Appendix B: Transition Relations

Let the current state be (e, h, k) where $k = (i, \text{env\_id}) \cdot k_{\text{rest}}$. The next state (e', h', k') is determined by the instruction P_i as follows:

Pass, Break, Continue, Global, Nonlocal

$$(e', h', k') = (e, h, (\text{next}[i], \text{env\_id}) \cdot k_{\text{rest}})$$

Expression Assignment

$$(e', h', k') = \begin{cases} (e_{\text{updated}}, h, (\text{next}[i], \text{env\_id}) \cdot k_{\text{rest}}) & \text{if val} \neq \text{error} \\ (e, h, (\text{err}[i], \text{env\_id}) \cdot k_{\text{rest}}) & \text{if val} = \text{error} \end{cases}$$

where

$$\begin{aligned} P_i &= \mathit{var} = \mathit{expr} \\ \mathit{val} &= \text{eval}(\mathit{expr}, \text{env\_id}, e, h) \\ \mathit{env}_{\text{target}} &= \text{lookup\_env}(\mathit{var}, \text{env\_id}, e, h) \\ e_{\text{updated}} &= e[\mathit{env}_{\text{target}} \leftarrow \mathit{env}_{\text{target}}[\mathit{var} \mapsto \mathit{val}]] \end{aligned}$$

If, While

$$(e', h', k') = \begin{cases} (e, h, (\text{true}[i], \text{env\_id}) \cdot k_{\text{rest}}) & \text{if val} = \text{true} \\ (e, h, (\text{false}[i], \text{env\_id}) \cdot k_{\text{rest}}) & \text{if val} = \text{false} \\ (e, h, (\text{err}[i], \text{env\_id}) \cdot k_{\text{rest}}) & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} P_i &= \text{if/while } \mathit{expr}: \\ \mathit{val} &= \text{eval}(\mathit{expr}, \text{env\_id}, e, h) \end{aligned}$$

Function Definition

$$(e', h', k') = \begin{cases} (e_{\text{updated}}, h, (\text{next}[i], \text{env\_id}) \cdot k_{\text{rest}}) & \text{if val} \neq \text{error} \\ (e, h, (\text{err}[i], \text{env\_id}) \cdot k_{\text{rest}}) & \text{if val} = \text{error} \end{cases}$$

where

$$\begin{aligned} P_i &= \text{def } \mathit{var}(\mathit{id}_1, \mathit{id}_2, \ldots, \mathit{id}_n): \\ f_{\text{entry}} &= \text{entry location of function block} \\ \mathit{closure} &= \mathit{Closure}(f_{\text{entry}}, \text{env\_id}, [\mathit{id}_1, \mathit{id}_2, \ldots, \mathit{id}_n]) \\ \mathit{env}_{\text{target}} &= \text{lookup\_env}(\mathit{var}, \text{env\_id}, e, h) \\ e_{\text{updated}} &= e[\mathit{env}_{\text{target}} \leftarrow \mathit{env}_{\text{target}}[\mathit{var} \mapsto \mathit{closure}]] \end{aligned}$$

Call Assignment

$$(e', h', k') = \begin{cases} (e'', h'', k'') & \text{if } |arg\_vals| = |formals| \land \forall v \in vals, v \neq \text{error} \\ (e, h, (\text{err}[i], \text{env\_id}) \cdot k_{\text{rest}}) & \text{otherwise} \end{cases}$$

where
$P_i = var = func\_var(args)$
$vals_j = \text{eval}(args_j, \text{env\_id}, e, h), 1 \leq j \leq |args|$
$closure = \text{lookup\_val}(func\_var, \text{env\_id}, e, h)$
$Closure(f_{\text{loc}}, par\_env\_id, formals) = closure$
$\text{env\_new\_id} = \text{create\_new\_env}(e)$
$env_{\text{new}} = \text{populate\_env}(closure, \text{env\_new\_id})$
$e'' = e + \{\text{env\_new\_id} \mapsto env_{\text{new}}\}$
$h'' = h + \{\text{env\_new\_id} \mapsto \text{par\_env\_id}\}$
$k'' = (f_{\text{loc}}, \text{env\_new\_id}) \cdot k$

Return

$(e', h', k') = \{(e'', h, (\text{next}[\text{ret\_loc})], \text{ret\_env\_id}) \cdot k'_{\text{rest}})$
where
$P_i = \text{return } expr$
$val = \text{eval}(expr, \text{env\_id}, e, h)$
$(i, \text{env\_id}) \cdot (\text{ret\_loc}, \text{ret\_env\_id}) \cdot k'_{\text{rest}} = k$
$P_{\text{ret\_loc}} = assign\_var = \ldots$
$e'' = e[\text{lookup\_env}(assign\_var, \text{ret\_env\_id}, e, h) \leftarrow env[assign\_var \mapsto val]]$

Descriptions of Helper Functions

The formal transition rules in the previous section utilize several helper functions to manage control flow, evaluate expressions, handle environments, and perform lookups according to Python's scoping rules within the SimpliPy subset. These functions are conceptually defined as follows:

evalexpr*(expr, env_id, e, h)* Evaluates a SimpliPy expression *expr* within the current execution context.

- **Inputs:** The expression *expr* to evaluate, the current environment identifier *env_id*, the global lexical map *e*, and the lexical hierarchy *h*.
- **Output:** The computed value of the expression, or a special 'error' marker if evaluation fails (e.g., type error, variable not found).
- **Behavior:**
  - If *expr* is a constant, returns the constant's value.

- If *expr* is a variable name, uses `lookupval`(*expr*, *env_ id*, *e*, *h*) to find its value. Returns 'error' if `lookupval` fails.
- If *expr* involves operators, recursively calls `evalexpr` on sub-expressions, performs the operation, and returns the result. Propagates 'error' if any sub-evaluation fails or if the operation is invalid for the operand types.

`lookupenv`*(var, env_ id, e, h)* Determines the target *environment ID* where a variable *var* exists or where a new binding for it should be created during an assignment. This function embodies Python's LEGB-like scope resolution for assignments and lookups.

- **Inputs:** The variable name *var*, the current environment identifier *env_ id*, the lexical map *e*, and the lexical hierarchy *h*.
- **Output:** The identifier of the environment where *var* is found or should be bound/updated, or 'error'.
- **Behavior:**
  - Consults static analysis results for the scope associated with *env_ id*.
  - If *var* is declared `global` in this scope, returns 0 (the global environment ID).
  - If *var* is declared `nonlocal`, searches ancestor environments starting from the parent of *env_ id* (using *h*) upwards towards (but *not* including) the global scope. Returns the ID of the first ancestor environment found that contains a binding for *var*. If not found, signals an error (e.g., returns -1 or raises an exception).
  - Otherwise (neither `global` nor `nonlocal`):
    - Traverses the lexical hierarchy starting from the current environment *env_ id* upwards towards and including the global environment (ID 0), using the parent links in *h*.
    - At each environment ID *curr_env_id* in this traversal, check if *var* exists as a key in the environment *e*[*curr_env_id*].
    - If *var* is found in *e*[*curr_env_id*], return *curr_env_id*.
    - If the traversal completes (reaches global scope and checks it) without finding *var*, return the *original* starting environment identifier *env_ id*. This indicates that if an assignment occurs, a new binding should be created in the current local scope.

`lookupval`*(var, env_ id, e, h)* Looks up the *value* associated with a variable name *var*. It first determines the correct environment using `lookupenv` and then retrieves the value.

- **Inputs:** The variable name *var*, the starting environment identifier *env_ id*, the lexical map *e*, and the lexical hierarchy *h*.
- **Output:** The value bound to *var*, or an 'error' marker if the variable is not found in any accessible scope (i.e., if `lookupenv` indicates it doesn't exist).
- **Behavior:**
  - Call `target_env_id = lookupenv(`*`var, env_id, e, h`*`)`.

- If $target\_env\_id$ indicates that the variable was not found during the lookup traversal, return 'error'.
- Otherwise (the variable was found in environment $target\_env\_id$), retrieve the environment map $env = e[target\_env\_id]$.
- Return the value associated with var in that map: $env[var]$.

`createnewenv`(e) Generates a unique identifier for a new environment frame.

- **Input:** The current lexical map e.
- **Output:** A new integer environment ID that is not currently a key in e.
- **Behavior:** Typically implemented by finding the maximum existing ID in 'dom(e)' and returning the next integer.

`populate_env`(closure, env_new_id, arg_vals) Initializes a new environment frame (env_new_id) for a function call. (Note: This function was used conceptually in the Call Assignment rule description).

- **Inputs:** The *closure* being invoked (containing formal parameters and definition environment ID), the ID for the new environment env_new_id, and the list of evaluated argument values arg_vals.
- **Output:** A new environment dictionary (mapping variable names to values) representing the initial state of the function's local scope.
- **Behavior:**
  - Creates bindings in the new environment dictionary mapping each formal parameter name (from the *closure*) to the corresponding value in arg_vals.
  - Identifies (via static analysis of the function's body) all other variables defined locally within that function.
  - Initializes these other local variables in the new environment dictionary to a special 'uninitialized' marker ($\perp$).