A Spytecode Intermediate Representation

A.1 Overview and Motivation

Static analyses operate most effectively over an intermediate representation (IR) rather than source code or execution bytecode. An IR can regularize language constructs, make implicit concepts explicit, and expose dataflow uniformly.

We target Python programs executed by *CPython*. CPython compiles to a stack-based bytecode optimized for execution. While suitable for interpretation, the implicit operand stack and incidental stack shuffling complicate static reasoning. We therefore translate CPython bytecode to a verification-oriented, register-like IR we call Spytecode. Spytecode names every intermediate value and decomposes operations into primitive instructions. Spytecode is an analysis abstraction; it is not intended as a new execution format.

Spytecode is designed to:

(1) make data dependencies explicit at the instruction level,
(2) model namespaces and objects through a uniform heap,
(3) separate *resolution* (what to call) from *invocation* (calling it),
(4) identify allocation sites precisely.

A.2 Programs and Control Flow

A Spytecode program is a control-flow graph (CFG) whose nodes are basic blocks of Spytecode instructions. Blocks execute sequentially; edges encode control transfer. Spytecode itself does not carry explicit jump opcodes.

Path conditions are modeled by AssumeEq: execution continues only if reference equality holds; otherwise, the path terminates. Iteration can be expressed by testing and assuming on iterator sentinels (e.g., a distinguished `StopIteration` value), but exceptions are not modeled.

A.3 Instruction Set and Memory Model

Values and Stores. Spytecode distinguishes *static* and *heap* values. Static values refer to immutable entities (functions, classes, constants) in a closed static store σ; static objects may reference only other static objects. The mutable heap H maps heap locations to field-keyed optional values. `LOCALS` and `GLOBALS` are modeled as fixed heap locations.

Field keys. A single key space is used for dictionary-like addressing: $name(s)$ (attributes, keyword arguments), $index(n)$ (positional elements/arguments), and $both(n, s)$ (positions with names, if needed).

Instruction categories. Spytecode instructions fall into the following groups.

Data movement. Mov, LoadConst, LoadLocal, LoadGlobal, SetLocal move values between temporaries and named variables (the latter stored in `LOCALS`/`GLOBALS`).

Special-method resolution (pure). LookupDunder resolve special methods (e.g., `__add__`, `__getitem__`) based on static types and σ, returning a static function. These steps are side-effect free.

Overload selection and binding. ResolveOverload selects a static target for a call given a candidate callable and argument bundles; it is pure and may consult H only to read metadata relevant to selection. Bind applies Python's argument-binding rules and allocates a *bound callable* in the heap; its effects are limited to constructing that bound object.

Asymmetric collection constructors. ConstructTuple and ConstructDict allocate and populate tuple and dictionary literals directly. These are modeled as first-class constructors rather than function calls. The asymmetry exists because bootstrapping these literals into Call requires pre-existing callable semantics for the very constructors being introduced.

Attribute access. GetAttr and SetAttr encapsulate Python's attribute protocol (instance dictionary, descriptors, inheritance). Attribute access is singled out because it does not reduce cleanly to a single special-method call in the general case.

Invocation and unpacking. Call invokes a (possibly bound) callable and may allocate results. Unpack consumes an iterable and writes its elements to destinations; its effects are those of the underlying iterator protocol.

Control. AssumeEq (path filtering) and Exit (termination) complete the set.

Allocation sites. Heap allocation occurs only at Bind (allocating the bound callable object), ConstructTuple, ConstructDict, Call (result objects as required by the callee semantics), and any allocation internally performed by Unpack. Resolution instructions (LookupDunder and ResolveOverload) and attribute lookup are pure with respect to heap writes.

A.4 Operational Semantics

Execution states are pairs $\langle S, H\rangle$ where S maps temporaries to values and H is the heap. The static store σ is global and immutable. Each instruction defines a small-step transition from $\langle S, H\rangle$ to $\langle S', H'\rangle$. Helper functions capture opaque Python semantics:

- $\mathsf{get\_class}(v) \to \mathsf{StaticID}$,
- $\mathsf{dunder\_lookup}(m, [c_1, \ldots, c_k]) \to \mathsf{StaticID}$,
- $\mathsf{overload}(H, f, a, k) \to \mathsf{StaticID}$ (pure selection),
- $\mathsf{bind}(H, f, a, k) \to (b, H')$ (allocate bound callable),
- $\mathsf{attribute\_lookup}(H, v, s) \to v'$, $\mathsf{attribute\_assign}(H, v, s, v') \to H'$,
- $\mathsf{construct\_tuple}(H, [v_1, \ldots, v_n]) \to (t, H')$,
- $\mathsf{construct\_dict}(H, [(k_i, v_i)]_i) \to (d, H')$,
- $\mathsf{apply}(H, f) \to (v, H')$,
- $\mathsf{unpack}(H, v, n) \to ([v_1, \ldots, v_n], H')$.

Any failing premise (e.g., missing attribute, arity mismatch) terminates the current path. This is a deliberate, intraprocedural "happy-path" model.

Key properties:

- *Purity of resolution.* LookupDunder and ResolveOverload do not mutate the heap.
- *Bound-callable discipline.* Bind allocates only the bound callable and initializes its fields from the provided argument bundles; semantic effects of the call occur at Call.
- *Literal construction.* ConstructTuple and ConstructDict allocate exactly one new object and populate its fields deterministically from inputs.
- *Static-store closure.* Static objects reference only static objects.

Instructions:

$i ::=$	$\mathsf{Mov}(\$d, \$s)$	// $\$d := \$s$
	$\mid \mathsf{LoadConst}(\$d, c)$	// $\$d := c$
	$\mid \mathsf{LoadLocal}(\$d, x)$	// $\$d := locals[x]$
	$\mid \mathsf{LoadGlobal}(\$d, x)$	// $\$d := globals[x]$
	$\mid \mathsf{SetLocal}(x, \$s)$	// $locals[x] := \$s$
	$\mid \mathsf{LookupDunder}(\$d, op, \$a)$	// pure: find __op__
	$\mid \mathsf{ResolveOverload}(\$d, \$f, \$a, \$k)$	// pure: choose static target
	$\mid \mathsf{Bind}(\$d, \$f, \$a, \$k)$	// effect: make bound callable
	$\mid \mathsf{ConstructTuple}(\$d, \overline{\$e})$	// effect: allocate/populate tuple
	$\mid \mathsf{ConstructDict}(\$d, \overline{(\$k, \$v)})$	// effect: allocate/populate dict
	$\mid \mathsf{GetAttr}(\$d, \$o, k)$	// $\$d := \$o.k$; effect on property access
	$\mid \mathsf{SetAttr}(\$o, k, \$v)$	// $\$o.k := \$v$; effect
	$\mid \mathsf{Unpack}(\overline{\$d}, \$s)$	// unpack iterator; effect on iterator
	$\mid \mathsf{Call}(\$d, \$f)$	// $\$d := \$f()$
	$\mid \mathsf{AssumeEq}(\$b, \$c)$	// continue iff $\$b = \$c$
	$\mid \mathsf{Exit}$	// terminate

Values:

$v ::= \quad static(sid) \mid heap(hid)$

$sid ::= func(name) \mid class(name) \mid const(c)$

$hid ::= loc(n) \mid locals \mid globals$

Field Keys:

$k ::= \quad name(s) \mid index(n) \mid both(n, s)$

Stores:

$\sigma : StaticID \to FieldKey \rightharpoonup StaticID$	// static store (closed)
$H : HeapLoc \to FieldKey \rightharpoonup Value$	// mutable heap

Figure 1: Spytecode syntax.

A.5 Translation from CPython Bytecode

Translation preserves each bytecode's stack effect while eliminating the operand stack. Simple loads/stores map to data-movement instructions. Operators and subscription lower to dunder lookup, optional overload selection, Bind, and Call. Attribute access uses GetAttr/SetAttr. Tuple and dictionary literals produced by `BUILD_TUPLE`/`BUILD_MAP` translate to ConstructTuple/ConstructDict without a trailing Call. List and set construction can be expressed via the regular call pipeline.

A.6 Design Decisions (scope and impact)

Uniform heap for namespaces. Modeling `LOCALS`/`GLOBALS` as ordinary heap objects avoids special-purpose rules for name binding and enables uniform reasoning about reads and writes. This affects all loads/stores but does not change observable Python behavior.

Separated resolution and invocation. Splitting resolution from invocation isolates target identification from side effects. Analyses can reason about call targets and argument shapes before considering callee effects.

Asymmetric literal construction. Tuple and dict literals are introduced as dedicated constructors. This avoids circularity in defining their semantics via Call while still allowing other collections to be expressed through the standard call pipeline.

Attribute access as a primitive. Attribute access remains a dedicated instruction pair because it subsumes instance dictionaries, descriptors, and inheritance. Reducing it to a single special-method call would obscure these behaviors.

A.7 Assumptions and Limitations

The model intentionally omits features that complicate static reasoning and are uncommon in the intended workloads:

- No exception handling; any failure terminates the path.
- No dynamic code execution (`eval`/`exec`) or runtime code generation.
- No closures or mutation of globals after initialization.
- No generators, coroutines, or context managers.
- No inheritance
- Classes and functions are immutable after creation.

$$\textsc{Mov}\ \frac{S(\$s) = v}{\langle S, H\rangle \xrightarrow{\text{Mov}(\$d,\$s)} \langle S[\$d \mapsto v], H\rangle} \qquad \textsc{LoadConst}\ \frac{static(const(c)) = v}{\langle S, H\rangle \xrightarrow{\text{LoadConst}(\$d,c)} \langle S[\$d \mapsto v], H\rangle} \qquad \textsc{LoadLocal}\ \frac{H(locals)(name(x)) = v}{\langle S, H\rangle \xrightarrow{\text{LoadLocal}(\$d,x)} \langle S[\$d \mapsto v], H\rangle}$$

$$\textsc{LoadGlobal}\ \frac{H(globals)(name(x)) = v}{\langle S, H\rangle \xrightarrow{\text{LoadGlobal}(\$d,x)} \langle S[\$d \mapsto v], H\rangle} \qquad \textsc{SetLocal}\ \frac{S(\$s) = v \qquad H' = H[locals \mapsto H(locals)[name(x) \mapsto v]]}{\langle S, H\rangle \xrightarrow{\text{SetLocal}(x,\$s)} \langle S, H'\rangle}$$

$$\textsc{LookupDunder}\ \frac{S(\$a) = \overline{v} \qquad \text{get\_class}(\overline{v}) = \overline{c} \qquad \text{dunder\_lookup}(m, [\overline{c}]) = f}{\langle S, H\rangle \xrightarrow{\text{LookupDunder}(\$d,m,\$a)} \langle S[\$d \mapsto static(f)], H\rangle}$$

$$\textsc{ResolveOverload}\ \frac{S(\$f) = v_f \quad S(\$a) = a \quad S(\$k) = k \qquad \text{overload}(H, v_f, a, k) = f \quad f \in StaticID}{\langle S, H\rangle \xrightarrow{\text{ResolveOverload}(\$d,\$f,\$a,\$k)} \langle S[\$d \mapsto static(f)], H\rangle}$$

$$\textsc{Bind}\ \frac{S(\$f) = static(f) \quad S(\$a) = a \quad S(\$k) = k \qquad \text{bind}(H, f, a, k) = (b, H')}{\langle S, H\rangle \xrightarrow{\text{Bind}(\$d,\$f,\$a,\$k)} \langle S[\$d \mapsto heap(b)], H'\rangle}$$

$$\textsc{ConstructTuple}\ \frac{S(\$e_1) = v_1\ \cdots\ S(\$e_n) = v_n \qquad \text{construct\_tuple}(H, [v_1, \ldots, v_n]) = (t, H') \quad t \in HeapLoc}{\langle S, H\rangle \xrightarrow{\text{ConstructTuple}(\$d,[\$e_1,\ldots,\$e_n])} \langle S[\$d \mapsto heap(t)], H'\rangle}$$

$$\textsc{GetAttr}\ \frac{S(\$o) = v \qquad \text{attribute\_lookup}(H, v, f) = v'}{\langle S, H\rangle \xrightarrow{\text{GetAttr}(\$d,\$o,f)} \langle S[\$d \mapsto v'], H\rangle} \qquad \textsc{SetAttr}\ \frac{S(\$o) = v_o \quad S(\$v) = v \qquad \text{attribute\_assign}(H, v_o, f, v) = H'}{\langle S, H\rangle \xrightarrow{\text{SetAttr}(\$o,f,\$v)} \langle S, H'\rangle}$$

$$\textsc{Call}\ \frac{S(\$f) = f \qquad \text{apply}(H, f) = (v, H')}{\langle S, H\rangle \xrightarrow{\text{Call}(\$d,\$f)} \langle S[\$d \mapsto v], H'\rangle} \qquad \textsc{AssumeEq}\ \frac{S(\$b) = S(\$c)}{\langle S, H\rangle \xrightarrow{\text{AssumeEq}(\$b,\$c)} \langle S, H\rangle} \qquad \textsc{Exit}\ \langle S, H\rangle \xrightarrow{\text{Exit}} \langle S, H\rangle$$

Notes Any premise may fail; failure terminates the current path.

Figure 2: Concrete operational semantics for Spytecode.

- Effectful properties (attribute access) are implemented but not modeled in the formal description.

These choices bound the semantic surface and keep the IR focused on dataflow and heap effects relevant to numerical and scientific code.

B Type System

B.1 Background

This appendix specifies the type system implemented in `type_system.py`. It is designed to support precise static modeling of scientific Python code (e.g., NumPy, SciPy) while avoiding complexity unnecessary for scientific/numerical workloads with predictable control flow and limited reflection.

Design rationale. The system adopts several deliberate departures from common type-system practice, trading generality for domain-specific tractability:

- **Uniform field/parameter handling:** Model both function parameters and class/protocol/module bindings as rows (ordered typed dictionaries), enabling uniform handling of lookup, subtyping, join, and unification with width-subtyping.
- **Variadic packs with incremental binding:** Allow pack variables to be partially instantiated and applied multiple times, mirroring Python's flexible calling conventions while leveraging the uniform row representation.
- **No nominal subtyping between user classes:** Avoid Python's method-resolution-order (MRO) complexity, which is unnecessary for scientific workloads.
- **Type effects with strict equality:** Annotate functions with effects for heap allocation, mutation, and argument type updates. Require equality rather than joins during function unification, prioritizing checkpointing precision over effect polymorphism.
- **Two-stage overload resolution:** First match parameter signatures, then join the return types of all successful matches, reflecting Python's runtime dispatch model.

Meta-properties and scope. While we do not prove formal soundness, the design aims for:

- **Decidability:** Type checking and inference are decidable for the restricted Python fragment (Section C).
- **Termination:** Unification and join terminate over the targeted subset; the type lattice is finite and usually very short. Dimensionality tracking of NumPy arrays is intentionally excluded due to its termination and complexity implications.
- **Precision:** Strict effect equality and the information ordering favor simplicity and precise state reasoning over broad applicability.
- **Transparency:** The unknown type any and dynamic features act as explicit, unsound escape hatches.

B.2 Syntax

Figure 4 defines the syntax of the type system. Type expressions $\mathcal{T}$ cover constants, variables (both ordinary and variadic), unions, records, classes, modules, polymorphic functions, overloads, instantiations, and literal values. Record types map field keys to types, where keys may carry both positional and nominal components. Effects annotate callable types with allocation and mutation.

Uniform use of rows. Rows ρ are the *only* finite mapping construct in the type system, used both for callable parameter lists and for class/module member dictionaries (see Section B.3). In Python, both are naturally modeled as ordered typed dictionaries: finite maps from *keys* (either positional indices or string labels) to types, with ordering preserved from their definition. This uniform abstraction allows the same operations—unify_ρ, subtyping, join/meet (Figure 6)—to apply in both contexts, simplifying the formalism and aligning with the structural nature of Python's attribute and parameter matching rules.

In our setting, parameter rows are not a distinct syntactic category from record rows: both are instances of the same ρ syntax, so parameter–argument unification is literally the same fieldwise unification used for member records. The only difference is in the *interpretation*: for arguments, missing keys may indicate partial application; for parameters, missing keys indicate an invalid call; for fields, missing keys indicate absent attributes (yielding $\bot$ on lookup).

The syntax of the type system is given in Figure 3 and Figure 4.

Design note. Effects $\epsilon^\#$ are *not* row-polymorphic: unification is only applied to positive rows, and unification of callables requires $\epsilon_1^\# = \epsilon_2^\#$. This sacrifices the flexibility of row-polymorphic effect systems in exchange for higher precision in mutation and allocation tracking, which is critical to our checkpointing analysis.

X, Y, Z	type variables
i	positional index $\in \mathbb{N}$
s	string label $\in \mathcal{L}$
ρ	generic row variable (unspecified polarity)
ρ^+	argument row (positive)
ρ^-	parameter row (negative)
$\rho^=$	default-argument row (positive)

Figure 3: Metavariables for row kinds and related constructs.

$$
\begin{aligned}
\tau ::= {} & c \quad \text{(qualified constant name)} \\
& \mid \ell \quad \text{(literal type)} \\
& \mid \mathsf{union}(\overline{\tau}) \\
& \mid \mathsf{module}(C, \rho) \\
& \mid \mathsf{protocol}(\rho, \overline{\tau}, \overline{X}) \\
& \mid \mathsf{class}(C, \rho, \overline{\tau}, \overline{X}) \\
& \mid \mathsf{overload}(\overline{F}) \\
& \mid \tau\langle\overline{\tau}\rangle \quad \text{(generic instantiation)} \\
& \mid X \mid X^* \mid \mathsf{star}(\overline{\tau}) \\
& \mid \mathsf{any} \\
\rho ::= {} & \{k_1 : \tau_1, \ldots, k_n : \tau_n\} \quad \text{(ordered typed dictionary)} \\
F ::= {} & \forall\overline{X}.\, \rho^- \mid \rho^= \xrightarrow{\epsilon^\#} \tau \quad \text{(function signature)} \\
& \mid \forall\overline{X} \xrightarrow{\epsilon^\#} \tau \quad \text{(property signature)} \\
k ::= {} & \langle i \rangle \mid \langle s \rangle \mid \langle i, s \rangle \quad \text{(field key)}
\end{aligned}
$$

Figure 4: Type system syntax. Effects $\epsilon^\#$ are described in Section B.11.

B.3 Unified Row Model

We use a single, uniform construct ρ to model all finite, ordered key–type mappings in Python. Rows appear in different *roles* (arguments/fields vs. parameters), but they share the *same* width-based order and lattice; variance is handled at function types.

Keys. A key k is composite: a parameter x at position 0 is $\langle 0, \text{"x"} \rangle$; a class attribute y is $\langle \text{"y"} \rangle$. Key compatibility:

$$\langle k \rangle \preccurlyeq \langle k \rangle, \quad \langle k \rangle \preccurlyeq \langle k, \_ \rangle, \quad \langle k \rangle \preccurlyeq \langle \_, k \rangle.$$

Row syntax.

$$\rho \;=\; \{\, k_1 : \tau_1,\; \ldots,\; k_n : \tau_n \,\}$$

Keys are unique; ordering is preserved for positional correctness.

Well-formedness. Let $\mathrm{dom}(\rho)$ be the set of keys in ρ.

$$\begin{aligned}\mathsf{wellformed}(\rho) \;\equiv\; &\big(\forall i < j.\ \langle i \rangle, \langle j \rangle \in \mathrm{dom}(\rho) \Rightarrow \langle i{-}1 \rangle \in \mathrm{dom}(\rho)\big) \wedge \\ &\big(\forall k \in \mathrm{dom}(\rho).\ \mathsf{wellformed}(\rho(k))\big).\end{aligned}$$

Contiguous-position predicate:

$$\mathsf{posrow}(\rho) \;\equiv\; \mathrm{dom}(\rho) \cap \mathbb{N} \;=\; \{0, \ldots, n{-}1\} \text{ for some } n.$$

Row order (width subtyping). Rows use a single width/depth order:

$$\begin{aligned}\rho_1 \;\le_\rho\; \rho_2 \quad \Longleftrightarrow \quad &\mathrm{dom}(\rho_2) \subseteq \mathrm{dom}(\rho_1) \\ &\wedge\ \forall k \in \mathrm{dom}(\rho_2).\ \rho_1(k) \;\le\; \rho_2(k)\end{aligned}$$

Intuition: more keys and/or more specific member types imply a more specific row.

Row lattice. Rows form a lattice $(\mathcal{R}, \le_\rho, \sqcup_\rho, \sqcap_\rho, \top_\rho, \bot_\rho)$:

$$\begin{array}{lcl}
\top_\rho & = & \{\}\ \text{(empty row)} \\
\bot_\rho & = & \text{inconsistent row} \\
\mathrm{dom}(\rho_1 \sqcup_\rho \rho_2) & = & \mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) \\
(\rho_1 \sqcup_\rho \rho_2)[k] & = & \rho_1[k] \;\sqcup\; \rho_2[k] \quad (k \text{ shared}) \\
\mathrm{dom}(\rho_1 \sqcap_\rho \rho_2) & = & \mathrm{dom}(\rho_1) \cup \mathrm{dom}(\rho_2) \\
(\rho_1 \sqcap_\rho \rho_2)[k] & = & \rho_1[k] \;\sqcap\; \rho_2[k] \quad (k \text{ shared})
\end{array}$$

Function variance (where contravariance lives). Contravariance arises at the function constructor:

$$(\rho_1 \xrightarrow{\epsilon^\#} \tau_1) \;\le\; (\rho_2 \xrightarrow{\epsilon^\#} \tau_2) \quad \Longleftrightarrow \quad \rho_2 \;\le_\rho\; \rho_1 \;\wedge\; \tau_1 \;\le\; \tau_2.$$

(Effects are compared as specified elsewhere; we require $\epsilon^\#$ equality in our implementation.)

Protocols (structural). Let $\Pi[\overline{\sigma}]$ have required row R_Π (after instantiation), and let R_T be the provided member row of T. Then

$$\begin{aligned}\Gamma \vdash T \;<:\; \Pi[\overline{\sigma}] \quad \Longleftrightarrow\ &\mathrm{dom}(R_\Pi) \subseteq \mathrm{dom}(R_T) \\ &\wedge\ \forall k \in \mathrm{dom}(R_\Pi).\ \Gamma \vdash R_T(k) \;<:\; R_\Pi(k)\end{aligned}$$

with method members checked by the function subtyping rule above.

B.4 Type Parametricity

Given $\forall \overline{X}.\ \rho^- \mid \rho^= \xrightarrow{\epsilon^\#} \tau_r$ and arguments ${\rho^+}_A$:

Binding. $(\theta, {\rho^-}') = \mathsf{matchRows}(\forall \overline{X}.\rho^-, {\rho^+}_A)$.

Effects. Apply θ to $\epsilon^\#$, then project to ${\rho^-}'$.

Residual. $\forall(\overline{X} \setminus \mathrm{dom}(\theta)).\ {\rho^-}' \mid {\rho^=}' \xrightarrow{\epsilon^{\#\prime}} \tau_r[\theta]$ where ${\rho^=}'$ is $\rho^=$ restricted/reindexed to ${\rho^-}'$.

Finish. If $\mathsf{matchRows}(\rho^=, {\rho^-}')$ has empty residual, return $\tau_r[\theta]$.

B.5 Variadic Packs

If ρ^- has one variadic positional $i : X^*$:

(1) Bind fixed params first.
(2) Match remaining contiguous positionals to X^*.
(3) Retain X^* in ${\rho^-}'$ for incremental binding in later applications.

B.6 Overloads

For overload $\{\varphi_1, \ldots, \varphi_m\}$ and ${\rho^+}_A$:

- Bind each case independently.
- Discard failures; group by identical matched-parameter sets.
- Each group becomes an overloaded residual callable.
- If all FinishApp is applicable to all residuals, join their return types.

B.7 Receiver Binding

Methods have first parameter as receiver:

$$\forall \overline{X}.\ (0 : \tau_{self},\ {\rho^-}') \mid \rho^= \xrightarrow{\epsilon^\#} \tau_r$$

Bind receiver, drop/reindex, continue as partial application.

Arity. Unmatched arguments → error; unmatched parameters → residual callable (possibly discharged by defaults).

B.8 Example

To illustrate partial application, consider a generic function taking two parameters. Let:

$$\varphi = \forall T, U.\ (0 : T,\ 1 : U) \mid \{\} \xrightarrow{\epsilon^\#} T$$

Apply $\{0 : \mathsf{int}\}$:

$$\theta = \{T \mapsto \mathsf{int}\}, \quad {\rho^-}' = \{0 : U\}$$

Residual: $\forall U.\ (0 : U) \mid \{\} \xrightarrow{\epsilon^\#[\theta]} \mathsf{int}$.

Apply $\{0 : \mathsf{str}\}$:

$$\theta' = \{U \mapsto \mathsf{str}\}$$

Now FinishApp is applicable, so $\Rightarrow \mathsf{int}$.

B.9 Modules, Classes, and Protocols

Entity differences. We distinguish three entity forms:

- **Classes** are nominal in that they do not support structural subtyping: two classes are related if and only if they have *exactly* the same name (and their type parameters can be unified). We model no nontrivial superclass relations. Class members are given by a row ρ; classes may be generic over type parameters $\overline{\alpha}$.
- **Modules** are singletons: a module $\mathsf{module}(C, \rho)$ has a fixed row of members and no generic parameters.

$$\frac{\text{posrow}({\rho^+}_A) \qquad \text{matchRows}(\forall \overline{X}.\ \rho^- \mid \rho^{=},\ {\rho^+}_A) \Downarrow (\theta,\ {\rho^-}')}{\forall \overline{X}.\ \rho^- \mid \rho^{=} \xrightarrow{\epsilon^\#} \tau \triangleright {\rho^+}_A \Downarrow \qquad \forall (\overline{X} \setminus \text{dom}(\theta)).\ {\rho^-}' \mid \rho^{=\prime} \xrightarrow{\text{project}(\epsilon^\#[\theta],\, {\rho^-}')} \tau[\theta]}\ \textsc{PartialApp}$$

$$\frac{\text{matchRows}(\rho^{=},\ {\rho^-}_{\text{rem}}) \Downarrow (\_,\ \varnothing)}{{\rho^-}_{\text{rem}} \mid \rho^{=} \xrightarrow{\epsilon^\#} \tau \triangleright \varnothing \Downarrow \tau}\ \textsc{FinishApp}$$

Auxiliary

$\text{posrow}(\rho^+)$	$\text{dom}(\rho^+) \cap \mathbb{N} = \{0, \ldots, n{-}1\}$
Argument compatibility	$\tau_a \leq: \tau_p$
$\text{matchRows}(\forall \overline{X}.\ \rho^-,\ \rho^+) \Downarrow (\theta,\ {\rho^-}')$	$\text{dom}(\rho^+) \subseteq \text{dom}(\rho^-) \ \wedge\ \forall k \in \text{dom}(\rho^+).\ {\rho^+}_k \leq: {\rho^-}_k[\theta]$, with ${\rho^-}' = \text{residual}(\rho^+[\theta],\ \rho^-)$.
$\text{residual}(\rho^+,\ \rho^-)$	$\text{subtract\_indices}(\rho^- \setminus \text{dom}(\rho^+),\ \text{dom}(\rho^+) \cap \mathbb{N})$
$\text{project}(\epsilon^\#,\ \rho^-)$	drop/rename parameter-indexed components to match ρ^-

Figure 5: Typing rules for function application, supporting partial application and default arguments in Python's calling model. PartialApp specializes parameters with arguments; FinishApp fires when remaining parameters are all satisfied by defaults.

- **Protocols** are structural interfaces with optional type parameters. Satisfaction is by structural subtyping (Figure 8), not by nominal identity.

Common mechanism: member access. All three entity kinds share the same lookup mechanism on their member row ρ (see Section B.3). Given an entity type τ and a key k (positional index or string label; cf. Figure 3), the lookup judgment

$$\tau \cdot k \Downarrow v$$

yields the member type v or $\bot$ if no match is found.

Lookup is defined as:

$$\rho \cdot k = \bigsqcup \{\tau \mid k' {:} \tau \in \rho \ \wedge\ k' <: k\} \qquad \text{(basic case; } \bot \text{ if no match)}$$

$$(\tau_1 + \tau_2) \cdot k = (\tau_1 \cdot k) \ \sqcup\ (\tau_2 \cdot k) \qquad \text{(union types)}$$

$$\text{module}(C, \rho) \cdot k = \rho \cdot k \qquad \text{(modules)}$$

$$\text{class}(C, \rho, \overline{\tau}, \overline{X}) \cdot k = \rho \cdot k \qquad \text{(classes)}$$

$$C[\overline{\sigma}] \cdot k = (\text{class}(C, \rho, \overline{\alpha}, \overline{X})[\alpha_i \mapsto \sigma_i]) \cdot k \qquad \text{(instantiated classes)}$$

If $\rho \cdot k$ yields an overload set, the overload combination rules of Section B.11 apply. If the member is callable, receiver binding (Section B.7) is applied, treating the receiver as the first positional argument $\langle 0 \rangle$. If the member is a property, the property-access rule applies.

Generic classes. A generic class type has the form

$$\text{class}(C, \rho, \overline{\alpha}, \overline{X})$$

where $\overline{\alpha} = (\alpha_1, \ldots, \alpha_m)$ are type parameters and ρ is the member row.

Instantiation: Given $\overline{\sigma}$ with $|\overline{\sigma}| = m$,

$$C[\overline{\sigma}] = (\text{class}(C, \rho, \overline{\alpha}, \overline{X}))[\alpha_i \mapsto \sigma_i]_{i=1}^{m}$$

Substitution applies to all member types and to any effects:

$$(\rho \xrightarrow{\epsilon^\#} \tau)[\alpha \mapsto \sigma] \ =\ (\rho[\alpha \mapsto \sigma]) \xrightarrow{\epsilon^\#[\alpha \mapsto \sigma]} (\tau[\alpha \mapsto \sigma]).$$

Class parameters are in scope for all members.

Method parameter shadowing. If a method is quantified $\forall \overline{\beta}.\ \rho \xrightarrow{\epsilon^\#} \tau$ and $\overline{\beta} \cap \overline{\alpha} \neq \varnothing$, each β_i shadows the corresponding α_j within the method's scope; no automatic renaming is performed.

Protocols. A protocol $\Pi[\overline{\alpha}] : \rho$ is a structural interface. A type T satisfies $\Pi[\overline{\sigma}]$ iff, writing R_Π for the instantiated requirement row and R_T for the provided member row of T:

$$\text{dom}(R_\Pi) \subseteq \text{dom}(R_T) \quad \text{and} \quad \forall k \in \text{dom}(R_\Pi).\, \Gamma \vdash R_T(k) \ <:\ R_\Pi(k),$$

where method members are checked by function subtyping (parameters contravariant, result covariant). Protocol type parameters are instantiated with $\overline{\sigma}$ before checking satisfaction.

Lattice structure of rows. As in Section B.3, rows form a lattice $(\rho, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ under the information ordering: more keys $\Rightarrow$ more specific.

- $\top_\rho$ = empty row
- $\bot_\rho$ = inconsistent row
- **Join** ($\sqcup$): intersection of domains, join types pointwise (join as in Figure 6)
- **Meet** ($\sqcap$): union of domains, meet types pointwise

Worked example.

Definition: $\text{class List}[T]$:

$\{\langle\text{"append"}\rangle : (\text{self} : \text{List}[T],\ x : T) \xrightarrow{\epsilon^\#} \text{None}\}$

Instantiation: $\text{List}[\text{int}] =$

$\{\langle\text{"append"}\rangle : (\text{self} : \text{List}[\text{int}],\ x : \text{int}) \xrightarrow{\epsilon^\#} \text{None}\}$

Lookup: $\text{List}[\text{int}] \cdot \langle\text{"append"}\rangle =$

$(\text{self} : \text{List}[\text{int}],\ x : \text{int}) \xrightarrow{\epsilon^\#} \text{None}$

Bind self: $\langle x \rangle : \text{int} \xrightarrow{\epsilon^\#} \text{None}$

Auxiliary Definitions

$\mathrm{dom}(\rho)$	Domain of keys in row ρ
$\rho \cdot k$	Lookup type at key k in ρ ($\bot$ if absent)
$\rho[k \mapsto \tau]$	Row update (replace or insert key k with type τ)
$\rho \setminus S$	Row with all keys in S removed
$k \preccurlyeq k'$	Key match: $\langle k\rangle \preccurlyeq \langle k\rangle$; $\langle k\rangle \preccurlyeq \langle k,\rangle$; $\langle k\rangle \preccurlyeq \langle ,k\rangle$
$\theta[\tau]$	Capture-avoiding substitution of θ into τ
$\theta_1 \sqcup \theta_2$	Substitution join: defined iff same RHS for overlapping vars (up to α)
$R_1 \sqcup R_2$	Row join: keep common keys, join their types
$R_1 \sqcap R_2$	Row meet: union of keys, meet their types
$\mathsf{unify}_{\mathrm{type}}(\tau_1, \tau_2)$	Binary type unification (Figure 9)
$\mathsf{unify}_{\rho}(\rho_1, \rho_2)$	Field-wise unification on $\mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2)$

Figure 6: Common predicates and operations for member access and structural satisfaction.

B.10 Static Semantics

Scope and soundness. The typing rules are intended to be sound for the restricted Python subset defined in Section C, but we make no claims for full Python. Features such as any and dynamic attribute creation are explicitly unsound escape hatches, admitted for practicality in scientific code. All rules are designed to ensure decidable type checking and termination of unification within the targeted subset.

Typing judgments are described in Figure 7.

Binary and unary operations. Binary and unary operations are resolved through a two-step process that makes dunder method lookup explicit. For binary operations $e_1\ op\ e_2$: first, the appropriate dunder method (e.g., `__add__` for +) is looked up on the left operand type via member access; second, the PartialApp rule is applied with the right operand as argument. Unary operations follow the same pattern but with no arguments. This approach unifies operator overloading with the general function application mechanism. Effects (allocation, mutation, aliasing) are extracted from the resolved function type and forwarded to the analysis domain, while the typing rules above show only the return types.

Subtyping. Subtyping supports reflexivity, transitivity, union decomposition, record width/depth subtyping, and structural subtyping for protocols. There is *no* nominal subtyping rule between arbitrary classes; two distinct non-protocol classes are unrelated unless one is a union containing the other. Protocol satisfaction is checked structurally: a type T is a subtype of $\Pi[\overline{\sigma}]$ if it has all required members and each member type unifies (fieldwise) with the corresponding protocol member type.

Relation to width subtyping. Our information ordering, where ρ rows with more fields are more specific, is equivalent to the standard *width subtyping* relation from record calculi . This correspondence ensures that meet and join operations behave predictably, while our strict key-matching rule simplifies unification and lattice reasoning at the cost of omitting Python's permissive keyword matching.

Type operations. Joins $\sqcup$ compute least upper bounds; meets $\sqcap$ compute greatest lower bounds. For ρ rows, these are as in Figure 6; otherwise joins are simple unions.

In the implementation, union types are *normalized* after construction: duplicates are removed, nested unions are flattened, and singleton unions collapse to the member type. This normalization ensures canonical forms for lattice comparisons and avoids spurious distinctions between equivalent unions.

Quantified function application. Application of a quantified function $\forall\overline{X}.\ \rho \xrightarrow{\epsilon^\#} \tau_r$ is governed by the binding algorithm in Section B.7, which handles instantiation, partial application, variadic packs, overloads, and receiver binding.

Unification. Unification $\Delta \vdash \tau_1 \doteq \tau_2 \rightsquigarrow \theta$ computes substitutions $\theta : \mathcal{V} \to \mathcal{T}$ and supports variadic parameters, instantiation, functions, and ρ rows. For functions, unification *requires* $\epsilon_1^\# = \epsilon_2^\#$; effects are not joined at this stage. For other structured types such as unions or classes, unification proceeds componentwise over their parameters, and is *undefined* when type constructors differ.
Note: The strict effect-equality requirement matches the implementation's precision goals in checkpointing analysis; other systems might use effect joins or subeffect relations.

B.11 Dynamic Semantics

Dynamic semantics here covers the runtime-analogous steps for overload resolution, attribute/index access, and effect composition, as modeled in our abstract interpreter.

Property evaluation. If $\tau \cdot k$ yields a callable ($\xrightarrow{\epsilon^\#} \sigma$), the result of $\tau \cdot k$ is σ (implicit call with empty arguments). Otherwise, if it is callable but not a property, $\tau \cdot k$ yields the bound method, with the receiver already bound via the mechanism described in Section B.7.

Overload resolution. Overload resolution is a two-stage process:

(1) **Static grouping:** Given an overload set $\{\varphi_1, \ldots, \varphi_n\}$ (e.g., from $\tau \cdot k$), each arm φ_i is unified with the argument row A via $\mathsf{unify}_{\rho}(\mathsf{params}(\varphi_i), A)$ from Section B.3. Successful arms produce residual cases; these are grouped by identical

Expression Typing $\Gamma \vdash e : \tau$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \Gamma \vdash v : \mathsf{Literal}(v)\ \textsc{Literal} \qquad \frac{\Gamma \vdash e : \tau \qquad \tau \cdot a \Downarrow \phi \qquad \phi \text{ is not callable}}{\Gamma \vdash e.a : \phi}\ \textsc{Member-Field}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \cdot a \Downarrow \forall \overline{X}.(0 : \tau_{self},\, R') \xrightarrow{\epsilon^\#} \sigma \qquad \mathsf{bind\_self}(\tau,\, \tau_{self},\, R',\, \overline{X}) = (R'',\, \overline{Y},\, \theta)}{\Gamma \vdash e.a : \forall \overline{Y}.\, R'' \xrightarrow{\epsilon^\#[\theta]} \sigma[\theta]}\ \textsc{Member-Method}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \cdot a \Downarrow \forall () \to \sigma}{\Gamma \vdash e.a : \sigma}\ \textsc{Member-Property} \qquad \frac{\Gamma \vdash e : \mathsf{type}[C[\overline{\alpha}]] \qquad \Gamma \vdash \tau_i : \mathsf{type}[T_i]\ (i = 1..n)}{\Gamma \vdash e[\tau_1, \ldots, \tau_n] : \mathsf{type}[C[T_1, \ldots, T_n]]}\ \textsc{Type-Inst}$$

$$\frac{\Gamma \vdash \overline{e} : \overline{\tau} \qquad \mathsf{dunder}(op,\, \overline{\tau}) = \sigma}{\Gamma \vdash op\ \overline{e} : \sigma}\ \textsc{LookupDunder}$$

Auxiliary Operations

$\tau \cdot a \Downarrow \phi$	Member lookup (Section B.11)
$\mathsf{bind\_self}(\tau_{recv}, \tau_{self}, R, \overline{X})$	Receiver binding (Section B.7)
$\mathsf{property}(\phi)$	Predicate: callable flagged as a property
$\mathsf{dunder}(op, \overline{\tau})$	look up the right dunder overloaded function given the participating types

Figure 7: Expression typing rules and row-level application. Constructor calls follow the same scheme after resolving the constructor protocol (e.g., `__init__`). Effects are forwarded to the combined domain; the relation here returns only types.

$$\Gamma \vdash \tau <: \tau\ \textsc{Refl} \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3}\ \textsc{Trans}$$

$$\frac{\forall i.\ \Gamma \vdash \tau_i <: \tau}{\Gamma \vdash (\tau_1 + \cdots + \tau_n) <: \tau}\ \textsc{Union-Sub}$$

$$\frac{\mathrm{dom}(R_2) \subseteq \mathrm{dom}(R_1) \qquad \forall k \in \mathrm{dom}(R_2).\ \Gamma \vdash R_1(k) <:_\rho R_2(k)}{\Gamma \vdash R_1 <:_\rho R_2}\ \textsc{Record-Sub}$$

$$\frac{\mathrm{dom}(R_{\Pi[\overline{\sigma}]}) \subseteq \mathrm{dom}(R_T) \qquad \forall k \in \mathrm{dom}(R_{\Pi[\overline{\sigma}]})\ .\ \Gamma \vdash R_T(k) <: (R_{\Pi[\overline{\sigma}]})(k)}{\Gamma \vdash T <: \Pi[\overline{\sigma}]}\ \textsc{Protocol}$$

Figure 8: Implemented subtyping rules. Rows use dom **and** $<:_\rho$ **as in Section B.3. Protocol satisfaction is structural, using member access +** unify_ρ**.**

$$\frac{X \notin \mathrm{dom}(\theta) \qquad X \notin \mathsf{FV}(\tau)}{\Delta \vdash X \doteq \tau \rightsquigarrow \theta[X \mapsto \tau]}\ \textsc{Var-L}$$

$$\frac{\Delta \vdash \overline{\tau_1} \doteq \overline{\tau_2} \rightsquigarrow \theta}{\Delta \vdash c\langle\overline{\tau_1}\rangle \doteq c\langle\overline{\tau_2}\rangle \rightsquigarrow \theta}\ \textsc{Inst}$$

$$\frac{X^* \notin \mathrm{dom}(\theta) \qquad X^* \notin \mathsf{FV}(\tau_1, \ldots, \tau_n)}{\Delta \vdash X^* \doteq \mathsf{star}(\overline{\tau}) \rightsquigarrow \theta[X^* \mapsto \mathsf{star}(\overline{\tau})]}\ \textsc{Var-Star}$$

$$\frac{X \notin \mathrm{dom}(\theta) \qquad \Delta \vdash \overline{\tau_1} \doteq \overline{\tau_2} \rightsquigarrow \theta'}{\Delta \vdash X\langle\overline{\tau_1}\rangle \doteq c\langle\overline{\tau_2}\rangle \rightsquigarrow \theta[X \mapsto c] \cup \theta'}\ \textsc{Inst-Var}$$

$$\frac{\Delta \vdash R_1\ \mathsf{unify}_\rho\ R_2 \rightsquigarrow \theta_1 \qquad \Delta \vdash \tau_1 \doteq \tau_2 \rightsquigarrow \theta_2 \qquad \epsilon_1^\# = \epsilon_2^\#}{\Delta \vdash (R_1 \xrightarrow{\epsilon_1^\#} \tau_1) \doteq (R_2 \xrightarrow{\epsilon_2^\#} \tau_2) \rightsquigarrow \theta_1 \cup \theta_2}\ \textsc{Fun}$$

$$\frac{\forall k \in \mathrm{dom}(R_1) \cap \mathrm{dom}(R_2).\ \Delta \vdash R_1(k) \doteq R_2(k) \rightsquigarrow \theta_k}{\Delta \vdash R_1\ \mathsf{unify}_\rho\ R_2 \rightsquigarrow \bigsqcup_k \theta_k}\ \textsc{Record}$$

Figure 9: Unification rules, using dom **and fieldwise** unify_ρ **as in Section B.3. Function unification requires equal effects; other structured types (e.g., unions, classes) are unified componentwise. Unification is undefined when type constructors differ.**

matched-parameter sets, allowing partially applied overloads to be resolved without re-analyzing unrelated arms.

(2) **Dynamic selection:** At an actual call site, the saturated arms—residuals where FinishApp is applicable—from the relevant group are taken, and their return types are joined pointwise using the type join from Figure 6.

This mirrors Python's runtime dispatch but enables precise static joins over all viable arms.

Attribute and index access. Attribute access $\tau \cdot k$ is resolved by the row lookup rules of Section B.9. Indexing $\tau[i]$ is modeled as:

$$\tau[i] \;=\; \begin{cases} \tau \cdot \langle i \rangle & \text{if } \langle i \rangle \in \mathrm{dom}(\rho_\tau) \\ \mathrm{apply}(\tau \cdot \langle \text{"\_\_getitem\_\_"} \rangle,\ \mathrm{type}(i)) & \text{otherwise} \end{cases}$$

where keys $\langle i \rangle$ and $\langle$"__getitem__"$\rangle$ follow the syntax in Figure 3 and Section B.3.

Effects. An effect $\epsilon^\#$ is a tuple

$$\epsilon^\# = (\mathsf{new},\ \mathsf{update},\ \mathsf{points\_to\_args},\ \mathsf{bound\_method})$$

where:

- new $\in \{\mathrm{true}, \mathrm{false}\}$ indicates allocation of a new object.
- update $\in \mathcal{T}_\rho^?$ is an optional row type ρ of updated fields.
- points_to_args $\in \{\mathrm{true}, \mathrm{false}\}$ indicates whether the result may point to arguments (used for aliasing).
- bound_method $\in \{\mathrm{true}, \mathrm{false}\}$ marks bound methods produced by attribute access.

Effects form a finite product lattice $(\mathcal{E}, \sqsubseteq_\mathcal{E}, \sqcup_\mathcal{E}, \sqcap_\mathcal{E}, \top_\mathcal{E}, \bot_\mathcal{E})$. Joins are computed pointwise over Boolean components, and via an optional-type join $\sqcup^?$ for the update component:

$$\tau_1^? \sqcup^? \tau_2^? = \begin{cases} \tau_1 \sqcup \tau_2 & \text{if both defined (join as in Figure 6),} \\ \tau_i & \text{if only } \tau_i \text{ defined,} \\ \mathit{None} & \text{if neither defined.} \end{cases}$$

If the update component is a row type ρ, its join is computed using the ρ-join rules from Figure 6.

Decidability. The fragment used in this paper admits a syntax-directed, terminating subtyping procedure. Rows are finite, records use width/depth subtyping, functions are contravariant in parameters and covariant in results, overloads are finite, and effects are compared by strict equality rather than via a subeffect lattice. There is no nominal class subtyping and, crucially, no subtyping rule that crosses ∀-quantifiers (quantification is handled by instantiation/binding rather than by subtyping). The subtyping algorithm decreases a well-founded size measure on types/rows at every rule (fieldwise checks, list traversals, and result recursion), so subtyping terminates and is therefore decidable. This yields decidable limited type inference for the targeted Python subset.

These claims hold for the restricted system specified here: uniform rows for parameters and members, structural protocol checking via the same row machinery, finite overload resolution, and effects with equality—not joins. Features marked as escape hatches (e.g., any, dynamic attributes) do not introduce non-termination in the checker, and we intentionally avoid the known undecidable combinations (e.g., F<:-style bounded or F-bounded quantification with subtyping under ∀).

C Analysis Assumptions and Design Trade-offs

To ensure a sound and tractable static analysis of a highly dynamic language, our framework targets a well-defined and practical subset of Python programs. This section outlines the scope of our analysis and the key design trade-offs that balance implementation complexity with analytical precision.

Target Program Scope. Our analysis requires that programs adhere to the following properties, which enable a sound interpretation of control and data flow:

- **No Dynamic Code Evaluation:** Constructs such as `eval`, `exec`, and `getattr` with dynamically computed string arguments is disallowed. `getattr` with literal strings (e.g., `getattr(x, "mean")`) is supported, as it's semantically equivalent to a standard attribute access (`x.mean`). These features prevent the static resolution of the program's control-flow graph.
- **Statically-Resolvable Calls:** Function calls must be resolvable at analysis time using the provided type signatures. The framework does not model complex higher-order control flow where functions are passed as first-class values to unknown call sites.
- **Type annotations**: Like most contemporary type analyses, we do not track dimensionality of numpy's ndarrays (but see [23]). Unfortunately, this means the analysis cannot distinguish between subscriptions that return a float and subscriptions that return a slice of an ndarray (which is an object). We therefore use simple `get_float()` helpers for the former—a form of type annotations required from the programmer. Alternatively, the programmer can be trusted to use slice syntax 'a[i:]' if and only if the index is not an `ndarray` and the result is an `ndarray`; this allows standard function overloading to do its job.
- **Explicit Generic Instantiations:** To avoid relying on runtime type propagation, generic collections must be explicitly instantiated with their type parameters (e.g., `list[int]()`), especially when empty.
- **Simplified NumPy Aliasing:** The analysis assumes that NumPy array variables refer to distinct memory objects unless explicitly constructed via view-creating operations. It does not model mutation through view-based aliasing where multiple arrays may share the same underlying data buffer.
- **Operator implementation is regular:** operator lookup rules in Python are complex. Usually, however, one can assume that all candidate implementations do generally the same thing. Similar assumptions are required for soundness in type checkers such as MyPy.
- **No reentrance** functions called can be overapproximated using effect annotation, and do not depend on the behavior of the callsite except the information explicitly passed through arguments and self-binding.

These assumptions are satisfied by a wide class of numerical programs that follow idiomatic NumPy usage: preallocated buffers, explicit data copying, no implicit sharing, and simple iteration over typed arrays. These assumptions enable a sound, precise static analysis tailored to checkpointing in deterministic, structured Python programs.

Precision and Design Trade-offs. Within this scope, our analysis embodies several conscious design trade-offs. Some choices simplify the abstract domain at the cost of precision, while others introduce complexity to the type system to more faithfully model Python's idioms and avoid hardcoding.

- **Dimensionality-Agnostic Array Types:** The type system abstracts all `numpy.ndarray` objects as containing floating-point numbers but does not track their dimensionality or shape. This design greatly simplifies the typing of numerical operations but means the analysis cannot distinguish between a vector and a matrix, which in some cases may require additional user hints to ensure type precision.
- **Wildcard for Collection Elements:** To handle collections of arbitrary size and for accesses that are not precisely known, our pointer analysis models all element access (e.g., via subscripting) using a single wildcard field, `*`. This is efficient and scalable but merges the abstract state of all elements, meaning a write to one index will appear to affect all others.
- **Literal Types for Precision:** We chose to add complexity by incorporating literal types (e.g., `Literal["mean"]`) into the type system. While this makes the type hierarchy more complex, it enables a fully generic, type-driven model for attribute access. It allows the analysis to resolve expressions like `x.mean` by treating it as a subscription on `x`'s type with the literal value, avoiding hardcoded heuristics for method names.
- **Variadic Generics for Generality:** The type system supports variadic generics (e.g., `*Args`). This required a more complex unification algorithm but was useful to accurately model common Python constructors like `tuple()` without special-casing them in the analyzer. This design allows the framework to be more extensible and handle a wider range of idiomatic Python code in a principled way.
- **Reliance on Annotations:** The soundness of the analysis is contingent on the correctness and completeness of the provided type and side-effect annotations (e.g., new,update). This is particularly true for external library functions, which are modeled as black boxes whose behavior is determined entirely by these summaries. Verifying these annotations is currently a manual process.

D Pointer Analysis

D.1 Scope and Goal

The pointer analysis abstracts heap shape, aliasing, and mutations over the Spytecode IR (Section A) to support checkpointing. It is flow-sensitive and intraprocedural along exception-free paths. Instruction–level effects (allocation at Bind, ConstructTuple, ConstructDict, Call, and iteration in Unpack) are interpreted here. Callable *invocation* effects come from the type system (Section B) as an effect triple $\epsilon^{\#} = (\text{new}, \text{update}, \text{points\_to\_args})$ and are translated to heap updates by the transfer functions below.

D.2 Notation and State

We write basic blocks $l \in L$ and instruction positions i within block l, giving a program point l. Variables are split into named locals $\mathcal{V}_{\text{named}}$ and stack stack indices $\mathbb{N}$. Let $\text{Live}_{l,i} \subseteq \mathcal{V}_{\text{named}}$ be liveness at l.

Abstract objects and fields. Abstract objects O include

(1) `Loc` l — allocation site.
(2) `Param` p — external input.
(3) `Imm` τ — immutable of type τ.
(4) scopes —`LOCALS`, `GLOBALS`.

Fields $\mathcal{K}$ are keys from the IR: name(s), index(n), both(n, s), and $\star$ (wildcard).

Abstract domains (cf. Fig. 10).

Conventions. We store named local variables as fields of roots: $H[\texttt{LOCALS}][\text{name}(x)]$ for $x \in \mathcal{V}_{\text{named}}$, and $S[\$k]$ for $\$k \in \mathbb{N}$. We write O_x for a set of objects and $\biguplus$ for union over a family.

D.3 Auxiliary Updates

Weak/strong update on points-to:

$$\text{Upd}(H, O_{\text{tgt}}, f, O_{\text{new}}) = \begin{cases} \text{for the unique } o \in O_{\text{tgt}}: & \\ \quad H[o][f] \leftarrow O_{\text{new}} & \text{if } |O_{\text{tgt}}| = 1 \\ \text{for all } o \in O_{\text{tgt}}: & \\ \quad H[o][f] \leftarrow H[o][f] \cup O_{\text{new}} & \text{otherwise} \end{cases} \tag{1}$$

Dirty update: $\text{UpdD}(D, O_{\text{tgt}}, f)$ adds f to $D[o]$ for each $o \in O_{\text{tgt}}$.

We use two auxiliary resolvers:

$$\text{RetObjects}(l, i, \epsilon^{\#}, \overline{O_a}) = \begin{cases} \{\texttt{Loc}\ l\} & \text{if } \epsilon^{\#}.\text{new} = \text{true} \\ \left(\biguplus_k O_{a_k}\right) \cup \{\text{Ret}^{\uparrow}\} & \text{otherwise} \end{cases}$$

(where $\text{Ret}^{\uparrow}$ is a distinguished summary return object), and a target resolver for effect updates that follows argument/receiver links on bound callables:

$\text{Targets}(H, O_f, \text{slot}) \subseteq O$ (slot is receiver or an argument index).

D.4 Transfer Functions for Spytecode

The abstract state is a four-tuple $\Sigma = (H, S, T, D)$ where H is the heap points-to map, S is the stack map, T the object-type map, D the dirty-field map. We write $S[d \mapsto O]$ for functional update, and $\biguplus \mathcal{X}$ for set union over a family $\mathcal{X}$. Heap updates use $\text{Upd}(H, O_{\text{tgt}}, f, O_{\text{new}})$ (weak/strong update depending on $|O_{\text{tgt}}|$), and dirty marks use $\text{UpdD}(D, O_{\text{tgt}}, f)$. Let `Loc` l denote the allocation site at instruction index l. Pure operations never mutate H; the stack map S changes only for instructions that assign to a stack.

D.5 Transfer Functions for Spytecode

The abstract state is a four-tuple $\Sigma = (H, S, T, D)$ where H is the heap points-to map, T the object-type map, D the dirty-field map, and S the stack map for stack variables. Pure operations never mutate H. Only instructions that assign to a destination stack variable can mutate S.

Notes. (i) *Lookup purity.* Results of LookupDunder/ResolveOverload are static; we leave S unchanged and rely on the type component to constrain subsequent Bind/Call. (ii) *Binding vs. calling.* Bind allocates a bound-callable object b with explicit `args`/`kwargs` edges; Call interprets $\epsilon^{\#}$ to allocate results, propagate aliasing, and perform effect-guided updates on receivers/arguments discovered via *Targets*. (iii) *Subscriptions.* Indexing sugar lowers to *__getitem__*/*__setitem__* and thus uses the same Bind/Call machinery.

Heap Domain (H)

$$H \in \mathcal{H} = \mathcal{O} \to (\mathcal{K} \to \mathscr{P}(\mathcal{O}))|\bot_{\mathcal{H}}$$
$$P_1 \sqsubseteq P_2 \iff \forall o.\ P_1[o] \subseteq P_2[o]$$
$$(P_1 \sqcup P_2)[o] = P_1[o] \cup P_2[o]$$

Stack Domain (S)

$$S \in \mathcal{S} = \mathbb{N} \to \mathscr{P}(\mathcal{O})|\bot_{\mathcal{S}}$$
$$S_1 \sqsubseteq S_2 \iff \forall i \in \mathbb{N}.\ S_1[i] \subseteq S_2[i]$$
$$(S_1 \sqcup S_2)[i] = S_1[i] \cup S_2[i]$$

Type Domain (T)

$$T \in \mathcal{T} = \mathcal{O} \to \mathit{TypeExpr}|\bot_{\mathcal{T}}$$
$$T_1 \sqsubseteq T_2 \iff \forall o.\ T_1[o] \leq: T_2[o]$$
$$(T_1 \sqcup T_2)[o] = T_1[o] \sqcup T_2[o]$$

Dirty Domain (D)

$$D \in \mathcal{D} = \mathcal{O} \to \mathscr{P}(\mathcal{K})|\bot_{\mathcal{D}}$$
$$D_1 \sqsubseteq D_2 \iff \forall o.\ D_1[o] \subseteq D_2[o]$$
$$(D_1 \sqcup D_2)[o] = D_1[o] \cup D_2[o]$$

Combined domain (Σ)

$$\Sigma = (H, S, T, D) \in \mathcal{H} \times \mathcal{S} \times \mathcal{T} \times \mathcal{D}|\bot_{\Sigma}$$
$$\Sigma_1 \sqsubseteq \Sigma_2 \iff P_1 \sqsubseteq P_2 \wedge S_1 \sqsubseteq S_2 \wedge T_1 \sqsubseteq T_2 \wedge D_1 \sqsubseteq D_2$$
$$\Sigma_1 \sqcup \Sigma_2 = (P_1 \sqcup P_2,\ S_1 \sqcup S_2,\ T_1 \sqcup T_2,\ D_1 \sqcup D_2)$$

Figure 10: Abstract domains for heap analysis combining pointer tracking, type information, and mutation tracking.

D.6 Reachability, Pruning, and Dirty Roots

Roots at l are $R_{l,i} = \{\, H[\texttt{LOCALS}][\mathsf{name}(x)] \mid x \in \mathsf{Live}_{l,i} \,\}$. Reachability is the least fixed point

$$\mathsf{Reach}(R, H) = \mu X.\ R \ \cup\ \{\, o' \mid \exists\, o \in X,\ f \in \mathcal{K}.\ o' \in H[o][f] \,\}.$$

Abstract GC restricts to $V = \mathsf{Reach}(R_{l,i}, H)$:

$$\mathsf{GC}(\Sigma, R_{l,i}) = (H|_V,\ T|_V,\ D|_V).$$

Dirty roots:

$$\mathsf{DirtyRoots}(\Sigma, R_{l,i}) = \left\{\, x \in \mathsf{Live}_{l,i} \,\middle|\, \exists o \in \mathsf{Reach}(\{H[\texttt{LOCALS}][\mathsf{name}(x)]\}, H).\ D[o] \neq \emptyset \,\right\}.$$

D.7 Soundness

Let γ map abstract states to sets of concrete heaps. Soundness requires:

$$\forall \Sigma^{\#} = (H, S, T, D),\ \forall H \in \gamma(\Sigma^{\#}),\ \forall o, f.\quad H(o).f \in H[o][f].$$

Whenever the concrete semantics performs a write $o.f \leftarrow v$, the abstract transformer must mark $f \in D[o]$. Monotonicity of transformers and the product lattice yields a standard worklist fixpoint and a post-fixpoint invariant. Local simulation lemmas follow the Spytecode operational semantics (lookups are pure; Bind/Construct$*$/Call/Unpack introduce only the abstract updates shown above).

(A) Data movement

$$Trans^{\#}\big((H,S,T,D),\mathsf{Mov}(\$d,\$s)\big) = (H,\ S[\![d \mapsto S[s]]\!],\ T,\ D)$$

(B) Constant lookup

$$Trans^{\#}\big((H,S,T,D),\mathsf{LoadConst}(\$d,c)\big) = (H,\ S[\![d \mapsto \{\,\mathsf{Imm}(type(c))\,\}]\!],\ T,\ D)$$

$$Trans^{\#}\big((H,S,T,D),\mathsf{LookupDunder}(\$d,op,\$a)\big) = (H,S,T,D[\![d \mapsto \mathsf{lookup\_dunder}(T,op,S[\$a])]\!])$$

$$Trans^{\#}\big((H,S,T,D),\mathsf{ResolveOverload}(\$d,\$f,\$a,\$k)\big) = (H,S,T,D[\![d \mapsto \mathsf{resolve\_overload}(T,\ f,\ S[\$a],\ S[\$k])]\!])$$

(C) Attribute read and named variable access (pure w.r.t. heap)

$$Trans^{\#}\big((H,S,T,D),\mathsf{GetAttr}(\$d,\$o,f)\big) = \big(H,\ S[\![d \mapsto \mathsf{Attr}(H,S[o],f)]\!],\ T,\ D\big)$$

$$Trans^{\#}\big((H,S,T,D),\mathsf{LoadLocal}(\$d,x)\big) = (H,\ S[\![d \mapsto H[\mathtt{LOCALS}][name(x)]]\!],\ T,\ D)$$

$$Trans^{\#}\big((H,S,T,D),\mathsf{LoadGlobal}(\$d,x)\big) = (H,\ S[\![d \mapsto H[\mathtt{GLOBALS}][name(x)]]\!],\ T,\ D)$$

(D) Allocation schemata (allocate & wire, then write stack)

$$Trans^{\#}\big((H,S,T,D),\mathsf{ConstructTuple}(\$d,\overline{\$e}),\ q\big) = \big(H',\ S[\![d \mapsto \{\,\mathtt{Loc\ q}\,\}]\!],\ T',\ D\big)$$
$$\text{where}\quad (H',T') \stackrel{\text{def}}{=} \mathsf{AllocTuple}(H,T,\mathtt{Loc\ q},\ \langle S[\overline{\$e}]\rangle)$$

$$Trans^{\#}\big((H,S,T,D),\mathsf{ConstructDict}(\$d,\overline{(\$k,\$v)}),\ q\big) = \big(H'',S[\![d \mapsto \{\,\mathtt{Loc\ q}\,\}]\!],\ T'',\ D\ \big)$$
$$\text{where}\quad (H'',T'') \stackrel{\text{def}}{=} \mathsf{AllocDict}(H,\langle\overline{(S[\$k],S[\$v])}\rangle,T,\mathtt{Loc\ q})$$

$$Trans^{\#}\big((H,S,T,D),\mathsf{Bind}(\$d,\$f,\$a,\$k),\ q\big) = \big(H''',\ S[\![d \mapsto \{\,\mathtt{Loc\ q}\,\}]\!],\ T''',\ D\big)$$
$$\text{where}\quad (H''',T''') \stackrel{\text{def}}{=} \mathsf{AllocBind}(H,S[k],T,\mathtt{Loc\ q},\ S[a])$$

(E) Heap updates via fields and calls

$$Trans^{\#}\big((H,S,T,D),\mathsf{SetLocal}(x,\$s)\big) = \big(Upd(H,\{\mathtt{LOCALS}\},name(x),S[s]),\ S,\ T,\ D\big)$$

$$Trans^{\#}\big((H,S,T,D),\mathsf{Unpack}(\overline{\$d},\$s)\big) = \big(H,\ S',\ T,\ D\big) \qquad \text{where}\quad S'[d_j] \stackrel{\text{def}}{=} Ind(H,S[s],j)$$

$$Trans^{\#}\big((H,S,T,D),\mathsf{SetAttr}(\$o,f,\$v)\big) = \big(Upd(H,S[o],name(f),S[v]),\ S,\ T,\ UpdD(D,S[o],name(f))\big)$$

$$Trans^{\#}\big((H,S,T,D),\mathsf{Call}(\$d,\$f),\ q\big) = \big(H^*,\ S[\![d \mapsto O_{\text{res}}]\!],\ T^*,\ D^*\big)$$
$$\text{where}\quad (H^*,T^*,D^*,O_{\text{res}}) \stackrel{\text{def}}{=} \mathsf{Invoke}\big(H,T,D;\ \mathsf{Effect}(\$f),\ S[f],\ \mathsf{Args}(H,S,S[f]),\ q\big)$$

(F) Control (no state change)

$$Trans^{\#}\big((H,S,T,D),\mathsf{AssumeEq}(\$b,\$c)\big) = (H,S,T,D), \quad Trans^{\#}\big((H,S,T,D),\mathsf{Exit}\big) = (H,S,T,D)$$

Figure 11: Abstract transformer $Trans^{\#}$. Only destination-producing instructions write to the stack map S. Heap H is mutated at allocation sites (ConstructTuple, ConstructDict, Bind), explicit writes (SetLocal, SetAttr), and via Call according to $\epsilon^{\#}$. LoadConst, LookupDunder and ResolveOverload are pure.

$$\mathsf{Attr}(H, O, f) \stackrel{\text{def}}{=} \biguplus_{o \in O} H[o][name(f)]$$

$$\mathsf{Ind}(H, O, j) \stackrel{\text{def}}{=} \left(\biguplus_{o \in O} H[o][index(j)] \right) \cup \left(\biguplus_{o \in O} H[o][\star] \right)$$

$$\mathsf{AllocTuple}(H, T, t, \langle O_0, \ldots, O_{n-1} \rangle) \stackrel{\text{def}}{=} \Big(H[t][index(j)] \supseteq O_j,\ H[t][\star] \supseteq O_j\ \forall j;\ T[t] \sqcup = \mathsf{TupleType}(\langle O_j \rangle) \Big)$$

$$\mathsf{AllocDict}(H, T, d, \langle (K_m, V_m) \rangle_{m=1}^{M}) \stackrel{\text{def}}{=} \Big(\begin{array}{ll} H[d][name(s)] \supseteq V_m & \text{if } K_m = \{\, \mathsf{Imm}(\texttt{str}(s)) \,\}; \Big) \\ H[d][\star] \supseteq V_m & \text{otherwise;} \\ T[d] \sqcup = \mathsf{DictType}(\langle K_m, V_m \rangle) & \end{array}$$

$$\mathsf{AllocBind}(H, T, b, O_a, O_k) \stackrel{\text{def}}{=} \Big(H[b][\texttt{"args"}] \supseteq O_a,\ H[b][\texttt{"kwargs"}] \supseteq O_k;\ T[b] \sqcup = \mathsf{ResidualCallableType}(b) \Big)$$

$$\mathsf{Args}(H, S, O_f) \stackrel{\text{def}}{=} \text{actual argument objects resolved from } S \text{ and binding edges hanging off } O_f$$

$$\mathsf{Effect}(\$f) \stackrel{\text{def}}{=} \text{the effect summary } \epsilon^{\#} \text{ retrieved from the type component at } \$f$$

$$\mathsf{Invoke}(H, T, D;\ \epsilon^{\#}, O_f, O_a, q) \stackrel{\text{def}}{=} \text{apply } \epsilon^{\#}\text{: allocate results (fresh site } \texttt{Loc q} \text{ if needed),}$$
$$\text{propagate aliasing (} \star \text{ edges) when } \epsilon^{\#}.\text{points\_to\_args, and perform field updates}$$
$$\text{on targets discovered by } Targets(H, O_f, \cdot) \text{ using Upd and UpdD; also update } T.$$

Figure 12: Helper definitions for abstract transformer.