

Code Transformation for Distributed Python ML Code

1 Python Abstract Syntax

<i>module</i>	::=	<i>stmt</i> * <i>type_ignore</i>	(MODULEDEF)
<i>stmt</i>	::=	(@ <i>expr</i>)* def <i>id</i> (<i>args</i>) (-> <i>expr</i>)? : (# type:s)? <i>stmt</i> *	(FUNDEF)
		(@ <i>expr</i>)* async def <i>id</i> (<i>args</i>) (-> <i>expr</i>)? : (# type:s)? <i>stmt</i> *	(ASYNCFUNDEF)
		(@ <i>expr</i>)* class <i>id</i> (<i>expr</i> * <i>keyword</i> *) : <i>stmt</i> *	(CLASSDEF)
		return <i>expr</i> ?	(RETURN)
		delete <i>expr</i> *	(DELETE)
		<i>expr</i> * = <i>expr</i> (# type:s)?	(ASSIGN)
		<i>expr</i> <i>binop</i> = <i>expr</i>	(AUGASSIGN)
		<i>expr</i> : <i>expr</i> (= <i>expr</i>)?	(ANNASSIGN)
		(# type:s)? for <i>expr</i> in <i>expr</i> : <i>stmt</i> * (else : <i>stmt</i>)?	(FORLOOP)
		(# type:s)? async for <i>expr</i> in <i>expr</i> : <i>stmt</i> * (else : <i>stmt</i>)?	(ASYNCFORLOOP)
		while (<i>expr</i>) : <i>stmt</i> * (else : <i>stmt</i>)?	(WHILELOOP)
		if (<i>expr</i>) : <i>stmt</i> * (else : <i>stmt</i>)?	(IF)
		(# type:s)? with <i>with_item</i> * : <i>stmt</i> *	(WITH)
		(# type:s)? async with <i>with_item</i> * : <i>stmt</i> *	(ASYNCWITH)
		match <i>expr</i> : <i>match_case</i> *	(MATCH)
		raise <i>expr</i> ? (from <i>expr</i>)?	(RAISE)
		try : <i>stmt</i> * <i>exc_handler</i> * (else : <i>stmt</i>)? (finally : <i>stmt</i>)?	(TRY)
		assert <i>expr</i> <i>expr</i> ?	(ASSERT)
		import <i>alias</i> *	(IMPORT)
		from <i>i id</i> ? import <i>alias</i> *	(IMPORTFROM)
		global <i>id</i> *	(GLOBAL)
		nonlocal <i>id</i> *	(NONLOCAL)
		<i>expr</i>	(EXPRSTMT)
		pass	(PASS)
		break	(BREAK)
		continue	(CONTINUE)

<i>expr</i>	<code>::=</code>	<code>expr boolop expr</code>	(BOOLOP)
		<code>expr := expr</code>	(NAMEDEXPR)
		<code>expr binop expr</code>	(BINARYOP)
		<code>unop expr</code>	(UNARYOP)
		<code>lambda args : expr</code>	(LAMBDA)
		<code>expr if expr else expr</code>	(IFEXPR)
		<code>{(expr : expr)* (**expr)*}</code>	(DICTIONARY)
		<code>{expr*}</code>	(SET)
		<code>[expr*]</code>	(LIST)
		<code>(expr*)</code>	(TUPLE)
		<code>{expr:expr comprehension*}</code>	(DICTIONARYCOMP)
		<code>{expr comprehension*}</code>	(SETCOMP)
		<code>[expr comprehension*]</code>	(LISTCOMP)
		<code>(expr comprehension*)</code>	(GENERATORCOMP)
		<code>await expr</code>	(AWAIT)
		<code>yield expr?</code>	(YIELD)
		<code>yield from expr</code>	(YIELDFROM)
		<code>expr (compop expr)*</code>	(COMPOP)
		<code>expr (expr* keyword*)</code>	(CALL)
		<code>{expr (!i)? (:expr)?}</code>	(FORMATTEDVALUE)
		<code>expr*</code>	(JOINEDSTR)
		<code>constant</code>	(CONSTANT)
		<code>expr.id</code>	(ATTRIBUTE)
		<code>expr[expr]</code>	(SUBSCRIPT)
		<code>*expr</code>	(STARRED)
		<code>**expr</code>	(DOUBLESTARRED)
		<code>id</code>	(NAME)
		<code>expr? (:expr)? (:expr)?</code>	(SLICE)
<i>boolop</i>	<code>::=</code>	<code>and or</code>	(BOOLOPERATOR)
<i>binop</i>	<code>::=</code>	<code>+ - * @ / ** << >> ^ & // %</code>	(BINOPERATOR)
<i>unop</i>	<code>::=</code>	<code>~ not + -</code>	(UNOPERATOR)
<i>compop</i>	<code>::=</code>	<code>== != < <= > >= is is not in not in</code>	(COMPOP)
<i>comprehension</i>	<code>::=</code>	<code>for expr in expr (if expr)*</code>	(COMPREHENSION)
		<code>async for expr in expr (if expr)*</code>	(ASYNCCOMPREHEN)
<i>exc.handler</i>	<code>::=</code>	<code>except expr? (as id)? : stmt*</code>	(EXCHANDLER)
<i>args</i>	<code>::=</code>	<code>(arg (= expr)?)*, (arg (= expr)?)*, arg?, (arg (= expr)?)*, arg?</code>	(ARGUMENTS)
<i>arg</i>	<code>::=</code>	<code>id expr? s?</code>	(ARGUMENT)
<i>keyword</i>	<code>::=</code>	<code>id? = expr</code>	(KEYWORD)
<i>alias</i>	<code>::=</code>	<code>id (.id)* (as id)?</code>	(ALIAS)
<i>with.item</i>	<code>::=</code>	<code>expr (as expr)?</code>	(WITHITEM)

<i>match_case</i>	<code>::= case pattern (if expr)? : stmt*</code>	(MATCHCASE)
<i>pattern</i>	<code>::= expr</code>	(MATCHVALUE)
	<i>constant</i>	(MATCHSINGLETON)
	[<i>pattern</i> *]	(MATCHSEQUENCE)
	<i>*(id)?</i>	(MATCHSTAR)
	{(<i>expr</i> : <i>pattern</i>)* <i>id</i> ?}	(MATCHMAPPING)
	<i>expr</i> (<i>pattern</i> * (<i>id</i> = <i>pattern</i>)*)	(MATCHCLASS)
	(<i>pattern</i> as)? <i>id</i>	(MATCHAS)
	<i>pattern</i> <i>pattern</i>	(MATCHOR)
	-	(MATCHWILDCARD)
<i>constant</i>	<code>::= None</code>	(NONELITERAL)
	<i>i</i>	(INTLITERAL)
	<i>f</i>	(FLOATLITERAL)
	<i>c</i>	(COMPLEXLITERAL)
	<i>s</i>	(STRINGLITERAL)
	<i>b</i>	(BOOLEANLITERAL)
	(<i>constant</i> *)	(TUPLELITERAL)
	...	(ELLIPSIS)
<i>type_ignore</i>	<code>::= i*</code>	(TYPEIGNORE)
<i>id</i>	$\in Id$	
<i>s</i>	$\in Str$	
<i>b</i>	$\in \{\text{True}, \text{False}\}$	
<i>i</i>	$\in \mathbb{Z}$	
<i>f</i>	$\in \mathbb{R}$	
<i>c</i>	$\in \mathbb{C}$	

Note: there may be more constant terms such as an immutable container containing only constant elements. Please update the *constant* rule if you know such terms.

2 Transformation Rule for TensorFlow ML Training Code

2.1 Restrictions

1. All import statements must be placed at the top of a module.
2. The `tensorflow` module must be assigned to a variable only using an import statement.
3. Members of the `tensorflow` module must not be aliased.
4. `print` function call expressions must not introduce side-effects.
5. A dataset and an optimizer object must only be created once via an assignment statement with a function call expression and must not be aliased.
6. Variables storing a dataset or a optimizer object must not be reassigned to store another value that is not a dataset or an optimizer object.
7. A dataset and an optimizer object must not be created conditionally.
8. `optimizer.apply_gradients` function call expressions must be expression statements or a direct RHS of an assignment statement.
9. Global variables storing an `optimizer` object and referred in functions must be defined before the functions' definitions and must not be changed after their initializations.
10. A checkpoint object must only be created once via an assignment statement with a function call expression and must not be aliased.

2.2 Rules

2.2.1 Types and Auxiliary Functions

$\tau ::=$	<i>Module</i>	(MODULES)
	<i>Stmt</i>	(STATEMENTS)
	<i>Expr</i>	(EXPRESSIONS)
	<i>Comprehension</i>	(COMPREHENSIONS)
	<i>ExcHandler</i>	(EXCEPTION HANDLERS)
	<i>Alias</i>	(ALIASES)
	<i>WithItem</i>	(WITHITEMS)
	<i>Pattern</i>	(PATTERNS)
	<i>Id</i>	(IDENTIFIERS)
	<i>Str</i>	(STRINGS)
	τ list	(LIST OF ELEMENTS TYPED τ)

$\sigma \in \Sigma = Str \xrightarrow{\text{fin}} Id$ ENVIRONMENT STORING MAPPINGS FROM STRINGS TO IDENTIFIERS

.1	:	$\tau \times \tau \rightarrow \tau$	GET THE FIRST ELEMENT OF THE GIVEN PAIR
@	:	$\tau \text{ list} \rightarrow \tau \text{ list} \rightarrow \tau \text{ list}$	CONCATENATE TWO LISTS
::	:	$\tau \rightarrow \tau \text{ list} \rightarrow \tau \text{ list}$	APPEND AN ELEMENT TO A LIST (RIGHT-ASSOCIATIVE)
<:	:	$Expr \times Expr \rightarrow \mathbb{B}$	WHETHER CLASSES SPECIFIED BY THE EXPRESSIONS ARE SUBCLASSES

2.3 Rule: TensorFlow 2.x Keras API Pattern

2.3.1 Transformation Rules

$$\boxed{trans_M : Module \rightarrow Module}$$

$$trans_M \llbracket stmt^* \text{ type_ignore} \rrbracket = trans_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma).1 \text{ type_ignore}$$

$$\boxed{trans_{\bar{S}} : Stmt \text{ list} \rightarrow \Sigma \rightarrow (Stmt \text{ list} \times \Sigma)}$$

$$\begin{aligned} trans_{\bar{S}} \llbracket stmt_1 \text{ stmt}_2 \dots stmt_n \rrbracket(\sigma) = & \text{LET } stmt_1', \sigma_1 = trans_S \llbracket stmt_1 \rrbracket(\sigma) \text{ IN} \\ & \text{LET } stmt_2', \sigma_2 = trans_S \llbracket stmt_2 \rrbracket(\sigma_1) \text{ IN} \\ & \dots \\ & \text{LET } stmt_n', \sigma_n = trans_S \llbracket stmt_n \rrbracket(\sigma_{n-1}) \text{ IN} \\ & (stmt_1' @ stmt_2' @ \dots @ stmt_n', \sigma_n) \end{aligned}$$

$$\boxed{trans_S : Stmt \rightarrow \Sigma \rightarrow (Stmt \text{ list} \times \Sigma)}$$

$$\begin{aligned} trans_S \llbracket (@expr_1)^* \text{ def } id (args) (-> expr_2)? : (\#type:s)? stmt^* \rrbracket(\sigma) = \\ ([(@expr_1)^* \text{ def } id (args) (-> expr_2)? : (\#type:s)? trans_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma).1], \sigma) \end{aligned}$$

$$\begin{aligned} trans_S \llbracket (@expr_1)^* \text{ async def } id (args) (-> expr_2)? : (\#type:s)? stmt^* \rrbracket(\sigma) = \\ ([(@expr_1)^* \text{ async def } id (args) (-> expr_2)? : (\#type:s)? trans_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma).1], \sigma) \end{aligned}$$

$$\begin{aligned} trans_S \llbracket (@expr_1)^* \text{ class } id (expr_2^* keyword^*) : stmt^* \rrbracket(\sigma) = \\ ([(@expr_1)^* \text{ class } id (expr_2^* keyword^*) : trans_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma).1], \sigma) = \end{aligned}$$

$$trans_S \llbracket \text{return } expr? \rrbracket(\sigma) = ([\text{return } (trans_E \llbracket expr \rrbracket(\sigma))], \sigma)$$

$$trans_S \llbracket \text{delete } expr^* \rrbracket(\sigma) = ([\text{delete } expr^*], \sigma)$$

A strict form of assignment statements

$$trans_S \llbracket id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) (\#type:s)? \rrbracket(\sigma) =$$

Learning rate scheduler scaling

IF $expr_1 <: \text{"tensorflow.keras.optimizers.schedules.PiecewiseConstantDecay"}$ **THEN**

$([id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) (\#type:s)?],$
 $\sigma["lr_scheduler"] \mapsto id_r])$

ELIF $expr_1 <: \text{tensorflow.keras.optimizers.schedules.LearningRateSchedule}$ **THEN**

IF $id_i = \text{initial_learning_rate}$ **WHEN** $1 \leq i \leq k$ **THEN**

$([id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots id_i = expr_{2i} * \text{hvd.size}()$
 $\dots (id_k =)? expr_{2k}) (\#type:s)?], \sigma["lr_scheduler"] \mapsto id_r])$

ELSE

$([id_r = expr_1 (expr_{11} * \text{hvd.size}() \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) (\#type:s)?],$
 $\sigma["lr_scheduler"] \mapsto id_r])$

ELIF $expr_1 <: \text{tensorflow.compat.v1.train.exponential_decay}$ **THEN**

IF $id_i = \text{learning_rate}$ **WHEN** $1 \leq i \leq k$ **THEN**

$([id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots id_i = expr_{2i} * \text{hvd.size}()$
 $\dots (id_k =)? expr_{2k}) (\#type:s)?], \sigma["lr_scheduler"] \mapsto id_r])$

ELSE

$([id_r = expr_1 (expr_{11} * hvd.size() \dots expr_{1n} \ (id_1 =)? \ expr_{21} \dots (id_k =)? \ expr_{2k}) \ (\#type:s)?],$
 $\sigma["lr_scheduler" \mapsto id_r])$

Optimizer learning rate scaling and wrapping

ELIF $expr_1 <: tensorflow.keras.optimizers.Optimizer$ **THEN**

IF $id_i = learning_rate$ **WHEN** $1 \leq i \leq k$ **THEN**

$([id_r = expr_1 (expr_{11} \dots expr_{1n} \ (id_1 =)? \ expr_{21} \dots id_i = expr_{2i} * hvd.size()$
 $\dots (id_k =)? \ expr_{2k}) \ (\#type:s)?$

$id_r = hvd.DistributedOptimizer(id_r)], \sigma["optimizer" \mapsto id_r])$

ELSE

$([id_r = expr_1 (expr_{11} * hvd.size() \dots expr_{1n} \ (id_1 =)? \ expr_{21} \dots (id_k =)? \ expr_{2k}) \ (\#type:s)?],$
 $id_r = hvd.DistributedOptimizer(id_r)], \sigma["optimizer" \mapsto id_r])$

Model related

ELIF $expr_1 <: tensorflow.keras.Model$ **THEN**

$([id_r = expr_1 (expr_{11} \dots expr_{1n} \ (id_2 =)? \ expr_{21} \dots (id_k =)? \ expr_{2k})], \sigma["model" \mapsto id_r])$

ELIF $id_m = "model"$ **AND** $expr_1 = id_t.evaluate$ **THEN**

IF $id_i = verbose$ **WHEN** $1 \leq i \leq k$ **THEN**

$([id_r = expr_1 (expr_{11} \dots expr_{1n} \ (id_1 =)? \ expr_{21} \dots id_i = 1 \text{ if } hvd.rank() == 0 \text{ else } 0$
 $\dots (id_k =)? \ expr_{2k}) \ (\#type:s)?], \sigma)$

ELSE

$([id_r = expr_1 (expr_{11} \ \ expr_{12} \ 1 \text{ if } hvd.rank() == 0 \text{ else } 0 \dots expr_{1n}$
 $(id_1 =)? \ expr_{21} \dots (id_k =)? \ expr_{2k}) \ (\#type:s)?], \sigma)$

Checkpoint

ELIF $\sigma("tensor_flow") = id_t$ **AND** $expr_1 = id_t.train.Checkpoint$ **THEN**

$([id_r = expr_1 (expr_{11} \dots expr_{1n} \ (id_1 =)? \ expr_{21} \dots (id_k =)? \ expr_{2k}) \ (\#type:s)?],$
 $\sigma["checkpoint" \mapsto id_r])$

Default

ELSE $([id_r = trans_E \llbracket expr_1 (expr_{11} \dots expr_{1n} \ (id_2 =)? \ expr_{21} \dots (id_k =)? \ expr_{2k}) \rrbracket(\sigma) \ (\#type:s)?],$
 $\sigma)$

$trans_S \llbracket expr_0 = expr_1 \ (\#type:s)? \rrbracket(\sigma) =$

IF $\sigma("os") = id_{os}$ **AND** $expr_0 = id_{os}.environ \ ['CUDA_VISIBLE_DEVICES']$

THEN $([], \sigma)$

A general form of assignment statements except for the strict form

$trans_S \llbracket expr_1^* = expr_2 \ (\#type:s)? \rrbracket(\sigma) = ([expr_1^* = trans_E \llbracket expr_2 \rrbracket(\sigma) \ (\#type:s)?], \sigma)$

$trans_S \llbracket expr_1 \ binop = expr_2 \rrbracket(\sigma) = ([expr_1 \ binop = trans_E \llbracket expr_2 \rrbracket(\sigma)], \sigma)$

$trans_S \llbracket expr_1 : expr_2 \ (= \ expr_3)? \rrbracket(\sigma) = ([expr_1 : expr_2 \ (= \ trans_E \llbracket expr_3 \rrbracket(\sigma))?), \sigma)$

$trans_S \llbracket (\#type:s)? \text{ for } expr_1 \text{ in } expr_2 : stmt_1^* \text{ (else : } stmt_2^*)? \rrbracket(\sigma) =$

$([(\#type:s)? \text{ for } expr_1 \text{ in } trans_E \llbracket expr_2 \rrbracket(\sigma) :$

$trans_{\bar{S}} \llbracket stmt_1^* \rrbracket(\sigma).1 \text{ (else : } trans_{\bar{S}} \llbracket stmt_2^* \rrbracket(\sigma).1?), \sigma)$

$trans_S \llbracket (\#type:s)? \text{ async for } expr_1 \text{ in } expr_2 : stmt_1^* (\text{else} : stmt_2^*)? \rrbracket (\sigma) =$
 $((\#type:s)? \text{ async for } expr_1 \text{ in } trans_E \llbracket expr_2 \rrbracket (\sigma) :$
 $trans_{\bar{S}} \llbracket stmt_1^* \rrbracket (\sigma).1 (\text{else} : trans_{\bar{S}} \llbracket stmt_2^* \rrbracket (\sigma).1)?], \sigma)$

$trans_S \llbracket \text{while } (expr) : stmt_1^* (\text{else} : stmt_2^*)? \rrbracket (\sigma) =$
 $([\text{while } (trans_E \llbracket expr \rrbracket (\sigma)) : trans_{\bar{S}} \llbracket stmt_1^* \rrbracket (\sigma).1 (\text{else} : trans_{\bar{S}} \llbracket stmt_2^* \rrbracket (\sigma).1)?], \sigma)$

$trans_S \llbracket \text{if } (expr) : stmt^* (\text{else} : stmt^*)? \rrbracket (\sigma) =$
 $([\text{if } (trans_E \llbracket expr \rrbracket (\sigma)) : trans_{\bar{S}} \llbracket stmt^* \rrbracket (\sigma).1 (\text{else} : trans_{\bar{S}} \llbracket stmt^* \rrbracket (\sigma).1)?], \sigma)$

$trans_S \llbracket (\#type:s)? \text{ with } with_item^* : stmt^* \rrbracket (\sigma) = ((\#type:s)? \text{ with } with_item^{*'} : stmt^{*'}], \sigma_2)$

$trans_S \llbracket (\#type:s)? \text{ async with } with_item^* : stmt^* \rrbracket (\sigma) =$
 $((\#type:s)? \text{ async with } with_item^{*'} : stmt^{*'}], \sigma_2)$

$trans_S \llbracket \text{match } expr : match_case^* \rrbracket (\sigma) =$
 $([\text{match } trans_E \llbracket expr \rrbracket (\sigma) : (trans_C \llbracket match_case \rrbracket (\sigma))^*], \sigma)$

$trans_S \llbracket \text{raise } expr_1? (\text{from } expr_2)? \rrbracket (\sigma) = ([\text{raise } expr_1? (\text{from } expr_2)?], \sigma)$

$trans_S \llbracket \text{try} : stmt_1^* exc_handler^* (\text{else} : stmt_2^*)? (\text{finally} : stmt_3^*)? \rrbracket (\sigma) =$
 $([\text{try} : trans_{\bar{S}} \llbracket stmt_1^* \rrbracket (\sigma).1 (trans_H \llbracket exc_handler \rrbracket (\sigma))^*$
 $(\text{else} : trans_{\bar{S}} \llbracket stmt_2^* \rrbracket (\sigma).1)? (\text{finally} : trans_{\bar{S}} \llbracket stmt_3^* \rrbracket (\sigma).1)?], \sigma)$

$trans_S \llbracket \text{assert } expr_1 \text{ } expr_2? \rrbracket (\sigma) = ([\text{assert } trans_E \llbracket expr_1 \rrbracket (\sigma) \text{ } expr_2?], \sigma)$

$trans_S \llbracket \text{import } alias^* \rrbracket (\sigma) =$
LET $\sigma_1 = trans_{\bar{A}} \llbracket alias^* \rrbracket (\sigma)$ **IN**
IF $\sigma_1 \setminus \sigma = [\text{"tensor_flow"} \mapsto id]$ **THEN**
 $([\text{import } alias^*,$
 $\text{import horovod.tensorflow as hvd},$
 $\text{hvd.init()},$
 $\text{gpus} = id.config.experimental.list_physical_devices('GPU'),$
 $\text{for gpu in gpus: } id.config.experimental.set_memory_growth(\text{gpu}, \text{True}),$
 $\text{if gpus: } id.config.experimental.set_visible_devices(\text{gpus}[\text{hvd.local_rank()}], 'GPU')], \sigma_1)$
ELSE $([\text{import } alias^*], \sigma_1)$

$trans_S \llbracket \text{from } i \text{ id? import } alias^* \rrbracket (\sigma) = ([\text{from } i \text{ id? import } alias^*], \sigma_1)$

$trans_S \llbracket \text{global } id^* \rrbracket (\sigma) = ([\text{global } id^*], \sigma)$

$trans_S \llbracket \text{nonlocal } id^* \rrbracket (\sigma) = ([\text{nonlocal } id^*], \sigma)$

A strict form of expr statements

```

transS[[ expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) ]](σ) =
  # Config
  IF idt = σ("tensor_flow") AND
    expr1 = idt.config.experimental.set_visible_devices THEN ([], σ)
  # model.compile
  ELIF idm = σ("model") AND expr1 = idm.compile THEN
    # string "adam" case
    IF idi = optimizer WHEN 2 ≤ i ≤ k AND expr2i = "adam" THEN
      ([optim = tf.optimizers.Adam(learning_rate=0.001 * hvd.size),
        optim = hvd.DistributedOptimizer(optim),
        expr1 (expr11 ... expr1n (id1 = )? expr21 ... idi = optim ... (idk = )? expr2k)], σ)
    ELIF expr11 = "adam" THEN
      ([optim = tf.optimizers.Adam(learning_rate=0.001 * hvd.size),
        optim = hvd.DistributedOptimizer(optim),
        expr1 (optim ... expr1n (id1 = )? expr21 ... (idk = )? expr2k)], σ)
    ELSE
      [expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k)], σ)
  # model.fit
  ELIF idm = σ("model") AND expr1 = idm.fit THEN
    IF idi = callbacks WHEN 2 ≤ i ≤ k THEN
      ([callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)],
        if hvd.rank() == 0: callbacks.append(expr2i),
        expr1 (optim ... expr1n (id1 = )? expr21 ... callbacks = callbacks ... (idk = )? expr2k)], σ)
    ELSE
      ([callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)],
        expr1 (optim ... expr1n (id1 = )? expr21 ... (idk = )? expr2k... callbacks = callbacks)], σ)
  # Root Rank Blocking
  ELIF idm = σ("model") AND (expr1 = idm.write OR
    expr1 = idm.summary OR expr1 = idm.save_weights OR
    expr1 = expr.load_weights OR expr1 = idm.save) OR
    idc = σ("checkpoint") AND expr1 = idc.save OR
    idt = σ("tensor_flow") AND expr1 = idt.print OR
    expr1 = print THEN
    ([if hvd.rank() == 0: expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k)], σ)
  ELSE
    [transE[[ expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) ]](σ)], σ)

```

A general form of expr statements except for the strict form

```
transS[[ expr ]](σ) = ([transE[[ expr ]](σ)], σ)
```

```
transS[[ pass ]](σ) = ([pass], σ)
```

```
transS[[ break ]](σ) = ([break], σ)
```


$$trans_S \llbracket \text{continue} \rrbracket(\sigma) = ([\text{continue}], \sigma)$$

$$\boxed{trans_E : Expr \rightarrow \Sigma \rightarrow Expr}$$

$$trans_E \llbracket expr_1 \text{ boolop } expr_2 \rrbracket(\sigma) = trans_E \llbracket expr_1 \rrbracket(\sigma) \text{ boolop } trans_E \llbracket expr_2 \rrbracket(\sigma)$$

$$trans_E \llbracket expr_1 := expr_2 \rrbracket(\sigma) = expr_1 := trans_E \llbracket expr_2 \rrbracket(\sigma)$$

$$trans_E \llbracket expr_1 \text{ binop } expr_2 \rrbracket(\sigma) = trans_E \llbracket expr_1 \rrbracket(\sigma) \text{ binop } trans_E \llbracket expr_2 \rrbracket(\sigma)$$

$$trans_E \llbracket unop \text{ expr } \rrbracket(\sigma) = unop \text{ } trans_E \llbracket expr \rrbracket(\sigma)$$

$$trans_E \llbracket \text{lambda } args : expr \rrbracket(\sigma) = \text{lambda } args : trans_E \llbracket expr \rrbracket(\sigma)$$

$$trans_E \llbracket expr_1 \text{ if } expr_2 \text{ else } expr_3 \rrbracket(\sigma) = \\ trans_E \llbracket expr_1 \rrbracket(\sigma) \text{ if } trans_E \llbracket expr_2 \rrbracket(\sigma) \text{ else } trans_E \llbracket expr_3 \rrbracket(\sigma)$$

$$trans_E \llbracket expr_1 \text{ if } expr_2 \text{ else } expr_3 \rrbracket(\sigma) = \\ trans_E \llbracket expr_1 \rrbracket(\sigma) \text{ if } trans_E \llbracket expr_2 \rrbracket(\sigma) \text{ else } trans_E \llbracket expr_3 \rrbracket(\sigma)$$

$$trans_E \llbracket (expr_1 : expr_2)^* (**expr_3)^* \rrbracket(\sigma) = \{(expr_1 : trans_E \llbracket expr_2 \rrbracket(\sigma))^* (**expr_3)^*\}$$

$$trans_E \llbracket \{expr^*\} \rrbracket(\sigma) = \{(trans_E \llbracket expr \rrbracket(\sigma))^*\}$$

$$trans_E \llbracket [expr^*] \rrbracket(\sigma) = [(trans_E \llbracket expr \rrbracket(\sigma))^*]$$

$$trans_E \llbracket (expr^*) \rrbracket(\sigma) = ((trans_E \llbracket expr \rrbracket(\sigma))^*)$$

$$trans_E \llbracket \{expr_1 : expr_2 \text{ comprehension}^*\} \rrbracket(\sigma) = \{expr_1 : trans_E \llbracket expr_2 \rrbracket(\sigma) (trans_O \llbracket comprehension \rrbracket(\sigma))^*\}$$

$$trans_E \llbracket \{expr \text{ comprehension}^*\} \rrbracket(\sigma) = \{trans_E \llbracket expr \rrbracket(\sigma) (trans_O \llbracket comprehension \rrbracket(\sigma))^*\}$$

$$trans_E \llbracket [expr \text{ comprehension}^*] \rrbracket(\sigma) = [(trans_E \llbracket expr \rrbracket(\sigma) (trans_O \llbracket comprehension \rrbracket(\sigma))^*)]$$

$$trans_E \llbracket (expr \text{ comprehension}^*) \rrbracket(\sigma) = (trans_E \llbracket expr \rrbracket(\sigma) (trans_O \llbracket comprehension \rrbracket(\sigma))^*)$$

$$trans_E \llbracket \text{await } expr \rrbracket(\sigma) = \text{await } trans_E \llbracket expr \rrbracket(\sigma)$$

$$trans_E \llbracket \text{yield } expr? \rrbracket(\sigma) = \text{yield } (trans_E \llbracket expr \rrbracket(\sigma))?$$

$$trans_E \llbracket \text{yield from } expr \rrbracket(\sigma) = \text{yield from } trans_E \llbracket expr \rrbracket(\sigma)$$

$$trans_E \llbracket expr_1 (compop\ expr_2)^* \rrbracket(\sigma) = trans_E \llbracket expr_1 \rrbracket(\sigma) (compop\ trans_E \llbracket expr_2 \rrbracket(\sigma))^*$$

$$\begin{aligned} trans_E \llbracket expr_1 (expr_{11} \dots expr_{1n} \ (id_1 =)? \ expr_{21} \dots (id_k =)? \ expr_{2k}) \rrbracket(\sigma) = \\ trans_E \llbracket expr_1 \rrbracket(\sigma) (trans_E \llbracket expr_{11} \rrbracket(\sigma) \dots trans_E \llbracket expr_{1n} \rrbracket(\sigma) \\ (id_1 =)? \ trans_E \llbracket expr_{21} \rrbracket(\sigma) \dots (id_k =)? \ trans_E \llbracket expr_{2k} \rrbracket(\sigma)) \end{aligned}$$

$$trans_E \llbracket \{expr_1 \quad (!i)? \quad (:expr_2)?\} \rrbracket(\sigma) = \{expr_1 \quad (!i)? \quad (:expr_2)?\}$$

$$trans_E \llbracket expr^* \rrbracket(\sigma) = expr^*$$

$$trans_E \llbracket constant \rrbracket(\sigma) = constant$$

$$trans_E \llbracket expr.id \rrbracket(\sigma) = (trans_E \llbracket expr \rrbracket(\sigma)).id$$

$$trans_E \llbracket expr_1[expr_2] \rrbracket(\sigma) = trans_E \llbracket expr_1 \rrbracket(\sigma)[trans_E \llbracket expr_2 \rrbracket(\sigma)]$$

$$trans_E \llbracket *expr \rrbracket(\sigma) = *expr$$

$$trans_E \llbracket **expr \rrbracket(\sigma) = **expr$$

$$trans_E \llbracket id \rrbracket(\sigma) = id$$

$$trans_E \llbracket expr_1? \ (:expr_2)? \ (:expr_3)? \rrbracket(\sigma) = trans_E \llbracket expr_1 \rrbracket(\sigma)? \ (:trans_E \llbracket expr_2 \rrbracket(\sigma))? \ (:trans_E \llbracket expr_3 \rrbracket(\sigma))?$$

$$\boxed{trans_O : Comprehension \rightarrow \Sigma \rightarrow Comprehension}$$

$$trans_O \llbracket \text{for } expr_1 \text{ in } expr_2 \text{ (if } expr_3)^* \rrbracket(\sigma) = \text{for } expr_1 \text{ in } trans_E \llbracket expr_2 \rrbracket(\sigma) \text{ (if } trans_E \llbracket expr_3 \rrbracket(\sigma))^*$$

$$\boxed{trans_H : ExcHandler \rightarrow \Sigma \rightarrow ExcHandler}$$

$$trans_H \llbracket \text{except } expr? \text{ (as } id)? : stmt^* \rrbracket(\sigma) = \text{except } expr? \text{ (as } id)? : trans_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma).1$$

$$\boxed{trans_{\bar{A}} : Alias\ list \rightarrow \Sigma \rightarrow \Sigma}$$

$$trans_{\bar{A}} \llbracket alias_1 \dots alias_{n-1} alias_n \rrbracket(\sigma) = trans_A \llbracket alias_n \rrbracket(trans_A \llbracket alias_{n-1} \rrbracket(\dots trans_A \llbracket alias_1 \rrbracket(\sigma)))$$

$$\boxed{trans_A : Alias \rightarrow \Sigma \rightarrow \Sigma}$$

$trans_A \llbracket id \rrbracket(\sigma) =$
LET $id = \text{tensorflow}$ **THEN** $\sigma[\text{"tensorflow"} \mapsto id]$
LET $id = \text{os}$ **THEN** $\sigma[\text{"os"} \mapsto id]$
ELSE σ

$trans_A \llbracket id_1 \text{ as } id_2 \rrbracket(\sigma) =$
LET $id_1 = \text{tensorflow}$ **THEN** $\sigma[\text{"tensorflow"} \mapsto id_2]$
LET $id_1 = \text{os}$ **THEN** $\sigma[\text{"os"} \mapsto id_2]$
ELSE σ

$trans_A \llbracket id_1 . id_2 (.id_3)^* (\text{as } id_2)? \rrbracket(\sigma) = \sigma$

$trans_{\overline{W}} : WithItem \text{ list} \rightarrow \Sigma \rightarrow (WithItem \text{ list} \times \Sigma)$

$trans_{\overline{W}} \llbracket with_item_1 with_item_2 \dots with_item_n \rrbracket(\sigma) =$
LET $with_item_1', \sigma_1 = trans_W \llbracket with_item_1 \rrbracket(\sigma)$ **IN**
LET $with_item_2', \sigma_2 = trans_W \llbracket with_item_2 \rrbracket(\sigma_1)$ **IN**
 \dots
LET $with_item_n', \sigma_n = trans_W \llbracket with_item_n \rrbracket(\sigma_{n-1})$ **IN**
 $(with_item_1' :: with_item_2' :: \dots :: [with_item_n'], \sigma_n)$

$trans_W : WithItem \rightarrow \Sigma \rightarrow (WithItem \times \Sigma)$
--

$trans_W \llbracket expr \rrbracket(\sigma) = (trans_E \llbracket expr \rrbracket(\sigma), \sigma)$

$trans_W \llbracket expr_1 \text{ as } expr_2 \rrbracket(\text{smodenv}) =$
 $(trans_E \llbracket expr_1 \rrbracket(\sigma) \text{ as } expr_2, \sigma)$

$trans_C : MatchCase \rightarrow \Sigma \rightarrow MatchCase$
--

$trans_C \llbracket \text{case } pattern \text{ (if } expr)? : stmt^* \rrbracket(\sigma) =$
 $\text{case } trans_P \llbracket pattern \rrbracket(\sigma) \text{ (if } trans_E \llbracket expr \rrbracket(\sigma))? : trans_{\overline{S}} \llbracket stmt^* \rrbracket(\sigma).1$

$trans_P : Pattern \rightarrow \Sigma \rightarrow Pattern$
--

$trans_P \llbracket expr \rrbracket(\sigma) = trans_E \llbracket expr \rrbracket(\sigma)$

$trans_P \llbracket constant \rrbracket(\sigma) = constant$

$trans_P \llbracket [pattern^*] \rrbracket(\sigma) = [trans_P \llbracket pattern \rrbracket(\sigma)]^*$

$trans_P \llbracket *(id)? \rrbracket(\sigma) = *(id)?$

$trans_P \llbracket \{(expr : pattern)^* id?\} \rrbracket(\sigma) = \{(expr : trans_P \llbracket pattern \rrbracket(\sigma))^* id?\}$

$trans_P \llbracket expr (pattern_1^* (id = pattern_2)^*) \rrbracket(\sigma) = expr (trans_P \llbracket pattern_1 \rrbracket(\sigma)^* (id = trans_P \llbracket pattern_2 \rrbracket(\sigma))^*)$

$$\text{trans}_P \llbracket (\text{pattern } \text{as})? \text{ id} \rrbracket(\sigma) = (\text{trans}_P \llbracket \text{pattern} \rrbracket(\sigma) \text{ as})? \text{ id}$$

$$\text{trans}_P \llbracket \text{pattern}_1 \mid \text{pattern}_2 \rrbracket(\sigma) = \text{trans}_P \llbracket \text{pattern}_1 \rrbracket(\sigma) \mid \text{trans}_P \llbracket \text{pattern}_2 \rrbracket(\sigma)$$

$$\text{trans}_P \llbracket - \rrbracket(\sigma) = -$$

2.4 Rule: TensorFlow 2.x GradientTape API Pattern

2.4.1 Transformation Rules

$\text{trans}_M : \text{Module} \rightarrow \text{Module}$
--

$$\text{trans}_M \llbracket \text{stmt}^* \text{ type_ignore} \rrbracket = \text{trans}_{\bar{S}} \llbracket \text{stmt}^* \rrbracket(\sigma).1 \text{ type_ignore}$$

$\text{trans}_{\bar{S}} : \text{Stmt list} \rightarrow \Sigma \rightarrow (\text{Stmt list} \times \Sigma)$

$$\begin{aligned} \text{trans}_{\bar{S}} \llbracket \text{stmt}_1 \text{ stmt}_2 \dots \text{stmt}_n \rrbracket(\sigma) = & \text{LET } \text{stmt}_1^{*'}, \sigma_1 = \text{trans}_S \llbracket \text{stmt}_1 \rrbracket(\sigma) \text{ IN} \\ & \text{LET } \text{stmt}_2^{*'}, \sigma_2 = \text{trans}_S \llbracket \text{stmt}_2 \rrbracket(\sigma_1) \text{ IN} \\ & \dots \\ & \text{LET } \text{stmt}_n^{*'}, \sigma_n = \text{trans}_S \llbracket \text{stmt}_n \rrbracket(\sigma_{n-1}) \text{ IN} \\ & (\text{stmt}_1^{*'} @ \text{stmt}_2^{*'} @ \dots @ \text{stmt}_n^{*'}, \sigma_n) \end{aligned}$$

$\text{trans}_S : \text{Stmt} \rightarrow \Sigma \rightarrow (\text{Stmt list} \times \Sigma)$
--

$$\begin{aligned} \text{trans}_S \llbracket (@\text{expr}_1)^* \text{ def } id (args) (-> \text{expr}_2)? : (\#type:s)? \text{ stmt}^* \rrbracket(\sigma) = \\ ([(@\text{expr}_1)^* \text{ def } id (args) (-> \text{expr}_2)? : (\#type:s)? \text{ trans}_{\bar{S}} \llbracket \text{stmt}^* \rrbracket(\sigma).1], \sigma) \\ \\ \text{trans}_S \llbracket (@\text{expr}_1)^* \text{ async def } id (args) (-> \text{expr}_2)? : (\#type:s)? \text{ stmt}^* \rrbracket(\sigma) = \\ ([(@\text{expr}_1)^* \text{ async def } id (args) (-> \text{expr}_2)? : (\#type:s)? \text{ trans}_{\bar{S}} \llbracket \text{stmt}^* \rrbracket(\sigma).1], \sigma) \\ \\ \text{trans}_S \llbracket (@\text{expr}_1)^* \text{ class } id (\text{expr}_2^* \text{ keyword}^*) : \text{ stmt}^* \rrbracket(\sigma) = \\ ([(@\text{expr}_1)^* \text{ class } id (\text{expr}_2^* \text{ keyword}^*) : \text{ trans}_{\bar{S}} \llbracket \text{stmt}^* \rrbracket(\sigma).1], \sigma) \\ \\ \text{trans}_S \llbracket \text{return } \text{expr}? \rrbracket(\sigma) = ([\text{return } (\text{trans}_E \llbracket \text{expr} \rrbracket(\sigma))?, \sigma) \\ \\ \text{trans}_S \llbracket \text{delete } \text{expr}^* \rrbracket(\sigma) = ([\text{delete } \text{expr}^*], \sigma) \end{aligned}$$

A strict form of assignment statements

$$\begin{aligned} \text{trans}_S \llbracket id_r = \text{expr}_1 (\text{expr}_{11} \dots \text{expr}_{1n} (id_1 =)? \text{expr}_{21} \dots (id_k =)? \text{expr}_{2k}) (\#type:s)? \rrbracket(\sigma) = \\ \text{\textcolor{blue}{\# Dataset}} \\ \text{IF } \sigma(\text{"tensor_flow"}) = id_t \text{ AND } \text{expr}_1 = id_t.\text{data.Dataset.expr}_3 \text{ THEN} \\ ([id_r = \text{expr}_1 (\text{expr}_{11} \dots \text{expr}_{1n} (id_1 =)? \text{expr}_{21} \dots (id_k =)? \text{expr}_{2k}) (\#type:s)?], \\ \sigma[\text{"dataset"} \mapsto id_r]) \\ \text{\textcolor{blue}{\# Variable Broadcasting}} \end{aligned}$$

```

ELIF  $\sigma(\text{"optimizer"}) = id_t$  AND  $expr_1 = id_t.apply\_gradients$  THEN
  IF  $id_i = grads\_and\_vars$  WHEN  $1 \leq i \leq k$  THEN
    LET  $id_z = NewID()$  IN
      ( $id_z = expr_{2i}$ ,
       $id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 = )? expr_{21} \dots id_i = id_z \dots (id_k = )? expr_{2k}) (\#type:s)?$ ,
      global  $hvd\_broadcast\_done$ ,
      if not  $hvd\_broadcast\_done$ : [ $hvd.broadcast\_variables([x[1] \text{ for } x \text{ in } id_z], root\_rank=0)$ ,
       $hvd.broadcast\_variables(id_t.variables(), root\_rank=0)$ ,
       $hvd\_broadcast\_done = True$  ]],  $\sigma$ )
    ELSE
      LET  $id_z = NewID()$  IN
        ( $id_z = expr_{11}$ ,
         $id_r = expr_1 (id_z expr_{12} \dots expr_{1n} (id_1 = )? expr_{21} \dots (id_k = )? expr_{2k}) (\#type:s)?$ ,
        global  $hvd\_broadcast\_done$ ,
        if not  $hvd\_broadcast\_done$ : [ $hvd.broadcast\_variables([x[1] \text{ for } x \text{ in } id_z], root\_rank=0)$ ,
         $hvd.broadcast\_variables(id_t.variables(), root\_rank=0)$ ,
         $hvd\_broadcast\_done = True$  ]],  $\sigma$ )
  # Learning rate scheduler scaling
  IF  $expr_1 <: tensorflow.keras.optimizers.schedules.PiecewiseConstantDecay$  THEN
    ( $id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 = )? expr_{21} \dots (id_k = )? expr_{2k}) (\#type:s)?$ ],
     $\sigma[\text{"lr\_scheduler"} \mapsto id_r]$ )
  ELIF  $expr_1 <: tensorflow.keras.optimizers.schedules.LearningRateSchedule$  THEN
    IF  $id_i = initial\_learning\_rate$  WHEN  $1 \leq i \leq k$  THEN
      ( $id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 = )? expr_{21} \dots id_i = expr_{2i} * hvd.size()$ 
       $\dots (id_k = )? expr_{2k}) (\#type:s)?$ ],  $\sigma[\text{"lr\_scheduler"} \mapsto id_r]$ )
    ELSE
      ( $id_r = expr_1 (expr_{11} * hvd.size() \dots expr_{1n} (id_1 = )? expr_{21} \dots (id_k = )? expr_{2k}) (\#type:s)?$ ],
       $\sigma[\text{"lr\_scheduler"} \mapsto id_r]$ )
  ELIF  $expr_1 <: tensorflow.compat.v1.train.exponential\_decay$  THEN
    IF  $id_i = learning\_rate$  WHEN  $1 \leq i \leq k$  THEN
      ( $id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 = )? expr_{21} \dots id_i = expr_{2i} * hvd.size()$ 
       $\dots (id_k = )? expr_{2k}) (\#type:s)?$ ],  $\sigma[\text{"lr\_scheduler"} \mapsto id_r]$ )
    ELSE
      ( $id_r = expr_1 (expr_{11} * hvd.size() \dots expr_{1n} (id_1 = )? expr_{21} \dots (id_k = )? expr_{2k}) (\#type:s)?$ ],
       $\sigma[\text{"lr\_scheduler"} \mapsto id_r]$ )
  # Optimizer learning rate scaling and wrapping
  IF  $expr_1 <: tensorflow.keras.optimizers.Optimizer$  THEN
    ( $id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 = )? expr_{21} \dots (id_k = )? expr_{2k}) (\#type:s)?$ ],
     $\sigma[\text{"optimizer"} \mapsto id_r]$ )
  # Model related
  ELIF  $expr_1 <: tensorflow.keras.Model$  THEN
    ( $id_r = expr_1 (expr_{11} \dots expr_{1n} (id_2 = )? expr_{21} \dots (id_k = )? expr_{2k})$ ],  $\sigma[\text{"model"} \mapsto id_r]$ )
  ELIF  $id_m = \text{"model"}$  AND  $expr_1 = id_t.evaluate$  THEN
    IF  $id_i = verbose$  WHEN  $1 \leq i \leq k$  THEN

```

```

([idr = expr1 (expr11 ... expr1n (id1 = )? expr21 ... idi = 1 if hvd.rank() == 0 else 0
... (idk = )? expr2k) (#type:s)?], σ)
ELSE
([idr = expr1 (expr11 expr12 1 if hvd.rank() == 0 else 0 ... expr1n
(id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?], σ)
# Checkpoint
ELIF σ(“tensor_flow”) = idt AND expr1 = idt.train.Checkpoint THEN
([idr = expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?],
σ[“checkpoint” ↦ idr])
# Default
ELSE ([idr = transE[[ expr1 (expr11 ... expr1n (id2 = )? expr21 ... (idk = )? expr2k) ]](σ) (#type:s)?],
σ)

```

A general form of assignment statements except for the strict form

```

transS[[ expr1* = expr2 (#type:s)? ]](σ) =
# CUDA Visible Devices
IF σ(“os”) = idos AND expr0 = idos.environ [‘CUDA_VISIBLE_DEVICES’] THEN
([], σ)
ELSE ([expr1* = transE[[ expr2 ]](σ) (#type:s)?], σ)

```

trans_S[[expr₁ binop = expr₂]](σ) = ([expr₁ binop = trans_E[[expr₂]](σ)], σ)

```

transS[[ expr1 : expr2 (= expr3)? ]](σ) =
# Dataset
IF expr1 = id1 AND σ(“tensor_flow”) = id2
AND expr3 = id2.data.Dataset.expr4 (expr5* keyword*) THEN
([expr1 : expr2 = expr3 (#type:s)?], σ[“dataset” ↦ id1])
ELSE ([expr1 : expr2 (= transE[[ expr3 ]](σ))?), σ)

```

```

transS[[ (#type:s)? for expr1 in expr2 : stmt1* (else : stmt2*)? ]](σ) =
# Adjust Steps
IF σ(“config”) = idc AND (expr2 = range(idc.iterations_per_epoch) OR
expr2 = tqdm(range(idc.iterations_per_epoch))) THEN
([(#type:s)? for expr1 in range(idc.iterations_per_epoch // hvd.size()) :
transS[[ stmt1* ]](σ).1 (else : transS[[ stmt2* ]](σ).1)?], σ)
ELSE ([(#type:s)? for expr1 in transE[[ expr2 ]](σ) :
transS[[ stmt1* ]](σ).1 (else : transS[[ stmt2* ]](σ).1)?], σ)

```

```

transS[(#type:s)? async for expr1 in expr2 : stmt1* (else : stmt2*)? ](σ) =
  # Adjust Steps
  IF σ(“config”) = idc AND (expr2 = range(idc.iterations_per_epoch) OR
    expr2 = tqdm(range(idc.iterations_per_epoch))) THEN
    ((#type:s)? async for expr1 in range(idc.iterations_per_epoch // hvd.size()) :
      transS[ stmt1* ](σ).1 (else : transS[ stmt2* ](σ).1)?, σ)
  ELSE ((#type:s)? async for expr1 in transE[ expr2 ](σ) :
    transS[ stmt1* ](σ).1 (else : transS[ stmt2* ](σ).1)?, σ)

transS[ while (expr) : stmt1* (else : stmt2*)? ](σ) =
  ([while (transE[ expr ](σ)) : transS[ stmt1* ](σ).1 (else : transS[ stmt2* ](σ).1)?, σ)

transS[ if (expr) : stmt* (else : stmt*)? ](σ) =
  ([if (transE[ expr ](σ)) : transS[ stmt* ](σ).1 (else : transS[ stmt* ](σ).1)?, σ)

transS[ (#type:s)? with with_item* : stmt* ](σ) =
  LET with_item*, σ1 = transW[ with_item* ](σ) IN
  LET stmt*, σ2 = transS[ stmt* ](σ1) IN
  # Wrapping Gradient Tape
  IF σ1 \ σ = [“gradient_tape” ↦ id] THEN
    ((#type:s)? with with_item* : stmt*,
      id = hvd.DistributedGradientTape(id), σ2)
  ELSE ((#type:s)? with with_item* : stmt*, σ2)

transS[ (#type:s)? async with with_item* : stmt* ](σ) =
  LET with_item*, σ1 = transW[ with_item* ](σ) IN
  LET stmt*, σ2 = transS[ stmt* ](σ1) IN
  # Wrapping Gradient Tape
  IF σ1 \ σ = [“gradient_tape” ↦ id] THEN
    ((#type:s)? async with with_item* : stmt*,
      id = hvd.DistributedGradientTape(id), σ2)
  ELSE ((#type:s)? async with with_item* : stmt*, σ2)

transS[ match expr : match_case* ](σ) =
  ([match transE[ expr ](σ) : (transC[ match_case ](σ))*], σ)

transS[ raise expr1? (from expr2)? ](σ) = ([raise expr1? (from expr2)?], σ)

transS[ try : stmt1* exc_handler* (else : stmt2*)? (finally : stmt3*)? ](σ) =
  ([try : transS[ stmt1* ](σ).1 (transH[ exc_handler ](σ))*
    (else : transS[ stmt2* ](σ).1)? (finally : transS[ stmt3* ](σ).1)?, σ)

transS[ assert expr1 expr2? ](σ) = ([assert transE[ expr1 ](σ) expr2?], σ)

```

```

transS[[ import alias* ]](σ) =
  LET σ1 = transA[[ alias* ]](σ) IN
  # import
  IF σ1 \ σ = [“tensor_flow” ↦ id] THEN
    ([import alias*,
     import horovod.tensorflow as hvd,
     hvd.broadcast_done = False,
     hvd.init(),
     gpus = id.config.experimental.list_physical_devices('GPU'),
     for gpu in gpus: id.config.experimental.set_memory_growth(gpu, True),
     if gpus: id.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')], σ1)
  ELSE ([import alias*], σ1)

```

```

transS[[ global id* ]](σ) = ([global id*], σ)

```

```

transS[[ nonlocal id* ]](σ) = ([nonlocal id*], σ)

```

A strict form of expr statements

```

transS[[ expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) ]](σ) =
  # Variable Broadcasting
  IF σ(“optimizer”) = idt AND expr1 = idt.apply_gradients THEN
    IF idi = grads_and_vars WHEN 1 ≤ i ≤ k THEN
      LET idz = NewID() IN
      ([idz = expr2i,
       expr1 (expr11 ... expr1n (id1 = )? expr21 ... idi = idz ... (idk = )? expr2k),
       global hvd.broadcast_done,
       if not hvd.broadcast_done: [ hvd.broadcast_variables([x[1] for x in idz], root_rank=0),
        hvd.broadcast_variables(idt.variables(), root_rank=0),
        hvd.broadcast_done = True ], σ)
      ELSE
        LET idz = NewID() IN
        ([idz = expr11,
         expr1 (idz expr12 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k),
         global hvd.broadcast_done,
         if not hvd.broadcast_done: [ hvd.broadcast_variables([x[1] for x in idz], root_rank=0),
          hvd.broadcast_variables(idt.variables(), root_rank=0),
          hvd.broadcast_done = True ], σ)
    # Config
    IF idt = σ(“tensor_flow”) AND
      expr1 = idt.config.experimental.set_visible_devices THEN ([], σ)
    # Root Rank Blocking
    ELIF idm = σ(“model”) AND (expr1 = idm.write OR
      expr1 = idm.summary OR expr1 = idm.save_weights OR

```


$expr_1 = expr.load_weights$ **OR** $expr_1 = id_m.save$ **OR**
 $id_c = \sigma(\text{"checkpoint"})$ **AND** $expr_1 = id_c.save$ **OR**
 $id_t = \sigma(\text{"tensor_flow"})$ **AND** $expr_1 = id_t.print$ **OR**
 $expr_1 = print$ **THEN**
 $((\text{if } hvd.rank() == 0: expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}), \sigma)$
ELSE
 $[trans_E[expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k})](\sigma), \sigma)$

A general form of expr statements except for the strict form

$trans_S[expr](\sigma) = ([trans_E[expr](\sigma)], \sigma)$

$trans_S[\text{pass}](\sigma) = ([\text{pass}], \sigma)$

$trans_S[\text{break}](\sigma) = ([\text{break}], \sigma)$

$trans_S[\text{continue}](\sigma) = ([\text{continue}], \sigma)$

$trans_E : Expr \rightarrow \Sigma \rightarrow Expr$
--

$trans_E[expr_1 \text{ boolop } expr_2](\sigma) = trans_E[expr_1](\sigma) \text{ boolop } trans_E[expr_2](\sigma)$

$trans_E[expr_1 := expr_2](\sigma) = expr_1 := trans_E[expr_2](\sigma)$

$trans_E[expr_1 \text{ binop } expr_2](\sigma) = trans_E[expr_1](\sigma) \text{ binop } trans_E[expr_2](\sigma)$

$trans_E[unop \text{ expr}](\sigma) = unop \ trans_E[expr](\sigma)$

$trans_E[\text{lambda } args : expr](\sigma) = \text{lambda } args : trans_E[expr](\sigma)$

$trans_E[expr_1 \text{ if } expr_2 \text{ else } expr_3](\sigma) =$
 $trans_E[expr_1](\sigma) \text{ if } trans_E[expr_2](\sigma) \text{ else } trans_E[expr_3](\sigma)$

$trans_E[expr_1 \text{ if } expr_2 \text{ else } expr_3](\sigma) =$
 $trans_E[expr_1](\sigma) \text{ if } trans_E[expr_2](\sigma) \text{ else } trans_E[expr_3](\sigma)$

$trans_E[\{(expr_1 : expr_2)^* (**expr_3)^*\}](\sigma) = \{(expr_1 : trans_E[expr_2](\sigma))^* (**expr_3)^*\}$

$trans_E[\{expr^*\}](\sigma) = \{(trans_E[expr](\sigma))^*\}$

$trans_E[[expr^*]](\sigma) = [(trans_E[expr](\sigma))^*]$

$$\text{trans}_E \llbracket (expr^*) \rrbracket(\sigma) = ((\text{trans}_E \llbracket expr \rrbracket(\sigma))^*)$$

$$\text{trans}_E \llbracket \{expr_1 : expr_2 \text{ comprehension}^*\} \rrbracket(\sigma) = \{expr_1 : \text{trans}_E \llbracket expr_2 \rrbracket(\sigma) \text{ (trans}_O \llbracket \text{comprehension} \rrbracket(\sigma))^* \}$$

$$\text{trans}_E \llbracket \{expr \text{ comprehension}^*\} \rrbracket(\sigma) = \{\text{trans}_E \llbracket expr \rrbracket(\sigma) \text{ (trans}_O \llbracket \text{comprehension} \rrbracket(\sigma))^* \}$$

$$\text{trans}_E \llbracket [expr \text{ comprehension}^*] \rrbracket(\sigma) = [\text{trans}_E \llbracket expr \rrbracket(\sigma) \text{ (trans}_O \llbracket \text{comprehension} \rrbracket(\sigma))^*]$$

$$\text{trans}_E \llbracket (expr \text{ comprehension}^*) \rrbracket(\sigma) = (\text{trans}_E \llbracket expr \rrbracket(\sigma) \text{ (trans}_O \llbracket \text{comprehension} \rrbracket(\sigma))^*)$$

$$\text{trans}_E \llbracket \text{await } expr \rrbracket(\sigma) = \text{await } \text{trans}_E \llbracket expr \rrbracket(\sigma)$$

$$\text{trans}_E \llbracket \text{yield } expr? \rrbracket(\sigma) = \text{yield } (\text{trans}_E \llbracket expr \rrbracket(\sigma))?$$

$$\text{trans}_E \llbracket \text{yield from } expr \rrbracket(\sigma) = \text{yield from } \text{trans}_E \llbracket expr \rrbracket(\sigma)$$

$$\text{trans}_E \llbracket expr_1 (\text{compop } expr_2)^* \rrbracket(\sigma) = \text{trans}_E \llbracket expr_1 \rrbracket(\sigma) (\text{compop } \text{trans}_E \llbracket expr_2 \rrbracket(\sigma))^*$$

$$\text{trans}_E \llbracket expr_1 (expr_{11} \dots expr_{1n} \text{ (id}_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) \rrbracket(\sigma) =$$

Adjust Steps

IF $\sigma(\text{"dataset"}) = id_t$ **AND** $expr_1 = id_t.\text{take}$ **THEN**

IF $id_i = \text{count}$ **WHEN** $1 \leq i \leq k$ **THEN**

$expr_1 (expr_{11} \dots expr_{1n} \text{ (id}_1 =)? expr_{21} \dots id_i = expr_{2i} // \text{hvd.size()} \dots (id_k =)? expr_{2k})$

ELSE

$expr_1 (expr_{11} // \text{hvd.size()} \dots expr_{1n} \text{ (id}_1 =)? expr_{21} \dots (id_k =)? expr_{2k})$

ELSE

$\text{trans}_E \llbracket expr_1 \rrbracket(\sigma) (\text{trans}_E \llbracket expr_{11} \rrbracket(\sigma) \dots \text{trans}_E \llbracket expr_{1n} \rrbracket(\sigma)$

$(id_1 =)? \text{trans}_E \llbracket expr_{21} \rrbracket(\sigma) \dots (id_k =)? \text{trans}_E \llbracket expr_{2k} \rrbracket(\sigma))$

$$\text{trans}_E \llbracket \{expr_1 \text{ (!i)? (:expr}_2\text{)?} \} \rrbracket(\sigma) = \{expr_1 \text{ (!i)? (:expr}_2\text{)?} \}$$

$$\text{trans}_E \llbracket expr^* \rrbracket(\sigma) = expr^*$$

$$\text{trans}_E \llbracket \text{constant} \rrbracket(\sigma) = \text{constant}$$

$$\text{trans}_E \llbracket expr.id \rrbracket(\sigma) = (\text{trans}_E \llbracket expr \rrbracket(\sigma)).id$$

$$\text{trans}_E \llbracket expr_1 [expr_2] \rrbracket(\sigma) = \text{trans}_E \llbracket expr_1 \rrbracket(\sigma) [\text{trans}_E \llbracket expr_2 \rrbracket(\sigma)]$$

$$\text{trans}_E \llbracket *expr \rrbracket(\sigma) = *expr$$

$$trans_E \llbracket **expr \rrbracket (\sigma) = **expr$$

$$trans_E \llbracket id \rrbracket (\sigma) = id$$

$$trans_E \llbracket expr_1? (:expr_2)? (:expr_3)? \rrbracket (\sigma) = trans_E \llbracket expr_1 \rrbracket (\sigma)? (:trans_E \llbracket expr_2 \rrbracket (\sigma))? (:trans_E \llbracket expr_3 \rrbracket (\sigma))?$$

$$\boxed{trans_O : Comprehension \rightarrow \Sigma \rightarrow Comprehension}$$

$$trans_O \llbracket \text{for } expr_1 \text{ in } expr_2 \text{ (if } expr_3)^* \rrbracket (\sigma) = \text{for } expr_1 \text{ in } trans_E \llbracket expr_2 \rrbracket (\sigma) \text{ (if } trans_E \llbracket expr_3 \rrbracket (\sigma))^*$$

$$\boxed{trans_H : ExcHandler \rightarrow \Sigma \rightarrow ExcHandler}$$

$$trans_H \llbracket \text{except } expr? \text{ (as } id)? : stmt^* \rrbracket (\sigma) = \text{except } expr? \text{ (as } id)? : trans_{\bar{S}} \llbracket stmt^* \rrbracket (\sigma).1$$

$$\boxed{trans_{\bar{A}} : Alias \text{ list} \rightarrow \Sigma \rightarrow \Sigma}$$

$$trans_{\bar{A}} \llbracket alias_1 \dots alias_{n-1} alias_n \rrbracket (\sigma) = trans_A \llbracket alias_n \rrbracket (trans_A \llbracket alias_{n-1} \rrbracket (\dots trans_A \llbracket alias_1 \rrbracket (\sigma)))$$

$$\boxed{trans_A : Alias \rightarrow \Sigma \rightarrow \Sigma}$$

$$trans_A \llbracket id \rrbracket (\sigma) =$$

LET $id = \text{tensorflow}$ **THEN** $\sigma[\text{"tensorflow"} \mapsto id]$
LET $id = \text{os}$ **THEN** $\sigma[\text{"os"} \mapsto id]$
LET $id = \text{config}$ **THEN** $\sigma[\text{"config"} \mapsto id]$
ELSE σ

$$trans_A \llbracket id_1 \text{ as } id_2 \rrbracket (\sigma) =$$

LET $id_1 = \text{tensorflow}$ **THEN** $\sigma[\text{"tensorflow"} \mapsto id_2]$
LET $id_1 = \text{os}$ **THEN** $\sigma[\text{"os"} \mapsto id_2]$
LET $id = \text{config}$ **THEN** $\sigma[\text{"config"} \mapsto id_2]$
ELSE σ

$$trans_A \llbracket id_1 . id_2 (.id_3)^* \text{ (as } id_2)? \rrbracket (\sigma) = \sigma$$

$$\boxed{trans_{\bar{W}} : WithItem \text{ list} \rightarrow \Sigma \rightarrow (WithItem \text{ list} \times \Sigma)}$$

$$trans_{\bar{W}} \llbracket with_item_1 with_item_2 \dots with_item_n \rrbracket (\sigma) =$$

LET $with_item_1', \sigma_1 = trans_W \llbracket with_item_1 \rrbracket (\sigma)$ **IN**
LET $with_item_2', \sigma_2 = trans_W \llbracket with_item_2 \rrbracket (\sigma_1)$ **IN**
 \dots
LET $with_item_n', \sigma_n = trans_W \llbracket with_item_n \rrbracket (\sigma_{n-1})$ **IN**
 $(with_item_1' :: with_item_2' :: \dots :: [with_item_n'], \sigma_n)$

$$\boxed{trans_W : WithItem \rightarrow \Sigma \rightarrow (WithItem \times \Sigma)}$$

$$trans_W \llbracket expr \rrbracket (\sigma) = (trans_E \llbracket expr \rrbracket (\sigma), \sigma)$$

$trans_W \llbracket expr_1 \text{ as } expr_2 \rrbracket(\text{smode}nv) =$
IF $\sigma(\text{"tensor_flow"}) = id_1$ **AND** $expr_1 = id_1.\text{GradientTape}()$ **AND** $expr_2 = id_2$ **THEN**
 $(expr_1 \text{ as } expr_2, \sigma[\text{"gradient_tape"} \mapsto id_2])$
ELSE $(trans_E \llbracket expr_1 \rrbracket(\sigma) \text{ as } expr_2, \sigma)$

$trans_C : MatchCase \rightarrow \Sigma \rightarrow MatchCase$
--

$trans_C \llbracket \text{case pattern (if expr)? : stmt}^* \rrbracket(\sigma) =$
 $\text{case } trans_P \llbracket pattern \rrbracket(\sigma) \text{ (if } trans_E \llbracket expr \rrbracket(\sigma))? : trans_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma).1$

$trans_P : Pattern \rightarrow \Sigma \rightarrow Pattern$
--

$trans_P \llbracket expr \rrbracket(\sigma) = trans_E \llbracket expr \rrbracket(\sigma)$

$trans_P \llbracket constant \rrbracket(\sigma) = constant$

$trans_P \llbracket [pattern]^* \rrbracket(\sigma) = [trans_P \llbracket pattern \rrbracket(\sigma)]^*$

$trans_P \llbracket \star(id)? \rrbracket(\sigma) = \star(id)?$

$trans_P \llbracket \{(expr : pattern)^* id?\} \rrbracket(\sigma) = \{(expr : trans_P \llbracket pattern \rrbracket(\sigma))^* id?\}$

$trans_P \llbracket expr \ (pattern_1^* (id = pattern_2)^*) \rrbracket(\sigma) = expr \ (trans_P \llbracket pattern_1 \rrbracket(\sigma)^* (id = trans_P \llbracket pattern_2 \rrbracket(\sigma))^*)$

$trans_P \llbracket (pattern \text{ as})? id \rrbracket(\sigma) = (trans_P \llbracket pattern \rrbracket(\sigma) \text{ as})? id$

$trans_P \llbracket pattern_1 \mid pattern_2 \rrbracket(\sigma) = trans_P \llbracket pattern_1 \rrbracket(\sigma) \mid trans_P \llbracket pattern_2 \rrbracket(\sigma)$

$trans_P \llbracket - \rrbracket(\sigma) = -$

2.5 Rule: TensorFlow 1.x Mainscript API Pattern

2.5.1 Transformation Rules

$trans_M : Module \rightarrow Module$

$trans_M \llbracket stmt^* \text{ type_ignore} \rrbracket = trans_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma).1 \text{ type_ignore}$

$trans_{\bar{S}} : Stmt \text{ list} \rightarrow \Sigma \rightarrow (Stmt \text{ list} \times \Sigma)$
--

$$\begin{aligned}
trans_{\bar{S}} \llbracket stmt_1 \quad stmt_2 \dots stmt_n \rrbracket(\sigma) &= \text{LET } stmt_1^{*'}, \sigma_1 = trans_S \llbracket stmt_1 \rrbracket(\sigma) \text{ IN} \\
&\text{LET } stmt_2^{*'}, \sigma_2 = trans_S \llbracket stmt_2 \rrbracket(\sigma_1) \text{ IN} \\
&\dots \\
&\text{LET } stmt_n^{*'}, \sigma_n = trans_S \llbracket stmt_n \rrbracket(\sigma_{n-1}) \text{ IN} \\
&(stmt_1^{*'} @ stmt_2^{*'} @ \dots @ stmt_n^{*'}, \sigma_n)
\end{aligned}$$

$trans_S : Stmt \rightarrow \Sigma \rightarrow (Stmt \text{ list } \times \Sigma)$
--

$$\begin{aligned}
trans_S \llbracket (@expr_1)^* \text{ def } id (args) (-> expr_2)? : (\#type:s)? stmt^* \rrbracket(\sigma) = \\
([(@expr_1)^* \text{ def } id (args) (-> expr_2)? : (\#type:s)? trans_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma).1], \sigma)
\end{aligned}$$

$$\begin{aligned}
trans_S \llbracket (@expr_1)^* \text{ async def } id (args) (-> expr_2)? : (\#type:s)? stmt^* \rrbracket(\sigma) = \\
([(@expr_1)^* \text{ async def } id (args) (-> expr_2)? : (\#type:s)? trans_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma).1], \sigma)
\end{aligned}$$

$$\begin{aligned}
trans_S \llbracket (@expr_1)^* \text{ class } id (expr_2^* keyword^*) : stmt^* \rrbracket(\sigma) = \\
([(@expr_1)^* \text{ class } id (expr_2^* keyword^*) : trans_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma).1], \sigma) =
\end{aligned}$$

$$trans_S \llbracket \text{return } expr? \rrbracket(\sigma) = ([\text{return } (trans_E \llbracket expr \rrbracket(\sigma))?, \sigma)$$

$$trans_S \llbracket \text{delete } expr^* \rrbracket(\sigma) = ([\text{delete } expr^*], \sigma)$$

A strict form of assignment statements

$$trans_S \llbracket id_r = expr_1 (expr_{11} \dots expr_{1j} \quad (id_{11} =)? expr_{21} \dots (id_{1k} =)? expr_{2k})$$

Minimized Optimizer

$$\begin{aligned}
&id_m (expr_{31} \dots expr_{3l} \quad (id_{41} =)? expr_{41} \dots (id_{4n} =)? expr_{4n}) (\#type:s)? \rrbracket(\sigma) = \\
&\text{IF } expr_1 <: \text{tensorflow.compat.v1.train.Optimizer} \text{ AND } id_m = \text{minimize} \text{ THEN} \\
&\quad ([id_r = expr_1 (expr_{11} \dots expr_{1j} \quad (id_1 =)? expr_{21} \dots id_i = expr_{2i} * \text{hvd.size}() \\
&\quad \dots (id_k =)? expr_{2k}).id_m (expr_{31} \dots expr_{3l} \quad (id_{41} =)? expr_{41} \dots (id_{4n} =)? expr_{4n}) (\#type:s)?, \\
&\quad id_r = \text{hvd.DistributedOptimizer}(id_r)], \sigma[\text{"optimizer"} \mapsto id_r]) \\
&\text{ELSE} \\
&\quad ([id_r = expr_1 (expr_{11} * \text{hvd.size}() \dots expr_{1j} \quad (id_1 =)? \quad expr_{21} \dots (id_k =)? expr_{2k}) \\
&\quad .id_m (expr_{31} \dots expr_{3l} \quad (id_{41} =)? expr_{41} \dots (id_{4n} =)? expr_{4n}) (\#type:s)?, \\
&\quad id_r = \text{hvd.DistributedOptimizer}(id_r)], \sigma[\text{"optimizer"} \mapsto id_r])
\end{aligned}$$

$$trans_S \llbracket id_r = expr_1 (expr_{11} \dots expr_{1n} \quad (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) (\#type:s)? \rrbracket(\sigma) =$$

Config Proto

$$\begin{aligned}
&\text{IF } id_t = \text{"tensorflow"} \text{ AND } expr_1 = id_t.\text{ConfigProto} \text{ THEN} \\
&\quad (id_r = expr_1 (expr_{11} \dots expr_{1n} \quad (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) (\#type:s)?, \\
&\quad \sigma[id_r \mapsto \text{"config_proto"}])
\end{aligned}$$

Learning rate scheduler scaling

$$\begin{aligned}
&\text{IF } expr_1 <: \text{"tensorflow.keras.optimizers.schedules.PiecewiseConstantDecay"} \text{ THEN} \\
&\quad ([id_r = expr_1 (expr_{11} \dots expr_{1n} \quad (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) (\#type:s)?, \\
&\quad \sigma[\text{"lr_scheduler"} \mapsto id_r])
\end{aligned}$$

$$\text{ELIF } expr_1 <: \text{tensorflow.keras.optimizers.schedules.LearningRateSchedule} \text{ THEN}$$

$$\begin{aligned}
&\text{IF } id_i = \text{initial_learning_rate} \text{ WHEN } 1 \leq i \leq k \text{ THEN} \\
&\quad ([id_r = expr_1 (expr_{11} \dots expr_{1n} \quad (id_1 =)? expr_{21} \dots id_i = expr_{2i} * \text{hvd.size}()
\end{aligned}$$

```

... (idk = )? expr2k) (#type:s)?], σ["lr_scheduler" ↦ idr])
ELSE
  ([idr = expr1 (expr11 * hvd.size() ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?],
   σ["lr_scheduler" ↦ idr])
ELIF expr1 <: tensorflow.compat.v1.train.exponential_decay THEN
  IF idi = learning_rate WHEN 1 ≤ i ≤ k THEN
    ([idr = expr1 (expr11 ... expr1n (id1 = )? expr21 ... idi = expr2i * hvd.size()
      ... (idk = )? expr2k) (#type:s)?], σ["lr_scheduler" ↦ idr])
  ELSE
    ([idr = expr1 (expr11 * hvd.size() ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?],
     σ["lr_scheduler" ↦ idr])
# Optimizer learning rate scaling and wrapping
ELIF expr1 <: tensorflow.keras.optimizers.Optimizer THEN
  IF idi = learning_rate WHEN 1 ≤ i ≤ k THEN
    ([idr = expr1 (expr11 ... expr1n (id1 = )? expr21 ... idi = expr2i * hvd.size()
      ... (idk = )? expr2k) (#type:s)?
      idr = hvd.DistributedOptimizer(idr)], σ["optimizer" ↦ idr])
  ELSE
    ([idr = expr1 (expr11 * hvd.size() ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?,
      idr = hvd.DistributedOptimizer(idr)], σ["optimizer" ↦ idr])
# Model related
ELIF expr1 <: tensorflow.keras.Model THEN
  ([idr = expr1 (expr11 ... expr1n (id2 = )? expr21 ... (idk = )? expr2k)], σ["model" ↦ idr])
ELIF idm = "model" AND expr1 = idt.evaluate THEN
  IF idi = verbose WHEN 1 ≤ i ≤ k THEN
    ([idr = expr1 (expr11 ... expr1n (id1 = )? expr21 ... idi = 1 if hvd.rank() == 0 else 0
      ... (idk = )? expr2k) (#type:s)?], σ)
  ELSE
    ([idr = expr1 (expr11 expr12 1 if hvd.rank() == 0 else 0 ... expr1n
      (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?], σ)
# Checkpoint
ELIF σ("tensor_flow") = idt AND expr1 = idt.train.Checkpoint THEN
  ([idr = expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?],
   σ["checkpoint" ↦ idr])
# Default
ELSE ([idr = transE[[ expr1 (expr11 ... expr1n (id2 = )? expr21 ... (idk = )? expr2k) ]](σ) (#type:s)?],
  σ)

```

trans_S[[expr₀ = expr₁ (#type:s)?]](σ) =

```

IF σ("os") = idos AND expr0 = idos.environ ["CUDA_VISIBLE_DEVICES"]
THEN ([], σ)

```

A general form of assignment statements except for the strict form

trans_S[[expr₁^{*} = expr₂ (#type:s)?]](σ) = ([expr₁^{*} = trans_E[[expr₂]](σ) (#type:s)?], σ)

$$trans_S \llbracket expr_1 \text{ binop } = expr_2 \rrbracket(\sigma) = ([expr_1 \text{ binop } = trans_E \llbracket expr_2 \rrbracket(\sigma)], \sigma)$$

$$trans_S \llbracket expr_1 : expr_2 (= expr_3)? \rrbracket(\sigma) = ([expr_1 : expr_2 (= trans_E \llbracket expr_3 \rrbracket(\sigma))?], \sigma)$$

$$trans_S \llbracket (\#type:s)? \text{ for } expr_1 \text{ in } expr_2 : stmt_1^* (\text{else} : stmt_2^*)? \rrbracket(\sigma) = \\ ([(\#type:s)? \text{ for } expr_1 \text{ in } trans_E \llbracket expr_2 \rrbracket(\sigma) : \\ trans_{\overline{S}} \llbracket stmt_1^* \rrbracket(\sigma).1 (\text{else} : trans_{\overline{S}} \llbracket stmt_2^* \rrbracket(\sigma).1)?], \sigma)$$

$$trans_S \llbracket (\#type:s)? \text{ async for } expr_1 \text{ in } expr_2 : stmt_1^* (\text{else} : stmt_2^*)? \rrbracket(\sigma) = \\ ([(\#type:s)? \text{ async for } expr_1 \text{ in } trans_E \llbracket expr_2 \rrbracket(\sigma) : \\ trans_{\overline{S}} \llbracket stmt_1^* \rrbracket(\sigma).1 (\text{else} : trans_{\overline{S}} \llbracket stmt_2^* \rrbracket(\sigma).1)?], \sigma)$$

$$trans_S \llbracket \text{while } (expr) : stmt_1^* (\text{else} : stmt_2^*)? \rrbracket(\sigma) = \\ ([\text{while } (trans_E \llbracket expr \rrbracket(\sigma)) : trans_{\overline{S}} \llbracket stmt_1^* \rrbracket(\sigma).1 (\text{else} : trans_{\overline{S}} \llbracket stmt_2^* \rrbracket(\sigma).1)?], \sigma)$$

$$trans_S \llbracket \text{if } (expr) : stmt^* (\text{else} : stmt^*)? \rrbracket(\sigma) = \\ ([\text{if } (trans_E \llbracket expr \rrbracket(\sigma)) : trans_{\overline{S}} \llbracket stmt^* \rrbracket(\sigma).1 (\text{else} : trans_{\overline{S}} \llbracket stmt^* \rrbracket(\sigma).1)?], \sigma)$$

$$trans_S \llbracket (\#type:s)? \text{ with } with_item^* : stmt^* \rrbracket(\sigma) = ([(\#type:s)? \text{ with } with_item^{*'} : stmt^{*'}], \sigma_2)$$

$$trans_S \llbracket (\#type:s)? \text{ async with } with_item^* : stmt^* \rrbracket(\sigma) = \\ ([(\#type:s)? \text{ async with } with_item^{*'} : stmt^{*'}], \sigma_2)$$

$$trans_S \llbracket \text{match } expr : match_case^* \rrbracket(\sigma) = \\ ([\text{match } trans_E \llbracket expr \rrbracket(\sigma) : (trans_C \llbracket match_case \rrbracket(\sigma))^*], \sigma)$$

$$trans_S \llbracket \text{raise } expr_1? (\text{from } expr_2)? \rrbracket(\sigma) = ([\text{raise } expr_1? (\text{from } expr_2)?], \sigma)$$

$$trans_S \llbracket \text{try} : stmt_1^* \text{ exc_handler}^* (\text{else} : stmt_2^*)? (\text{finally} : stmt_3^*)? \rrbracket(\sigma) = \\ ([\text{try} : trans_{\overline{S}} \llbracket stmt_1^* \rrbracket(\sigma).1 (trans_H \llbracket exc_handler \rrbracket(\sigma))^* \\ (\text{else} : trans_{\overline{S}} \llbracket stmt_2^* \rrbracket(\sigma).1)? (\text{finally} : trans_{\overline{S}} \llbracket stmt_3^* \rrbracket(\sigma).1)?], \sigma)$$

$$trans_S \llbracket \text{assert } expr_1 \text{ } expr_2? \rrbracket(\sigma) = ([\text{assert } trans_E \llbracket expr_1 \rrbracket(\sigma) \text{ } expr_2?], \sigma)$$

$$trans_S \llbracket \text{import } alias^* \rrbracket(\sigma) = \\ \text{IF } \sigma_1 \setminus \sigma = [\text{“tensor_flow_compat”} \mapsto id] \text{ THEN} \\ ([\text{import } alias^*, \\ \text{import horovod.tensorflow as hvd,} \\ \text{hvd.init()}], \sigma_1) \\ \text{ELSE } ([\text{import } alias^*], \sigma_1)$$

$$trans_S \llbracket \text{from } i \text{ id? import } alias^* \rrbracket(\sigma) = ([\text{from } i \text{ id? import } alias^*], \sigma_1)$$

$$trans_S \llbracket \text{global } id^* \rrbracket(\sigma) = ([\text{global } id^*], \sigma)$$

$trans_S \llbracket \text{nonlocal } id^* \rrbracket(\sigma) = ([\text{nonlocal } id^*], \sigma)$

A strict form of expr statements

$trans_S \llbracket expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) \rrbracket(\sigma) =$

Config

IF $id_t = \sigma(\text{"tensor_flow"})$ **AND**

$expr_1 = id_t.config.experimental.set_visible_devices$ **THEN** $([], \sigma)$

Root Rank Blocking

ELIF $id_m = \sigma(\text{"model"})$ **AND** $(expr_1 = id_m.write$ **OR**

$expr_1 = id_m.summary$ **OR** $expr_1 = id_m.save_weights$ **OR**

$expr_1 = expr.load_weights$ **OR** $expr_1 = id_m.save)$ **OR**

$id_c = \sigma(\text{"checkpoint"})$ **AND** $expr_1 = id_c.save$ **OR**

$id_t = \sigma(\text{"tensor_flow"})$ **AND** $expr_1 = id_t.print$ **OR**

$expr_1 = print$ **THEN**

$([if \text{ hvd.rank}() == 0: expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k})], \sigma)$

ELSE

$[trans_E \llbracket expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) \rrbracket(\sigma)], \sigma)$

A general form of expr statements except for the strict form

$trans_S \llbracket expr \rrbracket(\sigma) =$

Variable Broadcasting

IF $\sigma(\text{"session"}) = id_s$ **AND** $\sigma(\text{"tensor_flow_compat"}) = id_t$ **AND**

$expr = id_s.run(id_t.global_variables_initializer())$ **THEN**

$([expr,$

$id_s.run(hvd.broadcast_global_variables(root_rank=0))], \sigma)$

ELIF $\sigma(\text{"tensor_flow_compat"}) = id_t$ **AND**

$expr = id_t.global_variables_initializer().run()$ **THEN**

$([expr,$

$hvd.broadcast_global_variables(root_rank=0).run()], \sigma)$

ELSE $([trans_E \llbracket expr \rrbracket(\sigma)], \sigma)$

$trans_S \llbracket \text{pass} \rrbracket(\sigma) = ([\text{pass}], \sigma)$

$trans_S \llbracket \text{break} \rrbracket(\sigma) = ([\text{break}], \sigma)$

$trans_S \llbracket \text{continue} \rrbracket(\sigma) = ([\text{continue}], \sigma)$

$trans_E \llbracket expr_1 \ expr_2 \dots expr_n \rrbracket(\sigma) = trans_E \llbracket expr_1 \rrbracket(\sigma) \ trans_E \llbracket expr_2 \rrbracket(\sigma) \dots trans_E \llbracket expr_n \rrbracket(\sigma)$

$trans_E : Expr \rightarrow \Sigma \rightarrow Expr$
--

$trans_E \llbracket expr_1 \ \text{boolop} \ expr_2 \rrbracket(\sigma) = trans_E \llbracket expr_1 \rrbracket(\sigma) \ \text{boolop} \ trans_E \llbracket expr_2 \rrbracket(\sigma)$

$$\text{trans}_E \llbracket \text{expr}_1 \text{ := } \text{expr}_2 \rrbracket(\sigma) = \text{expr}_1 \text{ := } \text{trans}_E \llbracket \text{expr}_2 \rrbracket(\sigma)$$

$$\text{trans}_E \llbracket \text{expr}_1 \text{ binop } \text{expr}_2 \rrbracket(\sigma) = \text{trans}_E \llbracket \text{expr}_1 \rrbracket(\sigma) \text{ binop } \text{trans}_E \llbracket \text{expr}_2 \rrbracket(\sigma)$$

$$\text{trans}_E \llbracket \text{unop } \text{expr} \rrbracket(\sigma) = \text{unop } \text{trans}_E \llbracket \text{expr} \rrbracket(\sigma)$$

$$\text{trans}_E \llbracket \text{lambda } \text{args} : \text{expr} \rrbracket(\sigma) = \text{lambda } \text{args} : \text{trans}_E \llbracket \text{expr} \rrbracket(\sigma)$$

$$\begin{aligned} \text{trans}_E \llbracket \text{expr}_1 \text{ if } \text{expr}_2 \text{ else } \text{expr}_3 \rrbracket(\sigma) = \\ \text{trans}_E \llbracket \text{expr}_1 \rrbracket(\sigma) \text{ if } \text{trans}_E \llbracket \text{expr}_2 \rrbracket(\sigma) \text{ else } \text{trans}_E \llbracket \text{expr}_3 \rrbracket(\sigma) \end{aligned}$$

$$\begin{aligned} \text{trans}_E \llbracket \text{expr}_1 \text{ if } \text{expr}_2 \text{ else } \text{expr}_3 \rrbracket(\sigma) = \\ \text{trans}_E \llbracket \text{expr}_1 \rrbracket(\sigma) \text{ if } \text{trans}_E \llbracket \text{expr}_2 \rrbracket(\sigma) \text{ else } \text{trans}_E \llbracket \text{expr}_3 \rrbracket(\sigma) \end{aligned}$$

$$\text{trans}_E \llbracket (\text{expr}_1 : \text{expr}_2)^* (**\text{expr}_3)^* \rrbracket(\sigma) = \{(\text{expr}_1 : \text{trans}_E \llbracket \text{expr}_2 \rrbracket(\sigma))^* (**\text{expr}_3)^*\}$$

$$\text{trans}_E \llbracket \{\text{expr}^*\} \rrbracket(\sigma) = \{(\text{trans}_E \llbracket \text{expr} \rrbracket(\sigma))^*\}$$

$$\text{trans}_E \llbracket [\text{expr}^*] \rrbracket(\sigma) = [(\text{trans}_E \llbracket \text{expr} \rrbracket(\sigma))^*]$$

$$\text{trans}_E \llbracket (\text{expr}^*) \rrbracket(\sigma) = ((\text{trans}_E \llbracket \text{expr} \rrbracket(\sigma))^*)$$

$$\text{trans}_E \llbracket \{\text{expr}_1 : \text{expr}_2 \text{ comprehension}^*\} \rrbracket(\sigma) = \{\text{expr}_1 : \text{trans}_E \llbracket \text{expr}_2 \rrbracket(\sigma) (\text{trans}_O \llbracket \text{comprehension} \rrbracket(\sigma))^*\}$$

$$\text{trans}_E \llbracket \{\text{expr} \text{ comprehension}^*\} \rrbracket(\sigma) = \{\text{trans}_E \llbracket \text{expr} \rrbracket(\sigma) (\text{trans}_O \llbracket \text{comprehension} \rrbracket(\sigma))^*\}$$

$$\text{trans}_E \llbracket [\text{expr} \text{ comprehension}^*] \rrbracket(\sigma) = [\text{trans}_E \llbracket \text{expr} \rrbracket(\sigma) (\text{trans}_O \llbracket \text{comprehension} \rrbracket(\sigma))^*]$$

$$\text{trans}_E \llbracket (\text{expr} \text{ comprehension}^*) \rrbracket(\sigma) = (\text{trans}_E \llbracket \text{expr} \rrbracket(\sigma) (\text{trans}_O \llbracket \text{comprehension} \rrbracket(\sigma))^*)$$

$$\text{trans}_E \llbracket \text{await } \text{expr} \rrbracket(\sigma) = \text{await } \text{trans}_E \llbracket \text{expr} \rrbracket(\sigma)$$

$$\text{trans}_E \llbracket \text{yield } \text{expr}? \rrbracket(\sigma) = \text{yield } (\text{trans}_E \llbracket \text{expr} \rrbracket(\sigma))?$$

$$\text{trans}_E \llbracket \text{yield from } \text{expr} \rrbracket(\sigma) = \text{yield from } \text{trans}_E \llbracket \text{expr} \rrbracket(\sigma)$$

$$\text{trans}_E \llbracket \text{expr}_1 (\text{compop } \text{expr}_2)^* \rrbracket(\sigma) = \text{trans}_E \llbracket \text{expr}_1 \rrbracket(\sigma) (\text{compop } \text{trans}_E \llbracket \text{expr}_2 \rrbracket(\sigma))^*$$

$$\begin{aligned} trans_E \llbracket expr_1 (expr_{11} \dots expr_{1n} \ (id_1 =)? \ expr_{21} \dots (id_k =)? \ expr_{2k}) \rrbracket(\sigma) = \\ trans_E \llbracket expr_1 \rrbracket(\sigma) \ (trans_E \llbracket expr_{11} \rrbracket(\sigma) \dots trans_E \llbracket expr_{1n} \rrbracket(\sigma) \\ (id_1 =)? \ trans_E \llbracket expr_{21} \rrbracket(\sigma) \dots (id_k =)? \ trans_E \llbracket expr_{2k} \rrbracket(\sigma)) \end{aligned}$$

$$trans_E \llbracket \{expr_1 \quad (!i)? \quad (:expr_2)?\} \rrbracket(\sigma) = \{expr_1 \quad (!i)? \quad (:expr_2)?\}$$

$$trans_E \llbracket expr^* \rrbracket(\sigma) = expr^*$$

$$trans_E \llbracket constant \rrbracket(\sigma) = constant$$

$$trans_E \llbracket expr.id \rrbracket(\sigma) = (trans_E \llbracket expr \rrbracket(\sigma)).id$$

$$trans_E \llbracket expr_1[expr_2] \rrbracket(\sigma) = trans_E \llbracket expr_1 \rrbracket(\sigma)[trans_E \llbracket expr_2 \rrbracket(\sigma)]$$

$$trans_E \llbracket *expr \rrbracket(\sigma) = *expr$$

$$trans_E \llbracket **expr \rrbracket(\sigma) = **expr$$

$$trans_E \llbracket id \rrbracket(\sigma) = id$$

$$trans_E \llbracket expr_1? \ (:expr_2)? \ (:expr_3)? \rrbracket(\sigma) = trans_E \llbracket expr_1 \rrbracket(\sigma)? \ (:trans_E \llbracket expr_2 \rrbracket(\sigma))? \ (:trans_E \llbracket expr_3 \rrbracket(\sigma))?$$

$$\boxed{trans_O : Comprehension \rightarrow \Sigma \rightarrow Comprehension}$$

$$trans_O \llbracket \text{for } expr_1 \text{ in } expr_2 \text{ (if } expr_3)^* \rrbracket(\sigma) = \text{for } expr_1 \text{ in } trans_E \llbracket expr_2 \rrbracket(\sigma) \text{ (if } trans_E \llbracket expr_3 \rrbracket(\sigma))^*$$

$$\boxed{trans_H : ExcHandler \rightarrow \Sigma \rightarrow ExcHandler}$$

$$trans_H \llbracket \text{except } expr? \text{ (as } id)? : stmt^* \rrbracket(\sigma) = \text{except } expr? \text{ (as } id)? : trans_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma).1$$

$$\boxed{trans_{\bar{A}} : Alias \text{ list} \rightarrow \Sigma \rightarrow \Sigma}$$

$$trans_{\bar{A}} \llbracket alias_1 \dots alias_{n-1} alias_n \rrbracket(\sigma) = trans_A \llbracket alias_n \rrbracket(trans_A \llbracket alias_{n-1} \rrbracket(\dots trans_A \llbracket alias_1 \rrbracket(\sigma)))$$

$$\boxed{trans_A : Alias \rightarrow \Sigma \rightarrow \Sigma}$$

$$trans_A \llbracket id \rrbracket(\sigma) =$$

LET $id = \text{tensorflow.compat.v1}$ **THEN** $\sigma[\text{"tensorflow.compat"} \mapsto id]$

LET $id = \text{tensorflow}$ **THEN** $\sigma[\text{"tensorflow"} \mapsto id]$

LET $id = \text{os}$ **THEN** $\sigma[\text{"os"} \mapsto id]$

ELSE σ

$trans_A \llbracket id_1 \text{ as } id_2 \rrbracket(\sigma) =$
 $\text{LET } id_1 = \text{tensorflow.compat.v1} \text{ THEN } \sigma[\text{“tensorflow.compat”} \mapsto id_2]$
 $\text{LET } id_1 = \text{tensorflow} \text{ THEN } \sigma[\text{“tensorflow”} \mapsto id_2]$
 $\text{LET } id_1 = \text{os} \text{ THEN } \sigma[\text{“os”} \mapsto id_2]$
 $\text{ELSE } \sigma$

$trans_A \llbracket id_1 . id_2 (.id_3)^* (\text{as } id_2)? \rrbracket(\sigma) = \sigma$

$trans_{\overline{W}} : WithItem \text{ list} \rightarrow \Sigma \rightarrow (WithItem \text{ list} \times \Sigma)$

$trans_{\overline{W}} \llbracket with_item_1 with_item_2 \dots with_item_n \rrbracket(\sigma) =$
 $\text{LET } with_item_1', \sigma_1 = trans_W \llbracket with_item_1 \rrbracket(\sigma) \text{ IN}$
 $\text{LET } with_item_2', \sigma_2 = trans_W \llbracket with_item_2 \rrbracket(\sigma_1) \text{ IN}$
 \dots
 $\text{LET } with_item_n', \sigma_n = trans_W \llbracket with_item_n \rrbracket(\sigma_{n-1}) \text{ IN}$
 $(with_item_1' :: with_item_2' :: \dots :: [with_item_n'], \sigma_n)$

$trans_W : WithItem \rightarrow \Sigma \rightarrow (WithItem \times \Sigma)$

$trans_W \llbracket expr \rrbracket(\sigma) = (trans_E \llbracket expr \rrbracket(\sigma), \sigma)$

$trans_W \llbracket expr_1 \text{ as } expr_2 \rrbracket(\sigma) =$
 $(trans_E \llbracket expr_1 \rrbracket(\sigma) \text{ as } expr_2, \sigma)$

$trans_C : MatchCase \rightarrow \Sigma \rightarrow MatchCase$

$trans_C \llbracket \text{case pattern (if expr)? : stmt*} \rrbracket(\sigma) =$
 $\text{case } trans_P \llbracket pattern \rrbracket(\sigma) \text{ (if } trans_E \llbracket expr \rrbracket(\sigma))? : trans_{\overline{S}} \llbracket stmt* \rrbracket(\sigma).1$

$trans_P : Pattern \rightarrow \Sigma \rightarrow Pattern$

$trans_P \llbracket expr \rrbracket(\sigma) = trans_E \llbracket expr \rrbracket(\sigma)$

$trans_P \llbracket constant \rrbracket(\sigma) = constant$

$trans_P \llbracket [pattern^*] \rrbracket(\sigma) = [trans_P \llbracket pattern \rrbracket(\sigma)]^*$

$trans_P \llbracket *(id)? \rrbracket(\sigma) = *(id)?$

$trans_P \llbracket \{(expr : pattern)^* id?\} \rrbracket(\sigma) = \{(expr : trans_P \llbracket pattern \rrbracket(\sigma))^* id?\}$

$trans_P \llbracket expr (pattern_1^* (id = pattern_2)^*) \rrbracket(\sigma) = expr (trans_P \llbracket pattern_1 \rrbracket(\sigma)^* (id = trans_P \llbracket pattern_2 \rrbracket(\sigma))^*)$

$trans_P \llbracket (pattern \text{ as})? id \rrbracket(\sigma) = (trans_P \llbracket pattern \rrbracket(\sigma) \text{ as})? id$

$$trans_P \llbracket pattern_1 \mid pattern_2 \rrbracket(\sigma) = trans_P \llbracket pattern_1 \rrbracket(\sigma) \mid trans_P \llbracket pattern_2 \rrbracket(\sigma)$$

$$trans_P \llbracket - \rrbracket(\sigma) = -$$

3 Identifying trainig loop

3.1 Restrictions

1. Training loop must be defined in only one file.
2. Training loop type is either `distributed gradient tape` or `distributed optimizer`.
3. Each model must have only one type of training loop.
4. Function must not be assigned to the variables or passed as an argument.
5. Training loop must not be defined conditionally.

3.2 Rules

3.2.1 Summary

$$\begin{aligned}
Summary & ::= \text{ModuleSummary } id \ \sigma_{\top} \ tl & (\text{MODULESUMMARY}) \\
& \quad \text{FuncSummary } tl & (\text{FUNCSUMMARY}) \\
& \quad \text{ClassSummary } arg & (\text{CLASSSUMMARY}) \\
& \quad \text{ValueSummary } s & (\text{VALUESUMMARY}) \\
arg & ::= Model \mid \perp & (\text{ARG}) \\
tl & ::= GradTape \mid Optimizer \mid \perp & (\text{TRAININGLOOP}) \\
\sigma_{\top} & ::= \sigma \cup \top
\end{aligned}$$

$$\sigma \in \Sigma = Id \xrightarrow{\text{fin}} Summary \quad \text{ENVIRONMENT STORING MAPPINGS FROM IDS TO SUMMARIES}$$

3.2.2 Training Loop

$$\boxed{summary_M : Module \rightarrow \Sigma \rightarrow ModuleSummary}$$

$$\begin{aligned}
summary_M \llbracket id \ stmt^* \ type_ignore \rrbracket = \\
\quad \mathbf{LET} \ \sigma, tl = summary_{\overline{S}} \llbracket stmt^* \rrbracket(\sigma) \ \mathbf{IN} \\
\quad \text{ModuleSummary } id \ \sigma \ tl
\end{aligned}$$

$$\boxed{summary_{\overline{S}} : Stmt \ \mathbf{list} \rightarrow \Sigma \rightarrow (\Sigma \times tl)}$$

$$\begin{aligned}
summary_{\overline{S}} \llbracket stmt_1 \ stmt_2 \dots stmt_n \rrbracket(\sigma) = & \mathbf{LET} \ \sigma_1, tl_1 = summary_S \llbracket stmt_1 \rrbracket(\sigma) \ \mathbf{IN} \\
& \mathbf{LET} \ \sigma_2, tl_2 = summary_{\overline{S}} \llbracket stmt_2 \rrbracket(\sigma_1) \ \mathbf{IN} \\
& \dots \\
& \mathbf{LET} \ \sigma_n, tl_n = summary_{\overline{S}} \llbracket stmt_n \rrbracket(\sigma_{n-1}) \ \mathbf{IN} \\
& (\sigma_n, tl_1 \sqcup tl_2 \sqcup \dots tl_n)
\end{aligned}$$

$$\boxed{summary_S : Stmt \rightarrow \Sigma \rightarrow (\Sigma \times tl)}$$

$summary_S \llbracket (@expr_1)^* \text{ def } id (args) (-> expr_2)? : (\#type:s)? stmt^* \rrbracket(\sigma) =$
 $\text{LET } \sigma', tl = summary_{\bar{S}} \llbracket stmt^* \rrbracket \text{ IN}$
 $(\sigma[id \mapsto \text{FuncSummary } tl], \perp)$

$summary_S \llbracket (@expr_1)^* \text{ async def } id (args) (-> expr_2)? : (\#type:s)? stmt^* \rrbracket(\sigma) =$
 $\text{LET } \sigma', tl = summary_{\bar{S}} \llbracket stmt^* \rrbracket \text{ IN}$
 $(\sigma[id \mapsto \text{FuncSummary } tl], \perp)$

$summary_S \llbracket (@expr)^* \text{ class } id (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) : stmt^* \rrbracket(\sigma) =$
 $\text{IF } expr_{1i} = id_c.keras.Model \text{ WHEN } 1 \leq i \leq n \text{ AND}$
 $\sigma(id_c) = \text{ModuleSummary Id("tensorflow")} \sqcup \perp \text{ THEN } (\sigma[id \mapsto \text{ClassSummary } Model], \perp)$
 $\text{ELIF } expr_{2i} = id_c.keras.Model \text{ WHEN } 1 \leq i \leq k \text{ AND}$
 $\sigma(id_c) = \text{ModuleSummary Id("tensorflow")} \sqcup \perp \text{ THEN } (\sigma[id \mapsto \text{ClassSummary } Model], \perp)$
 $\text{ELSE } (\sigma[id \mapsto \text{ClassSummary } \perp], \perp)$

A strict form of assignment statements

$summary_S \llbracket id_r = id_c () (\#type:s)? \rrbracket(\sigma) =$
 $\text{IF } \sigma(id_c) = \text{ClassSummary } Model \text{ THEN } (\sigma[id_r \mapsto \text{ValueSummary "model.instance"}], \perp)$
 $\text{ELSE } (\sigma, \perp)$

$summary_S \llbracket (\#type:s)? \text{ for } expr_1 \text{ in } expr_2 : stmt_1^* (\text{else} : stmt_2^*)? \rrbracket(\sigma) =$
 $\text{LET } _, tl = summary_{\bar{S}} \llbracket stmt_1^* \rrbracket(\sigma) \text{ IN}$
 (σ, tl)

$summary_S \llbracket (\#type:s)? \text{ async for } expr_1 \text{ in } expr_2 : stmt_1^* (\text{else} : stmt_2^*)? \rrbracket(\sigma) =$
 $\text{LET } _, tl = summary_{\bar{S}} \llbracket stmt_1^* \rrbracket(\sigma) \text{ IN}$
 (σ, tl)

$summary_S \llbracket \text{ while } (expr) : stmt_1^* (\text{else} : stmt_2^*)? \rrbracket(\sigma) =$
 $\text{LET } _, tl = summary_{\bar{S}} \llbracket stmt_1^* \rrbracket(\sigma) \text{ IN}$
 (σ, tl)

$summary_S \llbracket (\#type:s)? \text{ with } with_item^* : stmt^* \rrbracket(\sigma) =$
 $\text{LET } _, tl = summary_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma) \text{ IN}$
 $(\sigma, summary_{\bar{W}} \llbracket with_item^* \rrbracket(\sigma) \sqcup tl)$

$summary_S \llbracket (\#type:s)? \text{ async with } with_item^* : stmt^* \rrbracket(\sigma) =$
 $\text{LET } _, tl = summary_{\bar{S}} \llbracket stmt^* \rrbracket(\sigma) \text{ IN}$
 $(\sigma, summary_{\bar{W}} \llbracket with_item^* \rrbracket(\sigma) \sqcup tl)$

$summary_S \llbracket \text{ try } : stmt_1^* \text{ exc_handler}^* (\text{else} : stmt_2^*)? (\text{finally} : stmt_3^*)? \rrbracket(\sigma) =$
 $(\sigma, summary_{\bar{S}} \llbracket stmt_1^* \rrbracket(\sigma))$

$summary_S \llbracket \text{ import } alias^* \rrbracket(\sigma) = (summary_{\bar{A}} \llbracket alias^* \rrbracket(\sigma), \perp)$

```

summaryS[[ from 0 id* import alias* ]](σ) =
  LET σ' = summaryA[[ alias* ]](σ) IN
  LET [id2 ↦ ModuleSummary id2 σ2 ⊥, id2 ↦ ModuleSummary id2 σ2 ⊥,
    ... idn ↦ ModuleSummary idn σn ⊥] = σ' \ σ IN
  σ ++ [id2 ↦ ModuleSummary (id*+id2) σ2 ⊥, id2 ↦ ModuleSummary (id*+id2) σ2 ⊥,
    ... idn ↦ ModuleSummary (id*+idn) σn ⊥]

```

```

summaryS[[ expr ]](σ) = (σ, summaryE[[ expr ]](σ))

```

```

summaryS[[ stmt ]](σ) = (σ, ⊥)

```

$summary_{\bar{A}} : Alias \text{ list} \rightarrow \Sigma \rightarrow \Sigma$
--

```

summaryA[[ alias1 ... aliasn-1 aliasn ]](σ) =
  summaryA[[ aliasn ]](summaryA[[ aliasn-1 ]](... summaryA[[ alias1 ]](σ) ...))

```

$trans_A : Alias \rightarrow \Sigma \rightarrow \Sigma$

```

transA[[ id1 id2 ... idn ]](σ) =
  LET σn = [idn ↦ ModuleSummary idn ⊤ ⊥]
  LET σn-1 = [idn-1 ↦ ModuleSummary idn-1 σn ⊥]
  ...
  LET σ2 = [id2 ↦ ModuleSummary id2 σ3 ⊥]
  LET σ1 = [id1 ↦ ModuleSummary id1 σ2 ⊥]
  σ ++ σ1

```

$summary_{\bar{W}} : WithItem \text{ list} \rightarrow \Sigma \rightarrow tl$

```

summaryW[[ with_item1 with_item2 ... with_itemn ]](σ) =
  summaryW[[ with_item1 ]](σ) ⊔ summaryW[[ with_item2 ]](σ) ⊔ ... summaryW[[ with_itemn ]](σ)

```

$summary_W : WithItem \rightarrow \Sigma \rightarrow tl$
--

```

summaryW[[ expr1 as expr2? ]](σ) = summaryE[[ expr1 ]](σ)

```

$summary_E : Expr \rightarrow \Sigma \rightarrow tl$
--

A strict form of call expression

```

summaryE[[ expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) ]](σ) =
  IF σ(idr) = ValueSummary "model_instance" AND
    expr1 = idr.fit THEN Optimizer
  ELIF σ(idr) = ModuleSummary Id("tensorflow") σ' tl AND
    expr1 = idr.GradientTape THEN GradTape
  ELIF σ(idr) = FuncSummary tl AND
    expr1 = idr THEN tl
  ELSE ⊥

```

```

summaryE[[ expr ]](σ) = ⊥

```