Dart Kernel Semantics
(draft)

June 13, 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.

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

1.3.2 String Concatenation

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

$$stringValue : \text{List}\langle\textbf{StringValue}\rangle \rightarrow \textbf{StringValue}$$

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.

$$extend : \textbf{Env} \times \text{List}\langle\textbf{VariableDeclaration}\rangle \times \text{List}\langle\textbf{Value}\rangle \rightarrow \textbf{Env}$$

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.

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

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, v. 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 v 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

$$\begin{aligned} v \in \textbf{LiteralValue} \quad = \quad & \textbf{int} + \textbf{bool} + \textbf{double} \\ & + \textbf{List} + \textbf{Map} + \textbf{String} + \textbf{Symbol} + \textbf{Type} \end{aligned}$$

1.11 Object values

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

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:

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

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 $\rho(x)$.

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

2.1.3 Boolean expressions

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

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

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

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

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

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

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

2.1.4 Let

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

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

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

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

with:

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

Expression list continuation application:

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

2.2 Statement execution

2.2.1 Variable Declaration

Let κ'_E = VarDeclarationK(ρ, x, κ_S) below:

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

2.2.2 Block statements

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

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

2.2.3 If statement

Let $\kappa'_E = \text{IfConditionK}(S_1, S_2, \rho, \mathsf{lbls}, \mathsf{clbls}, H, \mathsf{cst}, \mathit{cex}, \kappa_E, \kappa_S)$ below:

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

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

2.2.4 Return statement

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

2.2.5 Loops

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

with $\kappa'_E, \kappa'_S = \text{WhileCondK}(E, S, \rho, \mathsf{lbls}, \mathsf{clbls}, H, \mathsf{cst}, \mathit{cex}, \kappa_E, \kappa_S)$.

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

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.

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

with:

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

2.4 Switch

Kernel supports dispatching control among a number of cases with **switch** statements, where the target expression E is evaluated and matched against the different case clauses of the switch statement.

In Kernel, case clauses can have multiple constant expressions and there is no implicit fall-through between cases. Kernel supports continuing to execution of a preceding case clause with continue statements where the target of the continue statement is a preceding case clause, e.g, continue C with C = case $E_{1..i}$: S. To support continue, we add an optional clbls list, similar to the label list, which is set in a switch statement and unset otherwise.

The execution of a switch statement proceeds as follows:

$$\langle \textbf{switch } (E)SCs,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S\rangle_{\text{exec}} \quad \Rightarrow \quad \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E'\rangle_{\text{eval}}$$

with $\kappa_E' = \text{SwitchK}(SCs,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S)$

If there is a matching case clause, its statement is executed with a new statement continuation that will throw when reached. The new statement continuation is necessary because implicit fall-through is not supported and an explicit break of the flow is required (with either of **continue, break, return, throw**).

$$\begin{aligned}&\langle \text{SwitchK}(SC :: SCs,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S),\ V\rangle_{\text{cont}} \Rightarrow\\ &\qquad\qquad \langle S,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \text{ExitSwitchSK}()\rangle_{\text{exec}}\end{aligned}$$

with:

$$SC = \textbf{case } E_1, \ldots, E_i \textbf{ : } S \text{ and } SC \text{ is a matching } \textbf{case} \text{ clause}$$

A case clause is matching if is **default** or there is at least one of the constant expressions in the case clause identical to the target value of the switch.
If a non-matching **case** clause, SC, is the target of a continue statement, it installs a new continue label, in order to support execution of its statement with the appropriate configuration.

$$\begin{aligned}&\langle \text{SwitchK}(SC :: SCs,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S),\ V\rangle_{\text{cont}} \Rightarrow\\ &\qquad\qquad \langle \text{SwitchK}(SCs,\ \rho,\ lbls,\ clbl :: clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S),\ V\rangle_{\text{cont}}\end{aligned}$$

with:

$$\begin{aligned}&clbl = \text{ContinueL}(SC,\ \kappa_{switch}),\\ &\kappa_{switch} = \text{SwitchContinueK}(S,\ \rho,\ lbls,\ clbls,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S)\end{aligned}$$

Otherwise, the execution continues as follows:

$$\begin{aligned}&\langle \text{SwitchK}(SC :: SCs,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S),\ V\rangle_{\text{cont}} \Rightarrow\\ &\qquad\qquad \langle \text{SwitchK}(SCs,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_S),\ V\rangle_{\text{cont}}\end{aligned}$$

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

2.5 Exceptions

Kernel supports structured exception handling with **try/catch** and **try/finally** statements. Exceptions are thrown with **throw** and **rethrow** expressions. To support throwing exceptions, we add a handler, H, and a stacktrace, st, component to expression configurations (and correspondingly, to statement configurations). To support **rethrow**, we add an optional current error, cex, and current stack trace, cst, which are set when inside a catch block and unset otherwise.

$$\begin{aligned}\langle \textbf{throw } E,\ \rho,\ st, H,\ cst,\ cex,\ \kappa_E\rangle_{\text{eval}} &\Rightarrow \langle E,\ \rho,\ st, H,\ cst,\ cex,\ \text{ThrowK}((\textbf{throw } E) :: st,\ H)\rangle_{\text{eval}}\\ \langle \textbf{rethrow},\ \rho,\ st, H,\ inr(V),\ st',\ \kappa_E\rangle_{\text{eval}} &\Rightarrow \langle H,\ V,\ st'\rangle_{\text{throw}}\\ \langle \text{ThrowK}(st,\ H),\ V\rangle_{\text{cont}} &\Rightarrow \langle H,\ V,\ st\rangle_{\text{throw}}\end{aligned}$$

try/catch handlers contain a list of on-catch handlers, environment, break labels, continue switch labels, exception handler, stacktrace, return continuation, and statement continuation. The handlers are tried in order to see if they match against the type of the exception. If none match, the exception is rethrown.

Let κ'_E = Catch((**on T catch (e, s)** S) :: cs, ρ, $lbls$, $clbls$, st, H, κ_E, κ_S) be a catch continuation containing a non-empty list of on-catch handlers:

$$\begin{array}{lll}
\langle \kappa'_E,\ V,\ st' \rangle_{\text{throw}} & \Rightarrow & \langle S,\ \rho',\ lbls,\ clbls,\ st,\ H,\ inr(V),\ inr(st'),\ \kappa_E,\ \kappa_S \rangle_{\text{exec}} \quad \text{if V is T} \\
 & & \text{with } \rho' = extend(\rho, e :: s :: [], V :: st' :: []) \\
\langle \kappa'_E,\ V,\ st' \rangle_{\text{throw}} & \Rightarrow & \langle \text{Catch}(cs,\ \rho,\ lbls,\ clbls,\ st,\ H,\ \kappa_E,\ \kappa_S),\ V,\ st' \rangle_{\text{throw}} \quad \text{otherwise} \\
 & & \text{with } \rho' = extend(\rho, e :: s :: [], V :: st' :: [])
\end{array}$$

When a catch handler with no on-catch handlers is reached, the exception is rethrown.

$$\langle \text{Catch}([],\ \rho,\ lbls,\ clbls,\ st,\ H,\ \kappa_E,\ \kappa_S),\ V,\ st' \rangle_{\text{throw}} \quad \Rightarrow \quad \langle H,\ V,\ st' \rangle_{\text{throw}}$$

try/finally handlers contain a statement, environment, break labels, continue switch labels, exception handler, stacktrace, and return continuation. Note that they do not contain a statement continuation because when control falls off the end of the finally statement the exception is rethrown. The statement is unconditionally executed:

$$\begin{array}{l}
\langle \text{Finally}(S,\ \rho,\ lbls,\ clbls,\ st,\ H,\ \kappa_E),\ V,\ st' \rangle_{\text{throw}} \Rightarrow \\
\qquad \langle S,\ \rho,\ lbls,\ clbls,\ st,\ H,\ inl(),\ inl(),\ \kappa_E,\ \text{RethrowSK}(V,\ st',\ H) \rangle_{\text{exec}}
\end{array}$$

$$\langle \text{RethrowSK}(V,\ st,\ H),\ \_ \rangle_{\text{scont}} \Rightarrow \langle H,\ V,\ st \rangle_{\text{throw}}$$

try/catch statements execute their body with a new handler:

$$\begin{array}{l}
\langle \text{TryCatch}(S, cs),\ \rho,\ lbls,\ clbls,\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S \rangle_{\text{exec}} \Rightarrow \\
\qquad \langle S,\ \rho,\ lbls,\ clbls,\ st,\ H',\ cex,\ cst,\ \kappa_E,\ \kappa_S \rangle_{\text{exec}}
\end{array}$$

where H' = Catch(cs, ρ, $lbls$, $clbls$, st, H, κ_E, κ_S).
try/finally statements execute their body with a new handler and additionally install new break and continue switch labels, a new return continuation, and a new statement continuation:

$$\begin{array}{l}
\langle \text{TryFinally}(S_0, S_1),\ \rho,\ lbls,\ clbls,\ st,\ H,\ cex,\ cst,\ \kappa_E,\ \kappa_S \rangle_{\text{exec}} \Rightarrow \\
\langle S_0,\ \rho,\ lbls',\ clbls',\ st,\ H',\ cex,\ cst,\ \kappa'_E,\ \kappa'_S \rangle_{\text{exec}}
\end{array}$$

where:

$$H' = \text{Finally}(S_0,\ \rho,\ lbls,\ clbls,\ st,\ H,\ \kappa_E)$$

$$\kappa'_E = \text{FinallyReturnK}(S_1,\ \rho,\ lbls,\ clbls,\ st,\ H,\ \kappa_E)$$

$$\kappa'_S = \text{FinallySK}(S_1,\ \rho,\ lbls,\ clbls,\ st,\ H,\ \kappa_E,\ \kappa_S)$$

$$\begin{array}{l}
lbls' = \{\text{Label}(L, \kappa'_B) \mid \text{Label}(L, \kappa_B) \in lbls\} \\
\qquad \text{where } \kappa_B = \text{FinallyBreak}(S_1,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_B)
\end{array}$$

$$\begin{array}{l}
clbls' = \{\text{ContinueL}(C,\ \kappa'_{switch}) \mid \text{ContinueL}(C,\ \kappa_{switch}) \in clbls\} \\
\qquad \text{where } \kappa'_{switch} = \text{FinallyContinue}(S_1,\ \rho,\ lbls,\ clbls,\ st,\ H,\ cst,\ cex,\ \kappa_E,\ \kappa_{switch})
\end{array}$$

2.6 Break

break L statements break from an enclosing label **L** and execute the break continuation corresponding to the label in the list of labels $lbls$.

Let $\mathtt{lbl} = \mathit{Label}(\mathsf{L}', \kappa_B) \in \mathtt{lbls}$ with $\mathsf{L}' == \mathbf{L}$ below:

$$
\begin{array}{lcl}
\langle \textbf{break L}, \rho, \mathtt{lbls}, \mathtt{clbls}, \mathtt{st}, \mathsf{H}, \mathtt{cex}, \mathtt{cst}, \kappa_E, \kappa_S\rangle_{\text{exec}} & \Rightarrow & \langle \kappa_B\rangle_{\text{breakCont}} \\
\langle \text{Break}(\rho, \kappa_S)\rangle_{\text{breakCont}} & \Rightarrow & \langle \kappa_S, \rho\rangle_{\text{scont}} \\
\langle \text{FinallyBreak}(\mathsf{S}, \rho, \mathtt{lbls}, \mathtt{clbls}, \mathtt{st}, \mathsf{H}, \mathtt{cst}, \mathtt{cex}, \kappa_E, \kappa_B)\rangle_{\text{breakCont}} & \Rightarrow & \\
\quad \langle \mathsf{S}, \rho, \mathtt{lbls}, \mathtt{clbls}, \mathtt{st}, \mathsf{H}, \mathtt{cex}, \mathtt{cst}, \kappa_E, \text{BreakSK}(\kappa_B)\rangle_{\text{exec}} & & \\
\langle \text{BreakSK}(\kappa_B), \rho\rangle_{\text{scont}} & \Rightarrow & \langle \kappa_B\rangle_{\text{breakCont}}
\end{array}
$$

2.7 Continue

continue L statements continue to executing the statement of a preceding case clause, labelled with **L** and execute the continue continuation corresponding to the label in the list of labels $\mathtt{clbls}$.

Let $\mathtt{clbl} = \text{ContinueL}(\mathsf{L}', \kappa_{\mathtt{switch}}) \in \mathtt{clbls}$ with $\mathsf{L}' == \mathbf{L}$ below:

$$
\begin{array}{lc}
\langle \textbf{continue L}, \rho, \mathtt{lbls}, \mathtt{clbls}, \mathtt{st}, \mathsf{H}, \mathtt{cex}, \mathtt{cst}, \kappa_E, \kappa_S\rangle_{\text{exec}} & \Rightarrow \\
\quad \langle \kappa_{\mathtt{switch}}\rangle_{\text{switchCont}} & \\
\langle \text{SwitchContinueK}(\mathsf{S}, \rho, \mathtt{lbls}, \mathtt{clbls}, \mathsf{H}, \mathtt{cst}, \mathtt{cex}, \kappa_E, \kappa_S)\rangle_{\text{switchCont}} & \Rightarrow \\
\quad \langle \mathsf{S}, \rho, \mathtt{lbls}, \mathtt{clbls}, \mathtt{st}, \mathsf{H}, \mathtt{cex}, \mathtt{cst}, \kappa_E, \kappa_S\rangle_{\text{exec}} & \\
\langle \text{FinallyContinue}(\mathsf{S}, \rho, \mathtt{lbls}, \mathtt{clbls}, \mathtt{st}, \mathsf{H}, \mathtt{cst}, \mathtt{cex}, \kappa_E, \kappa_{\mathtt{switch}})\rangle_{\text{switchCont}} & \Rightarrow \\
\quad \langle \mathsf{S}, \rho, \mathtt{lbls}, \mathtt{clbls}, \mathtt{st}, \mathsf{H}, \mathtt{cex}, \mathtt{cst}, \kappa_E, \text{SwitchContinueSK}(\kappa_{\mathtt{switch}})\rangle_{\text{exec}} & \\
\langle \text{SwitchContinueSK}(\kappa_{\mathtt{switch}}), \rho\rangle_{\text{scont}} & \Rightarrow \\
\quad \langle \kappa_{\mathtt{switch}}\rangle_{\text{switchCont}} &
\end{array}
$$

2.8 Closures

2.8.1 Function expressions

Kernel supports encapsulating an executable unit of code with function expressions. To support function expressions we introduce $\mathtt{fv} = \text{FunctionValue}(\mathsf{S}_{\mathtt{body}}, \mathsf{As}, \rho) \in \textbf{FunctionValue}$ as a value that has the function statement body, $\mathsf{S}_{\mathtt{body}} \in \textbf{Stmt}$, its list of formal parameters, $\mathsf{As} \in \textbf{Formals}$, and the environment, $\rho \in \textbf{Env}$ in scope of the function expression. This ensures that there are no free variables in the body of $\mathtt{fv}$. $\mathtt{fv} \in \textbf{FunctionValue}$ has only one method, $\mathtt{call}$.

Let F be a function expression with a body $\mathsf{S}_{\mathtt{body}}$ and a list of formal parameters, As.

$$\langle \mathsf{F}, \rho, \mathtt{st}, \mathsf{H}, \mathtt{cex}, \mathtt{cst}, \kappa_E\rangle_{\text{eval}} \;\Rightarrow\; \langle \kappa_E, \text{FunctionValue}(\mathsf{As}, \mathsf{S}_{\mathtt{body}}, \rho)\rangle_{\text{cont}}$$

2.8.2 Function declaration

Kernel supports local function declaration, where a final variable stores a function value. Let F be a function with a body $\mathsf{S}_{\mathtt{body}}$ and a list of formal parameters, As, and let $\mathtt{fun} \in \textbf{VariableDeclaration}$ be the corresponding final variable declaration.

$$\langle \mathtt{fun}(\mathsf{As})\mathsf{S}_{\mathtt{body}}, \rho, \mathtt{lbls}, \mathtt{clbls}, \mathtt{st}, \mathsf{H}, \mathtt{cst}, \mathtt{cex}, \kappa_E, \kappa_S\rangle_{\text{exec}} \;\Rightarrow\; \langle \kappa_E, \text{FunctionValue}(\mathsf{S}_{\mathtt{body}}, \mathsf{As}, \rho')\rangle_{\text{cont}}$$

where $\rho' = \mathtt{extend}(\rho, \mathtt{fun}, \text{FunctionValue}(\mathsf{S}_{\mathtt{body}}, \mathsf{As}, \rho'))$.

2.9 New instance creation

The expression **new** invokes a constructor to produce a new instance. The target constructor **Q** has a list of formal parameters, As, a list of initializers, $\mathtt{Izs}$, and a statement body, $\mathsf{S}_{\mathtt{body}}$. The invocation of a constructor **Q** with **new Q**(Es) proceeds as follows:

$$\langle \textbf{new Q}(\mathsf{Es}), \rho, \mathtt{st}, \mathsf{H}, \mathtt{cex}, \mathtt{cst}, \kappa_E\rangle_{\text{eval}} \;\Rightarrow\; \langle \mathsf{Es}, \rho, \mathtt{st}, \mathsf{H}, \mathtt{cex}, \mathtt{cst}, \kappa'_A\rangle_{\text{evalList}}$$

where κ'_A = ConstructorApp(**Q**, st, H, cst, cex, κ_E).

$$\langle \text{ConstructorApp}(\mathbf{Q},\ \mathsf{st},\ \mathsf{H},\ \mathsf{cst},\ \mathsf{cex},\ \kappa_E),\ \mathsf{Vs}\rangle_{\text{acont}} \quad\Rightarrow\quad \langle \text{InitiK}(\mathsf{Q},\ \rho,\ \mathsf{st},\ \mathsf{H},\ \mathsf{cst},\ \mathsf{cex},\ \kappa_S),\ !\alpha\rangle_{\text{cont}}$$

where α is the location of the newly allocated instance in the store s.t.:

$$!\alpha = \text{ObjectValue}(\mathsf{class}, \mathsf{fields}) \in \textbf{ObjectValue}$$
$$\rho' = \mathsf{extend}(\rho_{\mathsf{empty}}, \mathsf{As}, \mathsf{Vs})$$
$$\kappa_S = \text{NewSK}(\kappa_E,\ \alpha)$$

$$\langle \text{NewSK}(\kappa_E,\ \alpha),\ \rho\rangle_{\text{scont}} \Rightarrow \langle \kappa_E,\ !\alpha\rangle_{\text{cont}}$$

Let $\mathsf{li} = \mathsf{last}(\text{Izs})$ be the last initializer in the list of initializers, Izs, of constructor **Q**.
When li is a redirecting initializer to target constructor **G**, then execution of constructor **Q** proceeds immediately with execution of its initializer list, Izs.

$$\langle \text{InitiK}(\mathsf{Q},\ \rho,\ \mathsf{st},\ \mathsf{H},\ \mathsf{cst},\ \mathsf{cex},\ \kappa_S),\ \mathsf{V}\rangle_{\text{cont}} \quad\Rightarrow\quad \langle \mathsf{Es},\ \rho,\ \mathsf{st}, \mathsf{H},\ \mathsf{cst},\ \mathsf{cex},\ \kappa_A\rangle_{\text{evalList}}$$
if $\text{Izs} = \mathsf{RedirectingI}(\mathsf{G}, \mathsf{Es}) :: []$

where:
κ_A = RedirectingApp(G, α, st, H, cst, cex, κ_S)
$!\alpha = \mathsf{V}$

$$\langle \text{InitiK}(\mathsf{Q},\ \rho,\ \mathsf{st},\ \mathsf{H},\ \mathsf{cst},\ \mathsf{cex},\ \kappa_S),\ \mathsf{V}\rangle_{\text{cont}} \quad\Rightarrow\quad \langle \mathsf{E},\ \rho,\ \mathsf{st}, \mathsf{H},\ \mathsf{cex},\ \mathsf{cst},\ \kappa'_E\rangle_{\text{eval}}$$
otherwise

where:
$\mathsf{fi} = \mathsf{Initializer}(\mathsf{E})$, first initializer in Izs
κ'_E = InitializerK(Izs, α, ρ, st, H, cst, cex, κ_S)
$!\alpha = \mathsf{V}$

$$\langle \text{RedirectingApp}(\mathsf{G},\ \alpha,\ \mathsf{st},\ \mathsf{H},\ \mathsf{cst},\ \mathsf{cex},\ \kappa_S),\ \mathsf{Vs}\rangle_{\text{acont}} \quad\Rightarrow\quad \langle \text{InitiK}(\mathsf{G},\ \rho',\ \mathsf{st},\ \mathsf{H},\ \mathsf{cst},\ \mathsf{cex},\ \kappa_S),\ !\alpha\rangle_{\text{cont}}$$

where:
$\rho' = \mathsf{extend}(\rho_{\mathsf{empty}}, \mathsf{As}, \mathsf{Vs})$

Otherwise, **Q** is a non-redirecting generative constructor, and execution of **Q** proceeds with evaluation of field initializers in the immediately enclosing class. Let **Q** be a constructor with non empty initializer list, Izs and a constructor body $\mathsf{S_{body}}$.

$$\langle \text{InitiK}(\mathsf{Q},\ \rho,\ \mathsf{st},\ \mathsf{H},\ \mathsf{cst},\ \mathsf{cex},\ \kappa_S),\ \mathsf{V}\rangle_{\text{cont}} \quad\Rightarrow\quad \langle \mathsf{Es},\ \rho,\ \mathsf{st}, \mathsf{H},\ \mathsf{cst},\ \mathsf{cex},\ \kappa_A\rangle_{\text{evalList}}$$

where:
$\rho' = \mathsf{extend}(\rho,\ \textbf{this},\ !\alpha)$
$!\alpha = \mathsf{V}$
Es = initializer expressions for instance fields
κ_A = FieldsApp(ρ, Izs, st, H, cst, cex, κ'_S)
κ'_S = BodySK($\mathsf{S_{body}}$, ρ', st, H, cst, cex, κ_S)

To further initialize the new instance, the body of the constructor will be executed. In Kernel the body of a constructor is executed after the execution of the initializer list and its super constructor. To support this, we add a new statement continuation, BodySK, that has the constructor body statement, the environment, the error handlers and stacktraces, and the next statement continuation.

$$\langle \text{BodySK}(\mathsf{S},\ \rho,\ \mathsf{st},\ \mathsf{H},\ \mathsf{cst},\ \mathsf{cex},\ \kappa_S),\ \_\rangle_{\text{scont}} \quad\Rightarrow\quad \langle \mathsf{S},\ \rho,\ [],\ [],\ \mathsf{st},\ \mathsf{H},\ \mathsf{cex},\ \mathsf{cst},\ \kappa_E,\ \kappa_S\rangle_{\text{exec}}$$