Dart Kernel Semantics
(draft)

June 12, 2017

The small-step operational semantics of Dart Kernel is given by an abstract machine in the style of the CESK machine. The machine is defined by a single step transition function where each step of the machine starts in a configuration and deterministically gives a next configuration.

1 Definitions

1.1 Conventions

- Symbols ":" and $\in$ are used interchangeably.
- Names of variables are italicized.
- Names of variables of syntactic domains start with an upper case letter.
- Names of domains are written in bold (e.g. **Expr**).
- Names of configuration and continuation kinds are written in normal text (e.g. VarSetK).
- Names of meta-functions start with lower case letter (e.g. extend).
- Symbol ":=" is read as "denotes".
- "List$\langle \mathbf{X} \rangle$" := domain of meta-lists of elements from domain "**X**". Note that the word "List" here is not in bold, so that it isn't confused with the domain **List** of Dart objects.

1.2 Domains

E, E_i	:	**Expr**	syntactic domain of expressions
Es	:	List⟨**Expr**⟩	
S, S_i	:	**Stmt**	syntactic domain of statements
Ss	:	List⟨**Stmt**⟩	
κ_E	:	**ExprCont**	domain of expression continuations
κ_A	:	**ApplCont**	domain of application continuations
κ_S	:	**StmtCont**	domain of statement continuations
κ_B	:	**BreakCont**	domain of break continuations
κ_{switch}	:	**SwitchCont**	domain of switch continuations
lbl	:	**Label**	domain of labels
lbl	:	List⟨**Label**⟩	
$clbl$	:	**SwitchLabel**	domain of switch labels
$clbl$	:	List⟨**SwitchLabel**⟩	
H	:	**Handler**	syntactic domain of exception handlers
st	:	List⟨**Expr**⟩	domain of stack traces
cex	:	$\emptyset +$ **Value**	domain of current expetion values
cst	:	$\emptyset +$ List⟨**Expr**⟩	domain of current exception stack traces
x	:	**VariableDeclaration**	domain of variable declarations
α	:	**Location**	domain of store locations
v	:	**Value**	domain of values
vs	:	List⟨**Value**⟩	
ρ	:	**Env**	domain of environments

A, A_i	:	**VariableDeclaration**	
As	:	**Formals** = List⟨**VariableDeclaration**⟩	domain of formals

1.3 Meta-functions

1.3.1 Dereferencing

Function "!" is used to "dereference" items stored in environments. It has an implicit argument which is the store of CESK machine.

$$! : \mathbf{Location} \to \mathbf{Value}$$
$$!\alpha = v, \text{ with } v \text{ the value in store at location } \alpha$$

1.3.2 String Concatenation

Function $stringValue$ concatenates strings from the given meta-list.

$$stringValue : \text{List}\langle\mathbf{StringValue}\rangle \to \mathbf{StringValue}$$

1.3.3 Updating Environment

Function "$extend$" creates a new environment by extending the provided environment with new bindings for the variable declarations to fresh labels for each of the provided values.

$$extend : \mathbf{Env} \times \text{List}\langle\mathbf{VariableDeclaration}\rangle \times \text{List}\langle\mathbf{Value}\rangle \to \mathbf{Env}$$

1.4 Notations

$[]$	:=	empty list
$X :: \mathtt{list}$	:=	a meta-list that is constructed by adding element X to the head of the meta-list list

1.5 Configurations for the CESK machine

The state space of the CESK machine contains various kinds of configurations, each containing components for applying the appropriate continuation in order to transition to the next configuration.

$$
\begin{array}{lcl}
\langle E,\ \rho,\ st, H,\ cex,\ cst,\ \kappa_E\rangle_{\mathrm{eval}} & : & \text{EvalConfiguration}\\
\langle Es,\ \rho,\ st, H,\ cex,\ cst,\ \kappa_E\rangle_{\mathrm{evalList}} & : & \text{EvalListConfiguration}\\
\langle S,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S\rangle_{\mathrm{exec}} & : & \text{ExecConfiguration}\\
\\
\langle \kappa_E,\ \nu\rangle_{\mathrm{cont}} & : & \text{ValuePassingConfiguration}\\
\langle \kappa_A,\ \nu s\rangle_{\mathrm{acont}} & : & \text{ApplicationConfiguration}\\
\langle \kappa_S,\ \rho\rangle_{\mathrm{acont}} & : & \text{ForwardConfiguration}\\
\\
\langle H,\ \nu,\ st\rangle_{\mathrm{throw}} & : & \text{ThrowConfiguration}\\
\langle \kappa_B\rangle_{\mathrm{breakCont}} & : & \text{BreakConfiguration}\\
\langle \kappa_{switch}\rangle_{\mathrm{switchCont}} & : & \text{SwitchConfiguration}
\end{array}
$$

1.6 Environment

The environment is a function that maps a variable to a location in the store.

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

1.7 Store

The store, s, maps a location, α, to a value, ν. The store is mutable and should not be confused with a function. It is possible to change the transition rules, so that the store is immutable, and right hand side configurations receive an updated copy of it. However, for the sake of simplicity, a global mutable store is assumed.

$$s : \textbf{Location} \to \textbf{Value}$$

Therefore a variable look-up will consist of looking up the address of a variable from the environment with $\alpha = \rho(x)$ and reading the stored value ν with $!\alpha$. For definition of "!" see Section 1.3.1

1.8 Continuations

Continuation is the function that represents the rest of the program and is has the information needed to resume the execution of the program. There are various types of continuations depending on the next statement to be executed or next expression to be evaluated.

1.9 Values

1.10 Literal values

$$
\begin{array}{lcl}
\nu \in \textbf{LiteralValue} & = & \textbf{int} + \textbf{bool} + \textbf{double}\\
& & +\textbf{List} + \textbf{Map} + \textbf{String} + \textbf{Symbol} + \textbf{Type}
\end{array}
$$

1.11 Object values

$$
\begin{array}{lll}
\textbf{ObjectValue} & : & \textbf{Class} \times \text{List}\langle\textbf{Location}\rangle \\
\textbf{Class} & : & superclass \times interfaces \times fields \times getters \times setters \times methods
\end{array}
$$

1.12 Function values

$$\textbf{FunctionValue} : \textbf{Formals} \times \textbf{Stmt} \times \textbf{Env}$$

2 Semantics

2.1 Expression evaluation

2.1.1 Basic literal evaluation

Kernel literals are evaluated to a value $V \in$ **LiteralValue**. Transitions of the CESK machine for basic literals are presented below:

$$
\begin{array}{lll}
\langle \text{IntLiteral}(v),\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} & \Rightarrow & \langle \kappa_E,\ V \rangle_{\text{cont}},\ \ V = \text{IntLiteral}(v) \in \textbf{int} \\
\langle \text{DoubleLiteral}(v),\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} & \Rightarrow & \langle \kappa_E,\ V \rangle_{\text{cont}},\ \ V = \text{DoubleLiteral}(v) \in \textbf{double} \\
\langle \text{BoolLiteral}(v),\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} & \Rightarrow & \langle \kappa_E,\ V \rangle_{\text{cont}},\ \ V = \text{BoolValue}(v) \in \textbf{bool} \\
\langle \text{StringLiteral}(v),\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} & \Rightarrow & \langle \kappa_E,\ V \rangle_{\text{cont}},\ \ V = \text{StringValue}(v) \in \textbf{String}
\end{array}
$$

2.1.2 Variable assignment and lookup

A variable x are accessed by reading the value stored at location $\rho(x)$ in the store s. Assigning a value to a variable will modify the store and the value stored at location $\rho(x)$.

$$
\begin{array}{lll}
\langle x,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} & \Rightarrow & \langle \kappa_E,\ !\rho(x) \rangle_{\text{cont}} \\
\langle x = E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} & \Rightarrow & \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \text{VarSetK}(\rho,\ x,\ \kappa_E) \rangle_{\text{eval}} \\
\langle \text{VarSetK}(\rho,\ x,\ \kappa_E),\ V \rangle_{\text{cont}} & \Rightarrow & \langle \kappa_E,\ V \rangle_{\text{cont}}, \quad !\rho(x) = V \text{ after transition}
\end{array}
$$

2.1.3 Boolean expresions

$$
\begin{array}{lll}
\langle \neg E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} & \Rightarrow & \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \text{NotK}(\kappa_E) \rangle_{\text{eval}} \\
\langle \text{NotK}(\kappa_E),\ \text{true} \rangle_{\text{cont}} & \Rightarrow & \langle \kappa_E,\ \text{false} \rangle_{\text{cont}} \\
\langle \text{NotK}(\kappa_E),\ \text{false} \rangle_{\text{cont}} & \Rightarrow & \langle \kappa_E,\ \text{true} \rangle_{\text{cont}}
\end{array}
$$

Let $\kappa'_E = \text{AndK}(E_2,\ \rho,\ st,\ H,\ cst,\ cex,\ \kappa_E)$ below:

$$
\begin{array}{lll}
\langle E_1 \wedge E_2,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} & \Rightarrow & \langle E_1,\ \rho,\ st, H,\ cst,\ cex,\ \kappa'_E \rangle_{\text{eval}} \\
\langle \text{AndK}(E,\ \rho,\ st,\ H,\ cst,\ cex,\ \kappa_E),\ \text{true} \rangle_{\text{cont}} & \Rightarrow & \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} \\
\langle \text{AndK}(E,\ \rho,\ st,\ H,\ cst,\ cex,\ \kappa_E),\ \text{false} \rangle_{\text{cont}} & \Rightarrow & \langle \kappa_E,\ \text{false} \rangle_{\text{cont}}
\end{array}
$$

Let $\kappa'_E = \text{OrK}(E_2,\ \rho,\ st,\ H,\ cst,\ cex,\ \kappa_E)$ below:

$$
\begin{array}{lll}
\langle E_1 \vee E_2,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} & \Rightarrow & \langle E_1,\ \rho,\ st, H,\ cst,\ cex,\ \kappa'_E \rangle_{\text{eval}} \\
\langle \text{OrK}(E,\ \rho,\ st,\ H,\ cst,\ cex,\ \kappa_E),\ \text{false} \rangle_{\text{cont}} & \Rightarrow & \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E \rangle_{\text{eval}} \\
\langle \text{OrK}(E,\ \rho,\ st,\ H,\ cst,\ cex,\ \kappa_E),\ \text{true} \rangle_{\text{cont}} & \Rightarrow & \langle \kappa_E,\ \text{true} \rangle_{\text{cont}}
\end{array}
$$

Let $\kappa'_E = \text{ConditionalK}(E_1,\ E_2,\ \rho,\ st,\ H,\ cst,\ cex,\ \kappa_E)$ below:

$$\begin{array}{lcl}
\langle E\ \mathbf{?}\ E_1\ \mathbf{:}\ E_2,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E\rangle_{\text{eval}} & \Rightarrow & \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa'_E\rangle_{\text{eval}} \\
\langle \kappa'_E,\ \text{true}\rangle_{\text{cont}} & \Rightarrow & \langle E_1,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E\rangle_{\text{eval}} \\
\langle \kappa'_E,\ \text{false}\rangle_{\text{cont}} & \Rightarrow & \langle E_2,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E\rangle_{\text{eval}}
\end{array}$$

2.1.4 Let

Let $\kappa'_E = \text{LetK}(E_2,\ \rho,\ x,\ st,\ H,\ cst,\ cex,\ \kappa_E)$ below:

$$\begin{array}{lcl}
\langle \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E\rangle_{\text{eval}} & \Rightarrow & \langle E_1,\ \rho,\ st, H,\ cst,\ cex,\ \kappa'_E\rangle_{\text{eval}} \\
\langle \kappa'_E,\ V\rangle_{\text{cont}} & \Rightarrow & \langle E_2,\ \rho',\ st, H,\ cst,\ cex,\ \kappa_E\rangle_{\text{eval}} \\
 & & \rho' = \mathit{extend}(\rho,\ x,\ V)
\end{array}$$

2.1.5 Static Invocation

$$\langle \mathbf{f}\ (Es),\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E\rangle_{\text{eval}} \ \Rightarrow\ \langle Es,\ \rho,\ st, H,\ cst,\ cex,\ \kappa'_A\rangle_{\text{evalList}}$$

with:

$$\mathbf{f} = \text{FunctionNode}(As, S_{\mathtt{body}}),$$

$$\kappa'_A = \text{StaticInvocationA}(As,\ S_{\mathtt{body}},\ \mathbf{f}\ (Es) :: st,\ H,\ cst,\ cex,\ \kappa_E)$$

2.1.6 Evaluation of list of expressions

$$\begin{array}{lcl}
\langle E :: Es,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_A\rangle_{\text{evalList}} & \Rightarrow & \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa'_E\rangle_{\text{eval}} \\
\langle [],\ \rho,\ st, H,\ cst,\ cex,\ \kappa_A\rangle_{\text{evalList}} & \Rightarrow & \langle \kappa_A,\ []\rangle_{\text{acont}}
\end{array}$$

with:

$$\kappa'_E = \text{ExpressionListK}((Es,\ \rho,\ st,\ cst,\ H,\ cex,\ \kappa_A)$$

Expression list continuation application:

$$\begin{array}{lcl}
\langle \text{ExpressionListK}((Es,\ \rho,\ st,\ cst,\ H,\ cex,\ \kappa_A),\ V\rangle_{\text{cont}} & \Rightarrow & \langle Es,\ \rho,\ st, H,\ cst,\ cex,\ \text{ValueA}(V,\ \kappa_A)\rangle_{\text{evalList}} \\
\langle \text{ValueA}(V,\ \kappa_A),\ Vs\rangle_{\text{acont}} & \Rightarrow & \langle \kappa_A,\ V :: Vs\rangle_{\text{acont}}
\end{array}$$

2.2 Statement execution

2.2.1 Variable Declaration

Let $\kappa'_E = \text{VarDeclarationK}(\rho,\ x,\ \kappa_S)$ below:

$$\begin{array}{lcl}
\langle \mathbf{var}\ x = E,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} & \Rightarrow & \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa'_E\rangle_{\text{eval}} \\
\langle \kappa'_E,\ V\rangle_{\text{cont}} & \Rightarrow & \langle \kappa_S,\ \mathit{extend}(\rho,\ x,\ V)\rangle_{\text{scont}}
\end{array}$$

2.2.2 Block statements

$$\begin{array}{lcl}
\langle \{\},\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} & \Rightarrow & \langle \kappa_S,\ \rho\rangle_{\text{scont}} \\
\langle \{E\},\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} & \Rightarrow & \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \text{ExpressionK}(\kappa_S,\ \rho)\rangle_{\text{eval}} \\
\langle S :: Ss,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} & \Rightarrow & \langle S,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa'_S\rangle_{\text{exec}} \\
\langle \text{BlockSK}(S :: Ss,\ \rho,\ lbls,\ clbls,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S),\ \rho'\rangle_{\text{scont}} & \Rightarrow & \langle S,\ \rho',\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa'_S\rangle_{\text{exec}} \\
\langle \text{BlockSK}([],\ \rho,\ lbls,\ clbls,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S),\ _\rangle_{\text{scont}} & \Rightarrow & \langle \kappa_S,\ \rho\rangle_{\text{scont}}
\end{array}$$

with $\kappa'_S = \text{BlockSK}(Ss,\ \rho,\ lbls,\ clbls,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S)$.

2.2.3 If statement

Let κ'_E = IfConditionK(S_1, S_2, ρ, lbls, clbls, H, cst, cex, κ_E, κ_S) below:

$$\langle \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2,\ \rho,\ \mathsf{lbls},\ \mathsf{clbls},\ \mathsf{st},\ H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} \quad\Rightarrow\quad \langle E,\ \rho,\ \mathsf{st}, H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa'_E\rangle_{\text{eval}}$$

$$\begin{aligned}
\langle \kappa'_E,\ \text{true}\rangle_{\text{cont}} &\quad\Rightarrow\quad \langle S_1,\ \rho,\ \mathsf{lbls},\ \mathsf{clbls},\ \mathsf{st},\ H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa_E,\ \kappa_S\rangle_{\text{exec}}\\
\langle \kappa'_E,\ \text{false}\rangle_{\text{cont}} &\quad\Rightarrow\quad \langle S_2,\ \rho,\ \mathsf{lbls},\ \mathsf{clbls},\ \mathsf{st},\ H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa_E,\ \kappa_S\rangle_{\text{exec}}
\end{aligned}$$

2.2.4 Return statement

$$\begin{aligned}
\langle \textbf{return } E,\ \rho,\ \mathsf{lbls},\ \mathsf{clbls},\ \mathsf{st},\ H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} &\quad\Rightarrow\quad \langle E,\ \rho,\ \mathsf{st}, H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa_E\rangle_{\text{eval}}\\
\langle \textbf{return},\ \rho,\ \mathsf{lbls},\ \mathsf{clbls},\ \mathsf{st},\ H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} &\quad\Rightarrow\quad \langle \kappa_E,\ \textbf{null}\rangle_{\text{cont}}
\end{aligned}$$

2.2.5 Loops

$$\langle \textbf{while } (E)S,\ \rho,\ \mathsf{lbls},\ \mathsf{clbls},\ \mathsf{st},\ H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} \quad\Rightarrow\quad \langle E,\ \rho,\ \mathsf{st}, H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa'_E\rangle_{\text{eval}}$$

with κ'_E, κ'_S = WhileCondK(E, S, ρ, lbls, clbls, H, cst, cex, κ_E, κ_S).

$$\begin{aligned}
\langle \kappa'_E,\ \text{false}\rangle_{\text{cont}} &\quad\Rightarrow\quad \langle \kappa_S,\ \rho\rangle_{\text{scont}}\\
\langle \kappa'_E,\ \text{true}\rangle_{\text{cont}} &\quad\Rightarrow\quad \langle S,\ \rho,\ \mathsf{lbls},\ \mathsf{clbls},\ \mathsf{st},\ H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa_E,\ \kappa'_S\rangle_{\text{exec}}\\
\langle \kappa'_S,\ _\rangle_{\text{scont}} &\quad\Rightarrow\quad \langle E,\ \rho,\ \mathsf{st}, H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa'_E\rangle_{\text{eval}}
\end{aligned}$$

Loops **do while**, **for in**, for can be desugared to while loops with transformations performed before interpreting the program.

2.3 Labels

Kernel supports labelling statements, **L:** S_L, and breaking to L, **break L**, which completes the execution of the labelled statement and proceeds to executing the rest of the program. To support breaking to a label, we add a labels component, lbls, to statement configurations. lbls represents a list of pairs mapping a labelled statement, **L:** S_L, to a break statement continuation, κ_B. Executing a labelled statement introduces a new break label, lbl to the list of labels lbls.

$$\langle \textbf{L: } S_L,\ \rho,\ \mathsf{lbls},\ \mathsf{clbls},\ \mathsf{st},\ H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} \quad\Rightarrow\quad \langle S_L,\ \rho,\ \mathsf{lbl} :: \mathsf{lbls},\ \mathsf{clbls},\ \mathsf{st},\ H,\ \mathsf{cst},\ \mathsf{cex},\ \kappa_E,\ \kappa_S\rangle_{\text{exec}}$$

with:

$$\mathsf{lbl} = \text{Label}(L, \text{Break}(\rho, \kappa_S), \mathsf{lbls})$$