FlowUnion: Guarded Disjunctive Types for Path-Sensitive Tensor Analysis

Retaining branch-conditional tensor shapes across control-flow joins

Leo J. Paggen
Maastricht University
lpaggen@gmail.com

Abstract

Static type checkers routinely use control flow to refine types within individual branches, but these refinements are often collapsed when branches rejoin. For programs involving tensors, this loss of information can obscure shape invariants and introduce infeasible combinations of types. We present FlowUnion, a static analysis engine that preserves branch-conditioned type information across control-flow joins. Rather than maintaining complete execution paths, FlowUnion merges abstract states at Control Flow Graph (CFG) joins while representing affected values as guarded alternatives. Subsequent tensor operations combine and refine these alternatives locally using SMT solver-backed feasibility checks and produce precise diagnostics. The implementation of FlowUnion presented in this paper detects non-symbolic shape mismatches statically in Python programs containing conditional control flow and non-recursive user-defined function calls.

Keywords: tensor analysis, static analysis, Python, program verification, SMT solver
Leo J. Paggen. 2026. FlowUnion: Guarded Disjunctive Types for Path-Sensitive Tensor Analysis

1 Introduction

Python programs which use tensor libraries such as PyTorch can fail at runtime because of incompatible shapes or types. These errors generally lie outside of the scope of conventional Python type checkers, which reason about objects such as "Tensor" but typically do not track dimensions and relationships required to validate tensor operations statically. As a result, ill-formed tensor programs may execute until an incompatible operation is reached, exposing errors only at runtime and often one at a time.

FlowUnion is motivated by the possibility of moving this class of errors from runtime to static analysis. Our goal is to detect and report all statically identifiable failure points involving tensors and tensor operations in a singular analysis pass. For the class of programs supported by FlowUnion, described precisely below, the analyzer aims to prove that tensor operations are shape-compatible across every feasible control-flow alternative, or otherwise produce a precise diagnostic identifying where and under which conditions the errors arise.

In this work, we present and explain each component of FlowUnion, illustrating the type checking and shape-mismatch detection logic with worked examples throughout. First, we explain how Python source code is transformed into a custom Intermediate Representation (IR); secondly we explain how entire programs and their dependencies are represented internally; and lastly we provide a section that goes in depth in the type checking and shape-mismatch detection logic. A quantitative evaluation is presented in Section 6.

The prototype described in this paper analyzes straight-line code and single-level `if`/`else` branching, together with calls between non-recursive, module-level function definitions. Loops, class definitions, structural pattern matching (`match`), and function definitions nested inside other functions lie outside this subset; because FlowUnion does not yet detect their presence syntactically, encountering one of these does not raise an error, and can instead result in false negatives — real shape errors located in or downstream of the unsupported construct going undetected. Programs using lazy or conditional imports are rejected outright during import resolution (Section 4) rather than analyzed. A full discussion of these boundaries, including which are enforced explicitly versus silently, is given in Section 8.

2 Architecture Overview

FlowUnion is written in Rust and Python, acting as a backend and a frontend, respectively. We opt to structure our backend entirely in Rust due to the type safety and speed of the language, and we opt for a Python frontend because Python's standard library provides us with the "ast" library which simplifies the parsing of Python files. Between Python and Rust is a Protobuf layer which converts Python IR objects into their Rust equivalents.

The Python frontend overrides Python's standard AST node visitors with custom logic to convert Python statements and objects into what we refer to as a "Semantic IR"; objects

are defined according to Python's AST documentation[**?**] and are supplied a "Source Span" which contains the exact location of each object in the original source code.

The Protobuf layer was chosen mainly for its speed over other formats such as JSON. FlowUnion aims to provide results with low latency for the best user experience, so a fast translation layer from Python to Rust is crucial.

The Rust layer converts the Protobuf message into a Rust IR, which enables us to use pattern matching on the objects we receive, a crucial feature of Rust we leverage extensively in our codebase. Beyond the conversion layer, the Rust backend handles everything from CFG construction, analysis, import resolution, to error detection and reporting.

2.1 Implementation Variants

FlowUnion has two implementations sharing the same frontend and intermediate representation. The native backend uses the z3 SMT solver[3] for all guard and constraint satisfiability checks, currently restricted to concrete integer tensor shapes (Section 5.1). A second backend, compiled to WebAssembly, powers a browser-based demonstration that requires no local installation. Because embedding z3 into a WebAssembly build is not currently practical with the Rust crate used in this software, this build instead relies on a lightweight boolean satisfiability solver written specifically for this purpose. Since both backends are currently restricted to concrete integer shapes, the two builds agree on every example presented in this paper; the distinction becomes relevant only once symbolic shape support (Section 9) is added to the native backend.

3 Python Source Code to Rust Objects

We begin this section by explaining how our Python IR looks like before proceeding to explain how we encode it into Protobuf and how objects are decoded to our Rust IR. As such, the section is divided into three sub-sections outlining each separate process. The procedure explained in this section provides the reader with the information necessary to convert Python source code into Rust objects. An example of this end-to-end process is presented in figure 1.

3.1 A Custom Python IR

Each program owns a set of fields meant to represent entire Python programs without losing important information. For example, the official Python documentation describes function definitions (FunctionDef) as a statement with the fields presented in the upper half of figure 2, and we opt to mirror this representation with the addition of the span of the function along with some scope-specific identifiers to help FlowUnion locate symbols correctly in later phases (Section 4.1).

The same mapping is applied to every Python object present in the official documentation. Python programs are abstracted into a data structure which contains the program's imports, statements, symbols along with their scopes, and its own unique identifier which is used during import resolution (Section 4).

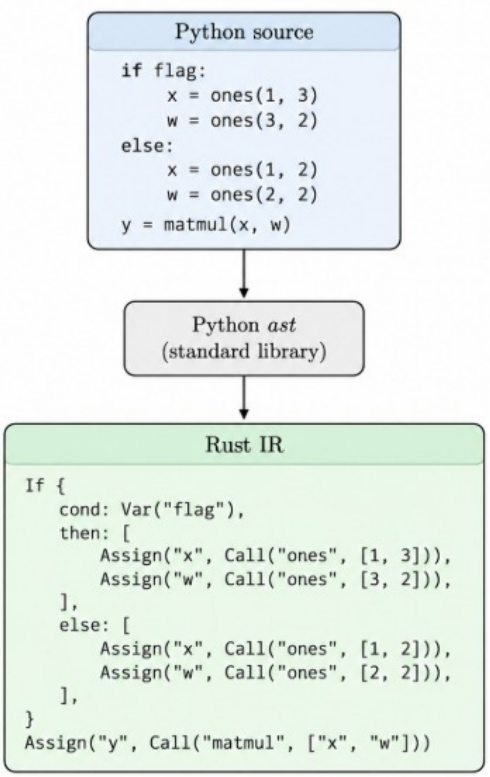

Figure 1. Python code example represented in our custom IR

3.2 Protobuf Translation Layer

Protobuf is a library developed by Google with the goal of transferring data across different machines at very fast speeds thanks to its custom binary format. Protobuf typically produces smaller artifacts than other serialization formats such as JSON[2].

The process we use is a two-step process: first, Python programs are converted to Protobuf messages; and secondly, the resulting Protobuf message is decoded into Rust IR objects which can be analyzed in later phases. We forgo the readily available data format provided by Protobuf in favor of data structures matching their Python counterparts as the latter allow us to fully use Rust's features, such as pattern-matching.

3.3 Decoding Protobuf Messages

Protobuf messages are decoded into Rust objects using the "Prost" Rust crate. FlowUnion is bundled with a decoder which analyzes Protobuf messages representing individual

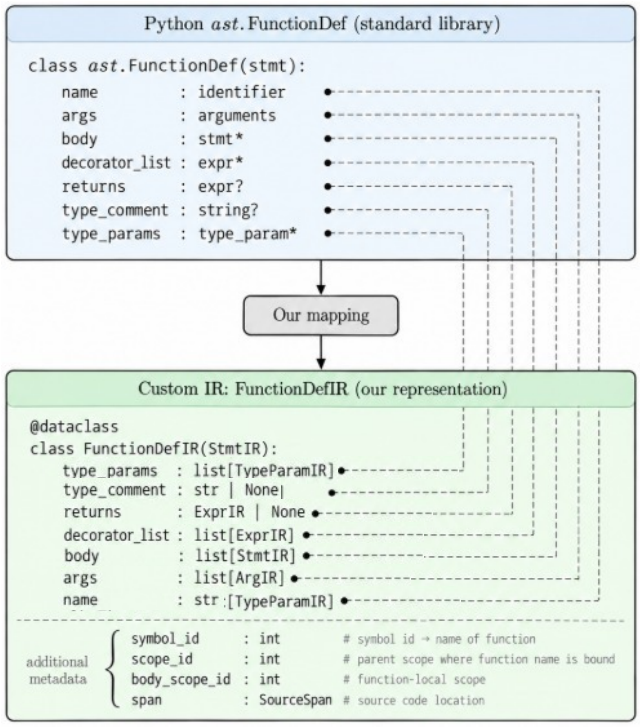

Figure 2. Official FunctionDef IR against our custom IR

Python programs in parallel. The procedure is straightforward, messages are received, and are recursively decoded to populate Rust-side program abstractions. Consider again our example presented in the previous sections. The attribute "returns" is converted into "Option<ExprIR>", and similarly "type_params" is converted into "Vec<ExprIR>", etc. The same is done for each object attribute; the Rust IR is a mirror of the custom Python IR.

4 Import Resolution

This section explains the process enabling FlowUnion to resolve local and supported external imports. Import resolution is necessary for later phases of the analysis to determine what a symbol encountered in the source program actually refers to. In particular, FlowUnion must recognize supported external operations such as PyTorch tensor constructors and operations regardless of how they are imported or aliased, while also resolving functions and variables imported from other local programs.

Import resolution allows FlowUnion to recognize objects from known third-party libraries such as PyTorch or Jax, and objects imported from other files in other Python programs.

4.1 Symbol Resolution

Each symbol of a program is assigned a unique identifier. Symbols are assigned a unique ID incrementally by the AST visitor, likewise programs are uniquely identified. Assuming a symbol with the lexeme "a" is the first symbol present in a program identified with the unique ID "2", we obtain the following mapping:

$$a \mapsto (1, 2)$$

Because functions and imports are equally treated as symbols, the same treatment applies to them. Local and external imports are differentiated:

$$\texttt{from torch import tensor} \mapsto \textit{local}$$
$$\texttt{from module\_a import A} \mapsto \textit{external}$$

4.2 External Imports and Aliases

The type resolver forces aliases to point to their non-aliased name. Expressions of the following form are supported:

```
import torch as th
```

In this case, "th" is given the same unique identifier as "torch", so that later phases may parse "th" and resolve it back to PyTorch. Similarly, the following style of import is permitted:

```
from torch import tensor as tn
```

The alias "tn" is assigned with a unique identifer which resolves back to the external target: "torch.tensor". FlowUnion supports all variations of Python imports, with one important exception: lazy imports. Lazy imports are syntactically valid, but they are not currently supported and their usage currently results in undefined behavior from the analyzer.

4.3 Local Imports

For a local import, one additional lookup is required. Consider the program "a":

```
from a import B
```

and the second program, named "b":

```
from b import A
```

Ignoring the possibility that this may be a circular import, executing this program generates the local symbol "B" inside module (todo explain module vs program) "a" and the local symbol "A" inside module "b". An implication of the way symbols are uniquely identified per module, the imported symbol "B" found in module "a" does not have the same identifier as the original symbol "B" found inside of module "b", the same fact holds for "A".

FlowUnion connects the two symbols by identifying the name of the program which it imports objects from, and accessing a table of the symbols which are lexically visible for importers. Consequently, "A" imported in module "b" is resolved to the local target "B" in module "b", vice-versa for "A". The table containing lexically visible symbols for

importers is built prior to the import resolution phase. This system enables the later type-checking phase (Section **??**) to locate symbols and resolve their types procedurally.

4.4 Locating Symbols

The type checker relies on this system in the following way. It first locates symbols using their unique identifier; if a symbol is not found, FlowUnion assumes the symbol was never declared in the relevant scope which needs to locate it. External targets are dispatched to supported library semantics, while local targets continue through the standard pipeline.

5 Flow-Dependent Typing System

FlowUnion introduces guarded types to Python. A guarded type is a symbol whose type depends on a logical predicate expressed in z3 SMT solver language. This allows the analysis to retain the set of conditions under which a symbol may hold a certain type. This system relies on Control Flow Graphs (CFGs) and a special union type to represent symbols whose type depends on execution flow. This union type allows the analyzer to express symbols as unions of guarded types.

Consider the program represented by figure 3. Blocks T and F eventually merge into a single block in which the type checker must infer the type of "y". Assume the conditional guard "flag?" which determines which one of blocks T or F is followwed during execution cannot be determined statically; FlowUnion avoids analyzing every path generated by "flag?" and instead represents "y" as:

$$\texttt{type(y)} \mapsto \texttt{(flag, float)} \cup \texttt{(!flag, str)}$$

The same system is used to express symbols whose types are tensors. User-defined functions are treated differently, as explained in section 5.8.

5.1 Supported Types

FlowUnion supports Python primitives. This includes integers, floats, booleans, strings, the special "None" type, bytes, complex numbers, and the special "Ellipsis" type. Tensor implementations of the PyTorch and Jax libraries are also supported. Tensor properties cannot be inferred directly from Python's type annotation system[1] (and therefore function calls, which permit annotations as return type hints). We therefore introduce two states to represent tensors: unresolved and resolved. The former is assigned to tensors whose shape and data type cannot be inferred trivially, while the latter represents tensors for which the shape is known as a combination of discrete (concrete integer, see section 8) values.

[1]Python allows annotating tensors, e.g. as `torch.Tensor`, but does not permit supplying dimensions in the annotation, discrete or otherwise.

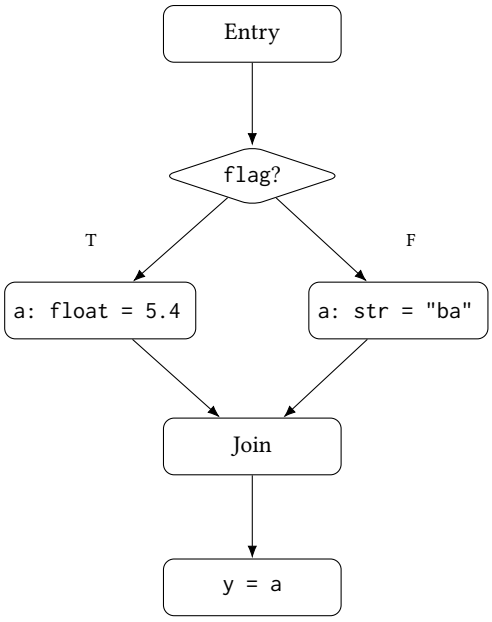

Figure 3. CFG join after branch-dependent assignments.

The shape of a tensor is treated as a combination of concrete values only when it can be soundly derived from existing program information (e.g. a literal tensor constructor). In unannotated Python, there is generally no principled basis for treating an otherwise-unresolved value as having a specific symbolic shape, since doing so would require the analysis to assume a rank or dimension it has no evidence for. FlowUnion therefore does not currently attempt symbolic shape tracking; Section 9 proposes a mandatory annotation syntax that resolves this by having the programmer state the shape rather than having the analyzer infer it.

The example presented in figure 4 highlights precisely why standard Python (as of version 3.14) does not provide the necessary features to determine tensor ranks statically.

Suppose for example the "read_csv" function returns a tensor of rank 2: "c = a + b" fails. This holds for any tensor whose rank is not 3. This topic and a solution to this issue are explained further in section 9.

5.2 Building CFGs

"Any static, global analysis of the expression and data relationships in a program requires a knowledge of the control flow of the program."[1]. CFGs are at the center of FlowUnion; they are the mechanism enabling the analyzer to produce the previously introduced union types.

CFGs allow us to represent arbitrary Python programs as directed graphs in which vertices are chunks of the program and edges represent the flow of the program's execution. The guarded-type propagation described in the remainder of this paper (Sections 5.5–**??**) currently operates fully only over

```python
import torch

# Shape depends on the contents of the file.
a: torch.Tensor = read_csv("some_path.csv")

# Explicitly 3D: shape (2, 2, 2).
b: torch.Tensor = torch.tensor([
    [[1, 2],
     [3, 4]],

    [[5, 6],
     [7, 8]]
])

c = a + b
```

Figure 4. Violation of the assumption that all tensors whose shape could not be determined statically have rank 2

`if`/`else`; `for` and `while` are recognized as terminators during CFG construction but their bodies are not yet analyzed for guarded tensor shapes (Section 8). Upon encountering such a statement, subsequent execution typically depends on boolean conditions, such as for example for-loops which keep iterating as long as a certain condition holds, or conditional branches which force execution flow to split across multiple branches holding their own condition. For example, consider the simple program of figure 5.

```python
import torch

flag = input(bool())

if flag:
    x = torch.tensor([[1, 2, 3]])      # [1,3]
    w = torch.tensor([
    [1, 2],
    [3, 4],
    [5, 6],
    ])                                  # [3,2]
else:
    x = torch.tensor([[1, 2]])          # [1,2]
    w = torch.tensor([
    [1, 2],
    [3, 4],
    ])                                  # [2,2]

y = torch.matmul(x, w)
```

Figure 5. Simple Python program involving conditional branching

The CFG for this program is shown in Figure 6. Both branches under the `flag` condition are merged into a single block regardless of which branch is followed. Joining the two

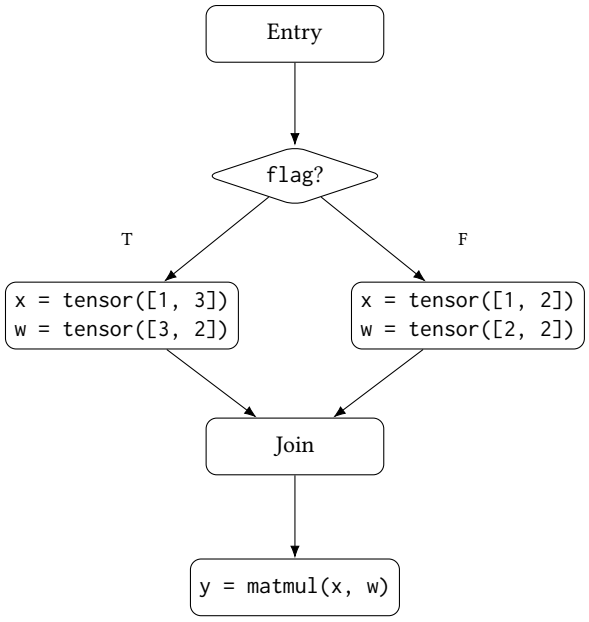

Figure 6. CFG join after branch-dependent tensor assignments.

states using a traditional type system may retain no more than:

$$\texttt{x: torch.Tensor}$$
$$\texttt{w: torch.Tensor}$$

Such a representation does not preserve information about the shapes of the tensors. A more informed, but still naive system may instead produce:

$$\texttt{x: [1,3]} \cup \texttt{[1,2]}$$
$$\texttt{w: [3,2]} \cup \texttt{[2,2]}$$

This merge admits infeasible combinations which cannot occur in a concrete execution because the shapes of "x" and "w" are related through the same branch condition.

FlowUnion preserves this information by associating each abstract tensor value with the branch predicate under which it was established. At the join, the two bindings therefore become:

$$\texttt{x: flag[1,3]} \cup \neg\texttt{flag[1,2]}$$
$$\texttt{w: flag[3,2]} \cup \neg\texttt{flag[2,2]}$$

The relationship established by the original branch is therefore preserved after the control-flow paths reconverge. When a subsequent tensor operation uses these values, FlowUnion reasons over the guarded alternatives and discards combinations that cannot correspond to a feasible execution.

The remainder of this section describes this process in detail.

5.3 Abstract State Representation

Each block of the CFG is attached to an environment which contains the information known by the analysis at a particular point in the CFG. We represent the environment E as

$$\texttt{R = (g, B, C)}$$

where g is the current control-flow guard, B maps symbols to typed bindings, and C is the logical formula accumulated by the analysis thus far. Each binding records both the abstract type of a symbol and its binding status, which is one of Bound, MaybeUnbound, or Unbound.

The guard g describes the control-flow condition under which the current abstract state is reachable. The formula C contains additional logical constraints learned during analysis. When no additional constraint has yet been established, C is simply `true`. Figure 7 represents a conditional branching under which this system can be observed. In this case, no constraints are accumulated in either branch, which leads to C being "true".

5.4 Transfer Through Branches

When a conditional terminator is encountered, FlowUnion creates separate successor states for the true and false edges.

Given a current guard g and branch condition p, the two successor guards are

$$g_T = g \wedge p \tag{1}$$

and

$$g_F = g \wedge \neg p. \tag{2}$$

For the running example, the entry guard is `true`. The two resulting guards are therefore

$$\texttt{true} \wedge \texttt{flag} \tag{3}$$

and

$$\texttt{true} \wedge \neg\texttt{flag}, \tag{4}$$

which simplify to `flag` and ¬`flag`, respectively.

Each successor state independently records the facts established along its own branch. The two states are siblings in the CFG: neither is derived from the other, and both represent alternative continuations from the same predecessor state. They remain logically disjoint until their control-flow paths reconverge at a later point.

5.5 Join Operation

When multiple predecessor states converge, FlowUnion merges their abstract states. For bindings whose abstract types agree across predecessors, the binding is preserved directly. When the types differ, the alternatives are retained together with the guards under which they were produced, yielding a guarded disjunctive type.

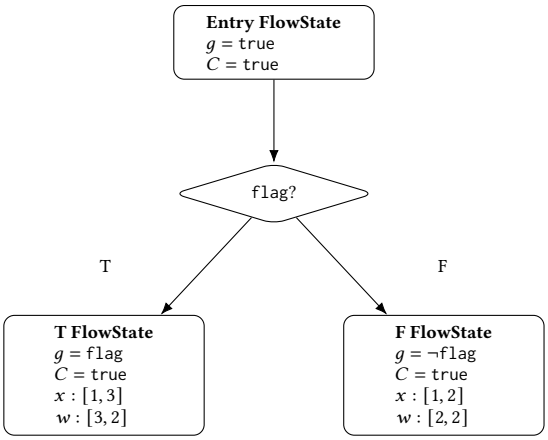

Figure 7. Propagation of abstract states across a conditional branch. The two successor states are mutually exclusive.

Assume the two abstract states of Figure 7 converge into a single state at the join shown in Figure 3. For x, the true branch provides `Tensor[1,3]` under guard `flag`, while the false branch provides `Tensor[1,2]` under guard `!flag`. The type of x therefore becomes:

$$x = \{\texttt{flag} \mapsto \texttt{Tensor}[1,3],\ \neg\texttt{flag} \mapsto \texttt{Tensor}[1,2]\}$$

Likewise, the merged binding for w becomes:

$$w = \{\texttt{flag} \mapsto \texttt{Tensor}[3,2],\ \neg\texttt{flag} \mapsto \texttt{Tensor}[2,2]\}$$

The resulting abstract state preserves not only the possible tensor shapes, but also the branch condition under which each shape is valid:

$$\begin{aligned} S_{\text{join}}: \quad & g = \mathit{true}, \\ & x = \{\texttt{flag} \mapsto \texttt{Tensor}[1,3], \\ & \qquad \neg\texttt{flag} \mapsto \texttt{Tensor}[1,2]\}, \\ & w = \{\texttt{flag} \mapsto \texttt{Tensor}[3,2], \\ & \qquad \neg\texttt{flag} \mapsto \texttt{Tensor}[2,2]\}. \end{aligned}$$

In this case, "g" is "true" because the block in which both states merge is reachable from either branch, and those incoming guards are:

$$\begin{aligned} \texttt{g}_T &= \texttt{flag} \\ \texttt{g}_F &= \neg\texttt{flag} \end{aligned}$$

Which when merged become:

$$\texttt{g}_{join} = \texttt{g}_T \vee \texttt{g}_F = \texttt{flag} \vee \neg\texttt{flag} = \texttt{true}$$

Consequently, the relationship between the values of x and w established by the original control-flow split is retained after the join.

5.6 Operations Over Guarded Values

Operations over guarded values must consider the alternatives retained at control-flow joins. Let $Alt(x)$ denote the guarded alternatives of x. In the running example,

$$Alt(x) = \{(\texttt{flag}, \texttt{Tensor}[1,3]), (\neg\texttt{flag}, \texttt{Tensor}[1,2])\},$$
$$Alt(w) = \{(\texttt{flag}, \texttt{Tensor}[3,2]), (\neg\texttt{flag}, \texttt{Tensor}[2,2])\}.$$

To evaluate `matmul`, FlowUnion forms the Cartesian product

$$Alt(x) \times Alt(w),$$

yielding four candidate operand pairs:

$$(\texttt{flag}, [1,3]) \times (\texttt{flag}, [3,2]),$$
$$(\texttt{flag}, [1,3]) \times (\neg\texttt{flag}, [2,2]),$$
$$(\neg\texttt{flag}, [1,2]) \times (\texttt{flag}, [3,2]),$$
$$(\neg\texttt{flag}, [1,2]) \times (\neg\texttt{flag}, [2,2]).$$

For each pair, FlowUnion constructs a candidate guard

$$g' = g_s \wedge g_x \wedge g_w, \tag{5}$$

where g_s is the current state's guard and g_x and g_w are the guards attached to the selected alternatives of x and w, respectively.

At this join, $g_s = true$. The four candidate guards therefore simplify to:

$$(\texttt{flag} \wedge \texttt{flag}) = \texttt{flag} \mapsto SAT$$
$$(\texttt{flag} \wedge \neg\texttt{flag}) = \texttt{false} \mapsto UNSAT$$
$$(\neg\texttt{flag} \wedge \texttt{flag}) = \texttt{false} \mapsto UNSAT$$
$$(\neg\texttt{flag} \wedge \neg\texttt{flag}) = \neg\texttt{flag} \mapsto SAT$$

Only candidates with satisfiable guards are retained. The two cross-branch combinations are therefore discarded, leaving exactly the operand pairs corresponding to feasible control-flow alternatives. Once a combination has been established as feasible, FlowUnion applies the constraint imposed by the tensor operation itself. For two-dimensional matrix multiplication, the last dimension of the left operand must equal the first dimension of the right operand:

$$x[-1] = w[0]. \tag{6}$$

For the two feasible alternatives of the running example, the solver therefore receives:

$$(\texttt{flag} \wedge (\texttt{3 == 3})) = \texttt{true} \mapsto SAT$$
$$(\neg\texttt{flag} \wedge (\texttt{2 == 2})) = \texttt{true} \mapsto SAT$$

Both paths are satisfiable. FlowUnion therefore establishes that the matrix multiplication is shape-compatible under every feasible control-flow alternative.

5.6.1 Operation Failure Under a Feasible Guard.

Consider a variation of the example in which w has shape $[3,2]$ under both branches:

$$x = \{\texttt{flag} \mapsto \texttt{Tensor}[1,3]\} \cup \{\neg\texttt{flag} \mapsto \texttt{Tensor}[1,2]\},$$
$$w = \{\texttt{flag} \mapsto \texttt{Tensor}[3,2]\} \cup \{\neg\texttt{flag} \mapsto \texttt{Tensor}[3,2]\}.$$

After infeasible cross-branch combinations are discarded, the two remaining alternatives are:

$$\{(\texttt{flag}, [1,3]) \times (\texttt{flag}, [3,2]),\ (\neg\texttt{flag}, [1,2]) \times (\neg\texttt{flag}, [3,2])\}$$

The control-flow guards `flag` and `!flag` are both satisfiable. Applying the matrix multiplication constraint produces:

$$(\texttt{flag} \wedge (\texttt{3 == 3})) = \texttt{true} \mapsto SAT$$
$$(\neg\texttt{flag} \wedge (\texttt{2 == 3})) = \texttt{false} \mapsto UNSAT$$

The second result does not imply that the `!flag` branch is unreachable. Indeed,

$$\neg\texttt{flag} \mapsto SAT$$

remains satisfiable. The conjunction becomes unsatisfiable only after introducing the constraint required by `matmul`. The false branch therefore represents a feasible execution in which the tensor operation itself is invalid.

This distinction is fundamental to the analysis: an unsatisfiable combination of guards represents an execution combination that cannot occur, whereas an unsatisfiable operation constraint under a satisfiable guard represents a genuine error on a feasible control-flow alternative.

5.7 Nested Conditional Control Flow

A consequence of our CFG join-mechanism is that our system carries on to nested conditional branches naturally. Each time execution crosses a branch, the guard of the successor state is constructed by conjoining the current guard with the new branch predicate.

Consider:

```
if a:
    if b:
        x = torch.tensor([[1, 2, 3]]) # [1,3]
    else:
        x = torch.tensor([[1, 2]]) # [1,2]
else:
    x = torch.tensor([[1, 2, 3, 4]]) # [1,4]

w = torch.tensor([
[1, 2],
[3, 4],
[5, 6],
]) # [3,2]

y = torch.matmul(x, w)
```

At the outer branch, the true successor receives guard a, while the false successor receives $\neg a$. When the nested condition b is encountered while the current guard is already a, the two nested successor guards become

$$a \wedge b \tag{7}$$

and

$$a \wedge \neg b. \tag{8}$$

After the branches reconverge, the abstract binding for x becomes:

$$\{(a \wedge b) \mapsto \texttt{Tensor}[1,3]\} \cup \{(a \wedge \neg b) \mapsto \texttt{Tensor}[1,2]\} \cup \{\neg a \mapsto \texttt{Tensor}[1,4]\}$$

The guards therefore accumulate the sequence of control-flow decisions under which each abstract value was established.

Note that this is also the source of the analysis's worst-case combinatorial cost: a binding touched by n nested branches can carry up to 2^n guarded alternatives, and an operation combining two such bindings (Section 5.6) pays a Cartesian-product cost in the number of resulting SMT queries. See Section 7 for how this compares to full path enumeration.

When "matmul(x,w)" is analyzed, the operation-specific shape constraint is checked independently under each feasible guarded alternative:

$$a \wedge b \wedge \texttt{(3 == 3)}\ \ SAT$$
$$a \wedge \neg b \wedge \texttt{(2 == 3)}\ \ UNSAT$$
$$\neg a \wedge \texttt{(4 == 3)}\ \ UNSAT$$

The first alternative is shape-compatible, while the remaining two correspond to feasible control-flow alternatives under which matrix multiplication fails. FlowUnion therefore retains the precise conditions under which each error occurs rather than collapsing the possible shapes of x into a single unguarded union.

5.8 Interprocedural Analysis

During CFG construction, FlowUnion treats function definitions differently than ordinary symbols. First, a fresh state is created for the function and the function's parameters are bound to this state with their respective type if provided, and are bound as "unknown" otherwise. A function body is treated like any other module-level CFG, guards and constraints are recorded and solved like in the previous examples. FlowUnion creates a contract for every function definition it encounters. A function contract holds the information callers are required to provide to call the function, and what they can expect in return. For example, consider the program defined in figure 8

In this example, encountering "def project(x: torch.Tensor, y: torch.Tensor) at line 3 creates a contract enforcing callers to supply two tensors of the Torch library for a valid call. The return type of the function is automatically inferred; it is the result of the multiplication of the tensors "x" and "y". FlowUnion does not enforce that the last dimension of "x" match the first dimension of "y", as this makes a strong assumption on the ranks of "x" and "y". While these parameters may be treated as symbolic dimensions in theory, we opt to specialize function contracts at call-sites; this is more flexible and makes no assumption about the dimensions of the tensors being passed to the function. In this case, "project" is called at line 29 when "chain" is called, the arguments "x" and "w1" are passed to "project" and constraints are checked against these call-site arguments:

$$\texttt{[1,2] @ [2,3]} \mapsto \texttt{(2 == 2) = true}\ \ SAT$$

Function contracts are cached upon being called; if "project" is called with tensors with the same properties are supplied to the function multiple times, a contract already exists for the function and analysis is skipped. Upon calling a function for which a specialized contract does not yet exist, FlowUnion pauses the CFG execution and resumes it from the start of the block which it was analyzing. This is not optimal, and in the future work will be done to resume from the exact statement it was analyzing instead (section 9).

We argue that FlowUnion needs to dynamically produce function contracts if it aims to support standard Pyhon syntax with the example of figure 9.

assuming b is passed as a tensor, the matrix multiplication is valid under different types of (a), namely if it is a tensor or an integer. If (a) is an integer, then no constraints are generated, the operation succeeds, but if a is also a tensor, then the analyzer needs to treat the function call as an abstract state analysis in which (a) and (b) are already bound and verify whether the types of (a) and (b) given the function's constraints result in a feasible return type or not.

5.9 Diagnostics

When an operation-specific constraint is unsatisfiable under a satisfiable guard, FlowUnion records the failure rather than discarding the corresponding control-flow alternative. Because the operation is associated with its source span and the analysis retains the guard under which the failure occurred, the diagnostic can identify both the location and the condition responsible for the error.

For the failing matrix multiplication introduced previously, FlowUnion may report:

```
Line 23, column 5
incompatible shapes for matmul:
[1,2] @ [3,2]
   ^ --- ^
2 != 3 (when !flag)
```

```python
import torch

def project(x: torch.Tensor, w: torch.Tensor):
    return torch.matmul(x, w)

def chain(x, w1, w2):
    first = project(x, w1)
    return project(first, w2)

# [1,2]
x = torch.tensor([[2, 3]])

# [2,3]
w1 = torch.tensor([
    [1, 2, 3],
    [4, 5, 6],
])

# first = [1,2] @ [2,3] -> [1,3]

# [3,2]
w2 = torch.tensor([
    [1, 2],
    [3, 4],
    [5, 6],
])

# result = [1,3] @ [3,2] -> [1,2]
result = chain(x, w1, w2)
```

Figure 8. Example of a chained function call

```python
import torch

def foo(a, b: torch.Tensor):
    return torch.matmul(a, b)
```

Figure 9. Legal Python function signature

The diagnostic therefore distinguishes a genuine shape error on a feasible branch from combinations of guarded values that were discarded because their guards were themselves unsatisfiable.

A complementary diagnostic for tensor operations with an unresolved (rather than infeasible) operand is discussed in Section 8.

6 Evaluation

[NEEDS WRITING — This section currently sits empty because of the relatively small subset of python we support. it will be expanded once we can reliably analyze bigger libraries]

7 Related Work

[NEEDS WRITING — More related work will be added once I freeze the project, there is a lot of similar work but much changes every day with this project.]

7.1 Python Type Checkers

The closest prior work is PyTea [4], a static analyzer for PyTorch code. PyTea statically traces every possible execution path through a program, collects the tensor-shape constraints implied by the tensor operations along each path, and checks them for unsatisfiability with an SMT solver; its authors report that, empirically, the number of surviving execution paths rarely grows large on real PyTorch code after their pruning step.

FlowUnion differs from this approach in what it enumerates. Rather than tracking one abstract state per execution path, FlowUnion merges to a single abstract state per program point and instead tracks, per program binding, a set of guarded alternative values reaching it (Section 5.5). As shown in Section 5.7, a binding touched by n nested branches can still carry up to 2^n guarded alternatives in the worst case, and an operation combining two such bindings pays a Cartesian-product cost in the number of SMT queries it issues (Section 5.6); FlowUnion's worst case is therefore not asymptotically better than full path enumeration. The practical difference is that this cost is local to the bindings a branch actually affects: a branch unrelated to any tensor value contributes no guarded alternatives, and CFG traversal itself remains a single pass regardless of how many guards are in flight elsewhere in the program. In PyTea's approach, by contrast, a path is inherently a whole-program concept spanning every branch encountered, so a branch with no bearing on any tensor shape can, in principle, still multiply the global path count unless specifically pruned.

7.2 Non-Python Systems

Liquid Types[5] pursues a closely related idea: they use SMT-backed constraints to preserve and infer value-dependent information that ordinary type would otherwise lose. FlowUnion applies this idea to tensor shapes and expresses branch-conditioned tensor types explicitely as guarded alternatives rather than collapsing them into a single refinement, but some idea from *Liquid Types* are potentially usable for FlowUnion if we pursue the strongly annotated system discussed in section 9. Under some conditions, *Liquid Types* is able to create reusable refinements (contracts) from generic functions by collecting constraints in the function's body. This system would enable callers to comply to strict function contracts rather than having to perform a new analysis every time a tensor or otherwise is passed to a function.

8 Limitations

FlowUnion's soundness holds only over the language subset described in Section 1. We distinguish between failure modes that are enforced explicitly and ones that are not yet enforced at all.

Rejected outright. Lazy and conditional imports are detected during import resolution (Section 4) and the enclosing program is rejected before analysis begins.

Silently unsupported (false-negative risk). Loops, class definitions, `match` statements, and function definitions nested inside other functions (Section 5.8) are not currently detected as out-of-scope constructs. FlowUnion still produces a report for programs containing them, but may silently omit real shape errors located in, or affected by, these constructs.

Unresolved tensor operands. A tensor whose shape cannot be determined (e.g. the result of an unannotated function argument, or of a call such as `to_tensor(input())` (Section 5.1)) is classified as `Tensor::Unresolved`. FlowUnion does not attempt to assign such a value a concrete or symbolic shape, since doing so in unannotated Python has no principled basis: the analysis has no information from which to justify a specific rank or dimension, and forcing a default (e.g. always assuming rank 2) would be equally unjustified. Currently, an operation applied to an `Unresolved` operand is skipped rather than checked, and no diagnostic is emitted.

Symbolic tensor shapes. FlowUnion does not currently support tensors whose shape includes a symbolic (non-integer-literal) dimension, on either backend (Section 2.1). The design reasoning is given in Section 5.1; Section 9 proposes a mandatory shape-annotation syntax that would resolve this by having the programmer state the shape rather than having the analyzer infer it.

9 Future Work

Loop support. Extending CFG construction and the guard-based fixed-point analysis (Section **??**) to loop back-edges.

Guard and alternative subsumption. Merging guarded alternatives that resolve to the same concrete type despite having distinct guards, as a way to bound the combinatorial growth discussed in Section 7.

Class and `match` support. Extending the IR and CFG construction to model class definitions – in particular `torch.nn.Module` subclasses, which represent the large majority of real-world PyTorch model code – and structural pattern matching.

A lightweight mandatory shape-annotation syntax. A small extension to Python's annotation syntax allowing declarations such as `tensor_a: "A@2B" = some_input()`, where each `@`-separated slot names a symbolic dimension variable, optionally with an integer coefficient (`2B` denotes a dimension that is always exactly twice whatever `B` resolves to at the call site).

Importantly, this style of annotation is not valid Python. Currently, Python 3.14 will not cause a runtime error if such annotations are provided, but this feature is part of the "future" module and is therefore subject to change. Implementation is straight-forward, but presently we cannot guarantee that standard type checkers will accept such annotations.

This type of language would reinforce the current system's capabilities by allowing FlowUnion to fully use z3's theory on tensors and other objects while minimizing the burden of annotation on the programmer; annotations are kept simple and refer strictly to tensor dimensions. One advantage of using such a system is that it can reuse much of the existing code developed for this project; parsing remains the same, the IR remains the same, only at type inference do we need to add a small parser for atomic operations such as "2@B" or "B@4H@P", potentially tensors of arbitrary ranks can be modeled easily.

Such a system would have to assert facts about relationships between tensors and their dimensions as it collects constraints. For example, if a function call establishes a relation between two symbolic dimensions at any point in the program and no prior relation existed, it would be assumed to hold throughout the entire program:

```
1  import torch
2
3
4  def project(x: "A@K", w: "J@B") -> "A@B":
5      return torch.matmul(x, w)
6
7  y = project(x, w1)   # "2@4"
8
9  w_bad: "5@4"
10
11 y = project(x, w_bad)
```

Figure 10. Caption

"matmul" introduces the additional constraint:

$$K = J,$$

so FlowUnion can infer the reusable contract

$$\texttt{A@K} \times \texttt{J@B} \wedge K = J \rightarrow \texttt{A@B}.$$

Should the programmer then instantiate the following tensors:

$$x1 : \text{"2@3"} = \texttt{some\_function()}$$
$$w1 : \text{"3@4"} = \texttt{other\_function()}$$

calling "project" and passing "x" and "w1" as arguments would result in a valid call because $K = 3$ and $J = 3$, so

the inferred constraint is satisfiable. The stronger function contract created for "project" could eliminate invalid calls without requiring a new analysis of the function's body. Suppose the programmer calls $project(x, w\_bad)$ with:

```
w_bad: "5@4" = some_function()
```

The call requires the following formula to be satisfiable:

$$K = J \wedge K = 3 \wedge J = 5,$$

but it is unsatisfiable given these arguments, so FlowUnion reports a shape mismatch.

This type of system would not require the programmer to annotate every tensor they create, as assigning targets to function calls with tensors whose properties are already known results in FlowUnion being able to assign the return type of said function to the target. The system would also resolve the issue raised in section 5.8 which discussed strong and invalid rank assumptions.

Unresolved-operand diagnostics. Described in Section 8; noted here as it is likely the smallest and most immediately actionable item in this list.

10 Conclusion

[NEEDS WRITING — Will complete once results are frozen.]

References

[1] Frances E. Allen. 1970. Control flow analysis. *ACM SIGPLAN Notices* 5, 7 (July 1970), 1–19. doi:10.1145/390013.808479

[2] The Protobuf Authors. [n. d.]. https://protobuf.dev/reference/rust/

[3] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[4] Ho Young Jhoo, Sehoon Kim, Woosung Song, Kyuyeon Park, DongKwon Lee, and Kwangkeun Yi. 2021. A Static Analyzer for Detecting Tensor Shape Errors in Deep Neural Network Training Code. arXiv:2112.09037 [cs.LG] https://arxiv.org/abs/2112.09037

[5] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. *SIGPLAN Not.* 43, 6 (June 2008), 159–169. doi:10.1145/1379022.1375602