

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 for TF2 Python 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)

2.2.2 Transformation Rules

$trans_M : Module \rightarrow Module$

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

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

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

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

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

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

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

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

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

A strict form of assignment statements

$trans_S \llbracket id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) (\#type:s)? \rrbracket(\sigma) =$
 $\quad \mathbf{IF} \sigma(\text{"tensor_flow"}) = id_t \text{ AND } expr_1 = id_t.data.Dataset.expr_3 \text{ THEN}$
 $\quad ([id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) (\#type:s)?],$
 $\quad \sigma[\text{"dataset"} \mapsto id_r])$
 $\quad \mathbf{ELIF} \sigma(\text{"tensor_flow"}) = id_t \text{ AND } expr_1 = id_t.train.Checkpoint \text{ THEN}$
 $\quad ([id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) (\#type:s)?],$
 $\quad \sigma[\text{"checkpoint"} \mapsto id_r])$
 $\quad \mathbf{ELIF} \sigma(\text{"tensor_flow"}) = id_t \text{ AND } expr_1 = id_t.optimizers.Adam \text{ THEN}$
 $\quad \mathbf{IF} id_i = \text{learning_rate} \text{ WHEN } 1 \leq i \leq k \text{ THEN}$
 $\quad ([id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots id_i = expr_{2i} * \text{hvd.size}()$
 $\quad \dots (id_k =)? expr_{2k}) (\#type:s)?], \sigma[\text{"optimizer"} \mapsto id_r])$
 $\quad \mathbf{ELSE}$
 $\quad ([id_r = expr_1 (expr_{11} * \text{hvd.size}() \dots expr_{1n} (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) (\#type:s)?],$
 $\quad \sigma[\text{"optimizer"} \mapsto id_r])$
 $\quad \mathbf{ELIF} \sigma(\text{"optimizers"}) = id_t \text{ AND } expr_1 = id_t.Adam \text{ THEN}$
 $\quad \mathbf{IF} id_i = \text{learning_rate} \text{ WHEN } 1 \leq i \leq k \text{ THEN}$
 $\quad ([id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots id_i = expr_{2i} * \text{hvd.size}()$

```

... (idk = )? expr2k) (#type:s)?], σ["optimizer" ↦ idr])
ELSE
  ([idr = expr1 (expr11 * hvd.size() ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?],
   σ["optimizer" ↦ idr])
ELIF 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,
       idr = expr1 (expr11 ... expr1n (id1 = )? expr21 ... idi = idz ... (idk = )? expr2k) (#type:s)?,
       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,
           idr = expr1 (idz expr12 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?,
           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)
          ELIF σ("checkpoint") = idt AND expr1 = idt.save THEN
            ([if hvd.rank() == 0: [idr = expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?]],
             σ)
            ELIF σ("keras") = idk AND expr1 = idk.models.Sequential THEN
              ([idr = expr1 (expr11 ... expr1n (id2 = )? expr21 ... (idk = )? expr2k)], σ["model" ↦ idr])
            ELIF σ("keras") = idt AND expr1 = idt.optimizers.Adam 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])
              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[[expr₁ binop = expr₂]](σ) = ([expr₁ binop = trans_E[[expr₂]](σ)], σ)

$trans_S \llbracket expr_1 : expr_2 (= expr_3)? \rrbracket(\sigma) = ([expr_1 : expr_2 (= trans_E \llbracket expr_3 \rrbracket(\sigma))?), \sigma)$
IF $expr_1 = id_1$ **AND** $\sigma(\text{"tensor_flow"}) = id_2$
AND $expr_3 = id_2.data.Dataset.expr_4 (expr_5^* keyword^*)$ **THEN**
 $([expr_1 : expr_2 = expr_3 (\#type:s)?], \sigma[\text{"dataset"} \mapsto id_1])$
ELSE $([expr_1 : expr_2 (= trans_E \llbracket expr_3 \rrbracket(\sigma))?), \sigma)$

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

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

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

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

$trans_S \llbracket (\#type:s)? \text{ with } with_item^* : stmt^* \rrbracket(\sigma) =$
LET $with_item^{*'}, \sigma_1 = trans_{\overline{W}} \llbracket with_item^* \rrbracket(\sigma)$ **IN**
LET $stmt^{*'}, \sigma_2 = trans_{\overline{S}} \llbracket stmt^* \rrbracket(\sigma_1)$ **IN**
IF $\sigma_1 \setminus \sigma = [\text{"gradient_tape"} \mapsto id]$ **THEN**
 $([(\#type:s)? \text{ with } with_item^{*'} : stmt^{*'},$
 $id = hvd.DistributedGradientTape(id)], \sigma_2)$
ELSE $([(\#type:s)? \text{ with } with_item^{*'} : stmt^{*'}], \sigma_2)$

$trans_S \llbracket (\#type:s)? \text{ async with } with_item^* : stmt^* \rrbracket(\sigma) =$
LET $with_item^{*'}, \sigma_1 = trans_{\overline{W}} \llbracket with_item^* \rrbracket(\sigma)$ **IN**
LET $stmt^{*'}, \sigma_2 = trans_{\overline{S}} \llbracket stmt^* \rrbracket(\sigma_1)$ **IN**
IF $\sigma_1 \setminus \sigma = [\text{"gradient_tape"} \mapsto id]$ **THEN**
 $([(\#type:s)? \text{ async with } with_item^{*'} : stmt^{*'},$
 $id = hvd.DistributedGradientTape(id)], \sigma_2)$
ELSE $([(\#type:s)? \text{ async with } with_item^{*'} : stmt^{*'}], \sigma_2)$

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

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

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

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

$trans_S \llbracket \text{import } alias^* \rrbracket(\sigma) =$
LET $\sigma_1 = trans_{\bar{A}} \llbracket alias^* \rrbracket(\sigma)$ **IN**
IF $\sigma_1 \setminus \sigma = [\text{"tensor_flow"} \mapsto id]$ **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: $id.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')$], σ_1)
ELSE ([import $alias^*$], σ_1)

$trans_S \llbracket \text{import } alias^* \rrbracket(\sigma) =$
LET $\sigma_1 = trans_{\bar{A}} \llbracket alias^* \rrbracket(\sigma)$ **IN**
IF $\sigma_1 \setminus \sigma = [\text{"tensor_flow"} \mapsto id]$ **THEN**
 ([import $alias^*$,
 import horovod.tensorflow as hvd,
 hvd.init(),
 gpus = $id.config.experimental.list_physical_devices('GPU')$,
 for gpu in gpus: $id.config.experimental.set_memory_growth(gpu, True)$,
 if gpus: $id.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')$], σ_1)
ELSE ([import $alias^*$], σ_1)

$trans_S \llbracket \text{from } i \text{ } id? \text{ import } alias^* \rrbracket(\sigma) =$
LET $\sigma_1 = trans_{\bar{A}} \llbracket alias^* \rrbracket(\sigma)$ **IN**
IF $id = \text{tensorflow}$ **AND** $\sigma_1 \setminus \sigma = [\text{"keras"} \mapsto id]$ **THEN**
 ([from $i \text{ } id?$ import $alias^*$,
 import horovod.tensorflow.keras as hvd,
 hvd.init(),
 gpus = $id.config.experimental.list_physical_devices('GPU')$,
 for gpu in gpus: $id.config.experimental.set_memory_growth(gpu, True)$,
 if gpus: $id.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')$], σ_1)
ELSE ([from $i \text{ } id?$ import $alias^*$], σ_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

$trans_S \llbracket expr_1 (expr_{11} \dots expr_{1n} \text{ } (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) \rrbracket(\sigma) =$
IF $\sigma(\text{"optimizer"}) = id_t$ **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
    ( $id_z$  =  $expr_{2i}$ ,
     $expr_1$  ( $expr_{11}$  ...  $expr_{1n}$  ( $id_1$  = )?  $expr_{21}$  ...  $id_i$  =  $id_z$  ... ( $id_k$  = )?  $expr_{2k}$ ),
    global hvd_broadcast_done,
    if not hvd_broadcast_done: [ hvd.broadcast_variables([x[1] for x 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
      ( $id_z$  =  $expr_{11}$ ,
       $expr_1$  ( $id_z$   $expr_{12}$  ...  $expr_{1n}$  ( $id_1$  = )?  $expr_{21}$  ... ( $id_k$  = )?  $expr_{2k}$ ),
      global hvd_broadcast_done,
      if not hvd_broadcast_done: [ hvd.broadcast_variables([x[1] for x in  $id_z$ ], root_rank=0),
        hvd.broadcast_variables( $id_t$ .variables(), root_rank=0),
        hvd_broadcast_done = True ],  $\sigma$ )
  ELIF  $expr_1$  = print THEN
    ([if hvd.rank() == 0:,
      [ $expr_1$  ( $expr_{11}$  ...  $expr_{1n}$  ( $id_1$  = )?  $expr_{21}$  ... ( $id_k$  = )?  $expr_{2k}$ )]],  $\sigma$ )
  ELIF  $\sigma$ ("checkpoint") =  $id_t$  AND  $expr_1$  =  $id_t$ .save THEN
    ([if hvd.rank() == 0: [ $expr_1$  ( $expr_{11}$  ...  $expr_{1n}$  ( $id_1$  = )?  $expr_{21}$  ... ( $id_k$  = )?  $expr_{2k}$ )]],  $\sigma$ )
  ELIF  $\sigma$ ("model") =  $id_t$  AND  $expr_1$  =  $id_t$ .fit THEN
    IF  $id_i$  = verbose WHEN  $1 \leq i \leq k$  THEN
      IF  $id_j$  = callbacks WHEN  $1 \leq j \leq k$  THEN
        ([callback = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)]
        if hvd.rank() == 0: callbacks.append( $expr_{2j}$ )
         $expr_1$  ( $expr_{11}$  ...  $expr_{1n}$  ( $id_1$  = )?  $expr_{21}$  ...  $id_i$  = 1 if hvd.rank() == 0 else 0
        ...  $id_j$  = callbacks ... ( $id_k$  = )?  $expr_{2k}$ )],  $\sigma$ )
      ELSE
        ([ $expr_1$  ( $expr_{11}$  ...  $expr_{1n}$  ( $id_1$  = )?  $expr_{21}$  ...
           $id_i$  = 1 if hvd.rank() == 0 else 0 ... ( $id_k$  = )?  $expr_{2k}$ 
          callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)]],  $\sigma$ )
      ELSE
        IF  $id_j$  = callbacks WHEN  $1 \leq j \leq k$  THEN
          ([callback = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)]
          if hvd.rank() == 0: callbacks.append( $expr_{2j}$ )
           $expr_1$  ( $expr_{11}$  ...  $expr_{1n}$  ( $id_1$  = )?  $expr_{21}$  ...  $id_j$  = callbacks ... ( $id_k$  = )?  $expr_{2k}$ 
          verbose = 1 if hvd.rank() == 0 else 0)],  $\sigma$ )
        ELSE
          ([ $expr_1$  ( $expr_{11}$  ...  $expr_{1n}$  ( $id_1$  = )?  $expr_{21}$  ... ( $id_k$  = )?  $expr_{2k}$ 
          verbose = 1 if hvd.rank() == 0 else 0
          callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(0)]],  $\sigma$ )
        ELIF  $\sigma$ ("model") =  $id_t$  AND  $expr_1$  =  $id_t$ .compile THEN
          IF  $id_i$  = optimizer AND  $expr_{2i}$  = "adam" WHEN  $1 \leq i \leq k$  THEN

```

```

    ([ $id_z$  = tf.optimizers.Adam(learning_rate=0.001 * hvd.size())
     $id_z$  = hvd.DistributedOptimizer( $id_z$ )
     $expr_1$  ( $expr_{11}$  ...  $expr_{1n}$  ( $id_1 =$  )?  $expr_{21}$  ...  $id_i = id_z$  ... ( $id_k =$  )?  $expr_{2k}$ )] ,  $\sigma$ )
ELIF  $expr_{11}$  = "adam" THEN
    ([ $id_z$  = tf.optimizers.Adam(learning_rate=0.001 * hvd.size())
     $id_z$  = hvd.DistributedOptimizer( $id_z$ )
     $expr_1$  ( $id_z$  ...  $expr_{1n}$  ( $id_1 =$  )?  $expr_{21}$  ... ( $id_k =$  )?  $expr_{2k}$ )] ,  $\sigma$ )
ELSE
    [ $trans_E$ [[ $expr_1$  ( $expr_{11}$  ...  $expr_{1n}$  ( $id_1 =$  )?  $expr_{21}$  ... ( $id_k =$  )?  $expr_{2k}$ ) ]]( $\sigma$ )] ,  $\sigma$ )
ELIF  $expr_1$  = model.summary THEN
    ([if hvd.rank() == 0:  $expr_1$  ( $expr_{11}$  ...  $expr_{1n}$  ( $id_1 =$  )?  $expr_{21}$  ... ( $id_k =$  )?  $expr_{2k}$ )] ,  $\sigma$ )
ELSE
    [ $trans_E$ [[ $expr_1$  ( $expr_{11}$  ...  $expr_{1n}$  ( $id_1 =$  )?  $expr_{21}$  ... ( $id_k =$  )?  $expr_{2k}$ ) ]]( $\sigma$ )] ,  $\sigma$ )

```

A general form of expr statements except for the strict form

$trans_S$ [[$expr$]](σ) = ([$trans_E$ [[$expr$]](σ)], σ)

$trans_S$ [[**pass**]](σ) = ([**pass**], σ)

$trans_S$ [[**break**]](σ) = ([**break**], σ)

$trans_S$ [[**continue**]](σ) = ([**continue**], σ)

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

$trans_E$ [[$expr_1$ **boolop** $expr_2$]](σ) = $trans_E$ [[$expr_1$]](σ) **boolop** $trans_E$ [[$expr_2$]](σ)

$trans_E$ [[$expr_1$ **:=** $expr_2$]](σ) = $expr_1$ **:=** $trans_E$ [[$expr_2$]](σ)

$trans_E$ [[$expr_1$ **binop** $expr_2$]](σ) = $trans_E$ [[$expr_1$]](σ) **binop** $trans_E$ [[$expr_2$]](σ)

$trans_E$ [[**unop** $expr$]](σ) = **unop** $trans_E$ [[$expr$]](σ)

$trans_E$ [[**lambda** $args$: $expr$]](σ) = **lambda** $args$: $trans_E$ [[$expr$]](σ)

$trans_E$ [[$expr_1$ **if** $expr_2$ **else** $expr_3$]](σ) =
 $trans_E$ [[$expr_1$]](σ) **if** $trans_E$ [[$expr_2$]](σ) **else** $trans_E$ [[$expr_3$]](σ)

$trans_E$ [[$expr_1$ **if** $expr_2$ **else** $expr_3$]](σ) =
 $trans_E$ [[$expr_1$]](σ) **if** $trans_E$ [[$expr_2$]](σ) **else** $trans_E$ [[$expr_3$]](σ)

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

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

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

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

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

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

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

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

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

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

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

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

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

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**

$expr_1 \text{ (} expr_{11} \dots expr_{1n} \text{ (} id_1 = \text{)? } expr_{21} \dots id_i = expr_{2i} // \text{ hvd.size() } \dots (id_k = \text{)? } expr_{2k} \text{)}$

ELSE

$expr_1 \text{ (} expr_{11} // \text{ hvd.size() } \dots expr_{1n} \text{ (} id_1 = \text{)? } expr_{21} \dots (id_k = \text{)? } expr_{2k} \text{)}$

ELSE

$trans_E \llbracket expr_1 \rrbracket(\sigma) \ (trans_E \llbracket expr_{11} \rrbracket(\sigma) \dots trans_E \llbracket expr_{1n} \rrbracket(\sigma)$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

LET $id = \text{keras}$ **THEN** $\sigma[\text{“keras”} \mapsto id]$

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

LET $id = \text{optimizers}$ **THEN** $\sigma[\text{“optimizers”} \mapsto id]$

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 $id_1 = \text{tensorflow.keras}$ **THEN** $\sigma[\text{“keras”} \mapsto id_2]$

LET $id_1 = \text{os}$ **THEN** $\sigma[\text{“os”} \mapsto id_2]$

LET $id = \text{optimizers}$ **THEN** $\sigma[\text{“optimizers”} \mapsto id]$

ELSE σ

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

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

$trans_{\overline{W}} \llbracket with_item_1 with_item_2 \dots with_item_n \rrbracket(\sigma) =$
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**
 $(with_item_1' :: with_item_2' :: \dots :: [with_item_n'], \sigma_n)$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2.2 Training Loop

$\textit{summary}_M : \textit{Module} \rightarrow \Sigma \rightarrow \textit{ModuleSummary}$

$$\begin{aligned}
 \textit{summary}_M \llbracket id \ stmt^* \ type_ignore \rrbracket = \\
 \quad \textbf{LET } \sigma, tl = \textit{summary}_{\overline{S}} \llbracket stmt^* \rrbracket (\sigma) \textbf{ IN} \\
 \quad \textit{ModuleSummary } id \ \sigma \ tl
 \end{aligned}$$

$\textit{summary}_{\overline{S}} : \textit{Stmt list} \rightarrow \Sigma \rightarrow (\Sigma \times tl)$

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

$\textit{summary}_S : \textit{Stmt} \rightarrow \Sigma \rightarrow (\Sigma \times tl)$

$$\begin{aligned}
 \textit{summary}_S \llbracket (@expr_1)^* \textbf{def } id \ (args) \ (-> expr_2)^? : (\#type:s)^? \ stmt^* \rrbracket (\sigma) = \\
 \quad \textbf{LET } \sigma', tl = \textit{summary}_{\overline{S}} \llbracket stmt^* \rrbracket \textbf{ IN} \\
 \quad (\sigma[id \mapsto \textit{FuncSummary } tl], \perp)
 \end{aligned}$$

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

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

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 _A : Alias list → Σ → Σ

```

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

```

trans _A : Alias → Σ → Σ

```

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 _W : WithItem list → Σ → tl

```

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

```

summary _W : WithItem → Σ → tl
--

```

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

```

summary _E : Expr → Σ → 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) = ⊥

```