

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

$\text{ClassNode} = Str \times \text{ClassNode?}$

$\text{ns} \in \text{Nodes} ::= \text{ClassNode list}$

$\text{vs} \in \text{Vars} ::= Str \text{ list} \xrightarrow{\text{fin}} \text{ClassNode}$

$\text{cg} \in \text{CG} = \text{Nodes} \times \text{Vars}$

.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)
.nodes	: $\text{ClassNode} \rightarrow \text{Nodes} = \text{.1}$	GET NODES OF CLASS GRAPH
.vars	: $\text{ClassNode} \rightarrow \text{Vars} = \text{.2}$	GET VARIABLES OF CLASS GRAPH

2.2.2 Transformation Rules

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

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

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

$$trans_{\bar{S}} \llbracket stmt_1 \text{ stmt}_2 \dots stmt_n \rrbracket(\sigma) = \text{LET } stmt_1^{*'}, \sigma_1 = trans_S \llbracket stmt_1 \rrbracket(\sigma) \text{ IN}$$

$$\text{LET } stmt_2^{*'}, \sigma_2 = trans_S \llbracket stmt_2 \rrbracket(\sigma_1) \text{ IN}$$

$$\dots$$

$$\text{LET } stmt_n^{*'}, \sigma_n = trans_S \llbracket stmt_n \rrbracket(\sigma_{n-1}) \text{ IN}$$

$$(stmt_1^{*'} @ stmt_2^{*'} @ \dots @ stmt_n^{*'}, \sigma_n)$$

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

$$trans_S \llbracket (@expr_1)^* \text{ def } id (args) (-> expr_2)? : (\#type:s)? stmt^* \rrbracket(\sigma) =$$

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

$$trans_S \llbracket (@expr_1)^* \text{ async def } id (args) (-> expr_2)? : (\#type:s)? stmt^* \rrbracket(\sigma) =$$

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

$$trans_S \llbracket (@expr_1)^* \text{ class } id (expr_2^* keyword^*) : stmt^* \rrbracket(\sigma) =$$

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

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

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

A strict form of assignment statements

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

$$\text{IF } \sigma(\text{"tensor_flow"}) = id_t \text{ AND } expr_1 = id_t.\text{data.Dataset.expr}_3 \text{ THEN}$$

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

$$\sigma(\text{"dataset"} \mapsto id_r])$$

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

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

$$\sigma(\text{"checkpoint"} \mapsto id_r])$$

$$\text{ELIF } \sigma(\text{"tensor_flow"}) = id_t \text{ AND } expr_1 = id_t.\text{optimizers.Adam} \text{ THEN}$$

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

$$([id_r = expr_1 (expr_{11} \dots expr_{1n} (id_1 =)? expr_{21} \dots id_i = expr_{2i} * \text{hvd.size}())$$

```

... (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("optimizers") = idt AND expr1 = idt.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)?], 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)

```

$trans_S \llbracket expr_0 = expr_1 \text{ (\#type:s) } ? \rrbracket(\sigma) =$
IF $\sigma(\text{"os"}) = id_{os}$ **AND** $expr_0 = id_{os}.\text{environ} \llbracket \text{'CUDA_VISIBLE_DEVICES'} \rrbracket$
THEN $([], \sigma)$

A general form of assignment statements except for the strict form

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

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

$trans_S \llbracket expr_1 : expr_2 (= expr_3) ? \rrbracket(\sigma) = ([expr_1 : expr_2 (= trans_E \llbracket expr_3 \rrbracket(\sigma)) ?], \sigma)$
IF $expr_1 = id_1$ **AND** $\sigma(\text{"tensor_flow"}) = id_2$
AND $expr_3 = id_2.\text{data.Dataset.expr}_4 \text{ (} expr_5^* \text{ keyword*)}$ **THEN**
 $([expr_1 : expr_2 = expr_3 \text{ (\#type:s) } ?], \sigma[\text{"dataset"} \mapsto id_1])$
ELSE $([expr_1 : expr_2 (= trans_E \llbracket expr_3 \rrbracket(\sigma)) ?], \sigma)$

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

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

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

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

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

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

```

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* ]](cg) = LET (ns, vs) = cg IN
  LET (ns1, vs1) = transA[[ alias* ]](cg) IN
  IF vs1 \ vs = [idt ↦ { “tensor_flow”, nodet }] THEN
    ([import alias*,
     import horovod.tensorflow as hvd,
     hvd.broadcast_done = False,
     hvd.init(),
     gpus = idt.config.experimental.list_physical_devices(‘GPU’),
     for gpu in gpus: idt.config.experimental.set_memory_growth(gpu, True),
     if gpus: idt.config.experimental.set_visible_devices(gpus[hvd.local_rank()], ‘GPU’)], (ns1, vs1))
  ELSE ([import alias*], cg)

transS[[ from i id? import alias* ]](σ) =
  LET σ1 = transA[[ alias* ]](σ) IN
  IF id = tensorflow AND σ1 \ σ = [“keras” ↦ id] THEN
    ([from i 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’)], σ1)
  ELSE ([from i id? 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) =
  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)
ELIF expr1 = print OR expr1 = expr.write OR expr1 = expr.summary OR
expr1 = expr.save_weights OR expr1 = expr.load_weights THEN
([if hvd.rank() == 0: expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k)], σ)
ELIF σ("checkpoint") = idt AND expr1 = idt.save THEN
([if hvd.rank() == 0: [expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k)]], σ)
ELIF σ("model") = idt AND expr1 = idt.fit THEN
IF idi = verbose WHEN 1 ≤ i ≤ k THEN
IF idj = callbacks WHEN 1 ≤ j ≤ k THEN
([callback = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)]
if hvd.rank() == 0: callbacks.append(expr2j)
expr1 (expr11 ... expr1n (id1 = )? expr21 ... idi = 1 if hvd.rank() == 0 else 0
... idj = callbacks ... (idk = )? expr2k)], σ)
ELSE
([expr1 (expr11 ... expr1n (id1 = )? expr21 ...
idi = 1 if hvd.rank() == 0 else 0 ... (idk = )? expr2k
callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)]]], σ)
ELSE
IF idj = callbacks WHEN 1 ≤ j ≤ k THEN
([callback = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)]
if hvd.rank() == 0: callbacks.append(expr2j)
expr1 (expr11 ... expr1n (id1 = )? expr21 ... idj = callbacks ... (idk = )? expr2k
verbose = 1 if hvd.rank() == 0 else 0)], σ)
ELSE
([expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k
verbose = 1 if hvd.rank() == 0 else 0
callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(0)]]], σ)
ELIF σ("model") = idt AND expr1 = idt.compile THEN
IF idi = optimizer AND expr2i = "adam" WHEN 1 ≤ i ≤ k THEN
([idz = tf.optimizers.Adam(learning_rate=0.001 * hvd.size())
idz = hvd.DistributedOptimizer(idz)

```

```

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

```

A general form of expr statements except for the strict form

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\text{trans}_E \llbracket expr_1? (:expr_2)? (:expr_3)? \rrbracket(\sigma) = \text{trans}_E \llbracket expr_1 \rrbracket(\sigma)? (: \text{trans}_E \llbracket expr_2 \rrbracket(\sigma))? (: \text{trans}_E \llbracket expr_3 \rrbracket(\sigma))?$$

$$\boxed{\text{trans}_O : \text{Comprehension} \rightarrow \Sigma \rightarrow \text{Comprehension}}$$

$$\text{trans}_O \llbracket \text{for } expr_1 \text{ in } expr_2 \text{ (if } expr_3)^* \rrbracket(\sigma) = \text{for } expr_1 \text{ in } \text{trans}_E \llbracket expr_2 \rrbracket(\sigma) \text{ (if } \text{trans}_E \llbracket expr_3 \rrbracket(\sigma))^*$$

$$\boxed{\text{trans}_H : \text{ExcHandler} \rightarrow \Sigma \rightarrow \text{ExcHandler}}$$

$$\text{trans}_H \llbracket \text{except } expr? \text{ (as } id)? : stmt^* \rrbracket(\sigma) = \text{except } expr? \text{ (as } id)? : \text{trans}_{\overline{S}} \llbracket stmt^* \rrbracket(\sigma).1$$

$$\boxed{\text{trans}_{\overline{A}} : \text{Alias list} \rightarrow \Sigma \rightarrow \Sigma}$$

$$\text{trans}_{\overline{A}} \llbracket alias_1 \dots alias_{n-1} alias_n \rrbracket(\sigma) = \text{trans}_A \llbracket alias_n \rrbracket(\text{trans}_A \llbracket alias_{n-1} \rrbracket(\dots \text{trans}_A \llbracket alias_1 \rrbracket(\sigma)))$$

$$\boxed{\text{trans}_A : \text{Alias} \rightarrow \text{CG} \rightarrow \text{CG}}$$

$$\text{trans}_A \llbracket id_1 . id_2 \dots . id_n \rrbracket(\text{cg}) =$$

$$\text{LET } (\text{ns}, \text{vs}) = \text{CG IN}$$

$$\text{LET } node_1 = \{ id_1, \cdot \}, node_i = \{ id_i, node_{i-1} \} \text{ WHEN } 2 \leq i \leq n \text{ IN}$$

$$\text{LET } var_k = id_1 \dots id_k \text{ WHEN } 1 \leq k \leq n \text{ IN}$$

$$(\text{ns}@[node_1, \dots, node_n], \text{vs}@[var_1 \mapsto node_1, \dots, var_n \mapsto node_n])$$

$$\text{trans}_A \llbracket id_1 . id_2 \dots . id_n \text{ as } id_{as} \rrbracket(\text{cg}) =$$

$$\text{LET } (\text{ns}, \text{vs}) = \text{CG IN}$$

$$\text{LET } node_1 = \{ id_1, \cdot \}, node_i = \{ id_i, node_{i-1} \} \text{ WHEN } 2 \leq i \leq n \text{ IN}$$

$$(\text{ns}@[node_1, \dots, node_n], \text{vs}@[id_{as} \mapsto node_n])$$

$$\boxed{\text{trans}_{\overline{W}} : \text{WithItem list} \rightarrow \Sigma \rightarrow (\text{WithItem list} \times \Sigma)}$$

$$\text{trans}_{\overline{W}} \llbracket with_item_1 with_item_2 \dots with_item_n \rrbracket(\sigma) =$$

$$\text{LET } with_item_1', \sigma_1 = \text{trans}_W \llbracket with_item_1 \rrbracket(\sigma) \text{ IN}$$

$$\text{LET } with_item_2', \sigma_2 = \text{trans}_W \llbracket with_item_2 \rrbracket(\sigma_1) \text{ IN}$$

inden ...

$$\text{LET } with_item_n', \sigma_n = \text{trans}_W \llbracket with_item_n \rrbracket(\sigma_{n-1}) \text{ IN}$$

$$(with_item_1' :: with_item_2' :: \dots :: [with_item_n'], \sigma_n)$$

$$\boxed{\text{trans}_W : \text{WithItem} \rightarrow \Sigma \rightarrow (\text{WithItem} \times \Sigma)}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2.2 Training Loop

$summary_M : Module \rightarrow \Sigma \rightarrow ModuleSummary$

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

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

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

$summary_S : Stmt \rightarrow \Sigma \rightarrow (\Sigma \times tl)$

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

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

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

A strict form of assignment statements

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

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

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

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

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

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

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

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

$summary_S \llbracket \text{ from } 0 \ id^* \text{ import } alias^* \rrbracket(\sigma) =$
LET $\sigma' = summary_{\bar{A}} \llbracket alias^* \rrbracket(\sigma)$ **IN**
LET $[id_2 \mapsto \text{ModuleSummary } id_2 \ \sigma_2 \ \perp, \ id_2 \mapsto \text{ModuleSummary } id_2 \ \sigma_2 \ \perp,$
 $\dots id_n \mapsto \text{ModuleSummary } id_n \ \sigma_n \ \perp] = \sigma' \setminus \sigma$ **IN**
 $\sigma \ ++ \ [id_2 \mapsto \text{ModuleSummary } (id^* + id_2) \ \sigma_2 \ \perp, \ id_2 \mapsto \text{ModuleSummary } (id^* + id_2) \ \sigma_2 \ \perp,$
 $\dots id_n \mapsto \text{ModuleSummary } (id^* + id_n) \ \sigma_n \ \perp]$

$summary_S \llbracket expr \rrbracket(\sigma) = (\sigma, summary_E \llbracket expr \rrbracket(\sigma))$

$summary_S \llbracket stmt \rrbracket(\sigma) = (\sigma, \perp)$

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

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

$trans_A : \text{Alias} \rightarrow \Sigma \rightarrow \Sigma$
--

$trans_A \llbracket id_1 id_2 \dots id_n \rrbracket(\sigma) =$
LET $\sigma_n = [id_n \mapsto \text{ModuleSummary } id_n \top \perp]$
LET $\sigma_{n-1} = [id_{n-1} \mapsto \text{ModuleSummary } id_{n-1} \sigma_n \perp]$
 \dots
LET $\sigma_2 = [id_2 \mapsto \text{ModuleSummary } id_2 \sigma_3 \perp]$
LET $\sigma_1 = [id_1 \mapsto \text{ModuleSummary } id_1 \sigma_2 \perp]$
 $\sigma \text{ ++ } \sigma_1$

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

$summary_{\overline{W}} \llbracket with_item_1 with_item_2 \dots with_item_n \rrbracket(\sigma) =$
 $summary_W \llbracket with_item_1 \rrbracket(\sigma) \sqcup summary_W \llbracket with_item_2 \rrbracket(\sigma) \sqcup \dots summary_W \llbracket with_item_n \rrbracket(\sigma)$

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

$summary_W \llbracket expr_1 \text{ as } expr_2? \rrbracket(\sigma) = summary_E \llbracket expr_1 \rrbracket(\sigma)$

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

A strict form of call expression

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

IF $\sigma(id_r) = \text{ValueSummary "model.instance"}$ **AND**
 $expr_1 = id_r.\text{fit}$ **THEN** *Optimizer*
ELIF $\sigma(id_r) = \text{ModuleSummary Id("tensorflow")}$ $\sigma' tl$ **AND**
 $expr_1 = id_r.\text{GradientTape}$ **THEN** *GradTape*
ELIF $\sigma(id_r) = \text{FuncSummary } tl$ **AND**
 $expr_1 = \text{idr}$ **THEN** tl
ELSE \perp

$summary_E \llbracket expr \rrbracket(\sigma) = \perp$