is executed on a machine[20].

5.1 Styles of Operational Semantics

Operational semantics is a fundamental tool in language design: it gives an unambigous definition of the behaviour of programs written in the language. It allows reasoning about properties of programs in terms of the constructs of the language and facilitates verification of properties, such as correctness or safety. There are two flavours of operational semantics: big-step, also known as natural semantics, and small-step operational semantics.

Natural Semantics Khan introduced natural semantics in the '80s[17]. In natural semantics we reason about the meaning of programs by looking at the value the program evaluates to, without necessarily specifying the steps to obtain that value. The steps from an initial state to a final state of an execution are implicit.

Small-step Semantics The small-step operational semantics, also known as reduction semantics, or small-step structural operational semantics, introduced by Plotkin[22][23], describes how a program is executed on an abstract machine in detail. It is defined in terms of atomic transitions that describe the local behaviour of a program. Unlike big-step operational semantics, which tells us what the final result of a program is, a step in small-step semantics represents only one step of a computation until the program has been reduced

to a value. With this approach we can define the exact order of evaluation, and the direct control of what is executed and when allows us to model the rich features of the Dart Kernel language, such as labelled statements, structured exception handling and asynchronous execution.

5.2 CESK machine

A common approach of specifying the operational semantics of a language is by giving a definitional interpreter that implements the semantics. In his paper "Definitional Interpreters for Higher-Order Programming Languages" Reynolds defines four categories of interpreters depending on whether the interpreter contains higher-order functions, and whether the order of application (i.e., call by value versus call by name) in the defined language depends upon the order of application in the defining language[24].

To define the operational semantics of Dart Kernel we consider the CESK machine, introduced by Felleisen and Friedman [4], that relates to Reynolds interpreter definitions by trampolining: instead of calling the eval, or apply, function in tail position, it returns the function's arguments which are tagged to indicate the next step of evaluation. The CESK-machine is a state machine where each state has the following components: Control String, Environment, Store and Continuation Code.

States are denoted as quadruples :

$$\sigma =< C, E, S, K > \in \Sigma,$$

where Σ is the domain of configurations.

Transitions from one state to another state are defined with a step function that deterministically produces the next state of the machine based on the control string or continuation code component. A step of the CESK machine starts in some configuration σ and does one step of computation. The step function is implied with the notation '$\Rightarrow$' and a transition from configuration σ_1 to configuration σ_2 is denoted as '$\sigma_1 \Rightarrow \sigma_2$'. The different components of a state are:

Control String The control string component of a configuration captures the code of the program currently being executed.

Environment The environment is a component that associates variables to locations in the store.

$$\rho \in \textbf{Env} = \textbf{VariableDeclaration} \mapsto \textbf{Location},$$

Store The store, similar to the environment, is a structure that maps locations to values. The store is denoted as:

$$s \in \textbf{Location} \mapsto \textbf{Value},$$

where $\textbf{Location} \mapsto \textbf{Value}$ is a map.

In the original CESK machine, locations are integers and values are closures.

Continuation Code The continuation code component captures the behaviour of the program when the control sequence can't be divided further.

6 Small-step Operational Semantics for Dart Kernel

In this section we present the techniques used for defining the small-step operational semantics of Dart Kernel. We present transitions for a subset of its expressions and statements. A more complete set of rules can be found in the report "Operational Semantics of Dart Kernel"[21].

6.1 Abstract machine for Dart Kernel

In Section 5.2 we presented the components of the CESK machine, which is similar to the abstract machine we use in this report. In this section we present the state-based machine used for specification of Dart Kernel's operational semantics and decribe in detail its components.

6.1.1 Configurations

The states of the machine are presented as elements of the domain of configurations, **Configuration**.

In this section we also mention elements of the domain of continuations, **Continuation**, denoted as κ_*. We define this domain in detail in Section 6.1.4.

We differentiate configurations that have a control code component which is part of the program, such as a Dart Kernel expression or a statement, and configurations whose control code component is an element from the semantic domain of continuations.

- Configurations with control component that is part of the program. If the control component is part of the program, it is either an expression, a statement or a semantic list of expressions. In this case the step function will produce the next configuration by focusing on one of the sub-terms of the given expression, statement or semantic list of expressions. Depending on the current control component, the transition step may also result with a configuration that has a continuation as control component.
- Configurations with control component from the semantic domain of continuations. If the control component is an element from the semantic domain of continuations, the step function will apply the continuation.

We differentiate the following states for the CESK machine for the small-step operational semantics of Dart Kernel: evaluation of expression, evaluation of a list of expressions, execution of a statement, handling of an exception or application of a continuation.

We present below the configuration with control component that is part of the program.

EvalConfiguration States in the domain **EvalConfiguration** capture an expression AST node, the current environment and additional components. The additional components will be explained in more detail in the following sections in the report and can be ignored here. We denote these configurations as follows:

$$\langle E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} \tag{1}$$

where

$E \in \mathbf{Expr}$	:	expression that is currently being evaluated
$\rho \in \mathbf{Env}$	:	current environment
$st, cst \in \text{List}\langle \mathbf{Expr} \rangle$	:	current stack traces
$H \in \mathbf{Continuation}$	:	exception handler
$cex \in \mathbf{Value}$	:	current exception
$\kappa_E \in \mathbf{Continuation}$	:	current continuation

The step function for elements of **EvalConfiguration** is defined with respect to the first component, the expression AST node of the configuration.

EvalListConfiguration The first component of the configurations in the domain **EvalListConfiguration** is a semantic list of expression AST nodes and the transition step for this configuration uses this component to produce the next state. The tuple representing these configurations is the following:

$$\langle Es,\ \rho,\ st, H,\ cex,\ cst,\ \kappa_A \rangle_{\text{evalList}} \tag{2}$$

where

$Es \in \text{List}\langle \mathbf{Expr} \rangle$	:	expressions currently being evaluated
$\rho \in \mathbf{Env}$	:	current environment
$st, cst \in \text{List}\langle \mathbf{Expr} \rangle$	:	current stack trace
$H \in \mathbf{Continuation}$	:	exception handler
$cex \in \mathbf{Value}$	:	current exception
$\kappa_A \in \mathbf{Continuation}$	:	current continuation

The step function for elements of **EvalListConfiguration** is defined with respect to the first component, the semantic list of expressions.

ExecConfiguration The first component of configurations in the domain **ExecConfiguration** is a statement AST node and the transition step for this configuration uses this component to produce the next state. The tuple representing these configurations is the following:

$$\langle S,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S \rangle_{\text{exec}} \tag{3}$$

where

S	:	statement currently being executed
ρ	:	current environment
$lbls$	:	current list of break continuations
$clbls$	:	current list of switch continuations
$st, cst \in \text{List}\langle\textbf{Expr}\rangle$	:	current stack trace
$H \in \textbf{Continuation}$	:	exception handler
$cex \in \textbf{Value}$	:	current exception
$\kappa_E \in \textbf{Continuation}$	:	current return continuation
$\kappa_S \in \textbf{Continuation}$	:	current continuation

The step function elements of **ExecConfiguration** is defined with respect to the first component, the statement.

Next we present the configurations whose control component is a continuation. The various kinds of continuations are explained in more detail in the next section.

ValuePassingConfiguration, ApplicationConfiguration, ForwardConfiguration The states from these domains capture an expression continuation (4), application continuation (5) and a statement continuation (6). They also capture a second component: a value, semantic list of values and an environment, respectively.

$$\langle \kappa_E,\ v \rangle_{\text{cont}} \tag{4}$$
$$\langle \kappa_A,\ vs \rangle_{\text{acont}} \tag{5}$$
$$\langle \kappa_S,\ \rho \rangle_{\text{scont}} \tag{6}$$

where

κ_E	:	current expression continuation
κ_A	:	current application continuation
κ_S	:	current statement continuation
v	:	current value
vs	:	current list of values
ρ	:	current environment

The step function of these configurations only applies the first component on the second one.

BreakConfiguration, SwitchConfiguration The states for the domains **BreakConfiguration** (7) and **SwitchConfiguration** (8) capture the corresponding continuations for break and switch labels.

$$\langle \kappa_B \rangle_{\text{breakCont}} \tag{7}$$
$$\langle \kappa_{switch},\ clbls \rangle_{\text{switchCont}} \tag{8}$$

The step function for these states applies the continuation to produce the next state.

ThrowConfiguration The states from the domain **ThrowConfiguration** have a continuation component, which is the exception handler and components that capture the current error and stack trace. Elements of this domain are represented as follows:

$$\langle H,\ v,\ st \rangle_{\text{throw}} \tag{9}$$

where

$H \in$ **Continuation**	:	current exception handler
$v \in$ **Value**	:	current exception
$st \in \text{List}\langle E \rangle$	:	current stack trace

The step function for these states applies the handler to the current exception and stack trace.

EventConfiguration The states from the domain **EventConfiguration** are represented as the tuple:

$$\langle \kappa_N \rangle_{\text{event}} \tag{10}$$

where

$$\kappa_N \in \textbf{Continuation} : \text{ current continuation}$$

6.1.2 Environments

We consider two kinds of environments for Dart Kernel : top-level environments and local environments. The domain of environments for Dart Kernel is defined as follows:

$$\textbf{Env} = (\textbf{VariableDeclaration} \cup \textbf{Member}) \to \textbf{Location}$$

Elements from the domain **Env** are defined as functions, therefore they can not be modified. This implies that we can safely capture environments, a property that is widely used in the remainder of the report.

Note that image domain of these functions is **Location**, rather than **Value**. We define the domain **Location** in Section 6.1.3.

Local environment DART KERNEL is lexically scoped and supports mutable variables. To support this feature, we define elements of the domain **Env**, denoted as ρ that we call local environments.

$$\rho \in \textbf{VariableDeclaration} \to \textbf{Location}$$

Top level environment DART KERNEL supports static and library fields, which are initialized when first accessed and mutations of values stored in these fields are visible in all subsequent execution of statements or evaluation of expressions. To support this feature, we introduce a top level environment, denoted as ρ^M that we call top level environment.

$$\rho^M \in \textbf{Member} \to \textbf{Location}$$

We define the following functions for manipulating elements of the domain **Env**:

Contains We define the function contains as follows:

$$\text{contains} \in \textbf{Env} \times (\textbf{VariableDeclaration} \cup \textbf{Member}) \to \textbf{bool}$$

$$\text{contains}(\rho, x) = \begin{cases} \texttt{true} & \text{if } x \in \mathit{dom}(\rho) \\ \texttt{false} & \text{otherwise} \end{cases}$$

where $\mathit{dom}(\rho)$ is the domain of the environment ρ, i.e., the set of variables or members for which ρ is defined.

Lookup The function lookup is defined as follows:

$$\text{lookup} \in \textbf{Env} \times (\textbf{VariableDeclaration} \cup \textbf{Member}) \to \textbf{Location}$$

$$\text{lookup}(\rho, x) = \rho(x)$$

where $x \in dom(\rho)$. Note that the function lookup is defined only for elements of $dom(\rho)$.

Extend We define the function extend as follows:

$\text{extend} \in \textbf{Env} \times (\textbf{VariableDeclaration} \cup \textbf{Member}) \times \textbf{Value} \to \textbf{Env}$
$\text{extend}(\rho, x, v) = \rho'$
where

$$\forall y \in dom(\rho) \cup x, \;\; \rho'(y) = \begin{cases} \alpha & \text{if } y = x \\ \rho(y) & \text{otherwise} \end{cases}$$

and α is a fresh location that stores the value v.

6.1.3 Store

The store for Dart Kernel, denoted as s, is defined as follows:

$$s \in \textbf{Store} = \textbf{Location} \to \textbf{Value}$$

The store is indexed by locations, elements of **Location** denoted as α. We consider the domain of **Location** to be countably infinite and fresh, unique elements of it can be generated within the CESK machine.

We define the following functions for the store s:

Read We define a function for reading a value from the store given its location as follows:

$\text{read} \in \textbf{Store} \times \textbf{Location} \to \textbf{Value}$
$\text{read}(s, \alpha) = s(\alpha)$

This function is defined only for elements of $dom(s)$.

Update We define a function for updating a location in the store with a given value:

$$\begin{aligned}&\text{update} \in \mathbf{Store} \times \mathbf{Location} \times \mathbf{Value} \to \mathbf{Store}\\&\text{update}(s,\, \alpha,\, v) = s'\\&\text{where}\\&\qquad \forall \alpha' \in dom(s),\;\; s'(\alpha') = \begin{cases} v & \text{if } \alpha' = \alpha \\ s(\alpha') & \text{otherwise} \end{cases}\end{aligned}$$

6.1.4 Continuations

In this section we present an overview of the different kinds of continuations, elements of subsets of the domain **Continuation**. We consider the following kinds of continuations: expression continuations, κ_E, statement continuations, κ_S, application continuations, κ_A, break continuations, κ_B, switch continuations, κ_{switch}, exception handlers, H, and event continuations, κ_N.

Expression Continuations We define the domain of expression continuations as **ExprCont** and we use κ_E for its elements. Expression continuations capture the behaviour after the evaluation of an expression to a value, i.e., when an expression can not be decomposed any further and evaluates to a value with the current transition. We say that $\kappa_E \in \mathbf{ExprCont}$ are applied to values.

We define the domain of statement continuations as **StmtCont** and we use the notation κ_S for its elements. Statement continuations capture the behaviour after the execution of a statement. Statements at the same level, that is a statement in a block, may extend the current local environment and the new bindings are visible for the subsequent statements. Therefore, statement continuations are applied to local environments.

Application Continuation We define the domain of application continuations as **ApplCont** and we use the notation κ_A for its elements. Application continuations capture the behaviour of applying a semantic list of values. They are used for the evaluation of semantic list of expressions, for example, for evaluation of the arguments of a function invocation. Therefore, application continuations are applied to list of values.

Break Continuations We define the domain of break continuations as **BreakCont** and we use the notation κ_B for its elements. Break continuations capture the behaviour of the program when a break to a label is executed.

Switch Continuations We define the domain of switch continuations as **SwitchCont** and we use the notation κ_{switch} for its elements. Switch continuations capture the behaviour of the program when a continue to a label is executed.

Exception Handlers We define the domain of exception handlers as **Handler** and we use the notation H for its elements. Exception handlers are also continuations and they capture the behaviour of the program when an exception in encountered. They are applied to an exception value and a stack trace.

Event Continuation We define the domain of event continuations as **EventCont** and we use the notation κ_N for its elements. Event continuations capture the behaviour for asynchronous executions and are the building components of the event loop G, introduced in Section 6.1.7 below.

For each sub-domain of **Continuation** there are various kinds of continuations. We will introduce the different kinds when they are required to understand the semantics in the report.

6.1.5 Break and Switch labels

In some of the configurations above we introduce two components for support of break and continue to a label in Dart Kernel. We denote these components as:

$$\begin{aligned} \mathit{lbl} &: \text{ break label} \\ \mathit{lbls} &: \text{ semantic list of break labels} \\ \mathit{clbl} &: \text{ continue label} \\ \mathit{clbls} &: \text{ semantic list of continue labels} \end{aligned}$$

Break Label We define a break label as follows:

$$\begin{aligned} &\mathit{lbl} \in \textbf{Stmt} \times \textbf{BreakCont} \\ &\mathit{lbl} = \text{BLabel}(S_{\text{L}},\, \kappa_B) \end{aligned}$$

where S_{L} is the Dart Kernel AST node for labelled statements and $\kappa_B \in$ **BreakCont** captures the next statement continuation and the corresponding environment.

Continue Label We define a continue label as follows:

$$clbl \in \textbf{SwitchCase} \times \textbf{SwitchCont}$$
$$clbl = \text{CLabel}(SC_{\text{L}},\ \kappa_{switch})$$

where SC_{L} is the DART KERNEL structure for labelled switch case and $\kappa_{switch} \in \textbf{SwitchCont}$ captures the statement continuation for executing the body of the case and the corresponding environment.

6.1.6 Exception components

DART KERNEL supports structured exception handling with `try/catch` and `try/finally`, and exception throwing with `throw` and `rethrow`. To support these features, we introduce the following exception components: exception handler, H, stack trace, st, current exception, cex, and current stack trace, cst, for support of exception handling.

Exception Handler A handler is a continuation $H \in \textbf{Handler}$, as defined in Section 6.1.4.

Stack trace The stack trace records the call expressions evaluated that have not returned yet preceeding a given configuration. It is represented as $st \in \text{List}\langle\textbf{Expr}\rangle$ and we use it to show the execution that has led to an exception.

Current exception The component $cex \in (\emptyset \cup \textbf{Value})$ represents the current exception and is set while executing the body of a catch statement and unset otherwise.

Current stack trace Similar to the component cex, this component is defined as $cst \in (\emptyset \cup \text{List}\langle\textbf{Expr}\rangle)$ and represents the stack trace that has led to the exception cex.

6.1.7 Event loop

DART KERNEL supports asynchronous execution and we introduce the event loop, G, to define the behaviour of asynchronous features of the language. The event loop is defined as:

$$G = \text{List}\langle\textbf{EventCont}\rangle$$

We support the usual list operations on the event loop, such as `head`, `tail` and `append` which produce the event continuation that is the first element of G, list of all elements of G except the `head` and list containing all elements of G and an additional element as the last element of the list, respectively.

6.1.8 Implicit components for Dart Kernel's CESK machine

In the remainder of the report we assume the following implicit components for the states of Dart Kernel's CESK machine:

$$\begin{aligned} \rho^M &: \text{ Top-level environment} \\ s &: \text{ Store} \\ G &: \text{ Event loop} \\ \tau &: \text{ Class table} \end{aligned}$$

We say that these components are present in all the states of Dart Kernel's CESK machine, but for the purpose of simplifying the notation, they are omitted in most of the transition rules.

When a component is omitted in a transition rule $C_1 \Rightarrow C_2$, we assume the component from the new configuration C_2 is the same component as the one in the initial configuration C_1. In some transition rules, we modify these components. In that case, we explicitly mention that a given component in the resulting configuration C_2 is not the same as the one in the previous configuration C_1.

6.2 Statements

In this section we present the CESK-transition function starting in configuration in the domain **ExecConfiguration** for execution of statements. In Dart Kernel the execution of a program starts with execution of the body of the `main` method, which is a block of statements.

We describe some of the features supported with statements and present the transition step for their execution.

6.2.1 Expression and block statements

Expressions appear as statements in Dart Kernel. They are evaluated with an expression continuation that discards the value the expression evaluates

to and applied the next statement continuation, as shown in the transition below.

$$\langle E,\, \rho,\, \mathit{lbls},\, \mathit{clbls},\, \mathit{st},\, H,\, \mathit{cex},\, \mathit{cst},\, \kappa_E,\, \kappa_S\rangle_{\text{exec}} \Rightarrow \langle E,\, \rho,\, \mathit{st}, H,\, \mathit{cst},\, \mathit{cex},\, \text{ExpressionEK}(\kappa_S,\, \rho)\rangle_{\text{eval}}$$

$$\langle \text{ExpressionEK}(\kappa_S,\, \rho),\, v\rangle_{\text{cont}} \Rightarrow \langle \kappa_S,\, \rho\rangle_{\text{scont}}$$

Block statements are list of statements in Dart Kernel. If a statement in this list is a variable or function declaration, it extends the current environment and the new binding is visible to the remaining statements in the list. Block statements are executed as follows:

$$\langle \{\, S_1 :: Ss \,\},\, \rho,\, \mathit{lbls},\, \mathit{clbls},\, \mathit{st},\, H,\, \mathit{cex},\, \mathit{cst},\, \kappa_E,\, \kappa_S\rangle_{\text{exec}} \Rightarrow \langle S_1,\, \rho,\, \mathit{lbls},\, \mathit{clbls},\, \mathit{st},\, H,\, \mathit{cex},\, \mathit{cst},\, \kappa_E,\, \kappa'_S\rangle_{\text{exec}}$$

$$\langle \text{BlockSK}(S :: Ss,\, \rho,\, \mathit{lbls},\, \mathit{clbls},\, H,\, \mathit{cst},\, \mathit{cex},\, \kappa_E,\, \kappa_S),\, \rho'\rangle_{\text{scont}} \Rightarrow \langle S,\, \rho',\, \mathit{lbls},\, \mathit{clbls},\, \mathit{st},\, H,\, \mathit{cst},\, \mathit{cex},\, \kappa_E,\, \kappa'_S\rangle_{\text{exec}}$$

$$\langle \text{BlockSK}([],\, \rho,\, \mathit{lbls},\, \mathit{clbls},\, H,\, \mathit{cst},\, \mathit{cex},\, \kappa_E,\, \kappa_S),\, \_\rangle_{\text{scont}} \Rightarrow \langle \kappa_S,\, \rho\rangle_{\text{scont}}$$

where $\kappa'_S = \text{BlockSK}(Ss,\, \rho,\, \mathit{lbls},\, \mathit{clbls},\, H,\, \mathit{cst},\, \mathit{cex},\, \kappa_E,\, \kappa_S)$. Note that statements from the statement list are executed with environment on which the BlockSK is applied, ρ', and only when the list is empty the current application continuation is applied to the captured environment ρ.

6.2.2 Variable Declaration

Variable declaration statements in Dart Kernel extend the environment with a new variable. The statement is executed as follows:

$$\langle \texttt{var}\, x,\, \rho,\, \mathit{lbls},\, \mathit{clbls},\, \mathit{st},\, H,\, \mathit{cex},\, \mathit{cst},\, \kappa_E,\, \kappa_S\rangle_{\text{exec}} \Rightarrow \langle \kappa_S,\, \text{extend}(\rho,\, x,\, \texttt{null})\rangle_{\text{scont}}$$

The variable declaration can optionally have an initializer expression. If the variable declaration has an initializer, the execution will proceed to evaluate the expression. If the expression evaluates to a value, that value will eventually be applied to the newly introduced variable declaration expression continuation, as shown in (12).

$$\langle \text{D},\, \rho,\, \mathit{lbls},\, \mathit{clbls},\, \mathit{st},\, H,\, \mathit{cex},\, \mathit{cst},\, \kappa_E,\, \kappa_S\rangle_{\text{exec}} \Rightarrow \langle E,\, \rho,\, \mathit{st}, H,\, \mathit{cst},\, \mathit{cex},\, \text{VarDeclarationEK}(\rho,\, x,\, \kappa_S)\rangle_{\text{eval}} \tag{11}$$

$$\langle \text{VarDeclarationEK}(\rho,\, x,\, \kappa_S),\, v\rangle_{\text{cont}} \Rightarrow \langle \kappa_S,\, \text{extend}(\rho,\, x,\, v)\rangle_{\text{scont}} \tag{12}$$

The transitions above show how the expression continuation is used to capture the behaviour of a variable declaration statement, when additional steps are needed to reduce the initializer expression to a value.

6.2.3 Return

To support returning from the body of a function, we add an expression continuation component to the configuration for execution of statements, **ExecConfiguration**. We call this continuation a return continuation.

An empty `return` statement applies this continuation immediately to a `null` value.

$$\langle \texttt{return}\,,\ \rho,\ \mathit{lbls},\ \mathit{clbls},\ \mathit{st},\ H,\ \mathit{cex},\ \mathit{cst},\ \kappa_E,\ \kappa_S \rangle_{\text{exec}} \Rightarrow \langle \kappa_E,\ \texttt{null} \rangle_{\text{cont}} \tag{13}$$

when a `return` expression is provided, this expression is evaluated first and the return continuation is added as the next expression continuation.

$$\begin{gathered} \langle \texttt{return}\, E,\ \rho,\ \mathit{lbls},\ \mathit{clbls},\ \mathit{st},\ H,\ \mathit{cex},\ \mathit{cst},\ \kappa_E,\ \kappa_S \rangle_{\text{exec}} \Rightarrow \\ \langle E,\ \rho,\ \mathit{st}, H,\ \mathit{cst},\ \mathit{cex},\ \kappa_E \rangle_{\text{eval}} \end{gathered} \tag{14}$$

6.2.4 Loops

Dart Kernel supports `while`, `for`, `do while` and `for in` loops. The transition rules for `while` and `for` loops can be found in "Operational Semantics of Dart Kernel"[21]. The semantics of `do while` loops is defined in terms of the `while` loop. Defining the semantics of loops does not require introduction of components in the CESK machine, therefore we will not present them in this report.

The semantics of `for in`, which has synchronous and asynchronous variant, relies on the built-in class "iterator" which was not specified in the context of this project.

6.2.5 Labelled statements and `break`

Dart Kernel supports labelled statements, $\mathrm{L} = l : S$ and `break` L statements. `break` statements break from the target labelled statement and proceed to execution of the rest of the program after it. Execution of labelled

statements modifies the break labels component of the current. It adds a new label as head of the break labels component *lbls*. The new label captures the current statement continuation and the corresponding environment.

$$\begin{aligned}\langle \mathrm{L},\ \rho,\ lbls,\ clbls,\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S\rangle_{\mathrm{exec}} \Rightarrow \\ \langle S,\ \rho,\ lbl :: lbls,\ clbls,\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S\rangle_{\mathrm{exec}}, \\ \text{where } \mathrm{L} = l : S,\ lbl = \mathrm{BLabel}(\mathrm{L},\ \kappa_B),\ \kappa_B = \mathrm{BreakBK}(\kappa_S,\ \rho)\end{aligned} \tag{15}$$

Execution of a `break` with target a labelled statement can only occur inside the labelled statement. When a `break` L is executed, the corresponding continuation is looked up in the break labels component *lbls*. The CESK machine then transitions to a **BreakConfiguration** state that applies the break continuation accordingly.

$$\begin{aligned}\langle \texttt{break}\,\mathrm{L},\ \rho,\ lbls,\ clbls,\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S\rangle_{\mathrm{exec}} \Rightarrow \langle \kappa_B\rangle_{\mathrm{breakCont}}, \\ \text{where } lbl = \mathrm{BLabel}(\mathrm{L}',\ \kappa_B) \in lbls \text{ such that } \mathrm{L}' = \mathrm{L}\end{aligned} \tag{16}$$

The break continuation introduced above, BreakBK(κ_S, ρ), applies the captured statement continuation to the environment.

6.2.6 Labelled `switch` cases and `continue`

Dart Kernel supports `switch` and `continue` statements. To support execution of these statements, we introduce a continue labels component *clbl* in the configuration for execution of statements **ExecConfiguration**. This component is modified when executing the bodies of the different switch cases. The execution of `switch` statement proceeds first with evaluation of the target expression. We introduce a new expression continuation, SwitchEK, that captures all the components from the **ExecConfiguration**. These components are needed to later execute the body of the `switch` statement, as well as the `switch` cases denoted as SCs.

$$\begin{aligned}\langle \texttt{switch (}E\texttt{)}\ SCs,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S\rangle_{\mathrm{exec}} \Rightarrow \\ \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa'_E\rangle_{\mathrm{eval}}, \\ \text{where } \kappa'_E = \mathrm{SwitchEK}(SCs,\ \rho,\ lbls,\ clbls',\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S)\end{aligned}$$

We also modify the *clbls* component to add the corresponding continue labels, CLabel. The new continue labels component *clbls*′ is constructed by adding

labels for all `switch` cases in the statement, so that the following holds:

$$
\begin{aligned}
&\forall \mathit{clbl} \in \mathit{clbls}' \\
&\quad \mathit{clbl} \in \mathit{clbls} \text{ or} \\
&\quad \mathit{clbl} = \mathrm{CLabel}(\mathit{SC},\ \kappa_{\mathit{switch}}) \\
&\qquad \text{where} \\
&\qquad\quad \kappa_{\mathit{switch}} = \mathrm{ContinueK}(\mathit{SC}\,s',\ S,\ \rho,\ \mathit{lbls},\ \mathit{clbls}\ st,\ H,\ \mathit{cex},\ \mathit{cst},\ \kappa_E, \kappa'_S) \\
&\qquad\quad \mathit{SC} = \texttt{case}\, v_1, \ldots, v_i : S \in \mathit{SC}\,s, \\
&\qquad\quad \mathit{SC}\,s' = \mathit{SC}\,s \setminus \mathit{SC} \\
&\qquad\quad \kappa'_S = \mathrm{SwitchExitSK}() \text{ if } \mathit{SC} \text{ is the last } \texttt{switch} \text{ case}, \\
&\qquad\quad \kappa'_S = \mathrm{ThrowingSwitchExitSK}() \text{ otherwise}
\end{aligned}
\tag{17}
$$

We introduce the continue continuation, ContinueK, that captures the components necessary for the execution of a continue statement in the body of the `switch` case. Note that we also capture the other `switch` cases in the current `switch` statement in each of these continuations. This is necessary to construct the correct continue label list when this continuation is applied.

We denote `switch` case as $\mathit{SC} = \texttt{case}\, v_1, \ldots, v_i : S$.

In Dart Kernel the case contains a list of expressions. We consider the expressions in a `switch` case as values to simplify the transitions shown below. To implement the correct behaviour, additional computations are needed in order to evaluate the list of expressions in the case to the list of values considered here.

The statement body of a matching `switch` case is executed with a statement continuation that will throw when reached if the list of remaining cases is non-empty. We add such a statement continuation because Dart Kernel does not support implicit fall-through cases and explicit change of flow is required with `return, throw, rethrow, break,` or `continue`.

$$
\begin{aligned}
&\langle \mathrm{SwitchEK}(\mathit{SC} :: \mathit{SC}\,s,\ \rho,\ \mathit{lbls},\ \mathit{clbls},\ st,\ H,\ \mathit{cst},\ \mathit{cex},\ \kappa_E,\ \kappa_S),\ v \rangle_{\mathrm{cont}} \Rightarrow \\
&\qquad \langle S,\ \rho,\ \mathit{lbls},\ \mathit{clbls}',\ st,\ H,\ \mathit{cex},\ \mathit{cst},\ \kappa_E,\ \kappa_S \rangle_{\mathrm{exec}}, \\
&\qquad\quad \text{where } \mathit{SC} = \texttt{case}\, v_1, \ldots, v_i : S,\ v \in v_1, \ldots, v_i \\
&\qquad\quad \mathit{clbls}' = \mathit{clbls} \setminus \mathrm{CLabel}(\mathit{SC},\ \kappa_{\mathit{switch}}) \\
&\qquad\quad \kappa_S = \mathrm{SwitchExitSK}() \text{ if } \mathit{SC} \text{ is the last } \texttt{switch} \text{ case}, \\
&\qquad\quad \kappa_S = \mathrm{ThrowingSwitchExitSK}() \text{ otherwise}
\end{aligned}
$$

We modify the continue label list and remove the label corresponding to the current `switch` case, since only other `switch` cases can be targets of an enclosed `continue` statement.

When a `switch` case is non-matching, we proceed with the next case, as folows:

$$\langle \text{SwitchEK}(SC :: SCs,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S),\ v\rangle_{\text{cont}} \Rightarrow \\ \langle \text{SwitchEK}(SCs,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S),\ v\rangle_{\text{cont}},$$

$$\langle \text{SwitchEK}([],\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S),\ v\rangle_{\text{cont}} \Rightarrow \langle \kappa_S,\ \rho\rangle_{\text{scont}}$$

A `continue` statement has another `switch` case as target and when executed, proceeds to executing the body of the corresponding `switch` case, from the current continue labels component, *clbls*.

$$\langle \texttt{continue}\ SC_{\text{L}},\ \rho,\ lbls,\ clbls,\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} \Rightarrow \\ \langle \kappa_{switch}\rangle_{\text{switchCont}}, \\ \text{where } clbl = \text{CLabel}(SC'_{\text{L}},\ \kappa_{switch}) \in clbls \text{ with } SC'_{\text{L}} == SC_{\text{L}}$$

The newly introduced continue continuation, ContinueK, is applied as follows:

$$\langle \text{ContinueK}(SCs',\ S,\ \rho,\ lbls,\ clbls\ st,\ H,\ cex,\ cst,\ \kappa_E, \kappa'_S)\rangle_{\text{switchCont}} \Rightarrow \\ \langle S,\ \rho,\ lbls,\ clbls',\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa'_S\rangle_{\text{exec}} \\ \text{where } clbls' \text{ is constructed from } clbls \text{ and } SCs' \text{ as in the equation (17)}$$

6.2.7 Exceptions

Dart Kernel supports structured exception handling with `try/finally` and `try/catch` statements.

try/finally `try/finally` statements ensure that a finalizer statement is executed before exiting the current `try/finally` statement. This implies that the finalizer statement is executed after the execution of the body, but also that it should be executed before an exception is thrown, a return statement, a break statement of an enclosing labelled statement or a continue statement to another switch case is executed.

Therefore, execution of `try/finally` statement modifies the current statement continuation, the exception handlers, the return continuation, the break and continue label components.

$$\langle \texttt{try}\ S_0\ \texttt{finally}\ S_1,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} \Rightarrow \\ \langle S_0,\ \rho,\ lbls',\ clbls',\ H',\ cex,\ cst,\ \kappa'_E,\ \kappa'_S\rangle_{\text{exec}}$$

where

$$
\begin{aligned}
H' &= \text{FinallyH}(S_1,\ \rho,\ \mathit{lbls},\ \mathit{clbls},\ \mathit{st},\ H,\ \kappa_E),\\
\kappa'_E &= \text{FinallyReturnEK}(S_1,\ \rho,\ \mathit{lbls},\ \mathit{clbls},\ \mathit{st},\ H,\ \kappa_E),\\
\kappa'_S &= \text{FinallySK}(S_1,\ \rho,\ \mathit{lbls},\ \mathit{clbls},\ \mathit{st},\ H,\ \kappa_E,\ \kappa_S),\\
\mathit{lbls}' &= \{\text{Label}(L, \kappa'_B) \mid \text{Label}(L, \kappa_B) \in \mathit{lbls}\},\\
\kappa'_B &= \text{FinallyBreak}(S_1,\ \rho,\ \mathit{lbls},\ \mathit{clbls},\ \mathit{st},\ H,\ \mathit{cst},\ \mathit{cex},\ \kappa_E,\ \kappa_B),\\
\mathit{clbls}' &= \{\text{CLabel}(SC_{\text{L}},\ \kappa'_{\mathit{switch}}) \mid \text{CLabel}(SC_{\text{L}},\ \kappa_{\mathit{switch}}) \in \mathit{clbls}\},\\
\kappa'_{\mathit{switch}} &= \text{FinallyContinue}(S_1,\ \rho,\ \mathit{lbls},\ \mathit{clbls},\ \mathit{st},\ H,\ \mathit{cst},\ \mathit{cex},\ \kappa_E,\ \kappa_{\mathit{switch}})
\end{aligned}
$$

We introduce a number of continuations here that ensure the execution of the finalizer statement. Their application is straight forward: they produce a state in **ExecConfiguration** that executes the finalizer statement with the appropriate components captured in the continuations. In the case of the introduced handler H', it additionally ensures that the next handler, H, is applied.

try/catch `try/catch` statements modify the exception handlers component and add an exception handler that captures the catch clauses of the statement and the remaining components needed for their execution.

$$
\begin{aligned}
&\langle \texttt{try}\ S\ \texttt{catch}\ cs,\ \rho,\ \mathit{lbls},\ \mathit{clbls},\ \mathit{st},\ H,\ \mathit{cex},\ \mathit{cst},\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} \Rightarrow\\
&\qquad \langle S,\ \rho,\ \mathit{lbls},\ \mathit{clbls},\ \mathit{st},\ H',\ \mathit{cex},\ \mathit{cst},\ \kappa_E,\ \kappa_S\rangle_{\text{exec}},\\
&\qquad\quad \text{where } H' = \text{CatchH}(cs,\ \rho,\ \mathit{lbls},\ \mathit{clbls},\ \mathit{st},\ H,\ \kappa_E,\ \kappa_S)
\end{aligned}
$$

To support `rethrow` expressions, we add the optional components current error and current stack trace that are only set when executing the body of a matching `catch` clause.

6.3 Expressions

In this section we present the CESK-transition function for evaluation of expressions. We consider transitions starting in states from the domains **EvalConfiguration** and **EvalListConfiguration**.

We describe some of the features supported with expressions. The rules described in this section are only for synchronous executions. Definitions of the CESK-transition function for the rest of DART KERNEL's expressions can be found in "Operational Semantics for DART KERNEL"[21].

Evaluation of list of expressions We first introduce steps starting in states from **EvalListConfiguration**. In DART KERNEL list of expressions do not occur as expressions. For simplifying the operational semantics we introduce a semantic list of expressions, which we denote as es. Elements of this list are elements that have an expression to be evaluated and an additional component, a syntactic identifier. The additional component is set when named argument expressions for function invocation are evaluated, and unset otherwise.

States from **EvalListConfiguration** dispatch on the component es and produce the next configuration as follows:

$$\begin{aligned}\langle e :: es,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_A\rangle_{\text{evalList}} \Rightarrow \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E\rangle_{\text{eval}},\\ \text{where } \kappa_E = \text{ExpressionsEK}(e,\ es,\ \rho,\ st,\ H,\ cst,\ cex,\ \kappa_A)\\ E = \pi_1(e)\end{aligned}$$

$$\langle [],\ \rho,\ st, H,\ cst,\ cex,\ \kappa_A\rangle_{\text{evalList}} \Rightarrow \langle \kappa_A,\ []\rangle_{\text{acont}}$$

We introduce an expression continuation, ExpressionsEK, that captures the current components and will proceed to evaluation of the rest of the expression list as shown:

$$\begin{aligned}\langle \text{ExpressionsEK}(e,\ es,\ \rho,\ st,\ H,\ cst,\ cex,\ \kappa_A),\ v\rangle_{\text{cont}} \Rightarrow\\ \langle es,\ \rho,\ st, H,\ cst,\ cex,\ \text{ValueA}(v,\ \kappa_A)\rangle_{\text{evalList}}\end{aligned}$$

We introduce an application continuation that captures the value for the current expression. This application continuation is applied as follows:

$$\langle \text{ValueA}(v,\ \kappa_A),\ vs\rangle_{\text{acont}} \Rightarrow \langle \kappa_A,\ v :: vs\rangle_{\text{acont}}$$

In the rest of the report, when the arrival state is in **EvalListConfiguration**, the steps presented in this section define the transitions from it and we will only present the transition from a **ApplicationConfiguration** when $\kappa_A \neq \text{ValueA}(\_, \_)$.

We assume that semantic values may have a syntactic identifier that corresponds to the syntactic identifier in the corresponding semantic expression e.

Evaluation of an expression The transition step from configurations starting in **EvalConfiguration** dispatches on the expression E, and based on it produces the next configuration. For some expressions it uses the handler H or the continuation κ_E to produce the next configuration. We can say that the step function for states in **EvalConfiguration** considers the following cases for E:

- It reduces to a value in one step.
- It can be decomposed to smaller terms from the program, sub-expressions.
- It implies execution of a statement.
- It throws.

We present below examples for each of the above mentioned cases.

6.3.1 Local Variables

Access The expression x evaluates to a value in one step by looking up the binding to x in the current environment:

$$\langle x,\, \rho,\, st, H,\, cst,\, cex,\, \kappa_E \rangle_{\text{eval}} \Rightarrow \langle \kappa_E,\, v \rangle_{\text{cont}}, \text{, where } v = s(\rho(x))$$

Assignment The expression $x = E$ proceeds to evaluation of the right-hand side of the expression.

$$\langle x = E,\, \rho,\, st, H,\, cst,\, cex,\, \kappa_E \rangle_{\text{eval}} \Rightarrow \langle E,\, \rho,\, st, H,\, cst,\, cex,\, \kappa'_E \rangle_{\text{eval}}$$

where $\kappa'_E = \text{VarSetEK}(x,\, \rho,\, \kappa_E)$.

We introduce the expression continuation VarSetEK that captures the component needed to update the value stored at location $\rho(x)$. This means that this transition step modifies the store. The update of the store on the

right-hand side is also shown below. This continuation is applied to a value and produces the next state as follows:

$$\langle \text{VarSetEK}(X,\ \rho,\ \kappa_E),\ v\rangle_{\text{cont}} \Rightarrow \langle \kappa_E,\ v\rangle_{\text{cont}}$$

where $s_R = \text{update}(s_L, \rho(X), v)$ after transition.

6.3.2 Instance Properties

Property extraction and assignment in Dart Kernel can be extraction or assignment of a property via its identifier or fully resolved.

In the first case, we use dynamic dispatch to lookup the property corresponding to the identifier in the class of the receiver.

When the property extraction or assignment is fully resolved, also called direct, the expression itself points to the Dart Kernel structure that represents the property, hence the lookup step is skipped.

6.3.3 Static Properties

In Dart Kernel static property accessors are fully resolved and the corresponding expressions have pointer to the member record. Static property extraction and assignment may modify the top level environment by adding a new binding to it.

Extraction Static property extraction may result in field access, execution of a user defined getter or a method tear-off.

We first consider field extraction. Static fields are only initialized when accessed, so static property extraction is reduced to a value in one step or implies evaluation of an expression.

When a property has been previously initialized, we have the following case:

$$\langle \{M\},\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E\rangle_{\text{eval}} \Rightarrow \langle \kappa_E,\ s(\rho^M(M))\rangle_{\text{cont}},$$
$$\text{where } M \text{ is a static field and } \text{contains}(\rho^M, M)$$

Otherwise, the property is accessed for the first time. When the accessed

field does not have an initializer, the evaluation proceeds as follows:

$$\langle \{M\},\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} \Rightarrow \langle \kappa_E,\ \texttt{null} \rangle_{\text{cont}},$$
$$\text{where } M \text{ is a static field without an initializer expression,}$$
$$not\,\text{contains}(\rho^M, M),$$
$$\rho_R^M = \text{extend}(\rho_L^M, M, \texttt{null})$$

Otherwise, the initializer expression is evaluated:

$$\langle \{M\},\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} \Rightarrow \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa'_E \rangle_{\text{eval}},$$
$$\text{where } M \text{ is a static field with initializer expression } E,$$
$$not\,\text{contains}(\rho^M, M),$$
$$\kappa'_E = \text{StaticGetEK}(M,\ \kappa_E)$$

We introduce StaticGetEK that captures the member and the expression continuation and it is applied as follows:

$$\langle \text{StaticGetEK}(M,\ \kappa_E),\ v \rangle_{\text{cont}} \Rightarrow \langle \kappa_E,\ v \rangle_{\text{cont}}, \text{where } \rho_R^M = \text{extend}(\rho_L^M,\ M,\ v)$$

When the property accessors is a user defined getter, the evaluation of the function proceeds as follows:

$$\langle \{M\},\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} \Rightarrow$$
$$\langle S,\ \rho_{empty},\ [],\ [],\ \{M\} :: st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S \rangle_{\text{exec}}$$
$$\text{where } M \text{ is a getter with body } S,\ \kappa_S = \text{ExitSK}(\kappa_E,\ \texttt{null})$$

Otherwise, we have a method tear-off and the evaluation of the expression proceeds as follows:

$$\langle \{M\},\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} \Rightarrow \langle \kappa_E,\ \text{FunctionValue}(As,\ S,\ \rho') \rangle_{\text{cont}},$$
$$\text{where } \rho' = \rho_{empty} \text{ after transition}$$
$$M \text{ is a method tear-off with formal parameters } As \text{ and body } S$$

Assignment Static property assignment modifies the value stored at the location corresponding to the property when it is a static field, or execute the user defined setter. They may modify the top level environment as well, if a field is accessed for the first time.

The evaluation of the expression proceeds with evaluation of the right-hand side expression.

$$\langle \{M\} = E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} \Rightarrow \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa'_E \rangle_{\text{eval}},$$
$$\text{where } \kappa'_E = \text{StaticSetEK}(M,\ \kappa_E)$$

We introduce StaticSetEK that will update the binding or execute the setter as follows:

$$\langle \text{StaticSetEK}(M,\ \kappa_E),\ v \rangle_{\text{cont}} \Rightarrow \langle \kappa_E,\ v \rangle_{\text{cont}},$$
$$\text{where } \rho_R^M = \text{extend}(\rho_L^M,\ M,\ v) \text{ and } \textit{not}\,\text{contains}(\rho_L^M, M)$$
$$s_R = \text{update}(s_L, \rho_L^M(M), v) \text{ and } \text{contains}(\rho_L^M, M)$$
$$\langle \text{StaticSetEK}(M,\ \kappa_E),\ v \rangle_{\text{cont}} \Rightarrow \langle S,\ \rho,\ [],\ [],\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S \rangle_{\text{exec}},$$
$$\text{where } M \text{ is a static setter with formal parameter } A \text{ and body } S$$
$$\kappa_S = \text{ExitSK}(\kappa_E,\ v),$$
$$\rho = \text{extend}(\rho_{empty}, A, v)$$

We introduce a new statement continuation ExitSK that ensures the correct execution of the program when explicit control flow with `return` is missing in the body of a statement.

6.4 Invocations

Dart Kernel has an in-memory representation of functions that is referenced in local functions, instance methods and constructors.

Function Invocations in Dart Kernel are expressed as static and instance invocations. Static invocations are fully resolved and the corresponding Dart Kernel expression has a reference to the function node to be invoked. Instance invocations appear in two flavours: with a reference to the function node to be invoked and with an identifier, X, where dynamic dispatch is necessary.

Constructor Constructors in Dart Kernel appear only as generative constructors. The Dart language has support for factory constructors, which are translated to static methods in Dart Kernel.

The CESK-transition function for invocations is presented in detail in the specification. It does not require special components in the CESK states, hence we are omitting them in this report. An interesting thing to note for invocations is that they modify the stack trace by appending the invocation expression to the stack trace component.

6.5 Exceptions

In Section 6.2.7 we present `try/catch` and `try/finally` statements that introduce exception handlers. Exceptions are thrown with the expressions `throw` and `rethrow`.

Throw `throw` expressions in DART KERNEL explicitly modify the flow of the execution of the program. The target of a `throw` is an expression, which is first evaluated to a value:

$$\begin{aligned}&\langle \texttt{throw}\, E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} \Rightarrow \\ &\quad \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \text{ThrowEK}((\texttt{throw}\, E) :: st,\ H) \rangle_{\text{eval}}\end{aligned}$$

We introduce an expression continuation, ThrowEK, that captures the current handler and stack trace. When applied, this continuation will produce the corresponding state in **ThrowConfiguration**:

$$\langle \text{ThrowEK}(st,\ H),\ v \rangle_{\text{cont}} \Rightarrow \langle H,\ v,\ st \rangle_{\text{throw}}$$

Rethrow `rethrow` expressions appear in the body of catch statements. They apply the current handler to the current exception and stack trace.

$$\langle \texttt{rethrow},\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} \Rightarrow \langle H,\ cex,\ cst \rangle_{\text{throw}}$$

The components current exception and stack trace are set accordingly when the body of a catch is executed.

6.6 Asynchronous execution

Asynchronous execution in DART is supported with `async` functions, `await` expressions, and asynchronous `for in` loops.

DART has an event loop on which asynchronous code is scheduled to be executed. The tasks scheduled in the event loop are called microtasks[12][19]. Since DART KERNEL has DART-like semantics, it also supports asynchornous execution, with the event loop component introduced in Section 6.1.7. In the DART 2.0 specification, execution of asynchronous functions starts synchronously, i.e., when the function is invoked. A microtask for the invocation is scheduled on the event loop and the execution is paused only when the body

of the function is suspended. One example when the execution is suspended is the evaluation of an `await` expression.

To support asynchronous execution we introduced the configuration domain **EventConfiguration**, which applies continuations from the domain of event continuations, $\kappa_N \in$ **EventCont**.

6.7 Execution of a program

The CESK machine has an external driver loop that starts in state C_{intial} and transitions with the defined step to the next state until the machine is in C_{final} state. When C_{final} is reached, the driver loop stops. The step function and states $C_{initial}$ and C_{final} are defined below.

Step function The transition function is defined as a step function from every configuration. Different configurations have different step functions and we have briefly introduced them in the previous sections of the report.

We define an initial state, $C_{initial} \in$ **ExecConfiguration** and a final state $C_{final} \in$ **Configuration**.

Initial state The initial state is defined as the tuple:

$$C_{initial} = \langle S,\ \rho_{empty},\ [],\ [],\ [],\ H,\ \emptyset,\ \emptyset,\ \kappa_E,\ \kappa_S \rangle_{\text{exec}}$$

where

S	:	the body of the `main` method,
ρ_{empty}	:	empty environment,
$[]$	:	empty lists of `break` and `continue` labels
$[]$	:	empty list of expressions for stack trace
H	=	TopH
$\emptyset$	:	empty current exception and stack trace
κ_E	=	MainEK
κ_S	=	MainSK

We introduce the following continuations, which produce an event configuration for executing asynchronous code or transition the machine to a final state.

- An exception handler, $H =$ TopH. When this continuation is applied, it transitions the machine to the final state.

$$\langle \text{TopH},\ v,\ st \rangle_{\text{throw}} \Rightarrow \langle\ \rangle_{final}$$

- A return continuation, κ_E = MainEK. When this continuation is applied and the event loop G is empty, it transitions the machine to the final state. Otherwise, it transitions to an event configuration.

$$\begin{array}{lll} \langle \text{MainEK}, v\rangle_{\text{cont}} & \Rightarrow \langle\,\rangle_{final} & \text{where } G = [] \\ \langle \text{MainEK}, v\rangle_{\text{cont}} & \Rightarrow \langle \texttt{head}(G)\rangle_{\text{event}} & \text{where } G_R = \texttt{tail}(G_L) \text{ otherwise} \end{array}$$

- A statement continuation, κ_S = MainSK. When this continuation is applied and the event loop G is empty, it transitions the machine to the final state. Otherwise, it transitions to an event configuration.

$$\begin{array}{lll} \langle \text{MainSK}, \rho\rangle_{\text{scont}} & \Rightarrow \langle\,\rangle_{final} & \text{where } G = [] \\ \langle \text{MainSK}, \rho\rangle_{\text{scont}} & \Rightarrow \langle \texttt{head}(G)\rangle_{\text{event}} & \text{where } G_R = \texttt{tail}(G_L) \text{ otherwise} \end{array}$$

Final state The final state is denoted as an empty tuple $C_{final} = \langle\,\rangle_{final}$ and it is used to detect the end of the execution of the program.

7 Implementation in Dart

We have implemented an interpreter for Dart Kernel that supports a subset of the features defined in the specification of the operational semantics.

We choose to define the semantics in a small-step manner with a state-machine which can easily be implemented as an interpreter, because it provides architecture for support of mutation, recursion, exceptions and asynchronous execution. We can support all these features with an implementation in applicative language, which is important since one of the goals of our implementation is to provide clarity. Additionally, this choice allows writing the abstract machine in Coq.

The implementation is in Dart, but many of the features of the Dart language are not used. A motivation for writing an interpreter for the operational semantics is to provide a reference implementation that can be used as testbed for new language features or to show correctness of transformations by demonstrating that they preserve the semantics. We use Dart in an applicative style in order to achieve clarity, so a deep understanding of how the Dart language works is not needed to understand the reference implementation of Dart Kernel. A meta-circular implementation in Dart, by using all of its features, would be of little value.

The implementation follows closely the described CESK machine in Section 5. We present the step function in trampolined style, where the execution is organized in a single driver loop. Trampolined style means that a single "scheduler" loop, called `trampoline` manages all transfers of control[5].

The execution of a Dart Kernel program proceeds in discrete steps by performing one computation at a time. The driver loop of the interpreter is shown in Code 1.

Example of how configurations are presented fot the CESK machine can be seen in Code 2 and in Code 3 for continuations.

Code 1: Driver loop in Dart

```
1  while(config != const FinalConfiguration()) {
2      config = config.step();
3  }
```

Code 2: Configurations implementation in Dart

```
1  abstract class Configuration{
2      Configuration step();
3  }
4
5  class FinalConfiguration extends Configuration {
```

```
6           const FinalConfiguration();
7
8           Configuration step() =>
9               throw 'step is not defined for FinalConfiguration.';
10      }
11
12      class EvalConfiguration extends Configuration {
13          final Expression expression;
14          final Environment environment;
15          final ExceptionComponents exceptionComponent;
16          final ExpressionContinuation continuation;
17
18          Configuration step() => eval(expression, this);
19
20          /// ...
21      }
22
23      class ExecConfiguration extends Configuration {
24          final Statement statement;
25          /// Represents the current state: exception components
26          /// environment and continuations.
27          final State state;
28
29          Configuration step() => exec(statement, environment, state);
30
31          /// ...
32      }
33
34      /// ...
```

Code 3: Continuations implmentation in DART

```
1       abstract class Continuation{}
2
3       abstract class ExpressionContinuation extends Continuation {
4           Configuration call(Value v);
5       }
6
7       abstract class StatementContinuation extends Continuation {
8           Configuration call(Environment env);
9       }
10
11      /// ...
```

8 Future work

While this project sets the grounds for formalization of the operational semantics of the core Dart language, Dart Kernel, it lacks proper formalization of some of Dart Kernel's features. In this section we talk how to further contribute to the formalization of the operational semantics of the core language Dart Kernel.

To achieve completeness, the specification of the asynchronous part of the language needs to be adapted to support the latest Dart semantics, where asynchronous functions are executed synchronously until explicitly suspended.

The reference interpreter lacks some of the features described in the specification, such as reference implementation of `switch` statements, `for` loops and `async` execution.

Dart Kernel also has a static semantics, with a formal type system that is sound and decidable, and a linking semantics. To achieve completeness in the operational semantics and the reference implementation, the static semantics of Dart Kernel needs to be formally specified. Dart Kernel also serves as a specification of a separately compiled modules. Some of the features in Dart Kernel do not have a runtime meaning, but have a link-time semantics, that needs to be specified.

This will allow us to formally specify Dart Kernel in Coq and show properties of the language.

9 Conclusion

In this report we presented the core Dart language, Dart Kernel and started the formalization of its operational semantics. We designed an abstract machine that implements the semantics with components to handle the rich features of the language.

We implemented a reference implementation for the Dart Kernel language as an interpreter in Dart, that can be used to execute Dart Kernel programs.

10 Acknowledgments

I would like to thank Professor Viktor Kuncak for supervising this project. I am grateful to Kevin Millikin, my supervisor at Google, for giving me the opportunity to work on this project and for following me closely and providing guidance. I would also like to thank my team at Google, the Dart Kernel team, and especially Dmitry Stefantsov, for working closely with me during the last 6 months.

References

[1] Tim Chevalier Andrew Tolmach and The GHC Team. *An External Representation for the GHC Core Language*. URL: www.haskell.org/ghc/docs/6.12.2/core.pdf.

[2] Nick Benton et al. "Shrinking Reductions in SML.NET". In: *Proceedings of the 16th International Conference on Implementation and Application of Functional Languages*. IFL'04. Lübeck, Germany: Springer-Verlag, 2005, pp. 142–159. ISBN: 3-540-26094-3, 978-3-540-26094-3. DOI: 10.1007/11431664_9. URL: http://dx.doi.org/10.1007/11431664_9.

[3] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381 –392. ISSN: 1385-7258. DOI: http://dx.doi.org/10.1016/1385-7258(72)90034-0. URL: http://www.sciencedirect.com/science/article/pii/1385725872900340.

[4] Matthias Felleisen and Daniel P. Friedman. "Control operators, the SECD-machine, and the lambda-calculus". In: *3rd Working Conference on the Formal Description of Programming Concepts*. Aug. 1986.

[5] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. "Trampolined Style". In: *SIGPLAN Not.* 34.9 (Sept. 1999), pp. 18–27. ISSN: 0362-1340. DOI: 10.1145/317765.317779. URL: http://doi.acm.org/10.1145/317765.317779.

[6] Google. *Dart Dev Compiler (DDC)*. URL: webdev.dartlang.org/tools/dartdevc.

[7] Google. *Dart Programming Language Specification*. 2017. URL: github.com/dart-lang/sdk/tree/master/docs/language.

[8] Google. *Dart VM*. URL: dartlang.org/dart-vm.

[9] Google. *dart2js: The Dart-to-JavaScript Compiler*. URL: webdev.dartlang.org/tools/dart2js.

[10] Google. *Fasta – Fully-resolved AST, Accelerated*. URL: github.com/dart-lang/sdk/tree/master/pkg/front_end/lib/src/fasta.

[11] Google. *Strong Mode Dart*. 2017. URL: www.dartlang.org/guides/language/sound-dart.

[12] Google. *The Event Loop and Dart*. URL: webdev.dartlang.org/articles/performance/event-loop.

[13] Google. *The new AdWords UI uses Dart — we asked why*. 2016. URL: http://news.dartlang.org/2016/03/the-new-adwords-ui-uses-dart-we-asked.html.

[14] David Morgan Google. *Strong Mode Dart*. 2017. URL: medium.com/dartlang/dart-gets-a-type-system-6bd3121772de.

[15] Neville Grech et al. "Static Analysis of Energy Consumption for LLVM IR Programs". In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '15. Sankt Goar, Germany: ACM, 2015, pp. 12–21. ISBN: 978-1-4503-3593-5. DOI: 10.1145/2764967.2764974. URL: http://doi.acm.org/10.1145/2764967.2764974.

[16] *Interpreter for Dart Kernel*. 2017. URL: https://github.com/dart-lang/sdk/tree/master/pkg/kernel/lib/interpreter.

[17] Gilles Kahn. "Natural Semantics". In: *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*. STACS '87. London, UK, UK: Springer-Verlag, 1987, pp. 22–39. ISBN: 3-540-17219-X. URL: http://dl.acm.org/citation.cfm?id=646503.696269.

[18] Pierre Lescanne and Jocelyne Rouyer-Degli. "Explicit Substitutions with De Bruijn's Levels". In: *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*. RTA '95. London, UK, UK: Springer-Verlag, 1995, pp. 294–308. ISBN: 3-540-59200-8. URL: http://dl.acm.org/citation.cfm?id=647194.720811.

[19] Erik Meijer, Kevin Millikin, and Gilad Bracha. "Spicing Up Dart with Side Effects". In: *Queue* 13.3 (Mar. 2015), 40:40–40:59. ISSN: 1542-7730. DOI: 10.1145/2742694.2747873. URL: http://doi.acm.org/10.1145/2742694.2747873.

[20] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. New York, NY, USA: John Wiley & Sons, Inc., 1992. ISBN: 0-471-92980-8.

[21] *Operational Semantics of Dart Kernel*. 2017. URL: https://github.com/zhivkag/dart-kernel-semantics.

[22] Gordon D. Plotkin. *A structural approach to operational semantics*. Lecture notes, Computer Science Department, Aarhus University, 1981. URL: http://homepages.inf.ed.ac.uk/gdp/publications/sos_jlap.pdf.

[23] Gordon D Plotkin. "The origins of structural operational semantics". In: *The Journal of Logic and Algebraic Programming* 60 (2004). Structural Operational Semantics, pp. 3 –15. ISSN: 1567-8326. DOI: http://dx.doi.org/10.1016/j.jlap.2004.03.009. URL: http://www.sciencedirect.com/science/article/pii/S1567832604000268.

[24] John C. Reynolds. "Definitional Interpreters for Higher-order Programming Languages". In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM '72. Boston, Massachusetts, USA: ACM, 1972, pp. 717–740. DOI: 10.1145/800194.805852. URL: http://doi.acm.org/10.1145/800194.805852.

[25] Rok Strniša, Peter Sewell, and Matthew Parkinson. "The Java Module System: Core Design and Semantic Definition". In: *SIGPLAN Not.* 42.10 (Oct. 2007), pp. 499–514. ISSN: 0362-1340. DOI: 10.1145/1297105.1297064. URL: http://doi.acm.org/10.1145/1297105.1297064.

[26] The Coq development team. *The Coq proof assistant*. URL: coq.inria.fr.

[27] The GHC Team. *The GHC compiler*. URL: www.haskell.org.ghc.