Code Transformation for Distributed Python ML Code

1 Identifying trainig loop

1.1 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.
5. Training loop must not be defined conditionally.

1.2 Rules

1.2.1 Summary

$Summary$	::=	`ModuleSummary` s $(SumMap)^*$ tl	(ModuleSummary)
		`FuncSummary` s $(SumMap)^*$ tl	(FuncSummary)
		`ClassSummary` s arg $(SumMap)^*$ tl	(ClassSummary)
$SumMap$	::=	$(s\ Summary)^*$	(SumMap)
arg	::=	???	(Arg)
tl	::=	$GradTape \mid Optimizer \mid NoTl$	(TrainingLoop)

1.2.2 Training Loop

$summary_M$: $Module \rightarrow ModuleSummary$

$summary_M[\![\ s\ \ stmt^*\ \ type\_ignore\ ]\!] =$
  LET $summary\_list, tl = summary_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)(\gamma)$ **IN**
  `ModuleSummary` s $summary\_list$ tl

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

$summary_S[\![\ stmt_1\ \ stmt_2\ ...\ stmt_n\ ]\!](\sigma)(\gamma) =$
  LET $summary_1, tl_1 = trans_S[\![\ stmt_1\ ]\!](\sigma)$ **IN**
  LET $summary_2, tl_2 = trans_S[\![\ stmt_2\ ]\!](\sigma)$ **IN**
  ...
  LET $summary_n, tl_n = trans_S[\![\ stmt_n\ ]\!](\sigma)$ **IN**
  LET $\gamma = summary_1\ summary_2\ \ldots\ summary_n$ **IN**
  γ, ???

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

$summary_S[\![\ (@expr_1)^*$ `def` id `(`$args$`)` $(\texttt{->}\ expr_2)?$: $(\texttt{\#type:}s)?\ stmt^*\ ]\!](\sigma)(\gamma) =$
  LET $\gamma', tl = summary_{\overline{S}}[\![\ stmt^*\ ]\!]$ **IN**
  `FuncSummary` id.name γ' tl, ???, ???

$summary_S[\![\ (@expr_1)^*$ `async def` id `(`$args$`)` $(\texttt{->}\ expr_2)?$: $(\texttt{\#type:}s)?\ stmt^*\ ]\!](\sigma)(\gamma) =$
  LET $\gamma', tl = summary_{\overline{S}}[\![\ stmt^*\ ]\!]$ **IN**
  `FuncSummary` id.name γ' tl, ???, ???

$summary_S[\![\ (@expr_1)^*$ `class` id `(`$expr_2^*\ keyword^*$`)` : $stmt^*\ ]\!](\sigma)(\gamma) =$
  LET $arg = summary_{\overline{A}}[\![\ expr_2^*\ keyword^*\ ]\!](\sigma)(\gamma)$ **IN**
  LET $\gamma', tl = summary_{\overline{S}}[\![\ stmt^*\ ]\!]$ **IN**
  `ClassSummary` arg id.name γ' tl, ???, ???

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

2 Python Abstract Syntax

module	::=	*stmt** *type_ignore*	(ModuleDef)
stmt	::=	(@*expr*)* `def` *id* `(`*args*`)` (`->` *expr*)? `:` (`#type:`*s*)? *stmt**	(FunDef)
	|	(@*expr*)* `async def` *id* `(`*args*`)` (`->` *expr*)? `:` (`#type:`*s*)? *stmt**	(AsyncFunDef)
	|	(@*expr*)* `class` *id* `(`*expr** *keyword**`)` `:` *stmt**	(ClassDef)
	|	`return` *expr*?	(Return)
	|	`delete` *expr**	(Delete)
	|	*expr** `=` *expr* (`#type:`*s*)?	(Assign)
	|	*expr* *binop* `=` *expr*	(AugAssign)
	|	*expr* `:` *expr* (`=` *expr*)?	(AnnAssign)
	|	(`#type:`*s*)? `for` *expr* `in` *expr* `:` *stmt** (`else :` *stmt**)?	(ForLoop)
	|	(`#type:`*s*)? `async for` *expr* `in` *expr* `:` *stmt** (`else :` *stmt**)?	(AsyncForLoop)
	|	`while` `(`*expr*`)` `:` *stmt** (`else :` *stmt**)?	(WhileLoop)
	|	`if` `(`*expr*`)` `:` *stmt** (`else :` *stmt**)?	(If)
	|	(`#type:`*s*)? `with` *with_item** `:` *stmt**	(With)
	|	(`#type:`*s*)? `async with` *with_item** `:` *stmt**	(AsyncWith)
	|	`match` *expr* `:` *match_case**	(Match)
	|	`raise` *expr*? (`from` *expr*)?	(Raise)
	|	`try :` *stmt** *exc_handler** (`else :` *stmt**)? (`finally :` *stmt**)?	(Try)
	|	`assert` *expr* *expr*?	(Assert)
	|	`import` *alias**	(Import)
	|	`from` *i* *id*? `import` *alias**	(ImportFrom)
	|	`global` *id**	(Global)
	|	`nonlocal` *id**	(NonLocal)
	|	*expr*	(ExprStmt)
	|	`pass`	(Pass)
	|	`break`	(Break)
	|	`continue`	(Continue)

expr	::=	*expr boolop expr*	(BoolOp)
	|	*expr* `:=` *expr*	(NamedExpr)
	|	*expr binop expr*	(BinaryOp)
	|	*unop expr*	(UnaryOp)
	|	`lambda` *args* `:` *expr*	(Lambda)
	|	*expr* `if` *expr* `else` *expr*	(IfExpr)
	|	{(*expr* `:` *expr*)* (`**`*expr*)*}	(Dictionary)
	|	{*expr**}	(Set)
	|	`[`*expr**`]`	(List)
	|	`(`*expr**`)`	(Tuple)
	|	{*expr*`:`*expr* *comprehension**}	(DictionaryComp)
	|	{*expr* *comprehension**}	(SetComp)
	|	`[`*expr* *comprehension**`]`	(ListComp)
	|	`(`*expr* *comprehension**`)`	(GeneratorComp)
	|	`await` *expr*	(Await)
	|	`yield` *expr*?	(Yield)
	|	`yield from` *expr*	(YieldFrom)
	|	*expr* (*compop expr*)*	(CompOp)
	|	*expr* `(`*expr** *keyword**`)`	(Call)
	|	{*expr* `(!`*i*`)?` `(:`*expr*`)?`}	(FormattedValue
	|	*expr**	(JoinedStr)
	|	*constant*	(Constant)
	|	*expr*`.`*id*	(Attribute)
	|	*expr*`[`*expr*`]`	(Subscript)
	|	`*`*expr*	(Starred)
	|	`**`*expr*	(DoubleStarred)
	|	*id*	(Name)
	|	*expr*? (`:`*expr*)? (`:`*expr*)?	(Slice)
boolop	::=	`and` | `or`	(BoolOperator)
binop	::=	`+` | `-` | `*` | `@` | `/` | `**` | `<<` | `>>` | `|` | `^` | `&` | `//` | `%`	(BinOperator)
unop	::=	∼ | `not` | `+` | `-`	(UnOperator)
compop	::=	`==` | `!=` | `<` | `<=` | `>` | `>=` | `is` | `is not` | `in` | `not in`	(CompOperator)
comprehension	::=	`for` *expr* `in` *expr* (`if` *expr*)*	(Comprehension)
	|	`async for` *expr* `in` *expr* (`if` *expr*)*	(AsyncComprehen
exc_handler	::=	`except` *expr*? (`as` *id*)? `:` *stmt**	(ExcHandler)
args	::=	(*arg* (= *expr*)?)*, (*arg* (= *expr*)?)*, *arg*?, (*arg* (= *expr*)?)*, *arg*?	(Arguments)
arg	::=	*id expr*? *s*?	(Argument)
keyword	::=	*id*? `=` *expr*	(Keyword)
alias	::=	*id* (.*id*)* (`as` *id*)?	(Alias)
with_item	::=	*expr* (`as` *expr*)?	(WithItem)

match_case	::=	`case` *pattern* (`if` *expr*)? `:` *stmt**	(MatchCase)
pattern	::=	*expr*	(MatchValue)
	|	*constant*	(MatchSingleton)
	|	`[`*pattern**`]`	(MatchSequence)
	|	`*`(*id*)?	(MatchStar)
	|	{(*expr* `:` *pattern*)* *id*`?`}	(MatchMapping)
	|	*expr* `(`*pattern** (*id* `=` *pattern*)*`)`	(MatchClass)
	|	(*pattern* `as`)? *id*	(MatchAs)
	|	*pattern* `|` *pattern*	(MatchOr)
	|	_	(MatchWildcard)
constant	::=	`None`	(NoneLiteral)
	|	*i*	(IntLiteral)
	|	*f*	(FloatLiteral)
	|	*c*	(ComplexLiteral)
	|	*s*	(StringLiteral)
	|	*b*	(BooleanLiteral)
	|	`(`*constant**`)`	(TupleLiteral)
	|	`...`	(Ellipsis)
type_ignore	::=	*i**	(TypeIgnore)
id	$\in$	*Id*	
s	$\in$	*Str*	
b	$\in$	{`True`, `False`}	
i	$\in$	$\mathbb{Z}$	
f	$\in$	$\mathbb{R}$	
c	$\in$	$\mathbb{C}$	

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.

3 Transformation for TF2 Python Code

3.1 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.

3.2 Rules

3.2.1 Types and Auxiliary Functions

τ	::=	$Module$	(Modules)
		$Stmt$	(Statements)
		$Expr$	(Expressions)
		$Comprehension$	(Comprehensions)
		$ExcHandler$	(Exception handlers)
		$Alias$	(Aliases)
		$WithItem$	(Withitems)
		$Pattern$	(Patterns)
		Id	(Identifiers)
		Str	(Strings)
		τ list	(List of elements typed τ)

$\sigma \in \Sigma = Str \xrightarrow{\text{fin}} Id$ Environment storing mappings from strings to identifiers

._1	:	$\tau \times \tau \to \tau$	Get the first element of the given pair
@	:	τ list $\to \tau$ list $\to \tau$ list	Concatenate two lists
::	:	$\tau \to \tau$ list $\to \tau$ list	Append an element to a list (right-associative)

3.2.2 Transformation Rules

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

$trans_M[\![\ stmt^*\ \ type\_ignore\ ]\!] \;=\; trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)\textbf{.\_1}\ \ type\_ignore$

$\boxed{trans_{\overline{S}} \;:\; Stmt\ \textbf{list} \;\rightarrow\; \Sigma \;\rightarrow\; (Stmt\ \textbf{list} \;\times\; \Sigma)}$

$$
\begin{aligned}
trans_{\overline{S}}[\![\ stmt_1\ \ stmt_2\ ...\ stmt_n\ ]\!](\sigma) \;=\;& \textbf{LET}\ \ stmt_1^{*\prime}, \sigma_1 \;=\; trans_S[\![\ stmt_1\ ]\!](\sigma)\ \ \textbf{IN}\\
& \textbf{LET}\ \ stmt_2^{*\prime}, \sigma_2 \;=\; trans_S[\![\ stmt_2\ ]\!](\sigma_1)\ \ \textbf{IN}\\
& \ldots\\
& \textbf{LET}\ \ stmt_n^{*\prime}, \sigma_n \;=\; trans_S[\![\ stmt_n\ ]\!](\sigma_{n-1})\ \ \textbf{IN}\\
& (stmt_1^{*\prime}\ \textbf{@}\ stmt_2^{*\prime}\ \textbf{@}\ ...\ \textbf{@}\ stmt_n^{*\prime}, \sigma_n)
\end{aligned}
$$

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

$trans_S[\![\ (@expr_1)^*\ \ \texttt{def}\ \ id\ \ \texttt{(}args\texttt{)}\ \ (\texttt{->}\ expr_2)?\ \ :\ \ (\texttt{\#type:}s)?\ \ stmt^*\ ]\!](\sigma) =$
$\quad([(@expr_1)^*\ \ \texttt{def}\ \ id\ \ \texttt{(}args\texttt{)}\ \ (\texttt{->}\ expr_2)?\ \ :\ \ (\texttt{\#type:}s)?\ \ trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)\textbf{.\_1}], \sigma)$

$trans_S[\![\ (@expr_1)^*\ \ \texttt{async def}\ \ id\ \ \texttt{(}args\texttt{)}\ \ (\texttt{->}\ expr_2)?\ \ :\ \ (\texttt{\#type:}s)?\ \ stmt^*\ ]\!](\sigma) =$
$\quad([(@expr_1)^*\ \ \texttt{async def}\ \ id\ \ \texttt{(}args\texttt{)}\ \ (\texttt{->}\ expr_2)?\ \ :\ \ (\texttt{\#type:}s)?\ \ trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)\textbf{.\_1}], \sigma)$

$trans_S[\![\ (@expr_1)^*\ \ \texttt{class}\ \ id\ \ \texttt{(}expr_2^*\ keyword^*\texttt{)}\ \ :\ \ stmt^*\ ]\!](\sigma) =$
$\quad([(@expr_1)^*\ \ \texttt{class}\ \ id\ \ \texttt{(}expr_2^*\ keyword^*\texttt{)}\ \ :\ \ trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)\textbf{.\_1}], \sigma) =$

$trans_S[\![\ \texttt{return}\ \ expr?\ ]\!](\sigma) = ([\texttt{return}\ \ (trans_E[\![\ expr\ ]\!](\sigma))?], \sigma)$

$trans_S[\![\ \texttt{delete}\ \ expr^*\ ]\!](\sigma) = ([\texttt{delete}\ \ expr^*], \sigma)$

A strict form of assignment statements

$trans_S[\![\ id_r$ = $expr_1$ (*expr*$_{11}$ ... *expr*$_{1n}$ (id_1 =)? *expr*$_{21}$ ... (id_k =)? *expr*$_{2k}$) (#type:s)? $]\!](\sigma) =$

IF σ(**"tensor_flow"**) = id_t **AND** $expr_1$ = id_t.data.Dataset.$expr_3$ **THEN**
([id_r = $expr_1$ (*expr*$_{11}$ ... *expr*$_{1n}$ (id_1 =)? *expr*$_{21}$ ... (id_k =)? *expr*$_{2k}$) (#type:s)?],
σ[**"dataset"**$\mapsto$ id_r])
ELIF σ(**"tensor_flow"**) = id_t **AND** $expr_1$ = id_t.train.Checkpoint **THEN**
([id_r = $expr_1$ (*expr*$_{11}$ ... *expr*$_{1n}$ (id_1 =)? *expr*$_{21}$ ... (id_k =)? *expr*$_{2k}$) (#type:s)?],
σ[**"checkpoint"**$\mapsto$ id_r])
ELIF σ(**"tensor_flow"**) = id_t **AND** $expr_1$ = id_t.optimizers.Adam **THEN**
IF id_i = learning_rate **WHEN** 1 ≤ i ≤ k **THEN**
([id_r = $expr_1$ (*expr*$_{11}$ ... *expr*$_{1n}$ (id_1 =)? *expr*$_{21}$ ... id_i = *expr*$_{2i}$ * hvd.size()
... (id_k =)? *expr*$_{2k}$) (#type:s)?], σ[**"optimizer"**$\mapsto$ id_r])
ELSE
([id_r = $expr_1$ (*expr*$_{11}$ * hvd.size() ... *expr*$_{1n}$ (id_1 =)? *expr*$_{21}$ ... (id_k =)? *expr*$_{2k}$) (#type:s)?],
σ[**"optimizer"**$\mapsto$ id_r])
ELIF σ(**"optimizer"**) = id_t **AND** $expr_1$ = id_t.apply_gradients **THEN**
IF id_i = grads_and_vars **WHEN** 1 ≤ i ≤ k **THEN**
LET id_z = **NewID()** **IN**
([id_z = *expr*$_{2i}$,
id_r = $expr_1$ (*expr*$_{11}$ ... *expr*$_{1n}$ (id_1 =)? *expr*$_{21}$ ... id_i = id_z ... (id_k =)? *expr*$_{2k}$) (#type:s)?,
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]], σ)
ELSE
LET id_z = **NewID()** **IN**
([id_z = *expr*$_{11}$,
id_r = $expr_1$ (id_z *expr*$_{12}$ ... *expr*$_{1n}$ (id_1 =)? *expr*$_{21}$ ... (id_k =)? *expr*$_{2k}$) (#type:s)?,
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]], σ)
ELIF σ(**"checkpoint"**) = id_t **AND** $expr_1$ = id_t.save **THEN**
([if hvd.rank() == 0: [id_r = $expr_1$ (*expr*$_{11}$ ... *expr*$_{1n}$ (id_1 =)? *expr*$_{21}$ ... (id_k =)? *expr*$_{2k}$) (#type:s)?]],
σ)
ELIF σ(**"keras"**) = id_k **AND** $expr_1$ = id_k.models.Sequential **THEN**
([id_r = $expr_1$ (*expr*$_{11}$ ... *expr*$_{1n}$ (id_2 =)? *expr*$_{21}$ ... (id_k =)? *expr*$_{2k}$)], σ[**"model"**$\mapsto$ id_k])
ELSE ([id_r = $trans_E[\![\ expr_1$ (*expr*$_{11}$ ... *expr*$_{1n}$ (id_2 =)? *expr*$_{21}$ ... (id_k =)? *expr*$_{2k}$) $]\!](\sigma)$ (#type:s)?],
σ)

A general form of assignment statements except for the strict form

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

$trans_S[\![\ expr_1\ binop$ = $expr_2\ ]\!](\sigma) = ([expr_1\ binop$ = $trans_E[\![\ expr_2\ ]\!](\sigma)]$, σ)

$trans_S[\![\ expr_1\ \texttt{:}\ expr_2\ (\texttt{=}\ expr_3)?\ ]\!](\sigma) = ([expr_1\ \texttt{:}\ expr_2\ (\texttt{=}\ trans_E[\![\ expr_3\ ]\!](\sigma))?], \sigma)$
IF $expr_1\ =\ id_1$ **AND** $\sigma(\textbf{“tensor\_flow”})\ =\ id_2$
AND $expr_3\ =\ id_2\texttt{.data.Dataset.}expr_4\ \texttt{(}expr_5^*\ keyword^*\texttt{)}$ **THEN**
$([expr_1\ \texttt{:}\ expr_2\ \texttt{=}\ expr_3\ (\texttt{\#type:}s)?], \sigma[\textbf{“dataset”} \mapsto id_1])$
ELSE $([expr_1\ \texttt{:}\ expr_2\ (\texttt{=}\ trans_E[\![\ expr_3\ ]\!](\sigma))?], \sigma)$

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

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

$trans_S[\![\ \texttt{while}\ \texttt{(}expr\texttt{)}\ \texttt{:}\ stmt_1^*\ (\texttt{else}\ \texttt{:}\ stmt_2^*)?\ ]\!](\sigma) =$
$([\texttt{while}\ (trans_E[\![\ expr\ ]\!](\sigma))\ \texttt{:}\ trans_{\overline{S}}[\![\ stmt_1^*\ ]\!](\sigma)\texttt{.\_}\textbf{1}\ (\texttt{else}\ \texttt{:}\ trans_{\overline{S}}[\![\ stmt_2^*\ ]\!](\sigma)\texttt{.\_}\textbf{1})?], \sigma)$

$trans_S[\![\ \texttt{if}\ \texttt{(}expr\texttt{)}\ \texttt{:}\ stmt^*\ (\texttt{else}\ \texttt{:}\ stmt^*)?\ ]\!](\sigma) =$
$([\texttt{if}\ (trans_E[\![\ expr\ ]\!](\sigma))\ \texttt{:}\ trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)\texttt{.\_}\textbf{1}\ (\texttt{else}\ \texttt{:}\ trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)\texttt{.\_}\textbf{1})?], \sigma)$

$trans_S[\![\ (\texttt{\#type:}s)?\ \texttt{with}\ with\_item^*\ \texttt{:}\ stmt^*\ ]\!](\sigma) =$
LET $with\_item^{*\prime}, \sigma_1 = trans_{\overline{W}}[\![\ with\_item^*\ ]\!](\sigma)$ **IN**
LET $stmt^{*\prime}, \sigma_2 = trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma_1)$ **IN**
IF $\sigma_1 \setminus \sigma\ =\ [\textbf{“gradient\_tape”} \mapsto id]$ **THEN**
$([(\texttt{\#type:}s)?\ \texttt{with}\ with\_item^{*\prime}\ \texttt{:}\ stmt^{*\prime},$
$id\ \texttt{=}\ \texttt{hvd.DistributedGradientTape(}id\texttt{)}], \sigma_2)$
ELSE $([(\texttt{\#type:}s)?\ \texttt{with}\ with\_item^{*\prime}\ \texttt{:}\ stmt^{*\prime}], \sigma_2)$

$trans_S[\![\ (\texttt{\#type:}s)?\ \texttt{async with}\ with\_item^*\ \texttt{:}\ stmt^*\ ]\!](\sigma) =$
LET $with\_item^{*\prime}, \sigma_1 = trans_{\overline{W}}[\![\ with\_item^*\ ]\!](\sigma)$ **IN**
LET $stmt^{*\prime}, \sigma_2 = trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma_1)$ **IN**
IF $\sigma_1 \setminus \sigma\ =\ [\textbf{“gradient\_tape”} \mapsto id]$ **THEN**
$([(\texttt{\#type:}s)?\ \texttt{async with}\ with\_item^{*\prime}\ \texttt{:}\ stmt^{*\prime},$
$id\ \texttt{=}\ \texttt{hvd.DistributedGradientTape(}id\texttt{)}], \sigma_2)$
ELSE $([(\texttt{\#type:}s)?\ \texttt{async with}\ with\_item^{*\prime}\ \texttt{:}\ stmt^{*\prime}], \sigma_2)$

$trans_S[\![\ \texttt{match}\ expr\ \texttt{:}\ match\_case^*\ ]\!](\sigma) =$
$([\texttt{match}\ trans_E[\![\ expr\ ]\!](\sigma)\ \texttt{:}\ (trans_C[\![match\_case]\!](\sigma))^*], \sigma)$

$trans_S[\![\ \texttt{raise}\ expr_1?\ \ (\texttt{from}\ expr_2)?\ ]\!](\sigma) = ([\texttt{raise}\ expr_1?\ \ (\texttt{from}\ expr_2)?], \sigma)$

$trans_S[\![\ \texttt{try}\ \texttt{:}\ stmt_1^*\ \ exc\_handler^*\ \ (\texttt{else}\ \texttt{:}\ stmt_2^*)?\ \ (\texttt{finally}\ \texttt{:}\ stmt_3^*)?\ ]\!](\sigma) =$
$([\texttt{try}\ \texttt{:}\ trans_{\overline{S}}[\![\ stmt_1^*\ ]\!](\sigma)\texttt{.\_}\textbf{1}\ \ (trans_H[\![exc\_handler]\!](\sigma))^*$
$(\texttt{else}\ \texttt{:}\ trans_{\overline{S}}[\![\ stmt_2^*\ ]\!](\sigma)\texttt{.\_}\textbf{1})?\ \ (\texttt{finally}\ \texttt{:}\ trans_{\overline{S}}[\![\ stmt_3^*\ ]\!](\sigma)\texttt{.\_}\textbf{1})?], \sigma)$

$trans_S[\![\ \texttt{assert}\ \ expr_1\ \ expr_2?\ ]\!](\sigma) = ([\texttt{assert}\ \ trans_E[\![\ expr_1\ ]\!](\sigma)\ \ expr_2?], \sigma)$

$trans_S[\![$ `import` $alias^*$ $]\!](\sigma) =$
LET $\sigma_1 = trans_{\overline{A}}[\![$ $alias^*$ $]\!](\sigma)$ **IN**
IF $\sigma_1 \setminus \sigma = [$**"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)

$trans_S[\![$ `from` i $id?$ `import` $alias^*$ $]\!](\sigma) =$
LET $\sigma_1 = trans_{\overline{A}}[\![$ $alias^*$ $]\!](\sigma)$ **IN**
IF $id =$ `tensorflow` **AND** $\sigma_1 \setminus \sigma = [$**"keras"**$\mapsto id]$ **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')`], σ_1)
ELSE ([`from` i $id?$ `import` $alias^*$], σ)

$trans_S[\![$ `global` id^* $]\!](\sigma) =$ ([`global` id^*], σ)

$trans_S[\![$ `nonlocal` id^* $]\!](\sigma) =$ ([`nonlocal` id^*], σ)

A strict form of expr statements

$trans_S[\![\ expr_1\ \texttt{(}expr_{11}\ ...\ expr_{1n}\quad (id_1\ \texttt{=}\ )?\ expr_{21}\ ...\ (id_k\ \texttt{=}\ )?\ expr_{2k}\texttt{)}\ ]\!](\sigma) =$

IF $\sigma($**"optimizer"**$)\ =\ id_t$ **AND** $expr_1\ =\ id_t$`.apply_gradients` **THEN**

IF $id_i\ =$ `grads_and_vars` **WHEN** $1 \leq \mathrm{i} \leq \mathrm{k}$ **THEN**

LET $id_z\ =$ **NewID() IN**

([$id_z\ \texttt{=}\ expr_{2i}$,

$expr_1$ `(`$expr_{11}\ ...\ expr_{1n}\quad (id_1$ `=` `)?` $expr_{21}\ ...\ id_i$ `=` $id_z\ ...\ (id_k$ `=` `)?` $expr_{2k}$`)`,

`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 ]]`, σ)

ELSE

LET $id_z\ =$ **NewID() IN**

([$id_z\ \texttt{=}\ expr_{11}$,

$expr_1$ `(`$id_z\ expr_{12}\ ...\ expr_{1n}\quad (id_1$ `=` `)?` $expr_{21}\ ...\ (id_k$ `=` `)?` $expr_{2k}$`)`,

`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 ]]`, σ)

ELIF $expr_1\ =$ `print` **THEN**

([`if hvd.rank() == 0:`,

[$expr_1$ `(`$expr_{11}\ ...\ expr_{1n}\quad (id_1$ `=` `)?` $expr_{21}\ ...\ (id_k$ `=` `)?` $expr_{2k}$`)`]], σ)

ELIF $\sigma($**"checkpoint"**$)\ =\ id_t$ **AND** $expr_1\ =\ id_t$`.save` **THEN**

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

ELIF $\sigma($**"model"**$)\ =\ id_t$ **AND** $expr_1\ =\ id_t$`.fit` **THEN**

IF $id_i\ =$ `callbacks` **WHEN** $1 \leq \mathrm{i} \leq \mathrm{k}$ **THEN**

([$expr_1$ `(`$expr_{11}\ ...\ expr_{1n}\quad (id_1$ `=` `)?` $expr_{21}\ ...$

id_i `=` $expr_{2i}$ `+ [hvd.callbacks.BroadcastGlobalVariablesCallback(0)]`

... (id_k `=` `)?` $expr_{2k}$`)`], σ)

ELSE

([$expr_1$ `(`$expr_{11}\ ...\ expr_{1n}\quad (id_1$ `=` `)?` $expr_{21}\ ...\ (id_k$ `=` `)?` $expr_{2k}$

`callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(0)])`], σ)

ELSE

[$trans_E[\![\ expr_1\ \texttt{(}expr_{11}\ ...\ expr_{1n}\quad (id_1\ \texttt{=}\ )?\ expr_{21}\ ...\ (id_k\ \texttt{=}\ )?\ expr_{2k}\texttt{)}\ ]\!](\sigma)$], σ)

A general form of expr statements except for the strict form

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

$trans_S[\![$ `pass` $]\!](\sigma) = ([$`pass`$], \sigma)$

$trans_S[\![$ `break` $]\!](\sigma) = ([$`break`$], \sigma)$

$trans_S[\![$ `continue` $]\!](\sigma) = ([$`continue`$], \sigma)$

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

$trans_E[\![\ expr_1 \;\; boolop \;\; expr_2\ ]\!](\sigma) = trans_E[\![\ expr_1\ ]\!](\sigma) \;\; boolop \;\; trans_E[\![\ expr_2\ ]\!](\sigma)$

$trans_E[\![\ expr_1 \;\; \texttt{:=} \;\; expr_2\ ]\!](\sigma) = expr_1 \;\; \texttt{:=} \;\; trans_E[\![\ expr_2\ ]\!](\sigma)$

$trans_E[\![\ expr_1 \;\; binop \;\; expr_2\ ]\!](\sigma) = trans_E[\![\ expr_1\ ]\!](\sigma) \;\; binop \;\; trans_E[\![\ expr_2\ ]\!](\sigma)$

$trans_E[\![\ unop \;\; expr\ ]\!](\sigma) = unop \;\; trans_E[\![\ expr\ ]\!](\sigma)$

$trans_E[\![\ \texttt{lambda} \;\; args \;\; \texttt{:} \;\; expr\ ]\!](\sigma) = \texttt{lambda} \;\; args \;\; \texttt{:} \;\; trans_E[\![\ expr\ ]\!](\sigma)$

$trans_E[\![\ expr_1 \;\; \texttt{if} \;\; expr_2 \;\; \texttt{else} \;\; expr_3\ ]\!](\sigma) =$
$\quad trans_E[\![\ expr_1\ ]\!](\sigma) \;\; \texttt{if} \;\; trans_E[\![\ expr_2\ ]\!](\sigma) \;\; \texttt{else} \;\; trans_E[\![\ expr_3\ ]\!](\sigma)$

$trans_E[\![\ expr_1 \;\; \texttt{if} \;\; expr_2 \;\; \texttt{else} \;\; expr_3\ ]\!](\sigma) =$
$\quad trans_E[\![\ expr_1\ ]\!](\sigma) \;\; \texttt{if} \;\; trans_E[\![\ expr_2\ ]\!](\sigma) \;\; \texttt{else} \;\; trans_E[\![\ expr_3\ ]\!](\sigma)$

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

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

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

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

$trans_E[\![\ \{expr_1\texttt{:}expr_2 \;\; comprehension^*\}\ ]\!](\sigma) = \{expr_1\texttt{:}trans_E[\![\ expr_2\ ]\!]\texttt{(}\sigma\texttt{)} \;\; (trans_O[\![comprehension]\!](\sigma))^*\}$

$trans_E[\![\ \{expr \;\; comprehension^*\}\ ]\!](\sigma) = \{trans_E[\![\ expr\ ]\!]\texttt{(}\sigma\texttt{)} \;\; (trans_O[\![comprehension]\!](\sigma))^*\}$

$trans_E[\![\ \texttt{[}expr \;\; comprehension^*\texttt{]}\ ]\!](\sigma) = \texttt{[}trans_E[\![\ expr\ ]\!]\texttt{(}\sigma\texttt{)} \;\; (trans_O[\![comprehension]\!](\sigma))^*\texttt{]}$

$trans_E[\![\ \texttt{(}expr \;\; comprehension^*\texttt{)}\ ]\!](\sigma) = \texttt{(}trans_E[\![\ expr\ ]\!](\sigma) \;\; (trans_O[\![comprehension]\!](\sigma))^*\texttt{)}$

$trans_E[\![\ \texttt{await} \;\; expr\ ]\!](\sigma) = \texttt{await} \;\; trans_E[\![\ expr\ ]\!](\sigma)$

$trans_E[\![\ \texttt{yield} \;\; expr?\ ]\!](\sigma) = \texttt{yield} \;\; (trans_E[\![\ expr\ ]\!](\sigma))?$

$trans_E[\![\ \texttt{yield from} \;\; expr\ ]\!](\sigma) = \texttt{yield from} \;\; trans_E[\![\ expr\ ]\!](\sigma)$

$trans_E[\![\ expr_1\ (compop\ expr_2)^*\ ]\!](\sigma) = trans_E[\![\ expr_1\ ]\!](\sigma)\ (compop\ trans_E[\![expr_2]\!](\sigma))^*$

$trans_E[\![\ expr_1\ \texttt{(}expr_{11}\ ...\ expr_{1n}\quad (id_1\ \texttt{=}\ )?\ expr_{21}\ ...\ (id_k\ \texttt{=}\ )?\ expr_{2k}\texttt{)}\ ]\!](\sigma) =$
IF $\sigma($**"dataset"**$)\ =\ id_t$ **AND** $expr_1\ =\ id_t$`.take` **THEN**
IF $id_i\ =$ `count` **WHEN** $1 \leq \mathrm{i} \leq \mathrm{k}$ **THEN**
$expr_1$ `(`$expr_{11}$ ... $expr_{1n}$ $(id_1$ `=` $)?$ $expr_{21}$ ... id_i `=` $expr_{2i}$ `//` `hvd.size()` ... $(id_k$ `=` $)?$ $expr_{2k}$`)`
ELSE
$expr_1$ `(`$expr_{11}$ `//` `hvd.size()` ... $expr_{1n}$ $(id_1$ `=` $)?$ $expr_{21}$ ... $(id_k$ `=` $)?$ $expr_{2k}$`)`
ELSE
$trans_E[\![\ expr_1\ ]\!](\sigma)$ `(`$trans_E[\![\ expr_{11}\ ]\!](\sigma)$ ... $trans_E[\![\ expr_{1n}\ ]\!](\sigma)$
$(id_1$ `=` $)?\ trans_E[\![\ expr_{21}\ ]\!](\sigma)$ ... $(id_k$ `=` $)?\ trans_E[\![\ expr_{2k}\ ]\!](\sigma)$`)`

$trans_E[\![\ \{expr_1\quad$ `(!`i`)?` `(:`$expr_2$`)?}` $]\!](\sigma) = \{expr_1\quad$ `(!`i`)?` `(:`$expr_2$`)?}`

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

$trans_E[\![\ constant\ ]\!](\sigma) = constant$

$trans_E[\![\ expr.id\ ]\!](\sigma) = (trans_E[\![\ expr\ ]\!](\sigma)).id$

$trans_E[\![\ expr_1$`[`$expr_2$`]` $]\!](\sigma) = trans_E[\![\ expr_1\ ]\!](\sigma)$`[`$trans_E[\![\ expr_2\ ]\!](\sigma)$`]`

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

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

$trans_E[\![\ id\ ]\!](\sigma) = id$

$trans_E[\![\ expr_1?\ (:expr_2)?\ (:expr_3)?\ ]\!](\sigma) = trans_E[\![\ expr_1\ ]\!](\sigma)?\ (:trans_E[\![\ expr_2\ ]\!](\sigma))?\ (:trans_E[\![\ expr_3\ ]\!](\sigma))?$

$trans_O$ `:` $Comprehension \rightarrow \Sigma \rightarrow Comprehension$
$trans_O[\![$ `for` $expr_1$ `in` $expr_2$ `(if` $expr_3)^*\ ]\!](\sigma) =$ `for` $expr_1$ `in` $trans_E[\![\ expr_2\ ]\!](\sigma)$ `(if` $trans_E[\![expr_3]\!](\sigma))^*$

$trans_H$ `:` $ExcHandler \rightarrow \Sigma \rightarrow ExcHandler$
$trans_H[\![$ `except` $expr?$ `(as` $id)?$ `:` $stmt^*\ ]\!](\sigma) =$ `except` $expr?$ `(as` $id)?$ `:` $trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)$**._1**

$trans_{\overline{A}}$ `:` $Alias$ **list** $\rightarrow \Sigma \rightarrow \Sigma$
$trans_{\overline{A}}[\![\ alias_1\ ...\ alias_{n-1}\ alias_n\ ]\!](\sigma) = trans_A[\![\ alias_n\ ]\!](trans_A[\![\ alias_{n-1}\ ]\!](...\ trans_A[\![\ alias_1\ ]\!](\sigma)))$

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

$trans_A[\![\ id\ ]\!](\sigma) =$
  LET id = `tensorflow` **THEN** σ[**"tensor_flow"**$\mapsto id$]
  LET id = `keras` **THEN** σ[**"keras"**$\mapsto id$]
  ELSE σ

$trans_A[\![\ id_1$ `as` $id_2\ ]\!](\sigma) =$
  LET id_1 = `tensorflow` **THEN** σ[**"tensor_flow"**$\mapsto id_2$]
  LET id_1 = `keras` **THEN** σ[**"tensor_flow"**$\mapsto id_2$]
  ELSE σ

$trans_A[\![\ id_1\ .id_2\ (.id_3)^*\ (\texttt{as}\ \ id_2)?\ ]\!](\sigma) = \sigma$

$\boxed{trans_{\overline{W}}\ :\ WithItem\ \textbf{list}\ \rightarrow\ \Sigma\ \rightarrow\ (WithItem\ \textbf{list}\ \times\ \Sigma)}$
$trans_{\overline{W}}[\![\ with\_item_1\ with\_item_2\ ...\ with\_item_n\ ]\!](\sigma) =$
  LET $with\_item_1'$, σ_1 = $trans_W[\![\ with\_item_1\ ]\!](\sigma)$ **IN**
  LET $with\_item_2'$, σ_2 = $trans_W[\![\ with\_item_2\ ]\!](\sigma_1)$ **IN**
  ...
  LET $with\_item_n'$, σ_n = $trans_W[\![\ with\_item_n\ ]\!](\sigma_{n-1})$ **IN**
  ($with\_item_1'$ **::** $with\_item_2'$ **::** ... **::** [$with\_item_n'$], σ_n)

$\boxed{trans_W\ :\ WithItem\ \rightarrow\ \Sigma\ \rightarrow\ (WithItem\ \times\ \Sigma)}$
$trans_W[\![\ expr\ ]\!](\sigma) = (trans_E[\![\ expr\ ]\!](\sigma), \sigma)$

$trans_W[\![\ expr_1$ `as` $expr_2\ ]\!]$(smodenv) =
  IF σ(**"tensor_flow"**) = id_1 **AND** $expr_1$ = id_1`.GradientTape()` **AND** $expr_2$ = id_2 **THEN**
    ($expr_1$ `as` $expr_2$, σ[**"gradient_tape"**$\mapsto id_2$])
  ELSE ($trans_E[\![\ expr_1\ ]\!](\sigma)$ `as` $expr_2$, σ)

$\boxed{trans_C\ :\ MatchCase\ \rightarrow\ \Sigma\ \rightarrow\ MatchCase}$
$trans_C[\![$ `case` $pattern$ (`if` $expr$)? : $stmt^*\ ]\!](\sigma) =$
  `case` $trans_P[\![\ pattern\ ]\!](\sigma)$ (`if` $trans_E[\![\ expr\ ]\!](\sigma)$)? : $trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)$**._1**

$\boxed{trans_P\ :\ Pattern\ \rightarrow\ \Sigma\ \rightarrow\ Pattern}$
$trans_P[\![\ expr\ ]\!](\sigma) = trans_E[\![\ expr\ ]\!](\sigma)$

$trans_P[\![\ constant\ ]\!](\sigma) = constant$

$trans_P[\![$ `[`$pattern^*$`]` $]\!](\sigma) =$ `[`$trans_P[\![pattern]\!](\sigma)^*$`]`

$trans_P[\![$ `*`$(id)?\ ]\!](\sigma) =$ `*`$(id)?$

$trans_P[\![\ \{(expr\ :\ pattern)^*\ \ id?\}\ ]\!](\sigma) = \{(expr\ :\ trans_P[\![pattern]\!](\sigma))^*\ \ id?\}$

$trans_P[\![\ expr\ \ (pattern_1^*\ (id = pattern_2)^*)\ ]\!](\sigma) = expr\ \ (trans_P[\![pattern_1]\!](\sigma)^*\ (id = trans_P[\![pattern_2]\!](\sigma))^*)$

$trans_P[\![\ (pattern\ \ \texttt{as})?\ id\ ]\!](\sigma) = (trans_P[\![\ pattern\ ]\!](\sigma)\ \texttt{as})?\ id$

$trans_P[\![\ pattern_1\ \texttt{|}\ pattern_2\ ]\!](\sigma) = trans_P[\![\ pattern_1\ ]\!](\sigma)\ \texttt{|}\ trans_P[\![\ pattern_2\ ]\!](\sigma)$

$trans_P[\![\ \_\ ]\!](\sigma) = \_$