

Code Transformation for Distributed Python ML Code

1 Python Abstract Syntax

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

τ	$::=$	<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

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

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

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

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

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

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

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

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

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

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

A strict form of assignment statements

```

transS[[ idr = expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)? ]](σ) =
  IF σ("tensor_flow") = idt AND expr1 = idt.data.Dataset.expr3 THEN
    ([idr = expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?],
     σ["dataset" ↦ idr])
  ELIF σ("tensor_flow") = idt AND expr1 = idt.train.Checkpoint THEN
    ([idr = expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) (#type:s)?],
     σ["checkpoint" ↦ idr])
  ELIF σ("tensor_flow") = 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)?], 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)
        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

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

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

$trans_S \llbracket expr_1 : expr_2 (= expr_3)? \rrbracket(\sigma) = ([expr_1 : expr_2 (= trans_E \llbracket expr_3 \rrbracket(\sigma))?], \sigma)$
IF $expr_1 = id_1$ **AND** $\sigma(\text{"tensor_flow"}) = id_2$
AND $expr_3 = id_2.data.Dataset.expr_4 (expr_5^* keyword^*)$ **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**
 $([(\#type:s)? \text{ with } with_item'^* : stmt'^*,$
 $id = \text{hvd.DistributedGradientTape}(id)], \sigma_2)$
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**
 $([(\#type:s)? \text{ async with } with_item'^* : stmt'^*,$
 $id = \text{hvd.DistributedGradientTape}(id)], \sigma_2)$
ELSE $([(\#type:s)? \text{ async with } with_item'^* : stmt'^*], \sigma_2)$

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

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

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

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

```

transS[[ import alias* ]](σ) =
  LET σ1 = transA[[ alias* ]](σ) IN
  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[[ from i id? import alias* ]](σ) = ([from i id? import alias*], σ)

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 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)
  ELSE
    [transE[[ expr1 (expr11 ... expr1n (id1 = )? expr21 ... (idk = )? expr2k) ]](σ)], σ)

```

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 → Σ → Expr

trans_E[[expr₁ boolop expr₂]](σ) = trans_E[[expr₁]](σ) boolop trans_E[[expr₂]](σ)

trans_E[[expr₁ := expr₂]](σ) = expr₁ := trans_E[[expr₂]](σ)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned}
& trans_E \llbracket expr_1 (expr_{11} \dots expr_{1n} \ (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) \rrbracket(\sigma) = \\
& \quad \text{IF } \sigma(\text{“dataset”}) = id_t \text{ AND } expr_1 = id_t.\text{take} \text{ THEN} \\
& \quad \quad \text{IF } id_i = \text{count} \text{ WHEN } 1 \leq i \leq k \text{ THEN} \\
& \quad \quad \quad expr_1 (expr_{11} \dots expr_{1n} \ (id_1 =)? expr_{21} \dots id_i = expr_{2i} // \text{hvd.size}() \dots (id_k =)? expr_{2k}) \\
& \quad \quad \text{ELSE} \\
& \quad \quad \quad expr_1 (expr_{11} // \text{hvd.size}() \dots expr_{1n} \ (id_1 =)? expr_{21} \dots (id_k =)? expr_{2k}) \\
& \quad \text{IF } \sigma(\text{“tensor_flow”}) = id_t \text{ AND } expr_1 = id_t.\text{keras.datasets.mnist.load_data} \text{ THEN} \\
& \quad \text{ELSE} \\
& \quad \quad trans_E \llbracket expr_1 \rrbracket(\sigma) (trans_E \llbracket expr_{11} \rrbracket(\sigma) \dots trans_E \llbracket expr_{1n} \rrbracket(\sigma) \\
& \quad \quad \quad (id_1 =)? trans_E \llbracket expr_{21} \rrbracket(\sigma) \dots (id_k =)? trans_E \llbracket expr_{2k} \rrbracket(\sigma)) \\
\\
& 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).\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}
\end{aligned}$$

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

$trans_A \llbracket id_1 \text{ as } id_2 \rrbracket(\sigma) =$
LET $id_1 = \text{tensorflow}$ **THEN** $\sigma[\text{“tensorflow”} \mapsto id_2]$
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**
 $(with_item_1' :: with_item_2' :: \dots :: [with_item_n'], \sigma_n)$

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

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

$trans_W \llbracket expr_1 \text{ as } expr_2 \rrbracket(\text{smodenv}) =$
IF $\sigma[\text{“tensorflow”}] = id_1$ **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 $(trans_E \llbracket expr_1 \rrbracket(\sigma) \text{ as } expr_2, \sigma)$

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

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

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

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

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

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

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

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

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

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

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

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