Code Transformation for Distributed Python ML Code

1 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.

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	::=	$Module$	(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

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

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

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

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

._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)
.nodes	:	`ClassNode`$\to$ `Nodes` = **._1**	Get nodes of class graph
.vars	:	`ClassNode`$\to$ `Vars` = **._2**	Get variables of class graph

2.2.2 Transformation Rules

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

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

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

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

$\boxed{trans_S \ : \ Stmt \ \to \ \texttt{CG} \ \to \ (Stmt \ \textbf{list} \ \times \ \texttt{CG})}$

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

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

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

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

... (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(**"optimizers"**) = id_t **AND** $expr_1$ = id_t.Adam **THEN**

IF id_i = learning_rate **WHEN** $1 \leq \mathrm{i} \leq \mathrm{k}$ **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** $1 \leq \mathrm{i} \leq \mathrm{k}$ **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_r])

ELIF σ(**"keras"**) = id_t **AND** $expr_1$ = id_t.optimizers.Adam **THEN**

IF id_i = learning_rate **WHEN** $1 \leq \mathrm{i} \leq \mathrm{k}$ **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)?

id_r = hvd.DistributedOptimizer(id_r)], 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)?

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

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

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

A general form of assignment statements except for the strict form
$trans_S[\![\ expr_1^*$ `=` $expr_2$ (`#type:`s)? $]\!](\sigma) = ([expr_1^*$ `=` $trans_E[\![\ expr_2\ ]\!](\sigma)$ (`#type:`s)?$], \sigma)$

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

$trans_S[\![\ expr_1$ `:` $expr_2$ (`=` $expr_3$)? $]\!](\sigma) = ([expr_1$ `:` $expr_2$ (`=` $trans_E[\![\ expr_3\ ]\!](\sigma)$)?$], \sigma)$
IF $expr_1$ $=$ id_1 **AND** $\sigma($**"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)?$], \sigma[$**"dataset"**$\mapsto id_1])$
ELSE $([expr_1$ `:` $expr_2$ (`=` $trans_E[\![\ expr_3\ ]\!](\sigma)$)?$], \sigma)$

$trans_S[\![$ (`#type:`s)? `for` $expr_1$ `in` $expr_2$ `:` $stmt_1^*$ (`else :` $stmt_2^*$)? $]\!](\sigma) =$
$([$(`#type:`s)? `for` $expr_1$ `in` $trans_E[\![\ expr_2\ ]\!](\sigma)$ `:`
$trans_{\overline{S}}[\![\ stmt_1^*\ ]\!](\sigma)$**._1** (`else :` $trans_{\overline{S}}[\![\ stmt_2^*\ ]\!](\sigma)$**._1**)?$], \sigma)$

$trans_S[\![$ (`#type:`s)? `async for` $expr_1$ `in` $expr_2$ `:` $stmt_1^*$ (`else :` $stmt_2^*$)? $]\!](\sigma) =$
$([$(`#type:`s)? `async for` $expr_1$ `in` $trans_E[\![\ expr_2\ ]\!](\sigma)$ `:`
$trans_{\overline{S}}[\![\ stmt_1^*\ ]\!](\sigma)$**._1** (`else :` $trans_{\overline{S}}[\![\ stmt_2^*\ ]\!](\sigma)$**._1**)?$], \sigma)$

$trans_S[\![$ `while (`$expr$`)` `:` $stmt_1^*$ (`else :` $stmt_2^*$)? $]\!](\sigma) =$
$([$`while (`$trans_E[\![\ expr\ ]\!](\sigma)$`)` `:` $trans_{\overline{S}}[\![\ stmt_1^*\ ]\!](\sigma)$**._1** (`else :` $trans_{\overline{S}}[\![\ stmt_2^*\ ]\!](\sigma)$**._1**)?$], \sigma)$

$trans_S[\![$ `if (`$expr$`)` `:` $stmt^*$ (`else :` $stmt^*$)? $]\!](\sigma) =$
$([$`if (`$trans_E[\![\ expr\ ]\!](\sigma)$`)` `:` $trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)$**._1** (`else :` $trans_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)$**._1**)?$], \sigma)$

$trans_S[\![$ (`#type:`s)? `with` $with\_item^*$ `:` $stmt^*\ ]\!](\sigma) =$
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 $\sigma_1 \setminus \sigma =$ [**"gradient_tape"**$\mapsto id$] **THEN**
$([$(`#type:`s)? `with` $with\_item^{*\prime}$ `:` $stmt^{*\prime}$,
id `= hvd.DistributedGradientTape(`id`)`$], \sigma_2)$
ELSE $([$(`#type:`s)? `with` $with\_item^{*\prime}$ `:` $stmt^{*\prime}], \sigma_2)$

$trans_S[\![$ (`#type:`s)? `async with` $with\_item^*$ `:` $stmt^*\ ]\!](\sigma) =$
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 $\sigma_1 \setminus \sigma =$ [**"gradient_tape"**$\mapsto id$] **THEN**
$([$(`#type:`s)? `async with` $with\_item^{*\prime}$ `:` $stmt^{*\prime}$,
id `= hvd.DistributedGradientTape(`id`)`$], \sigma_2)$
ELSE $([$(`#type:`s)? `async with` $with\_item^{*\prime}$ `:` $stmt^{*\prime}], \sigma_2)$

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

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

$trans_S[\![$ `try` `:` $stmt_1^*$ $exc\_handler^*$ $(\texttt{else}\ :\ stmt_2^*)?$ $(\texttt{finally}\ :\ stmt_3^*)?$ $]\!](\sigma) =$
$([\texttt{try}\ :\ trans_{\overline{S}}[\![$ $stmt_1^*$ $]\!](\sigma)\texttt{.\_1}\ \ (trans_H[\![exc\_handler]\!](\sigma))^*$
$(\texttt{else}\ :\ trans_{\overline{S}}[\![$ $stmt_2^*$ $]\!](\sigma)\texttt{.\_1})?\ \ (\texttt{finally}\ :\ trans_{\overline{S}}[\![$ $stmt_3^*$ $]\!](\sigma)\texttt{.\_1})?], \sigma)$

$trans_S[\![$ `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')`$], \sigma_1)$
ELSE $([$`import` $alias^*], \sigma_1)$

$trans_S[\![$ `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')`$], \sigma_1)$
ELSE $([$`from` i $id?$ `import` $alias^*], \sigma_1)$

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

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

A strict form of expr statements

$trans_S[\![$ $expr_1$ `(`$expr_{11}$ ... $expr_{1n}$ $(id_1$ `=` $)?$ $expr_{21}$ ... $(id_k$ `=` $)?$ $expr_{2k}$`)` $]\!](\sigma) =$
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 = $expr_{2i}$,
$expr_1$ ($expr_{11}$ ... $expr_{1n}$ (id_1 =)? $expr_{21}$ ... id_i = id_z ... (id_k =)? $expr_{2k}$),
global hvd_broadcast_done,
if not hvd_broadcast_done: [hvd.broadcast_variables([x[1] for x in id_z], root_rank=0),
hvd.broadcast_variables(id_t.variables(), root_rank=0),
hvd_broadcast_done = True]], σ)

ELSE

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

([id_z = $expr_{11}$,
$expr_1$ (id_z $expr_{12}$ ... $expr_{1n}$ (id_1 =)? $expr_{21}$ ... (id_k =)? $expr_{2k}$),
global hvd_broadcast_done,
if not hvd_broadcast_done: [hvd.broadcast_variables([x[1] for x in id_z], root_rank=0),
hvd.broadcast_variables(id_t.variables(), root_rank=0),
hvd_broadcast_done = True]], σ)

ELIF $expr_1$ = print **OR** $expr_1$ = $expr$.write **OR** $expr_1$ = $expr$.summary **OR** $expr_1$ = $expr$.save_weights **OR** $expr_1$ = $expr$.load_weights **THEN**

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

ELIF σ(**"checkpoint"**) = id_t **AND** $expr_1$ = id_t.save **THEN**

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

ELIF σ(**"model"**) = id_t **AND** $expr_1$ = id_t.fit **THEN**

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

IF id_j = callbacks **WHEN** $1 \leq \mathrm{j} \leq \mathrm{k}$ **THEN**

([callback = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)]
if hvd.rank() == 0: callbacks.append($expr_{2j}$)
$expr_1$ ($expr_{11}$ ... $expr_{1n}$ (id_1 =)? $expr_{21}$ ... id_i = 1 if hvd.rank() == 0 else 0
... id_j = callbacks ... (id_k =)? $expr_{2k}$)], σ)

ELSE

([$expr_1$ ($expr_{11}$ ... $expr_{1n}$ (id_1 =)? $expr_{21}$ ...
id_i = 1 if hvd.rank() == 0 else 0 ... (id_k =)? $expr_{2k}$
callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)])], σ)

ELSE

IF id_j = callbacks **WHEN** $1 \leq \mathrm{j} \leq \mathrm{k}$ **THEN**

([callback = [hvd.callbacks.BroadcastGlobalVariablesCallback(root_rank=0)]
if hvd.rank() == 0: callbacks.append($expr_{2j}$)
$expr_1$ ($expr_{11}$ ... $expr_{1n}$ (id_1 =)? $expr_{21}$ ... id_j = callbacks ... (id_k =)? $expr_{2k}$
verbose = 1 if hvd.rank() == 0 else 0)], σ)

ELSE

([$expr_1$ ($expr_{11}$ ... $expr_{1n}$ (id_1 =)? $expr_{21}$ ... (id_k =)? $expr_{2k}$
verbose = 1 if hvd.rank() == 0 else 0
callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(0)])], σ)

ELIF σ(**"model"**) = id_t **AND** $expr_1$ = id_t.compile **THEN**

IF id_i = optimizer **AND** $expr_{2i}$ = "adam" **WHEN** $1 \leq \mathrm{i} \leq \mathrm{k}$ **THEN**

([id_z = tf.optimizers.Adam(learning_rate=0.001 * hvd.size())
id_z = hvd.DistributedOptimizer(id_z)

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

ELIF $expr_{11}$ `==` `"adam"` **THEN**

([id_z `=` `tf.optimizers.Adam(learning_rate=0.001 * hvd.size())`

id_z `=` `hvd.DistributedOptimizer(`id_z`)`

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

ELSE

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

ELIF $expr_1$ `==` `model.summary` **THEN**

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

ELSE

[$trans_E$⟦ $expr_1$ (`(`$expr_{11}$ ... $expr_{1n}$ `)`? id_1 `=`)? $expr_{21}$ ... (id_k `=`)? $expr_{2k}$`)` ⟧(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 → $Expr$

$trans_E$⟦ $expr_1$ $boolop$ $expr_2$ ⟧(σ) = $trans_E$⟦ $expr_1$ ⟧(σ) $boolop$ $trans_E$⟦ $expr_2$ ⟧(σ)

$trans_E$⟦ $expr_1$ `:=` $expr_2$ ⟧(σ) = $expr_1$ `:=` $trans_E$⟦ $expr_2$ ⟧(σ)

$trans_E$⟦ $expr_1$ $binop$ $expr_2$ ⟧(σ) = $trans_E$⟦ $expr_1$ ⟧(σ) $binop$ $trans_E$⟦ $expr_2$ ⟧(σ)

$trans_E$⟦ $unop$ $expr$ ⟧(σ) = $unop$ $trans_E$⟦ $expr$ ⟧(σ)

$trans_E$⟦ `lambda` $args$: $expr$ ⟧(σ) = `lambda` $args$: $trans_E$⟦ $expr$ ⟧(σ)

$trans_E$⟦ $expr_1$ `if` $expr_2$ `else` $expr_3$ ⟧(σ) =
$trans_E$⟦ $expr_1$ ⟧(σ) `if` $trans_E$⟦ $expr_2$ ⟧(σ) `else` $trans_E$⟦ $expr_3$ ⟧(σ)

$trans_E$⟦ $expr_1$ `if` $expr_2$ `else` $expr_3$ ⟧(σ) =
$trans_E$⟦ $expr_1$ ⟧(σ) `if` $trans_E$⟦ $expr_2$ ⟧(σ) `else` $trans_E$⟦ $expr_3$ ⟧(σ)

$trans_E$⟦ {($expr_1$: $expr_2$)* (**$expr_3$)*} ⟧(σ) = {($expr_1$: $trans_E$⟦$expr_2$⟧(σ))* (**$expr_3$)*}

$trans_E$⟦ {$expr^*$} ⟧(σ) = {($trans_E$⟦$expr$⟧(σ))*}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ELSE

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

ELSE

$trans_E[\![\ expr_1\ ]\!](\sigma)\ \texttt{(}trans_E[\![\ expr_{11}\ ]\!](\sigma)\ ...\ trans_E[\![\ expr_{1n}\ ]\!](\sigma)$

$(id_1\ \texttt{=}\ )?\ trans_E[\![\ expr_{21}\ ]\!](\sigma)\ ...\ (id_k\ \texttt{=}\ )?\ trans_E[\![\ expr_{2k}\ ]\!](\sigma)\texttt{)}$

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

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

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

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

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

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

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

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

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

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

$trans_O[\![\ \texttt{for}\ \ expr_1\ \texttt{in}\ \ expr_2\ (\texttt{if}\ expr_3)^*\ ]\!](\sigma) = \texttt{for}\ \ expr_1\ \texttt{in}\ \ trans_E[\![\ expr_2\ ]\!](\sigma)\ (\texttt{if}\ trans_E[\![expr_3]\!](\sigma))^*$

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

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

$\boxed{trans_{\overline{A}}\ \ \texttt{:}\ \ Alias\ \ \textbf{list}\ \ \rightarrow\ \ \Sigma\ \rightarrow\ \ \Sigma}$

$trans_{\overline{A}}[\![\ alias_1\ ...\ alias_{n-1}\ alias_n\ ]\!](\sigma) = trans_A[\![\ alias_n\ ]\!](trans_A[\![\ alias_{n-1}\ ]\!](...\ trans_A[\![\ alias_1\ ]\!](\sigma)))$

$\boxed{trans_A\ \ \texttt{:}\ \ Alias\ \rightarrow\ \texttt{CG}\ \rightarrow\ \texttt{CG}}$

$trans_A[\![\ id_1\ .id_2\ ...\ .id_n\ ]\!](\texttt{cg}) =$
LET $(\texttt{ns}, \texttt{vs})\ \ =\ \texttt{CG}$ **IN**
  LET $node_1 = \{\ id_1, \cdot\ \}, node_i = \{\ id_i, node_{i-1}\ \}$ **WHEN** $2 \leq i \leq n$ **IN**
  LET $var_k = id_1\ ...\ id_k$ **WHEN** $1 \leq k \leq n$ **IN**
$(\texttt{ns@}[node_1, ..., node_n], \texttt{vs@}[var_1 \mapsto node_1, ..., var_n \mapsto node_n])$

$trans_A[\![\ id_1\ .id_2\ ...\ .id_n\ \texttt{as}\ \ id_{as}\ ]\!](\texttt{cg}) =$
LET $(\texttt{ns}, \texttt{vs})\ \ =\ \texttt{CG}$ **IN**
  LET $node_1 = \{\ id_1, \cdot\ \}, node_i = \{\ id_i, node_{i-1}\ \}$ **WHEN** $2 \leq i \leq n$ **IN**
$(\texttt{ns@}[node_1, ..., node_n], \texttt{vs@}[id_{as} \mapsto node_n])$

$\boxed{trans_{\overline{W}}\ \ \texttt{:}\ \ WithItem\ \ \textbf{list}\ \ \rightarrow\ \ \Sigma\ \rightarrow\ \ (WithItem\ \ \textbf{list}\ \times\ \Sigma)}$

$trans_{\overline{W}}[\![\ with\_item_1\ with\_item_2\ ...\ with\_item_n\ ]\!](\sigma) =$
  LET $with\_item_1', \sigma_1\ \ =\ trans_W[\![\ with\_item_1\ ]\!](\sigma)$ **IN**
  LET $with\_item_2', \sigma_2\ \ =\ trans_W[\![\ with\_item_2\ ]\!](\sigma_1)$ **IN**

inden ...
  LET $with\_item_n', \sigma_n\ \ =\ trans_W[\![\ with\_item_n\ ]\!](\sigma_{n-1})$ **IN**
  $(with\_item_1'\ \textbf{::}\ with\_item_2'\ \textbf{::}\ ...\ \textbf{::}\ [with\_item_n'], \sigma_n)$

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

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

$trans_W[\![\ expr_1\ \texttt{as}\ \ expr_2\ ]\!](\text{smodenv}) =$
IF $\sigma(\textbf{“tensor\_flow”}) = id_1$ **AND** $expr_1 = id_1\texttt{.GradientTape()}$ **AND** $expr_2 = id_2$ **THEN** $(expr_1\ \texttt{as}\ \ expr_2, \sigma[\textbf{“gradient\_tape”} \mapsto id_2])$
ELSE $(trans_E[\![\ expr_1\ ]\!](\sigma)\ \texttt{as}\ \ expr_2, \sigma)$

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

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

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

$trans_P[\![\ \texttt{[}pattern^*\texttt{]}\ ]\!](\sigma) = \texttt{[}trans_P[\![pattern]\!](\sigma)^*\texttt{]}$

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

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

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

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

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

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

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	(ModuleSummary)
		`FuncSummary` tl	(FuncSummary)
		`ClassSummary` arg	(ClassSummary)
		`ValueSummary` s	(ValueSummary)
arg	::=	$Model \mid \bot$	(Arg)
tl	::=	$GradTape \mid Optimizer \mid \bot$	(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

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

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

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

$summary_{\overline{S}}[\![\ stmt_1\ \ stmt_2\ ...\ stmt_n\ ]\!](\sigma) =$ **LET** $\sigma_1, tl_1 = summary_S[\![\ stmt_1\ ]\!](\sigma)$ **IN**
LET $\sigma_2, tl_2 = summary_{\overline{S}}[\![\ stmt_2\ ]\!](\sigma_1)$ **IN**
...
LET $\sigma_n, tl_n = summary_{\overline{S}}[\![\ stmt_n\ ]\!](\sigma_{n-1})$ **IN**
$(\sigma_n, tl_1 \sqcup tl_2 \sqcup \ \ldots\ tl_n)$

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

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

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

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

A strict form of assignment statements

$summary_S[\![\ id_r\ \texttt{=}\ id_c\ \texttt{()}\ (\texttt{\#type:}s)?\ ]\!](\sigma) =$
IF $\sigma(id_c)$ = `ClassSummary` *Model* **THEN** $(\sigma[id_r \mapsto$ `ValueSummary` **"model_instance"**$], \bot)$
ELSE $(\sigma, \bot)$

$summary_S[\![\ (\texttt{\#type:}s)?\ \texttt{for}\ expr_1\ \texttt{in}\ expr_2\ \texttt{:}\ stmt_1^*\ (\texttt{else :}\ stmt_2^*)?\ ]\!](\sigma) =$
LET $\_, tl = summary_{\overline{S}}[\![\ stmt_1^*\ ]\!](\sigma)$ **IN**
(σ, tl)

$summary_S[\![\ (\texttt{\#type:}s)?\ \texttt{async for}\ expr_1\ \texttt{in}\ expr_2\ \texttt{:}\ stmt_1^*\ (\texttt{else :}\ stmt_2^*)?\ ]\!](\sigma) =$
LET $\_, tl = summary_{\overline{S}}[\![\ stmt_1^*\ ]\!](\sigma)$ **IN**
(σ, tl)

$summary_S[\![\ \texttt{while}\ (expr)\ \texttt{:}\ stmt_1^*\ (\texttt{else :}\ stmt_2^*)?\ ]\!](\sigma) =$
LET $\_, tl = summary_{\overline{S}}[\![\ stmt_1^*\ ]\!](\sigma)$ **IN**
(σ, tl)

$summary_S[\![\ (\texttt{\#type:}s)?\ \texttt{with}\ with\_item^*\ \texttt{:}\ stmt^*\ ]\!](\sigma) =$
LET $\_, tl = summary_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)$ **IN**
$(\sigma, summary_{\overline{W}}[\![\ with\_item^*\ ]\!](\sigma) \sqcup\ tl)$

$summary_S[\![\ (\texttt{\#type:}s)?\ \texttt{async with}\ with\_item^*\ \texttt{:}\ stmt^*\ ]\!](\sigma) =$
LET $\_, tl = summary_{\overline{S}}[\![\ stmt^*\ ]\!](\sigma)$ **IN**
$(\sigma, summary_{\overline{W}}[\![\ with\_item^*\ ]\!](\sigma) \sqcup\ tl)$

$summary_S[\![\ \texttt{try :}\ stmt_1^*\ \ exc\_handler^*\ \ (\texttt{else :}\ stmt_2^*)?\ \ (\texttt{finally :}\ stmt_3^*)?\ ]\!](\sigma) =$
$(\sigma, summary_{\overline{S}}[\![\ stmt_1^*\ ]\!](\sigma))$

$summary_S[\![\ \texttt{import}\ alias^*\ ]\!](\sigma) = (summary_{\overline{A}}[\![\ alias^*\ ]\!](\sigma), \bot)$

$summary_S[\![\ \texttt{from}\ 0\ \ id^*\ \ \texttt{import}\ alias^*\ ]\!](\sigma) =$
LET $\sigma' = summary_{\overline{A}}[\![\ alias^*\ ]\!](\sigma)$ **IN**
LET $[id_2 \mapsto$ `ModuleSummary` $id_2\ \sigma_2\ \bot,\ \ id_2 \mapsto$ `ModuleSummary` $id_2\ \sigma_2\ \bot,$
... $id_n \mapsto$ `ModuleSummary` $id_n\ \sigma_n\ \bot] = \sigma' \setminus \sigma$ **IN**
σ ++ $[id_2 \mapsto$ `ModuleSummary` $(id^*{+}id_2)\ \sigma_2\ \bot,\ \ id_2 \mapsto$ `ModuleSummary` $(id^*{+}id_2)\ \sigma_2\ \bot,$
... $id_n \mapsto$ `ModuleSummary` $(id^*{+}id_n)\ \sigma_n\ \bot]$

$summary_S[\![\ expr\ ]\!](\sigma) = (\sigma, summary_E[\![\ expr\ ]\!](\sigma))$

$summary_S[\![\ stmt\ ]\!](\sigma) = (\sigma, \bot)$

$\boxed{summary_{\overline{A}}\ \texttt{:}\ Alias\ \textbf{list}\ \to\ \Sigma\ \to\ \Sigma}$
$summary_{\overline{A}}[\![\ alias_1\ ...\ alias_{n-1}\ alias_n\ ]\!](\sigma) =$
$summary_A[\![\ alias_n\ ]\!](summary_A[\![\ alias_{n-1}\ ]\!](...\ summary_A[\![\ alias_1\ ]\!](\sigma)\ ...))$

$\boxed{trans_A\ \texttt{:}\ Alias\ \to\ \Sigma\ \to\ \Sigma}$

$trans_A[\![\ id_1\ id_2\ ...\ id_n\ ]\!](\sigma) =$

LET $\sigma_n = [id_n \mapsto$ `ModuleSummary` $id_n \quad \top \quad \bot]$

LET $\sigma_{n-1} = [id_{n-1} \mapsto$ `ModuleSummary` $id_{n-1} \quad \sigma_n \quad \bot]$

...

LET $\sigma_2 = [id_2 \mapsto$ `ModuleSummary` $id_2 \quad \sigma_3 \quad \bot]$

LET $\sigma_1 = [id_1 \mapsto$ `ModuleSummary` $id_1 \quad \sigma_2 \quad \bot]$

$\sigma \ ++ \ \sigma_1$

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

$summary_{\overline{W}}[\![\ with\_item_1\ with\_item_2\ ...\ with\_item_n\ ]\!](\sigma) =$

$summary_W[\![\ with\_item_1\ ]\!](\sigma) \sqcup \ summary_W[\![\ with\_item_2\ ]\!](\sigma) \sqcup$ **...** $summary_W[\![\ with\_item_n\ ]\!](\sigma)$

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

$summary_W[\![\ expr_1$ `as` $expr_2?\ ]\!](\sigma) = summary_E[\![\ expr_1\ ]\!](\sigma)$

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

A strict form of call expression

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

IF $\sigma(id_r)$ = `ValueSummary` **"model_instance" AND**

$expr_1$ = id_r`.fit` **THEN** *Optimizer*

ELIF $\sigma(id_r)$ = `ModuleSummary` Id("tensorflow") $\sigma' \ tl$ **AND**

$expr_1$ = id_r`.GradientTape` **THEN** *GradTape*

ELIF $\sigma(id_r)$ = `FuncSummary` tl **AND**

$expr_1$ = `idr` **THEN** tl

ELSE $\bot$

$summary_E[\![\ expr\ ]\!](\sigma) = \bot$