RULE READADDRESS-GLOBALVARIABLES
```
⟨ readAddress(Addr:Int,String2Id("Global"))
  -----------------------------------------
                 V:Value                     ...⟩k
⟨ ListItem(N:Int) ...⟩contractStack
⟨ ⟨ N ⟩ctId
  ⟨... Addr |-> V ...⟩ctStorage ...⟩contractInstance
```

RULE READADDRESS-LOCALVARIABLES
```
⟨ readAddress(Addr:Int,String2Id("Local"))
  ----------------------------------------
                 V:Value                    ...⟩k
⟨ ListItem(N:Int) ...⟩contractStack
⟨ ⟨ N ⟩ctId
  ⟨... Addr |-> V ...⟩Memory ...⟩contractInstance
```

RULE WRITEADDRESS-GLOBALVARIABLES
```
⟨ writeAddress(Addr:Int,String2Id("Global"),
               V:Value)
  ------------------------------------------
                   V                          ...⟩k
⟨ ListItem(N:Int) ...⟩contractStack
⟨ ⟨ N ⟩ctId
  ⟨... Addr |-> _
       ----------
       Addr |-> V ...⟩ctStorage ...⟩contractInstance
```

RULE WRITEADDRESS-LOCALVARIABLES
```
⟨ writeAddress(Addr:Int,String2Id("Local"),
               V:Value)
  -----------------------------------------
                   V                         ...⟩k
⟨ ListItem(N:Int) ...⟩contractStack
⟨ ⟨ N ⟩ctId
  ⟨... Addr |-> _
       ----------
       Addr |-> V ...⟩Memory ...⟩contractInstance
```

RULE ALLOCATEADDRESS-GLOBALVARIABLES
```
⟨ allocateAddress(N:Int, Addr:Int,
    String2Id("Global"), V:Value)
  --------------------------------
                V                   ...⟩k
⟨ ⟨ N ⟩ctId
  ⟨ STORAGE:Map
    ---------------------------
    STORAGE (Addr |-> V) ⟩ctStorage ...⟩contractInstance
```

RULE ALLOCATEADDRESS-LOCALVARIABLES
```
⟨ allocateAddress(N:Int, Addr:Int,
    String2Id("Local"), V:Value)
  --------------------------------
                V                   ...⟩k
⟨ ⟨ N ⟩ctId
  ⟨ MEMORY:Map
    --------------------------
    MEMORY (Addr |-> V) ⟩Memory ...⟩contractInstance
```

RULE UPDATESTATE-MAIN-CONTRACT
```
⟨ updateState(X:Id)
  -----------------
          .          ...⟩k  ⟨ ⟨ X ⟩cName ...⟩contract
⟨ N:Int
  ---------
  N +Int 1 ⟩cntContracts  ⟨ T:Int
                            ---------
                            T +Int 1 ⟩cntTrans
⟨                  INS:Bag
  ---------------------------------------------------
  INS ⟨ ⟨ N ⟩ctId
        ⟨ X ⟩ctName ...⟩contractInstance ⟩contractInstances
⟨              Trans:Map
  ---------------------------------------
  Trans (T |-> "new contract") ⟩tranComputation
⟨ L:List
  ---------------
  ListItem(X) L ⟩newStack
⟨ .List ⟩functionStack
```

RULE UPDATESTATE-FUNCTION-CALL
```
⟨ updateState(X:Id)
  -----------------
          .          ...⟩k  ⟨ ⟨ X ⟩cName ...⟩contract
⟨ N:Int
  ---------
  N +Int 1 ⟩cntContracts
⟨                  INS:Bag
  ---------------------------------------------------
  INS ⟨ ⟨ N ⟩ctId
        ⟨ X ⟩ctName ...⟩contractInstance ⟩contractInstances
⟨ L:List
  ---------------
  ListItem(X) L ⟩newStack  ⟨ CallList:List ⟩functionStack
requires CallList =/=K .List
```

RULE ALLOCATESTORAGE
```
⟨       allocateStorage(X:Id)
  --------------------------------
  allocateStateVars(N -Int 1, Vars) ...⟩k
⟨ ⟨ X ⟩cName ⟨ Vars:List ⟩stateVars ...⟩contract
⟨ N:Int ⟩cntContracts
```

RULE ALLOCATESTATEVARIABLES
```
⟨ allocateStateVars(N:Int,
   ListItem(Var) Vars:List)
  --------------------------
     allocate(N, Var) ↷
  allocateStateVars(N, Vars)  ...⟩k
```

RULE ALLOCATESTATEVARIABLES-END
```
⟨ allocateStateVars(N:Int, .List)
  -------------------------------
                .                  ...⟩k
```

APPENDIX

A. ReadAddress

We evaluate global variables with READADDRESS-GLOBALVARIABLES and local variables with READADDRESS-LOCALVARIABLES. Particularly, global and local variables are stored in `ctStorage` and `Memory` in the corresponding contract instance, respectively.

B. WriteAddress

Similar to `readAddress`, we rewrite global variables with WRITEADDRESS-GLOBALVARIABLES and local variables with WRITEADDRESS-LOCALVARIABLES. Particularly, global and local variables are rewritten in `ctStorage` and `Memory` in the corresponding contract instance, respectively.

C. AllocateAddress

A memory slot is allocated for a variable through `allocateAddress`. First, we map the corresponding contract instance with its Id `N`. Then we add a new slot at `Addr` with the initial value `V`. Particularly, for global variables the slot is added in `ctStorage` as shown in ALLOCATEADDRESS-GLOBALVARIABLES. And for local variables, the slot is added in `Memory` as illustrated in ALLOCATEADDRESS-LOCALVARIABLES.

D. UpdateState

The blockchain state is updated through `updateState` when a new contract instance is created. To be specific, the number of contract instances is increased by 1 in `cntContracts`. A new contract instance cell is created with its Id `N` in `ctId` and the associated contract name `X` in `ctName`. In addition, as shown in UPDATESTATE-MAIN-CONTRACT, if this new contract instance creation is in the "Main" contract, the number of transactions will be increased by 1 in `cntTrans` and a new mapping will be created to record this new contract instance creation as a transaction in `tranComputation`. As shown in UPDATESTATE-FUNCTION-CALL, if this new contract instance creation is nested in a function call, no transaction information will be recorded since it is not an independent transaction. Finally, a new list item of `X` is added into `newStack` to indicate that we are in the process of a new contract instance creation.

RULE INITINSTANCE-NOCONSTRUCTOR

$$\left\langle \frac{\texttt{initInstance(X:Id,E:ExpressionList)}}{\texttt{N -Int 1}} \cdots \right\rangle_{k}$$
$$\left\langle \frac{\texttt{ListItem(X) L:List}}{\texttt{L}} \right\rangle_{newStack} \left\langle \texttt{N:Int} \right\rangle_{cntContracts}$$
$$\left\langle \left\langle \texttt{X} \right\rangle_{cName} \left\langle \texttt{false} \right\rangle_{Constructor} \cdots \right\rangle_{contract}$$

RULE INITINSTANCE-WITHCONSTRUCTOR

$$\left\langle \frac{\texttt{initInstance(X:Id,E:ExpressionList)}}{\begin{array}{c}\texttt{functionCall(C;N -Int 1;}\\ \texttt{String2Id("constructor");}\\ \texttt{E;\#msgInfo(C,N -Int 1,0,0))}\end{array}} \cdots \right\rangle_{k}$$
$$\left\langle \frac{\texttt{ListItem(X) L:List}}{\texttt{L}} \right\rangle_{newStack} \left\langle \texttt{N:Int} \right\rangle_{cntContracts}$$
$$\left\langle \texttt{ListItem(C:Int)} \cdots \right\rangle_{contractStack}$$
$$\left\langle \left\langle \texttt{X} \right\rangle_{cName} \left\langle \texttt{true} \right\rangle_{Constructor} \cdots \right\rangle_{contract}$$

RULE SWITCH-CONTEXT

$$\left\langle \frac{\texttt{switchContext(C:Int,R:Int,F:Id,M:Msg)}}{\texttt{createTransaction(L)}} \cdots \right\rangle_{k}$$
$$\left\langle \frac{\texttt{L:List}}{\texttt{ListItem(R) L}} \right\rangle_{contractStack} \left\langle \texttt{CNum} \right\rangle_{cntContracts}$$
$$\left\langle \frac{\texttt{M1}}{\texttt{M}} \right\rangle_{Msg} \left\langle \frac{\texttt{MsgList:List}}{\texttt{ListItem(M1) MsgList}} \right\rangle_{msgStack}$$
$$\left\langle \frac{\texttt{CallList:List}}{\begin{array}{c}\texttt{ListItem(\#state(RhoC,F,}\\ \texttt{\#return(false,0),CNum,false)) CallList}\end{array}} \right\rangle_{functionStack}$$
$$\left\langle \begin{array}{c} \left\langle \texttt{C} \right\rangle_{ctId} \left\langle \texttt{RhoG} \right\rangle_{globalContext} \\ \left\langle \frac{\texttt{RhoC}}{\texttt{RhoG}} \right\rangle_{ctContext} \end{array} \cdots \right\rangle_{contractInstance}$$

E. AllocateStorage

Memory slots are allocated for state variables in the new contract instance through `allocateStateVars`. As shown in ALLOCATESTATEVARIABLES, a memory slot is allocated for each variable `Var` with `allocate` sequentially until there are no more variables to process.

F. InitInstance

If there is a constructor in the contract for which a new instance is created, INITINSTANCE-WITHCONSTRUCTOR will be applied. Otherwise, INITINSTANCE-NOCONSTRUCTOR will be applied. The rule INITINSTANCE-NOCONSTRUCTOR simply returns the Id of the new instance. Furthermore, the associated contract name `X` is popped out of `newStack` to indicate that the new instance creation is finished. While in the rule INITINSTANCE-WITHCONSTRUCTOR, apart from removing the contract name `X` out of `newStack`, a function call is processed to execute the constructor. To be specific, the caller of this function is `C` and the recipient is the new instance `N - 1`. In addition, the function to be called is the constructor and `E` specifies the function arguments. The function name is stored as an identifier, so we transform the string "constructor" into the equivalent `Id` type with the built-in function `String2Id` in the K-framework. The last argument of `functionCall` is the "msg" information, denoted by `#msgInfo(msg.sender(Id_of_Caller), Id_of_Recipient, msg.value, msg.gas)`. We assume that there is no gas cost for executing the constructor.

G. SwitchContext

The first step for a function call is to switch to the recipient instance as shown in SWITCH-CONTEXT. This is achieved by adding the recipient `R` into `contractStack` which indicates the current contract instance. At the same time, we need to store the state information of this function call, presented as `#state(RhoC,F,#return(false,0),CNum,false)`, in `functionStack`. There are altogether five items in the state information. The first item `RhoC` is the variable context of the caller instance which can be used to read and write variables in that instance. The second one `F` is the name of the function to be called. The next one is the information for `return` with two fields. The first field indicates whether a return statement has already been encountered, while the second records the return value. We assume that the default return value is 0. After this, the next item in `#state` is the number of contract instances created before this function call which is `CNum`. This is necessary because new contract instances can be created in a function call, changing the number of contract instances. If this function call or transaction throws an exception, the new contract instances created in this function call should be deleted. The last item is a flag to indicate whether this function call throws an exception. All exception handling features in smart contracts, such as `revert`, `assert`, etc, share similar semantics, making it possible to handle them with the same mechanism.

The variable context of the caller instance `RhoC` is obtained from the cell `contractInstance` with the caller instance Id `C` in the sub-cell `ctId`. Apart from storing the previous variable context in `functionStack`, we rewrite it to `RhoG` which is stored in `globalContext` and only associated with state variables. The intension of this step is to remove the context of local variables. Furthermore, the current transaction information in `Msg` is rewritten to `M`, while the previous one `M1` is pushed into `msgStack`. Finally, we record the transaction information through `createTransaction`. This part is omitted here.

H. Internal-Function-Call

There are three sub-steps in handling internal function calls according to INTERNAL-FUNCTION-CALL. To be specific, `saveCurContext` is used to facilitate the semantics of exception handling. When a transaction throws an exception, the states of all the contract instances involved should revert to the ones before this transaction. The instance states prior to the transaction are saved through `saveCurContext(CNum,0)` where `CNum` is the number of contract instances created before this function call and `0` specifies the starting point. In other words, we store the states of contract instances whose Ids range from `0` to `CNum - 1`. If this is a nested call, the states of the involving instances will not be saved through `saveCurContext` since it aims to keep track of the states before a transaction. `call(searchFunction(F,checkCallData(Es,0)), Es)` is the actual call of the function `F` with arguments `Es`. Particularly, `searchFunction` is an expression that

RULE INTERNAL-FUNCTION-CALL

```
⟨  functionCall(F:Id;Es:Values)
   ──────────────────────────────────────────────
   saveCurContext(CNum,0) ↷
   call(searchFunction(F,checkCallData(Es,0)),Es)
   ↷ updateCurContext(CNum,0)                        ··· ⟩k
⟨ CNum ⟩cntContracts
```

RULE CALL

```
⟨  call(N:Int,Es:Values)
   ──────────────────────────────
   initFunParams(N,Es) ↷
   processFunQuantifiers(N)↷
   callFunBody(N)                  ··· ⟩k
```

RULE CALL-FUNCTION-BODY

```
⟨  callFunBody(N)
   ──────────────────────────────────────────
   funBody(B) ↷ updateReturnParams(N) ↷
   updateReturnValue(N)                        ··· ⟩k
⟨ ⟨ N ⟩fId ⟨ B ⟩Body ··· ⟩function
```

RULE FUNCTION-BODY

```
⟨  funBody(S:Statement Ss:Statements)
   ──────────────────────────────────
   exeStmt(S) ↷ funBody(Ss)             ··· ⟩k

⟨  funBody(.Statements)
   ────────────────────
   .                      ··· ⟩k
```

returns the Id of the function to be called. It is used to distinguish functions with the same name `F` through `checkCallData` which checks the call data specified by `Es`. The second argument of `checkCallData` records the number of parameters that have been checked, so we start from 0. Finally, we update the instance states we have saved to the ones after the function call through `updateCurContext(CNum,0)`. If an exception is encountered in this call, the states of the involving instances will not be updated through `updateCurContext`.

In dealing with CALL, we first initialize function parameters including input parameters and return parameters through `initFunParams`. Input parameters are initialized by the function call arguments `Es` and return parameters are initialized to be the default values, such as 0 for an integer and false for a Boolean type. The first argument of `initFunParams`, denoted by `N`, is the Id of the function to be called while the second `Es` specifies the values of the input parameters of the function. After this, `processFunQuantifiers` deals with function quantifiers, namely specifiers and modifiers, which may modify the function body and have an impact on the function execution. For instance, modifiers rewrite the function body by replacing "_;" that appears there with the function body. Finally, `callFunBody` executes the function body that has been modified by function quantifiers. `initFunParams` is the same as allocating memory slots for local variables. `processFunQuantifiers` deals with the rewriting of the function body. Let us go into the details of `callFunBody`.

There are three sub-steps in CALL-FUNCTION-BODY. The first one `funBody` is the execution of the function body `B` which is obtained by mapping the cell `function` with Id `N`. The second `updateReturnParams` binds the return parameter with the return value. For instance, if a function returns `1`, the value of its return parameter should be `1`. The last one `updateReturnValue` returns the value of the return parameter if there is no return statement in this function.

RULE RETURN-CONTEXT

```
⟨  returnContext(R:Int)
   ──────────────────────────────────────────
   clearRecipientContext(R,RhoG)↷
   clearCallerContext(C,Rho)↷
   propagateException(C,Exception)↷E:Value      ··· ⟩k
⟨  ListItem(R) ListItem(C) L:List
   ──────────────────────────────
   ListItem(C) L                    ⟩contractStack
⟨  M
   ──
   M1  ⟩Msg
⟨  ListItem(M1) MsgList:List
   ─────────────────────────
   MsgList                     ⟩msgStack
⟨  ListItem(#state(Rho,_,#return(_,E),
   _,Exception)) CallList:List
   ─────────────────────────────────────
   CallList                                ⟩functionStack
⟨ ⟨ R ⟩ctId ⟨ RhoG ⟩globalContext ··· ⟩contractInstance
```

Statements in a function body are executed sequentially. In the rule FUNCTION-BODY, every time the first statement `S` in a list of statements is extracted for execution through `exeStmt` and the remaining statements `Ss` are processed with this rule recursively until the list of statements becomes empty.

I. ReturnContext

RETURN-CONTEXT is the last step in FUNCTION-CALL to return to the contract instance before this external function call. In order to finish this function call, the recipient instance `R` is popped out of `contractStack` to switch back to the caller instance `C`. In the meantime, the state information for this function call is removed from `functionStack`. Furthermore, the "msg" information in `Msg` is rewritten to the previous one `M1` which is also popped out of `msgStack`. This rule has three sub-steps and ends with the return value of this function call. These three sub-steps are `clearRecipientContext`, `clearCallerContext`, and `propagateException`. To be specific, `clearRecipientContext` and `clearCallerContext` remove the local variable contexts in the instances of the recipient and caller, respectively. For the recipient instance, the variable context is set to be `RhoG` which is obtained from `globalContext` and only holds the context of state variables in `R`. For the caller instance, the variable context is set to be the previous one `Rho` retrieved from the state information in `functionStack`. `propagateException` deals with the propagation of exceptions based on the `exception` flag recorded in `functionStack`. If an exception is encountered in a function call, it should be propagated to the call stemming from the "Main" contract which is considered as an independent transaction. In this way, if an exception appears in any nested call, the whole transaction throws and the states of all the involving instances should revert to the ones before this transaction. Lastly, RETURN-CONTEXT returns the return value of this function call `E` which is obtained from `#return` in `functionStack`.

If the `exception` flag, the second field in `propagateException`, is true, which means that an exception has been encountered in the function body, another exception will be propagated to the caller instance `C` according to PROPAGATE-EXCEPTION-TRUE. Particularly, the Id of the caller instance is required to be larger than or equal to 0. This

RULE PROPAGATE-EXCEPTION-TRUE

```
⟨ propagateException(C,true)
  ─────────────────────────── ... ⟩k
  exception()
requires C >=Int 0
```

RULE PROPAGATE-EXCEPTION-FALSE

```
⟨ propagateException(C,false)
  ──────────────────────────── ... ⟩k
  .
requires C >=Int 0
```

RULE PROPAGATE-EXCEPTION-MAIN-CONTRACT

```
⟨ propagateException(C,Exception)
  ──────────────────────────────── ... ⟩k
  .
requires C <Int 0
```

RULE UPDATE-EXCEPTION-STATE

```
⟨ updateExceptionState()
  ────────────────────── ... ⟩k
  .
⟨ ListItem(#state(_,_,_,_,_))
  ─────────────────────────────── ... ⟩functionStack
  ListItem(#state(_,_,_,_,true))
```

RULE REVERT-STATE

```
⟨ revertState()
  ────────────────────────────────── ... ⟩k
  revertInContracts(PreCNum,0) ↷
  deleteNewContracts(PreCNum,CNum)
⟨ ListItem(#state(_,_,_,PreCNum,_)) ... ⟩functionStack
⟨ CNum ⟩cntContracts
```

RULE RETURN-VALUE

```
⟨ return E:Value
  ────────────── ... ⟩k
  1
⟨ ListItem(#state(_,_,
  #return(_,_),_,_))
  ──────────────────── ... ⟩functionStack
  ListItem(#state(_,_,
  #return(true,E),_,_))
```

RULE RETURN

```
⟨ return
  ────── ... ⟩k
  1
⟨ ListItem(#state(_,_,
  #return(_,_),_,_))
  ──────────────────── ... ⟩functionStack
  ListItem(#state(_,_,
  #return(true,true),_,_))
```

is because the propagation of exceptions stops in the function call whose caller is the "Main" contract. If the `exception` flag is false, which means that there is no exception encountered in the function body, exceptions will not be propagated according to PROPAGATE-EXCEPTION-FALSE. Also, as illustrated in PROPAGATE-EXCEPTION-MAIN-CONTRACT, exceptions are not propagated to the "Main" contract whose Id is "-1".

J. Exception Handling

In UPDATE-EXCEPTION-STATE, the `exception` flag, the last field in `#state` is rewritten to `true` to indicate that an exception has been encountered. In REVERT-STATE, two sub-steps are processed to revert to the state before the transaction. These sub-steps are `revertInContracts` and `deleteNewContracts`. To be specific, `revertInContracts(PreCNum,0)` deals with the reversion to the previous states of the contract instances that were created before this transaction. Particularly, the reversion starts from the instance with Id `0` and ends at the one with Id `PreCNum - 1` where `PreCNum` is the number of instances created before

RULE BLOCK

```
⟨ {S:Statements}
  ───────────────────────────────────── ... ⟩k
  pushBlockStack() ↷ S ↷ popBlockStack()
```

RULE PUSH-BLOCK-STACK

```
⟨ pushBlockStack()
  ──────────────── ... ⟩k   ⟨ ListItem(N:Int) ... ⟩contractStack
  .
⟨ ⟨ N ⟩ctId
  ⟨ Rho ⟩ctContext ... ⟩contractInstance
⟨ (.List ⇒ ListItem(Rho)) ... ⟩blockStack
```

RULE POP-BLOCK-STACK

```
⟨ popBlockStack()
  ─────────────── ... ⟩k   ⟨ ListItem(N:Int) ... ⟩contractStack
  .
⟨ ⟨ N ⟩ctId
  ⟨ _ ⇒ Rho ⟩ctContext ... ⟩contractInstance
⟨ (ListItem(Rho) ⇒ .List) ... ⟩blockStack
```

RULE STATEMENTS

```
⟨ S:Statement Ss:Statements
  ───────────────────────── ... ⟩k   ⟨ .Statements
  exeStmt(S)↷Ss                         ─────────── ... ⟩k
                                        .
```

this transaction which is recorded in `functionStack`. `deleteNewContracts(PreCNum,CNum)` deletes the new contract instances created in this transaction. Here, `CNum` is the current number of contract instances retrieved from `cntContracts`. The reversion to previous states is simply the rewriting of cell contents to previous ones and the deletion of new contract instances is the deletion of a set of cells. Detailed steps are omitted here.

K. Return

When `return` is encountered, the `return` flag, the first field in `#return`, is set to be `true` in `functionStack`. Particularly, if a value `E` is returned, the `return` value, the second field in `#return`, is rewritten to `E`. If there is no value to return, we assign `true` to the `return` value to indicate that this function call is successful. Both RETURN-VALUE and RETURN end with the integer 1 which indicates the end of the expression. Any integer followed by ";" will be rewritten to `.`, which represents the end of the `return` statement.

L. Statements

As shown in BLOCK and STATEMENTS, each statement in a block is executed sequentially with `exeStmt`. Please note that these rules are consistent with FUNCTION-BODY. A function body is actually a list of statements, possibly with blocks. Due to clarity in presentation, we use a separate rule FUNCTION-BODY to capture the logic.

As shown in EXE-STATEMENT, the execution of each statement in the function body is affected by the `return` and `exception` flags in `functionStack`. Generally speaking, a statement will be executed when both of the two flags are false, indicating that neither `return` nor an exception has been encountered. For statements in the "Main" contract, we simply execute them.

Please note that we limit the statement for execution to `NoBlockStatement` where no block structures are present.

RULE EXE-STATEMENT

$$\left\langle \frac{\texttt{exeStmt(S:NoBlockStatement)}}{\texttt{S}} \cdots \right\rangle_{k}$$
$$\left\langle \begin{array}{c}\texttt{ListItem(\#state(\_,\_,}\\ \texttt{\#return(false,\_),\_,false))}\end{array} \cdots \right\rangle_{functionStack}$$

RULE EXE-STATEMENT-END

$$\left\langle \frac{\texttt{exeStmt(S:NoBlockStatement)}}{\cdot} \cdots \right\rangle_{k}$$
$$\left\langle \begin{array}{c}\texttt{ListItem(\#state(\_,\_,}\\ \texttt{\#return(ReturnFlag,\_),}\\ \texttt{\_,ExceptionFlag))}\end{array} \cdots \right\rangle_{functionStack}$$

```
requires (ReturnFlag ==Bool true)
orBool (ExceptionFlag ==Bool true)
```

RULE EXE-STATEMENT-MAIN-CONTRACT

$$\left\langle \frac{\texttt{exeStmt(S:NoBlockStatement)}}{\texttt{S}} \cdots \right\rangle_{k}$$
$$\left\langle \ \texttt{.List}\ \right\rangle_{functionStack}$$

This is because we need to exclude block structures to keep all the other statements in the function body parallel to `return` and `exception` statements. In this way, each statement can be executed sequentially without any nested structures. Once `return` or an exception is encountered, the execution stops regardless of the structure of the statements. Statements with blocks, named as `BlockStatements`, can be transformed into a list of `NoBlockStatements`. The transformation rules for `If Statements` are shown below as an example.

RULE EXE-STATEMENT-BLOCKSTATEMENT

$$\left\langle \frac{\texttt{exeStmt(S:BlockStatement)}}{\texttt{S}} \cdots \right\rangle_{k}$$

RULE IF

$$\left\langle \frac{\texttt{if(true) S}}{\texttt{exeStmt(S)}} \cdots \right\rangle_{k} \quad \left\langle \frac{\texttt{if(false) S}}{\cdot} \cdots \right\rangle_{k}$$

RULE IF-ELSE

$$\left\langle \frac{\texttt{if(true) S else S1}}{\texttt{exeStmt(S)}} \cdots \right\rangle_{k} \quad \left\langle \frac{\texttt{if(false) S1 else S}}{\texttt{exeStmt(S)}} \cdots \right\rangle_{k}$$

Suppose we have a `If Statement` which is `if(x>0){x=x+1; return x;}` and a pre-condition that `x` is 1. This `If Statement` is encountered in a function body. According to FUNCTION-BODY, we have `exeStmt(if(x>0){x=x+1;return x;})`. First, it is processed with EXE-STATEMENT-BLOCKSTATEMENT which ends with the original `If Statement`. Next, this `If Statement` is processed with the rule IF which ends with `exeStmt({x=x+1;return x;})`. Then EXE-STATEMENT-BLOCKSTATEMENT is applied and ends with `{x=x+1;return x;}`. Subsequently, it is handled with BLOCK that ends with `x=x+1;return x;`. At this stage, STATEMENTS is applied to the two resulting statements. Finally, we have two `NoBlockStatements`, namely `exeStmt(x=x+1;)` and `exeStmt(return x;)`, to be processed with EXE-STATEMENT. In this way, a `BlockStatement` encountered in the function body can be transformed into a list of `NoBlockStatements`.