2024-4-(x): Manuscript for presentation at IPSJ-SIGPRO, d 1 2025.

Dynamic Version Checking for Gradual Updating

Satsuki Kasuya[1,a)] Yudai Tanabe[1,b)] Hidehiko Masuhara[1,c)]

Abstract: Programming with Version (PWV) is a programming paradigm that allows programmers to safely utilize multiple versions of the same package within a single program, facilitating flexible version updates of dependent packages. Existing PWV languages ensure consistent version usage so as not to break software behaviors by leveraging the type system of the base language. However, dynamically typed languages need a mechanism to support multiple versions with an efficient method of ensuring consistent version usage without a type system. To introduce PWV features into dynamically typed languages, we propose a dynamic version checking (DVC) mechanism. It records version information in a value, propagates it during evaluation, and checks inconsistency using version information recorded in values. When an inconsistency is detected, the mechanism suggests how to modify the program to resolve potential semantic errors from the inconsistency. We develop Vython, a Python-based PWV language with DVC, and implement its compiler. The compiler translates a Vython program into a Python program with bitwise operations. Our performance measurement shows the DVC mechanism's overhead is scalable and acceptable for small programs but requires further optimization for real-world use. Additionally, we conduct a case study and discuss future directions to facilitate smoother updates in practical development.

Keywords: Software maintenance, Software migration, Dependency management, Compiler, Python

1. Introduction

Updating the version of upstream packages is one of the most troublesome tasks for downstream developers [15], [18]. An incompatible new version can break the behavior of downstream programs [10], [14]. Each new release of upstream packages requires downstream developers to assess its impact and modify their source code accordingly.

Replacing an upstream package with its new version is automated by package managers such as pip[*1] in Python. For example, developers using NumPy [11] can automatically install the latest version by running `pip install --upgrade numpy`. Many developers benefit from this automation, as packages like NumPy are widely used across various domains, such as data analysis, deep learning, and image processing, in libraries like Pandas [27], PyTorch [22], and OpenCV [4].

Downstream developers carefully coordinate existing programs to update fundamental packages such as NumPy. The first major update of NumPy, version 2.0.0, was released in 2024. If any of the packages in use depends on NumPy 1.x series, the automatic installation of NumPy 2.0.0 via pip will fail. Manual installation, which is possible from the source, can break the existing behavior of downstream programs unintentionally, as some NumPy functions are incompatible with the old ones (see Appendix A.1).

Programming with Versions (PWV) [17], [24], [25] is a recent proposal designed to enable a gradual transition to new versions, thereby reducing update costs. The key ideas of PWV are (1) the simultaneous use of multiple versions, and (2) language mechanisms (i.e. types) that check version compatibilities. PWV languages ensure that programs use values created by compatible versions.

While previous research realized PWV in statically-typed languages, this research explores methods implementing PWV functionalities in dynamically-typed languages. To achieve this, we propose *dynamic version checking* (DVC) to alert when values of incompatible versions are used together at runtime. The DVC mechanism facilitates developers' communication regarding incompatibilities [16]; upstream developers specify compatibility for each function, allowing downstream developers to assess the impact of updates on their software through warnings.

The contributions of this paper are summarized as follows:

- We developed DVC, which records version information within values, propagates it during an evaluation, and detects inconsistencies based on the recorded version data.
- We implemented a Vython compiler, a PWV language with DVC. The Vython compiler translates Vython programs into Python programs, and the DVC functions are compiled into efficient bitwise operations.
- We evaluated the runtime performance of the Vython compiler. The results showed that Vython is scalable and acceptable for debugging small programs as an offline analysis tool but requires further optimization for

[1] Institute of Science Tokyo
Ookayama 2-12-1, Meguro, Tokyo 152–8552, Japan
a) satsuki.kasuya@prg.is.titech.ac.jp
b) yudaitnb@prg.is.titech.ac.jp
c) masuhara@acm.org

[*1] pip: The PyPA recommended tool for installing Python packages. `https://pip.pypa.io/` (Accessed December 6, 2024)

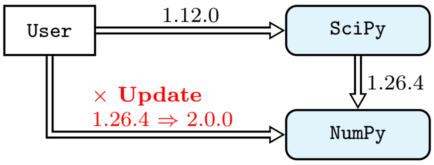

Fig. 1: Dependencies of the `User` program.

```
1 class SciPy: # SciPy 1.12.0:
2   def place_poles(A, B, poles):
3     return NumPy().solve(..) # Using NumPy 1.26.4
```

```
1 def my_place_poles(A, B, poles): # User Program
2   return NumPy().solve(..) # Using NumPy 2.0.0
3 NumPy().array_equal(
4   my_place_poles(A, B, poles),
5   scipy.place_poles(A, B, poles) ) # => False
```

Fig. 2: A program that uses NumPy and SciPy in Python

real-world use.

- We conducted a case study using gradual update scenarios with the Vython compiler. Based on these findings, we discussed how Vython could work effectively in more practical scenarios.

The rest of the paper is organized as follows. Section 2 introduces inflexible update scenarios that motivate our research, and Section 3 offers an overview of the proposed features. Section 4 explains the semantics of the DVC mechanism, and Section 5 details the implementation of the Vython compiler. Section 6 presents the performance evaluation and case study conducted using the compiler. Finally, Sections 7 and 8 discuss related work and provide concluding remarks.

2. Motivating Example

Consider a scenario where we update a user program that reimplements a function for solving pole placement problem[*2] and test its behavior against `SciPy` [26] implementation. Figure 1 shows the dependencies of the `User` program. `User` depends on `SciPy` version 1.12.0, which indirectly depends on `NumPy` 1.26.4, and `User` directly depends on `NumPy` and attempts to update it from version 1.26.4 to 2.0.0.

As shown in Figure 2, both `SciPy` and `User` use the `solve` function from `NumPy`[*3]. In `User` (Figure 2 bottom), `my_place_poles` is implemented using the `solve` function, and its results are compared against the existing implementation in `SciPy`. `place_poles` in `SciPy` 1.12.0 (Figure 2 top) directly returns the result of the `solve` function. We try to update `NumPy` in the `User` project.

Updating NumPy via pip

This attempt fails as follows.

```
$ pip install numpy==2.0.0
ERROR: scipy 1.12.0 requires numpy<1.29.0,>=1.22.4, but
    you have numpy 2.0.0 which is incompatible.
```

The error message indicates that this attempt resulted in broken dependencies, as the installed `SciPy` is locked to `NumPy` versions below 1.29.0. Similarly, other Python package managers, such as poetry[*4] and pyenv[YT: pyenv is not for version management for user package, for python version instead][*5], also conservatively reject the installation of multiple versions of the same package within the development environment.

Updating `NumPy` from the Source

A potential workaround for using `NumPy` 2.0.0 without waiting for `SciPy` updates is to use `NumPy` 1.26.4 for `SciPy` and 2.0.0 for `User` independently. As mentioned earlier, while standard Python package managers do not support the installation of multiple versions of a package simultaneously, the `importlib` package[*6] in the Python standard library allows dynamic switching to specific package versions (see Appendix A.2).

However, the workaround using the `importlib` package involves dynamically modifying module objects in `sys.modules`, which can result in unpredictable program behavior. For example, subtle differences between the two versions of `NumPy` could lead to unintended behavior in the `User` program. The `solve` implementation was incompatibly changed in the `NumPy` 2.0.0 release. As explained in Appendix A.1, the ambiguous broadcasting rule was corrected in 2.0.0, so the `solve` function in the two versions may return different outputs even with the same input. As a result, the test of `my_place_poles` against `place_poles` in Figure 2 line 5 fails, even if both implementations are logically the same.

Identifying the cause of this failure is challenging. Current build systems lack mechanisms to detect the mixed use of incompatible implementation versions. [24] Additionally, such incorrect version usage is often reported as Python semantic errors, which fail to pinpoint the root cause stemming from version incompatibilities. Consequently, programmers must engage in tedious tasks such as reading release notes and reviewing implementations of all upstream packages. To mitigate such unfavorable situations, re-importing the `NumPy` through the `importlib` package triggers a warning message cautioning against its use due to the risk of unpredictable behavior as follows.

```
UserWarning: The NumPy module was reloaded (imported
    a second time). This can in some cases result in
    small but subtle issues and is discouraged.
```

[*2] This is a common task in control theory, placing closed-loop poles in desired locations to control the system response. The SciPy implementation can be found at SciPy Manual. https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.place_poles.html (Accessed December 6, 2024)

[*3] These programs are simplified, but are essentially identical to the actual implementation. For more details, see Appendix A.2.

[*4] Poetry, https://python-poetry.org/ (Accessed December 6, 2024)

[*5] pyenv/pyenv, https://github.com/pyenv/pyenv (Accessed December 6, 2024)

[*6] importlib — The implementation of `import`, https://docs.python.org/3/library/importlib.html (Accessed December 6, 2024)

```
class NumPy!1.26.4():
  def solve(self, A, B):
    return res
```

```
class NumPy!2.0.0():
  def solve(self, A, B):
    return res.incomp(..)
```

```
class SciPy!1.12.0():
  def place_poles(A, B, poles):
    return NumPy!1.26.4().solve(..) # Using NumPy 1.26.4
```

```
1 def my_place_poles(A, B, poles): # User Program
2   return NumPy!2.0.0().solve(..) # Using NumPy 2.0.0
3 array_equal(
4   my_place_poles(A, B, poles),
5   SciPy!1.12.0().place_poles(A, B, poles) ) # => Warning!
```

Fig. 3: A program that uses NumPy and SciPy in Vython

```
    spec.loader.exec_module(numpy)
```

3. Safely Use Multiple Versions in Vython

Vython is a Python subset that implements the PWV update model (*gradual updating*), which splits the burden of program modifications caused by package updates. Vython is designed to mitigate the version-locking problem by enabling version selection at individual code sites and provides debugging information to help address incompatibility errors with mixed package versions through the following features:

- **Using multiple versions in a code**: The programmer can selectively use multiple versions of a class definition by specifying a version when instantiating.
- **Dynamic version checking (DVC)**: Vython records information about the class and its version used for creating a value, ensuring that programs use values created with consistent versions.

Vython differentiates multiple class versions internally, allowing for their selective use. As shown in Figure 3, the current naive implementation requires version annotations in the surface language. Additionally, DVC is intended to be enabled only in debug mode. Vython has a production mode that deploys programs without runtime checks.

Vython provides a mechanism for upstream developers to specify compatibility, which is utilized in DVC as follows.

Upstream Developer Specifies Compatibilities in Code

In Vython, upstream developers are responsible for specifying incompatibilities. In NumPy 2.0.0 (Figure 3 top right), the NumPy developer uses `incompatible()` (denoted as `incomp()` below for brevity) to mark an expression as incompatible with previous versions. Additionally, upstream developers can provide guidance (as shown below) to help downstream developers. This information is recorded along with the class definition in the source code.

```
[Changed in 2.0.0] (How it differes from 1.26.4)
```

Notifying Downstream Developers of Incompatibility Causes

The downstream developer using both NumPy versions benefits from DVC and the guidance for updates specified by the NumPy developer. In the user program (Figure 3 bottom), the DVC mechanism reports runtime warnings (as shown below) on lines 3-5 because `array_equal` uses values derived from incompatible versions of the `solve` function.

```
Incompatible version usage found in Lines 3-5:
  - NumPy 1.26.4
  - NumPy 2.0.0
[Changed in version 2.0] `NumPy().solve(a,b)`:
  - If `b` is 1-dim, it is treated as a column vector (M,).
  - Otherwise, it is treated as a stack of (M, K) matrices.
  - Previously, `b` was treated as a stack of (M,) vectors if `b.ndim` equaled `a.ndim - 1`.
```

4. Vython Semantics for DVC

4.1 Intuition to Vython Semantics

Vython associates version information with each object and ensures that method return values are created from a combination of consistent version implementations. This version information reflects the versions of implementation used to create the object. Version information is recorded in a format called version table (VT). Additionally, Vython considers any value derived from another object created using version V as also originating from version V. This subsection illustrates the principle behind the design decision through examples that clarify its principles.

4.1.1 Version Tables Recorded in Objects

In Vython, all class instances will record the information of their instantiated class and its version when it is created. For example, the following program creates an instance of the `NumPy` class in Vython, specifying version `2.0.0`.

```
x = NumPy!2.0.0()
```

The Vython runtime with DVC records version information in the `NumPy` instance, indicating that it was made from version `2.0.0` of the `NumPy` class. The VT of an instance of `NumPy` version `2.0.0` immediately after instantiation is represented as follows.

`class`	NumPy
`version`	2.0.0
`flag`	-

The `class` and `version` fields indicate the class and version of the implementation used to create the value, and the `flag` field indicates any incompatibility of the implementation with other versions. Here, the `flag` is set to -, and the VT is simply recording class and version information.

4.1.2 Version Information Propagation

At the time of returning a value from method invocation or field access, a runtime with DVC appends the version information of the receiver object to the return value. The following program performs a method invocation on an instance of `NumPy` version `2.0.0` on line 8.

```
1 class NumPy!2.0.0():
2   def add(self, a, b):
3     res = a + b
4     return res
5
6 x = NumPy!2.0.0()
7 a, b = 1, 2
8 y = x.add(a, b)
```

The version information recorded in the value created by `x.add(a, b)` merges the version information of the return value from the `add` method (on line 7) with the version information of the value bound to `x`, as follows.

`class`	Int	NumPy
`version`	1	2.0.0
`flag`	-	-

Here, the records for Int and NumPy come from the VTs of the return value of the `add` method and the `NumPy` instance bound to `x`, both of which are shown below.

`class`	Int
`version`	1
`flag`	-

`class`	NumPy
`version`	2.0.0
`flag`	-

4.1.3 Dynamic Version Checking

Vython checks whether the value is created using consistent version implementations on the return values of method calls and field accesses. If the value is created using potentially incompatible implementations, Vython considers the method call to be a usage of incompatible implementation and issues a warning to the programmer. The following program compares arrays created by incompatible implementations of the `solve` method.

```
1 class NumPy!1.26.4():
2   def solve(self, A, B):
3     return res
```

```
1 class NumPy!2.0.0():
2   def solve(self, A, B):
3     return res.incomp(..)
```

```
1 def array_equal(arr1, arr2):
2   ..
3   arr1 == arr2
4   ...
5 a, b = [..]
6 res1 = NumPy!1.26.4().solve(a, b)
7 res2 = NumPy!2.0.0().solve(a, b)
8 array_equal(res1, res2)
```

As described in Section 3, in Vython, upstream developers can use the `incomp` method to explicitly declare the incompatibility introduced in version `2.0.0` of the `solve` function, thereby encouraging downstream developers to use the `solve` function with caution regarding its incompatibility. On lines 6 and 7, Vython records the following VTs for the values bound to `res1` and `res2`, respectively.

`class`	NumPy	...
`version`	1.26.4	...
`flag`	-	...

VT of the value bound to `res1`

`class`	NumPy	...
`version`	2.0.0	...
`flag`	True	...

VT of the value bound to `res2`

Note that the `flag` field of the new column is set to True for the value bound to `res2` (right). For VTs where the `flag` field is set to True, Vython performs consistency checking at the `return` point of method calls and field accesses. For example, on line 8, the `array_equal` function is called with `res1` and `res2` as arguments. Inside the `array_equal` function, on line 3, the equality of the vectors `arr1` and `arr2` is checked. At this point, Vython performs a consistency checking on the return value of `==` on line 3 and outputs a warning and refactoring hints set by the upstream developer, as described in Section 3.

4.1.4 Difference between Vython and Existing PWV Languages

An important design decision for DVC is that the version information of function arguments is not directly reflected in the return value. This design decision leads to differences in consistency-checking capabilities compared to the existing PWV languages. For example, in cases where an argument is merely discarded during the function body, the version information of the argument is not propagated into the version information of the return value.

```
1 class A!1.0.0(): ..
2 class A!2.0.0(): ..
3 class Choose!1():
4   def discard_snd(self, fst, snd):
5     return first
6
7 c = Choose!1.0.0()
8 x = c.discard_snd(A!1.0.0(), A!2.0.0())
```

As shown in line 5, the `discard_snd` method does not use `snd` in the actual computation, and therefore, the VT of the return value does not include the VT of the second argument. On the other hand, existing PWV languages such as VL [23], [24], [25] and BatakJava [17] conservatively reflect the version information of arguments in the return value.

Another interesting aspect of DVC is path sensitivity.

```
1 class A!1.0.0(): ..
2 class A!2.0.0(): ..
3 if rand():
4   x = A!1.0.0()
5 else:
6   x = A!2.0.0()
7 y = x
```

With the DVC mechanism, the recorded version information in the value bound to `y` in line 7 is only the one associated with the path actually taken. Therefore, upon the completion of the above program's execution, the VT of the value assigned to `y` is the lower left VT if the `then` branch is executed, and lower right VT if the `else` branch is executed, as shown below.

`class`	A
`version`	1.0.0
`flag`	-

through `then` branch

`class`	A
`version`	2.0.0
`flag`	-

through `else` branch

In contrast, In existing PWV languages, a type-system-based analysis is performed for the above example; that is, the analysis enforces the result where the versions (and types) of the `then` and `else` branches completely match (meet), or become the join of both.

Vython semantics $\boxed{t \Downarrow v}$

$$\frac{t \Downarrow v \quad v = C(\overline{v_i}) \quad f_i \in \mathsf{fields}(C) \quad vt = v.\mathsf{get}()}{t.f_i \Downarrow v_i.\mathsf{set}^{\mathsf{wf}}_{\cup}(vt)}\ \text{(E-Field)} \qquad \frac{t_i \Downarrow v_i \quad vt = \mathsf{mk}(C, V, -)}{C!V(\overline{t_i}) \Downarrow C(\overline{v_i}).\mathsf{set}(vt)}\ \text{(E-New)} \qquad \frac{t \Downarrow v \quad vt = \mathsf{mk}(C, V, \mathbf{T})}{t.\mathsf{incomp}(C, V) \Downarrow v.\mathsf{set}_{\cup}(vt)}\ \text{(E-Incomp)}$$

$$\frac{t \Downarrow v \quad v = C(\ldots) \quad t_i \Downarrow v_i \quad \mathsf{mbody}(m, C) = (\overline{x_i}, t_b) \quad t_b[\overline{v_i/x_i}] \Downarrow v' \quad vt = v.\mathsf{get}()}{t.m(\overline{t_i}) \Downarrow v'.\mathsf{set}^{\mathsf{wf}}_{\cup}(vt)}\ \text{(E-Invk)} \qquad \frac{\mathsf{op} \in \mathsf{BuiltinOp} \quad \mathsf{op}(v_1, v_2) = v \quad vt_1 = v_1.\mathsf{get}() \quad vt_2 = v_2.\mathsf{get}()}{\mathsf{op}(v_1, v_2) \Downarrow v.\mathsf{set}_{\cup}(vt_1).\mathsf{set}^{\mathsf{wf}}_{\cup}(vt_2)}\ \text{(E-BuiltinOp)}$$

$$\text{where} \quad v.\mathsf{set}^{\mathsf{wf}}_{\cup}(vt) \triangleq vt' \leftarrow \mathsf{join}(vt, v.\mathsf{get}());\ \mathsf{wf}(vt');\ v.\mathsf{set}(vt')$$
$$v.\mathsf{set}_{\cup}(vt) \triangleq vt' \leftarrow \mathsf{join}(vt, v.\mathsf{get}());\ v.\mathsf{set}(vt')$$

Fig. 4: Vython semantics with the highlight of VT operations. get and set are getters and setters for the VT of each object, and they are assumed to be defined in all classes. BuiltinOp represents the set of operators (op) for Python's built-in types, such as `+`, `or`, and `=`, etc. The operator op also is a meta-level function for constant objects v_1 and v_2. For example, when $\mathsf{op} = +$, $\mathsf{op}(\mathsf{Int}(1), \mathsf{Int}(2)) = \mathsf{Int}(3)$ that has an empty VT.

4.2 The Vython Semantics

This section presents the Vython semantics, which are defined through DVC functions over VT. Before detailing the DVC extensions applied to the underlying Python semantics, we begin by defining version tables.

4.2.1 DVC Functions for VT

We define VT as follows.

Definition 4.1 (Version Tables)**.** A version table (VT) is a set of triples $\{\overline{(C, V, f)}\}$, where C represents the class, V represents the version associated with a constructor, method, or field implementation, and f indicates incompatibility with other versions of the implementation when $f = \mathbf{T}$.

Note that each VT does not have duplicate triples and the order of recording triples does not relate to the meaning of the VT.

Next, we define DVC functions for VTs. All operations on the VT are performed using the following functions.

Definition 4.2 (DVC functions)**.** The mk function takes a class name C, a version number V, and a flag f, and creates a VT of size 1. The join function takes multiple VTs and creates a union set of VTs. The wf function takes a VT and checks whether the VT is well-formed or not.

$$\mathsf{mk}(C, V, f) = \{(C, V, f)\} \qquad \mathsf{join}(vt_1, vt_2) = vt_1 \cup vt_2$$

$$\mathsf{wf}(vt) = \begin{cases} \texttt{incomp} & \exists V_1, V_2.\ V_1 \neq V_2 \\ & \quad \wedge\ (C, V_1, \mathbf{T}) \in vt \\ & \quad \wedge\ (C, V_2, f) \in vt \\ \texttt{comp} & \text{otherwise} \end{cases}$$

Here, `incomp` indicates that the value with the associated VT was produced by a computation involving a value originating from an incompatible class. At the implementation level, a warning is issued at the time when wf is called. Conversely, `comp` indicates that the value associated with the VT is derived from values generated with consistent class versions, allowing evaluation to proceed without any warnings.

To aid the reader's understanding, we provide several examples using DVC functions.

Example 4.1 (VT operations (join and mk))**.**

$$\begin{aligned} & \mathsf{join}(\mathsf{mk}(\text{NumPy}, 2.0.0, \mathbf{T}), \mathsf{mk}(\text{NumPy}, 2.0.0, \mathbf{T})) \\ = & \{(\text{NumPy}, 2.0.0, \mathbf{T})\} \\ & \mathsf{join}(\mathsf{mk}(\text{NumPy}, 2.0.0, \mathbf{T}), \mathsf{mk}(\text{NumPy}, 2.0.0, \text{-})) \\ = & \{(\text{NumPy}, 2.0.0, \mathbf{T}), (\text{NumPy}, 2.0.0, \text{-})\} \\ & \mathsf{join}(\mathsf{mk}(\text{NumPy}, 2.0.0, \mathbf{T}), \mathsf{mk}(\text{Array}, 1.0.3, \text{-})) \\ = & \{(\text{NumPy}, 2.0.0, \mathbf{T}), (\text{Array}, 1.0.3, \text{-})\} \end{aligned}$$

Example 4.2 (VT consistency checking (wf))**.**

$$\begin{aligned} \mathsf{wf}(\{(\text{NumPy}, 2.0.0, \mathbf{T}), (\text{NumPy}, 2.0.0, \text{-})\}) &= \texttt{comp} \\ \mathsf{wf}(\{(\text{NumPy}, 2.0.0, \mathbf{T}), (\text{NumPy}, 1.26.4, \text{-})\}) &= \texttt{incomp} \\ \mathsf{wf}(\{(\text{NumPy}, 2.0.0, \mathbf{T}), (\text{NumPy}, 1.26.4, \mathbf{T})\}) &= \texttt{incomp} \end{aligned}$$

4.2.2 The Vython Semantics Using DVC Functions

Using DVC functions, we define the Vython semantics. The current Vython is a minimal object-oriented language because it lacks support for abstract classes, interfaces, class inheritance, and method overriding.

To highlight VT operations, we present the big-step operational semantics as an extension of standard object-oriented language semantics like Featherweight Java [12] in a somewhat informal manner. Figure 4 illustrates the semantics for an excerpt of the Vython syntax. The meta-variables t, v, C, m, and f represent terms, values, method names, and field names, respectively. The meta-functions fields and mbody represent the lookup of field names and method arguments with their bodies for a given class. Method calls $t.m()$ and field accesses $t.f$ follow the standard notation, but note that $C!V(\overline{t_i})$ and $C(\overline{v_i})$ denotes instance creation (term) and class instance (value).

Furthermore, while the syntax allows the incompatible tag $t.\mathsf{incomp}(C, V)$ to be applied to any terms, we assume it will be used specifically in the return statements of methods or constructors. The arguments C and V represent the class and version in which the method or constructor is defined, and they are expected to be automatically inserted during

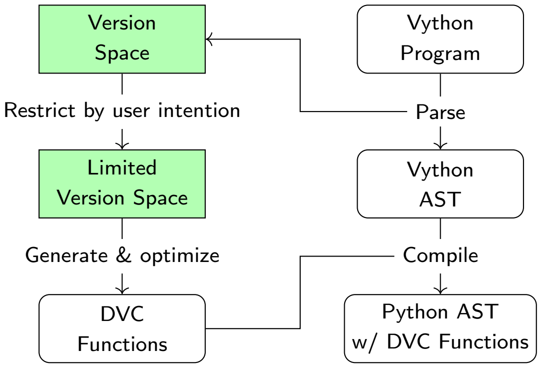

Fig. 5: Compilation flow of the Vython compiler.

the compiler's preprocess.*7

As discussed in Section 4.1, (E-Field) and (E-Invk) follow similar patterns since both field accesses and method calls involve accessing an instance's attributes. The VT of a method call's return value is determined by concatenating the VTs recorded in two components: the method's return value and the receiver object. Similarly, the VT for a field access result is derived by concatenating the VTs recorded in the field-assigned value and the target object. Lastly, both rules include a consistency check using wf.

(E-BuiltinOp) defines the rule for VT operations on built-in type values, such as boolean and numerical operations. In Vython, these operations are treated differently from method calls, as they are considered computations that always use the argument values. Therefore, the VTs of all arguments are joined into the VT of the resulting value.

(E-Incomp) defines the VT semantics for the incompatibility tag. The incomp function is the only mechanism for assigning the **T** flag, which triggers consistency checking at a wf call. In contrast, (E-New) specifies the VT to be recorded for a newly created class instance, recording a $\mathsf{mk}(C, V, -)$. Unlike (E-Incomp), the attached checker flag here is $-$, meaning it only records version information without triggering DVC except in the cases where computations involve values with **T** flag assigned by incomp.

5. Implementation

5.1 The Vython Compiler

We implemented the Vython compiler which translates the Vython program into the Python program. In the Python programs generated after compilation, the DVC mechanism is achieved by representing a VT as an attribute of each object and operating and checking VTs by predefined DVC functions, which are inserted against class methods, field accesses, and class instantiation. Since we treat all values as objects, Vython literals are compiled into predefined Python classes.

Figure 5 shows the compilation flow of the Vython compiler. First, the Vython program is parsed into a Vython

```
class A!1():
  def __init__(self, value):
    self.value = value
  def get(self):
    return self.value
```

(a)

```
1 class A_v_1:
2   def __init__(self, value):
3     self.vt = 1
4     self.value = value
5
6   @_vt_invk
7   def get(self):
8     return _vt_field(self, self.value)
9 ..
10 def _vt_invk(func):
11   def wrapper(*args, **kwargs):
12     result = func(*args, **kwargs)
13     if result is not None:
14       result = _vt_join(result, args[0])
15       if not _vt_well_formed(result):
16         _issue_warning(result, args[0])
17     return result
18   return wrapper
```

(b)

Fig. 6: Simple Vython program (a) before and (b) after compilation.

AST using the parser library lark*8. During parsing, the compiler extracts version information for classes and constructs version space (available class and version pairs) globally. Programmers can further restrict the size of version tables by specifying the classes they want to use with multiple versions as input to the compiler. The limited version space is then used to generate DVC functions. Finally, the Vython AST is compiled into Python AST, along with the insertion of generated DVC functions.

The Vython compiler inserts DVC functions by each evaluation rule. Figures 6a and 6b illustrate a program before and after compilation for a class `A` version 1, which only includes getter `get`. As shown in line 6 of Figure 6b, the VT operation for method calls is implemented through the decorator function `_vt_invk`, and the VT operation for field accesses is implemented through `_vt_field` in line 8. Additionally, as shown in line 3, the constructor bit-encodes the initial VT and assigns it to the `vt` field (mk).

Among the implementation functions introduced so far, we will use the implementation of `_vt_invk`, which corresponds to (E-Invk), as an example to explain. When a method with this decorator is invoked, the `_vt_invk` function performs VT operations according to (E-Invk). First, the method is executed in line 12. Then, VT propagation to the return value is handled by `_vt_join` (join) in line 14, and the VT consistency of the return value is checked by `_vt_well_formed` (wf) in line 15.

5.2 Optimization

To avoid the significant runtime overhead anticipated,

*7 The current Vython compiler does not yet support this feature. In the current implementation, incomp calls require both C and V in a surface program.

*8 `lark-parser/lark`, `https://github.com/lark-parser/lark` (Accessed December 6, 2024)

Algorithm 1: VT Encoding Algorithm

Data:
- $\mathcal{V}$: Restricted versions space, a map of class names to available versions $\{C_i \mapsto \overline{[V_{ij}]}\}$
- $vt \in \{(C_i, V_{ij}, f_i) \mid C_i \in \mathsf{dom}(\mathcal{V}), V_{ij} \in \mathcal{V}[C_i], f_i \in \{\mathbf{T}, -\}\}$

Result: A bit sequence b representing the vt under $\mathcal{V}$.

begin
  $|b| \leftarrow 2 \times \sum_{C \in \mathsf{dom}(\mathcal{V})} |\mathcal{V}[C]|$;
  `// ` $|\mathcal{V}[C]| = 2$ `in our assumption`
  for $i \leftarrow 0$ **to** $|b| - 1$ **do**
    $b[i] \leftarrow 0$;
  foreach $(C_i, V_{ij}, f_i) \in vt$ **do**
    $\mathit{offset} \leftarrow 2 \times (j + \sum_{k=0}^{i-1} |\mathcal{V}[C_k]|)$;
    `// ` $|\mathcal{V}[C_k]| = 2$ `for any` k `in our assumption`
    if $f_i = \mathbf{T}$ **then**
      $\mathit{index} \leftarrow \mathit{offset} + 1$;
    else if $f_i = -$ **then**
      $\mathit{index} \leftarrow \mathit{offset}$;
    $b[\mathit{index}] \leftarrow 1$;
  return b;

Vython compiles VTs into bit sequences and DVC functions into combinations of bitwise operations.

5.2.1 Encoding VTs as Bit Sequences

Algorithm 1 shows the algorithm for encoding a VT into a bit sequence. The algorithm takes as input the version space $\mathcal{V}$ (limited in size by the user) and the vt to be encoded, and outputs the bit sequence representation of vt under $\mathcal{V}$.

Note that the current implementation of Vython is restricted to two versions per class within a program. This limitation is based on the assumption that, in most cases, programmers are primarily concerned with compatibility between two specific versions. By leveraging this assumption, a bit-encoded VT is a bit sequence whose length is equal to four times the number of classes in the limited version space. Every grouped set of four bits records what version of a certain class was used to create the value.

For example, consider a program whose resulting limited version space consists of NumPy versions 1.26.4 and 2.0.0.

Example 5.1 (Bit-encoded VT)**.** The following version table is encoded into the bit sequence `1101` under the version space $\{\text{NumPy} \mapsto [1.26.4, 2.0.0]\}$.

`class`	NumPy	NumPy	NumPy
`version`	2.0.0	2.0.0	1.26.4
`flag`	True	-	-

The length of the bit sequence encoding the VT, $|b|$ in Algorithm 1, is 4. The first, second, third, and fourth bits correspond versions 1.26.4 with flag $-$, 1.26.4 with flag $\mathbf{T}$, 2.0.0 with flag $-$, and 2.0.0 with flag $\mathbf{T}$, respectively. Following the three elements contained in the version table, the encoding algorithm returns a bit sequence `1101` with the first, third, and fourth bits set to `1`.

5.2.2 Encoding Helper Functions as Bitwise Operations

Along with the encoding of VT into a bit string, DVC functions are also encoded into bitwise operations. The three DVC functions, mk, join, and wf are represented using bitwise operations as follows.

Definition 5.1 (DVC functions (bit-encoded))**.**

$$\mathsf{mk}(C, V, f) = [\![\{(C, V, f)\}]\!]_{\text{bit}} \quad \mathsf{join}(vt_1, vt_2) = vt_1 \mid vt_2$$

$$\mathsf{wf}(vt) = \begin{cases} \texttt{incomp} & (((vt \gg 1) \mathbin{\&} vt) \gg 1 \\ & \mid (vt \gg 3) \mathbin{\&} vt) \\ & \mathbin{\&} \mathsf{mask} \neq 0 \\ \texttt{comp} & \text{otherwise} \end{cases}$$

where mask is a bit sequence of the form $(0001)+$.

The mk function is compiled into a bit sequence determined by $[\![*]\!]_{\text{bit}}$, the encoding function defined in Algorithm 1. This function is evaluated at compile time, producing hard coded bit sequences in the Python AST, thus incurring no runtime overhead with $[\![*]\!]_{\text{bit}}$. The join function is simply compiled into a bitwise OR operation (`|`).

Compared to the other two DVC functions, wf is less straightforward and requires a detailed explanation. The purpose of wf is to detect when a VT element $(C, V_1, \mathbf{T})$ exists and another element (C, V_2, f) is present in the input VT, where $V_2 \neq V_1$ and f can be either $\mathbf{T}$ or $-$, returning `incomp` as a result. Assuming each class has only two versions, this detection corresponds to recognizing one of the following bit patterns.

- `1 _ _ 1`
- `1 _ 1 _`
- `_ 1 1 _`

Furthermore, we focus on (E-Incomp), which, as previously noted, is the sole mechanism for assigning the $\mathbf{T}$ flag, and optimizing its behavior at the implementation level. As defined in Definition 4.2, if a VT element $(C, V_1, \mathbf{T})$ exists in vt, adding $(C, V_1, -)$ – an element with the same class name and version name – to vt does not affect the results of the wf functions with other VT elements, such as (C, V_2, f). Accordingly, Vython implements (E-Incomp) with $vt = \mathsf{join}(\mathsf{mk}(C, V, \mathbf{T}), \mathsf{mk}(C, V, -))$ in the premise.

By leveraging this optimization, the bit pattern `1 _ 1 _` will always simplify to `1 1 1 1`, which is subsumed by the other two patterns. Therefore, it suffices to detect the following two patterns:

- `1 _ _ 1`
- `_ 1 1 _`

To achieve this, the `1 _ _ 1` pattern is detected by $(((vt \gg 1) \mathbin{\&} vt) \gg 1) \mathbin{\&} \mathsf{mask} \neq 0$, and the `_ 1 1 _` pattern is detected by $((vt \gg 3) \mathbin{\&} vt) \mathbin{\&} \mathsf{mask} \neq 0$. Further optimization is performed to eliminate redundant operations, resulting in the current wf definition in Definition 5.1.

6. Evaluation

6.1 Performance Evaluation of Debugging Mode

We conducted preliminary experiments on runtime performance. This performance evaluation focuses on the overhead introduced by DVC in Vython's debug mode. In contrast, the production mode (without the DVC feature) in-

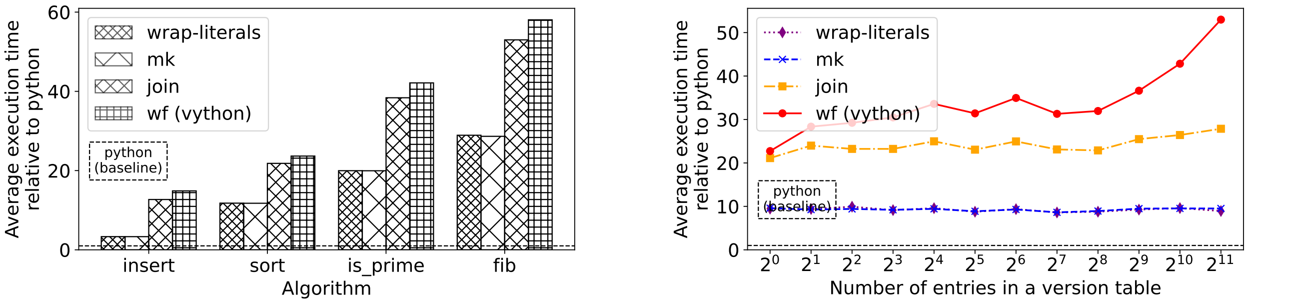

Fig. 7: Overhead of DVC functions in Vython (left) for simple benchmarks and (right) for repeating additions 2000 times with the number of VT entries.

curs no additional runtime cost, as it simply executes a Python program where multiple class versions are distinguished by their names.

6.1.1 Settings

We ran the following two benchmarks and calculated the average over 1000 iterations. The experiments were conducted with Python 3.12.1 on an Intel Core i5-10400F running Windows 11 23H2.

Benchmarks

- **Simple algorithms**: Vython programs implementing four simple algorithms insert, sort, fib, and is_prime (see Appendix A.3 for more details) using a VT with a maximum of two entries.
- **Scalability**: a Vython program that repeats additions 2000 times, with the number of VT entries doubling from 2^0 to 2^{11}.

The four algorithms in the simple algorithms benchmark were chosen to examine the overhead trends of the DVC feature in programs that can be written using the current Vython. insert and sort rely heavily on user-defined class instances, whereas fib and is_prime do not.

Each benchmark examines where the overhead occurs by disabling certain VT operations through the following Vython compiler options.

Compiler options

(1) **python**: baseline, production mode.
(2) **wrap-literals**: compiling literals as with VTs.
(3) **mk**: (2) + VT Initialization at instantiations.
(4) **join**: (3) + VT propagation.
(5) **wf (vython)**: (4) + consistency check.

6.1.2 Result and Discussion

Simple algorithms

Figure 7 (left) shows the execution times of simple algorithms relative to python. Focusing on each algorithm's case wf (vython), the program dominated by method calls to user-defined class instances, such as insert and sort, exhibit an overhead mostly within 20 to 30 times. In contrast, programs dominated by arithmetic or boolean operations, such as is_prime and fib, show a larger overhead ranging from 45 to 60 times.

Focusing on individual DVC functions, it is evident that weap-literals and join exhibit significant overhead across all benchmarks. In particular, these two DVC functions account for 90% of the overhead in sort, is_prime, and fib.

Compared to other dynamic analysis tools for Python (i.e. DynaPyt [9]), which generally exhibit an overhead of up to 20x, the current overhead of Vython is not practically acceptable and requires further optimization.

We believe the following optimizations could be effective in addressing this issue. For the overhead caused by wrap-literals, the current Vython attaches VT to all Python primitive values, even when they are involved in computations entirely unrelated to the classes of interest to the programmer. This negates bytecode optimizations for constant calculations. By discontinuing compile-time literal wrapping and instead performing dynamic casts only when computations interact with classes in the version space, we believe it is possible to reduce the overhead. Regarding the overhead of join, the current approach performs join on every field access, method call, and built-in operation. By incorporating static analysis, to compose DVC functions, we believe it will be possible to omit most join.

Scalability

Figure 7 (right) shows that overhead does not increase significantly as the VT size grows. Although an additional case study is needed to ensure that the maximum VT size does not grow excessively for practical programs, these results suggest that Vython is scalable.

6.2 Case Study

In the case study, we implement Keyword In Context (KWIC) according to the second modularization criteria presented by Parnas [21], and verify incompatible updates in downstream programs that use it. In [21], KWIC is described as follows:

> The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

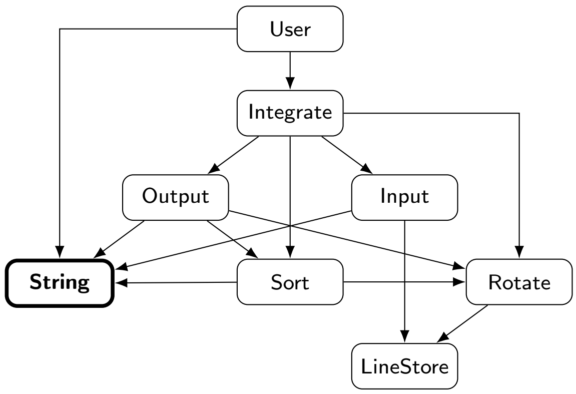

Fig. 8: Class dependency diagram.

6.2.1 Setting

Class Structure

Figure 8 shows the class dependencies of the KWIC implementation. We define that class A depends on class B if class A's definition includes at least one of the following: a B instantiation, a method call of a B object, or field access of a B object. When class A depends on class B, any modifications to class B necessitate a code review of class A's implementation.[*9] We implemented Parnas's module structure as individual classes shown in Figure 8. Additionally, we introduced the String class, which assumes standard library APIs and was implicitly used by each module in the original paper. The roles of the other classes remain the same as described in the original paper.

KWIC Implementation

Figure 9 shows the definition of a series of classes for implementing KWIC. Passing a list of `String` classes to `Integrate.main()` will output the corresponding KWIC index. These classes are implemented using version `1` of the `String` class. Notably, in the `sort` method of the `Sort` class, the program processes words one letter at a time, converting lowercase letters to uppercase. For further implementation details, please refer to the published GitHub repository.[*10]

Update Scenario

We define an update scenario for the downstream User's program which uses the KWIC implementation as follows. Initially, the User program relied on only version `1` of the `String` class. However, a requirement arises to use a new method introduced `include_pattern` in version `2` of the `String` class for a specific process, while keeping the use of the KWIC implemented using version `1` of the `String` class. Therefore, the downstream programmer selectively updates some of the call sites of the `String` class in their program to use version `2`.

Figure 10 shows the definitions of version `1` (top) and version `2` (bottom) of the `String` class. The update introduces two incompatible changes: (1) the addition of a new method, `include_pattern`, for checking the existence of a substring that matches a given regex, and (2) a modification to the behavior of the `get` method, which retrieves the character at a specified index. In the original implementation, `self.get(i)` returned the character code of the `i`th character in the string stored in `self`. After the update, it instead returns a single-character string.

```
class LineStore!1():
  def __init__(self):
      self.rows = []

  def set_char(self, row, word, offset, char):
    current_word = self.get_word(row, word)
    while offset >= len(current_word):
      current_word.append("")
    current_word[offset] = char

class Input!1():
  def input(texts):
    line_store = LineStore!1()
    for row_i in range(len(texts)):
      line = texts[row_i]
      split_line = line.split()
      for word_i in range(len(split_line)):
        word = split_line[word_i]
        for char_i in range(word.size()):
          line_store.set_char(row_i, word_i, char_i,
               word.get(char_i))
    return line_store

class Rotate!1():
  def __init__(self, line_store):
    self.line_store = line_store
    self.shift_table = []
    for row in range(line_store.num_rows()):
      for word in range(line_store.num_words(row)):
        self.shift_table.append((row, word))

class Sort!1():
  def __init__(self, rotate):
    self.rotate = rotate

  def first_shift_to_str(self, shift):
    keyword = String!1("")
    for char_i in range(self.rotate.num_chars(shift, 0))
        :
      char = (self.rotate.get_char(shift, 0, char_i))
      if String!1("a").get(0) <= char <= String!1("z").
           get(0):
        char -= 32
      keyword.add(String!1(char))
    # keyword = "".join(self.rotate.get_char(shift, 0,
         char) for char in range(self.rotate.num_chars(
         shift, 0)))
    return keyword

  def do_sort(self):
    self.row_indices = sorted(
      range(self.rotate.num_rows()),
      key=lambda r: self.first_shift_to_str(r),
    )
    return self

  class Output!1:
  def __init__(self, sort):
    self.sort = sort

  def output(self):
    ..

class Integrate!1:
  def main(self, titles):
    line_store = Input.input(titles)
    rotated = Rotate(line_store)
    sorted_rotate = Sort(rotated).do_sort()
    Output(sorted_rotate).output()
```

Fig. 9: Classes for KWIC

Downstream User's Program

Figure 11 shows the user program after the gradual update; the part that originally used version `1` of the `String`

[*9] It is important to note that the absence of a direct dependency from class A to class B does not guarantee that a review is unnecessary when class B is updated. Identifying such implicit errors is the purpose of DVC.

[*10] `prg-titech/kwic-vython`, `https://github.com/prg-titech/kwic-vython` (Accessed December 6, 2024)

```
class String!1():
  def __init__(self, value):
    if type(value) == int:
      self.value = chr(value)
    elif type(value) == str:
      self.value = value

  def get(self, i):
    if(i < 0):
      return 0
    if(len(self.value) <= i):
      return 0
    return ord(self.value[i])
```

```
class String!2():
  def __init__(self, value):
    if type(value) == int:
      self.value = chr(value)
    elif type(value) == str:
      self.value = value

  def get(self, i):
    if(i < 0):
      return 0
    if(len(self.value) <= i):
      return 0
    return self.value[i].incomp(String, 2)

  def include_patter(self, regex):
    ..
```

Fig. 10: Class **String** before and after update

```
1 ..
2 titles = [String!2("reverse-engineering␣weep␣ReLU␣
      networks"), String!2("Hi"), String!2("Reverse-
      engineering␣deep␣ReLU␣networks")]
3 Integrate().main(titles)
4 ..
5 # operation using newly introduced method in version 2
      of the String class
6 .. title.include_pattern(r'..') ..
```

Fig. 11: Updated Downstream User's Program

class has been selectively updated to use version 2 of the `String` class, as shown in line 2. In line 3, a list of `String!2` instances is passed to the `main` method of the `Integrate` class, which outputs the KWIC index corresponding to the given list.

6.2.2 Gradually Updating KWIC in Vython

Inconsistent Version Usage in Sort

When we execute the updated user program in Figure 11, a value derived from the `get` method for a `String!2` instance is assigned to the `char` in line 38 in Figure 9. Then, at the execution of line 39, where `char` is passed to the inequality operation, the following message is notified by both Python runtime and DVC.

```
TypeError: '<=' not supported between instances of 'int
      ' and 'str'
```

Error Message from Python Runtime

```
Incompatible version usage found:
- String 1
- String 2
[Changed in version 2] `String().get(i)`:
- returns a string whose length is 1 consisting of the
      i-th character.
- but returns the character code of the i-th character
      in version 1.
```

The Waring Issued from DVC

Here, Python runtime reports that its inequality operations are performed on a combination of values with different types (`int` and `str`). Specifically, the evaluating value of `String!1("a").get(0)` has a type of `int`, and the evaluating value of `char` has a type of `str`.

DVC reports that its inequality operations are performed on a combination of values created from an incompatible version of the implementation. Specifically, the evaluating value of `String!1("a").get(0)` has a VT of {(String, 1, -),..} and the evaluating value of `char` has a VT of {(String, 2, **T**),..}. Additionally, DVC also reports how these values can be incompatible. By combining this information, we identify the cause of the program's failure: the implementation assumes that the return value of the `get` method from the version 1 `String` class (an `int` value) is assigned to a `char`. However, due to the update, the return value of the version 2 `get` method (a `char` value) is being assigned instead.

```
class Sort!1:
  ..
  def first_shift_to_str(self, shift):
    keyword = String!1("")
    for char_i in range(self.rotate.num_chars(shift, 0))
        :
      char = self.rotate.get_char(shift, 0, char_i)
      if String! 2 ("a").get(0) <= char <= String! 2 ("z"
          ).get(0):
        char = char - 32
      keyword.add(String!1(char))
    return keyword
```

Refactored Definition of **first_shift_to_str**: 1

This cause analysis leads to a program revision, as exemplified in the above program. We refactor the program by changing the version of `get` method from version 1 to version 2 in the inequality operation. Then, executing the programs with this refactored program, the following results were produced.

```
TypeError: unsupported operand type(s) for -: 'int'
    and 'str'
```

Error Message from Python Runtime

The Python runtime raises an error at line 40 of Figure 9. This error reports that an `int` type value and an `str` type value were passed as operands to a subtraction operation. Specifically, the evaluating value of `32` has a type of `int`, and the evaluating value of `char` has a type of `str`.

However, no error is reported by the DVC mechanism in this case. This is because the return value of the version 2 `get` method assigned to `char` and the naive integer value 32 seem to be version consistent.

```
class Sort!1:
  ..
  def first_shift_to_str(self, shift):
    keyword = String!1("")
    for char_i in range(self.rotate.num_chars(shift, 0))
        :
      char = self.rotate.get_char(shift, 0, char_i)
      if String!2("a").get(0) <= char <= String!2("z").
          get(0):
        char = VInt(ord( char )) - 32
      keyword.add(String!1(char))
    return keyword
```

Refactored Definition of `first_shift_to_str`: 2

Therefore, in this case, we refactor the program as described above based solely on information provided by the Python runtime, without any feedback from the DVC mechanism. Specifically, we refactored the program to perform the subtraction of character codes using the `ord` method to explicitly obtain the character codes.

Executing the programs with this refactored program, it outputs the correct KWIC index as same as before an update along with countless warnings from Vython. However, these warnings are due to DVC's conservative design, and the results confirm that there is in fact no problem. Thus, we successfully done the gradual update of the program from version `1` of `String` class to version `2`.

6.2.3 Discussion

Through this case study, we demonstrate that Vython with DVC can support gradual updates. However, several limitations were also observed. For instance, during the execution of the program after the first refactoring, DVC did not issue any warnings about a type error caused by version-related incompatibilities. While such issues fall outside the current scope of DVC, they represent a class of problems that are likely to occur frequently in practice due to version incompatibilities. It will be necessary to extend DVC to address these structural incompatibility issues.

In addition, the current implementation of DVC is not able to identify which evaluation step each version information was attached in when inconsistent version information is detected. In addition to reporting inconsistency, it would be more helpful to be able to report what piece of program caused the inconsistency.

7. Related Work

7.1 Language-Based Approach Using Multiple Version in a Program

Programming with Versions is a programming paradigm proposed by Tanabe et al. that enables the use of multiple versions of packages, modules, or classes within a program, facilitating gradual updates. They highlight the incompatibility issues that arise when multiple versions of functions or classes coexist in a program and emphasize the importance of handling version consistency [24], [25] and structural compatibility [17] in their languages.

λ_{VL} [23], [24] is a functional core calculus with a type system that ensures version consistency. Its implementation, VL [25], enables the automatic selection of function versions using consistency-based type inference. On the other hand, BatakJava [17] is an extension of Featherweight Java [13] with a type system that ensures structural compatibility. Through BatakJava's type inference, the version of the class used for all instance creations is automatically determined. Programs that pass BatakJava's type inference are guaranteed to avoid method call or field access failures caused by version incompatibilities. Vython can be viewed as a dynamic checking mechanism that ensures version consistency. Exploring language extensions to guarantee structural compatibility presents an interesting direction for future work.

Carvalho and Seco propose a mechanism that allows version updates of programs to be expressed within the programming language itself, from the perspective of software evolution. Versioned Featherweight Java [5], [7] is an extension of Featherweight Java that introduces multi-branching and merge operations. Using program slicing, it extracts a well-typed single-version Featherweight Java code from a version-controlled codebase at compile time. They have also implemented a similar approach for Python [6], reducing the effort required for refactoring by automatically inserting some compatibility-absorbing code at compile time. While this approach cannot execute programs involving incompatible versions without pre-defined conversions, Vython allows programmers to run programs with any version combination and has a mechanism that dynamically detects issues arising from incompatibilities.

7.2 Dynamic Analysis Tools With an Implementation Similar to Vython

Some offline tools performing dynamic checks exhibit implementation methods that are partially similar to DVC. TaintCheck [20] is a dynamic analysis tool built on Valgrind [19], designed to detect vulnerabilities such as buffer overflows and format string exploits. TaintCheck marks input data from untrusted sources as tainted, tracks the propagation of tainted attributes during program execution (i.e., identifying which other data becomes tainted), and detects instances where tainted data is used in unsafe operations.

Similar approaches are found in the literature on dynamic information flow analysis. Austin and Flanagan [2], [3] introduced the semantics of λ_{info} [1], a variant of lambda calculus designed to derive the evaluation rules for Featherweight JavaScript, a subset of JavaScript. This calculus assigns security labels to values and dynamically verifies these labels to prevent the leakage of sensitive information by malicious JavaScript programs.

While these tools differ in purpose from Vython, they share a key characteristic: attaching metadata to values and defining how this metadata propagates. In Vython, version tables are used in place of security labels. These tools have been reported to incur runtime overheads ranging from several times (as seen in λ_{info}) to as much as 25 times (as observed in TaintCheck) compared to implementations without additional checks. Although the level of optimization varies significantly, incorporating techniques such as sparse labeling—used in λ_{info} to minimize the addition of check metadata—could potentially enhance the performance of Vython.

8. Conclusion and Future Work

We implement Vython and conduct a preliminary evaluation. The results indicate that while the current implementation is prototypical, its performance is scalable and acceptable for debugging small programs but requires further optimization for real-world applications. We plan to undertake the following future work.

Compatibility Management Toward Better Feedback

We plan to extend the DVC mechanism to accommodate the diverse compatibility requirements of real-world software. Practical software packages often have numerous versions and evolve non-linearly [8] through experimental features and language extensions. Currently, however, the DVC mechanism assumes a linear evolution involving only two versions or classes. The sole incompatibility annotation available to upstream developers, incomp, simply indicates that a version is incompatible with all others. This limitation it impossible to express incompatibilities between specific versions in scenarios involving three or more versions, as commonly encountered in real-world development. Leveraging tools to manage source code differences and incompatibilities, it becomes possible to synthesize feedback that accounts for the history of updates.

Surface Language Design

The current Vython requires specifying class versions in the surface program. We plan to develop a method to automatically infer versions working on Python programs. This will help minimize the annotations given by downstream developers, identify dependencies on old versions, and automate updates.

Acknowledgments The authors would like to thank the members of the PRG lab for their daily discussions. The authors also thank the audiences of APLAS & ATVA 2024 and PPL 2024 for their valuable comments on the preliminary stages of this research. This work was supported by JSPS KAKENHI Grant Numbers JP23K19961 and JP23K28058.

References

[1] Austin, T. H.: Dynamic Information Flow Analysis for Featherweight JavaScript Technical Report # UCSC-SOE-11-19, (online), available from ⟨https://api.semanticscholar.org/CorpusID:16308311⟩ (2011).

[2] Austin, T. H. and Flanagan, C.: Efficient purely-dynamic information flow analysis, *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, New York, NY, USA, Association for Computing Machinery, p. 113–124 (online), DOI: 10.1145/1554339.1554353 (2009).

[3] Austin, T. H. and Flanagan, C.: Permissive dynamic information flow analysis, *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/1814217.1814220 (2010).

[4] Bradski, G.: The OpenCV Library, *Dr. Dobb's Journal of Software Tools* (2000).

[5] Carvalho, L. and Costa Seco, J. a.: Deep Semantic Versioning for Evolution and Variability, *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming*, PPDP '21, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3479394.3479416 (2021).

[6] Carvalho, L. and Costa Seco, J. a.: A Language-Based Version Control System for Python, *38th European Conference on Object-Oriented Programming (ECOOP 2024)* (Aldrich, J. and Salvaneschi, G., eds.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313, Dagstuhl, Germany, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 9:1–9:27 (online), DOI: 10.4230/LIPIcs.ECOOP.2024.9 (2024).

[7] Carvalho, L. and Costa Seco, J.: Software Evolution with a Typeful Version Control System, *Software Engineering and Formal Methods* (Ölveczky, P. C. and Salaün, G., eds.), Cham, Springer International Publishing, pp. 145–161 (2019).

[8] Conradi, R. and Westfechtel, B.: Version models for software configuration management, *ACM Comput. Surv.*, Vol. 30, No. 2, p. 232–282 (online), DOI: 10.1145/280277.280280 (1998).

[9] Eghbali, A. and Pradel, M.: DynaPyt: a dynamic analysis framework for Python, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, New York, NY, USA, Association for Computing Machinery, p. 760–771 (online), DOI: 10.1145/3540250.3549126 (2022).

[10] Foo, D., Chua, H., Yeo, J., Ang, M. Y. and Sharma, A.: Efficient static checking of library updates, *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, New York, NY, USA, Association for Computing Machinery, p. 791–796 (online), DOI: 10.1145/3236024.3275535 (2018).

[11] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del R'ıo, J. F., Wiebe, M., Peterson, P., G'erard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C. and Oliphant, T. E.: Array programming with NumPy, *Nature*, Vol. 585, No. 7825, pp. 357–362 (online), DOI: 10.1038/s41586-020-2649-2 (2020).

[12] Igarashi, A., Pierce, B. C. and Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ, *ACM Trans. Program. Lang. Syst.*, Vol. 23, No. 3, p. 396–450 (online), DOI: 10.1145/503502.503505 (2001).

[13] Igarashi, A., Pierce, B. C. and Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ, *ACM Transactions on Programming Languages and Systems*, Vol. 23, No. 3, pp. 396–450 (online), DOI: 10.1145/503502.503505 (2001).

[14] Jayasuriya, D., Terragni, V., Dietrich, J. and Blincoe, K.: Understanding the Impact of APIs Behavioral Breaking Changes on Client Applications, *Proc. ACM Softw. Eng.*, Vol. 1, No. FSE (online), DOI: 10.1145/3643782 (2024).

[15] Kula, R. G., German, D. M., Ouni, A., Ishio, T. and Inoue, K.: Do developers update their library dependencies?, *Empirical Software Engineering*, Vol. 23, No. 1, pp. 384–417 (online), DOI: 10.1007/s10664-017-9521-5 (2018).

[16] Lam, P., Dietrich, J. and Pearce, D. J.: Putting the semantics into semantic versioning, *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2020, New York, NY, USA, Association for Computing Machinery, p. 157–179 (online), DOI: 10.1145/3426428.3426922 (2020).

[17] Lubis, L. A., Tanabe, Y., Aotani, T. and Masuhara, H.: BatakJava: An Object-Oriented Programming Language with Versions, *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, Auckland New Zealand, ACM, pp. 222–234 (online), DOI: 10.1145/3567512.3567531 (2022).

[18] Mirhosseini, S. and Parnin, C.: Can automated pull requests encourage software developers to upgrade out-of-date dependencies?, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, Urbana-Champaign, IL, USA, IEEE Press, pp. 84–94 (2017).

[19] Nethercote, N. and Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation, *SIGPLAN Not.*, Vol. 42, No. 6, p. 89–100 (online), DOI: 10.1145/1273442.1250746 (2007).

[20] Newsome, J. and Song, D. X.: Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGener-

ation of Exploits on Commodity Software, *Network and Distributed System Security Symposium*, (online), available from ⟨https://valgrind.org/docs/newsome2005.pdf⟩ (2005).

[21] Parnas, D. L.: On the criteria to be used in decomposing systems into modules, *Commun. ACM*, Vol. 15, No. 12, p. 1053–1058 (online), DOI: 10.1145/361598.361623 (1972).

[22] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S.: PyTorch: An Imperative Style, High-Performance Deep Learning Library, *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., pp. 8024–8035 (online), available from ⟨http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf⟩ (2019).

[23] Tanabe, Y., Aotani, T. and Masuhara, H.: A Context-Oriented Programming Approach to Dependency Hell, *Proceedings of the 10th ACM International Workshop on Context-Oriented Programming: Advanced Modularity for Run-Time Composition*, COP '18, New York, NY, USA, Association for Computing Machinery, p. 8–14 (online), DOI: 10.1145/3242921.3242923 (2018).

[24] Tanabe, Y., Lubis, L. A., Aotani, T. and Masuhara, H.: A Functional Programming Language with Versions, *The Art, Science, and Engineering of Programming*, Vol. 6, No. 1, pp. 5:1–5:30 (online), DOI: 10.22152/programming-journal.org/2022/6/5 (2021).

[25] Tanabe, Y., Lubis, L. A., Aotani, T. and Masuhara, H.: Compilation Semantics for a Programming Language with Versions, *Programming Languages and Systems* (Hur, C.-K., ed.), Singapore, Springer Nature Singapore, pp. 3–23 (2023).

[26] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P. and SciPy 1.0 Contributors: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, *Nature Methods*, Vol. 17, pp. 261–272 (online), DOI: 10.1038/s41592-019-0686-2 (2020).

[27] Wes McKinney: Data Structures for Statistical Computing in Python, *Proceedings of the 9th Python in Science Conference* (Stéfan van der Walt and Jarrod Millman, eds.), pp. 56 – 61 (online), DOI: 10.25080/Majora-92bf1922-00a (2010).

Appendix

A.1 Incompatible Behaviours Between NumPy 2.0.0 and 1.26.4

This section outlines some of the major incompatibilities between NumPy 1.26.4 and NumPy 2.0.0. All 11 examples we collect, along with the scripts to reproduce them, are available on the GitHub repository (`https://github.com/prg-titech/numpy_diff`).

A.1.1 Incompatibilities in `numpy.linalg.solve`

The `numpy.linalg.solve` function solves a linear matrix equation. It solves the equation $ax = b$ for x, where a is a square matrix and b is a vector or matrix provided as arguments to the function. The implementation was changed in NumPy 2.0.0. The following note is from the official NumPy documentation:

> *Changed in version 2.0*: The `b` array is only treated as a shape `(M,)` column vector if it is exactly 1-dimensional. In all other instances it is treated as a stack of `(M, K)` matrices. Previously `b` would be treated as a stack of `(M,)` vectors if `b.ndim` was equal to `a.ndim - 1`.

As a result, when b is not strictly one-dimensional, the output of the `solve` function differs for the same input. For example, consider the following program run with NumPy 1.26.4 and NumPy 2.0.0.

```
import numpy as np

# Shape (2, 2, 2)
a = np.array(
  [ [[3, 1], [1, 2]]
  , [[2, 1], [1, 3]] ])
# Shape (2, 2)
b = np.array(
  [ [9, 8]
  , [7, 10] ])

x = np.linalg.solve(a, b)
print(x)
```

When we run the above program with NumPy versions 1.26.4 and 2.0.0, we get the following different outputs due to incompatibility in broadcasting rules.

```
@ Running linalg_solve.py with numpy 1.26.4
[[2. 3. ]
 [2.2 2.6]]
@ Running linalg_solve.py with numpy 2.0.0
[[[2.2 1.2]
  [2.4 4.4]]

 [[4. 2.8]
  [1. 2.4]]]
```

The reason for this difference lies in how the `b` array is treated in different versions of NumPy. In version 1.26.4, if `b`'s number of dimensions (`b.ndim`) is equal to one less than the number of dimensions of `a` (`a.ndim - 1`), `b` is interpreted as a stack of `(M,)` vectors. This means that in version 1.26.4, the `b` array is treated as a stack of 1-dimensional vectors, each corresponding to a 2x2 matrix in `a`. Therefore, the program is interpreted as follows:

$$\begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 9 \\ 8 \end{pmatrix}, \quad \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 7 \\ 10 \end{pmatrix}.$$

However, in version 2.0.0, the behavior was modified such that the `b` array is treated as a column vector only if it is strictly 1-dimensional. In all other cases, it is treated as a stack of `(M, K)` matrices. Consequently, for the given input, `b` is treated as a stack of 2-dimensional matrices. Therefore, the program is interpreted as follows:

$$\begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix} = \begin{pmatrix} 9 & 8 \\ 7 & 10 \end{pmatrix}, \quad \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} y_1 & y_2 \\ y_3 & y_4 \end{pmatrix} = \begin{pmatrix} 9 & 8 \\ 7 & 10 \end{pmatrix}.$$

A.1.2 Incompatibilities in Other Functions

In addition to `numpy.linalg.solve`, NumPy 2.0.0 introduces several other backward-incompatible modifications.

Among the programs we collected that produce different outputs solely due to version differences in NumPy, we list some notable input-output pairs below. For other examples where downstream developers might easily notice incompatibilities due to Python runtime errors, such as differences in output types, please refer to the repository.

`numpy.nonzero`

The function return the indices of the elements that are non-zero. The function previously ignored whitespace so that a string only containing whitespace was considered `False`, however, whitespace is now considered `True` in string arrays newly in NumPy 2.0.0.

```
import numpy as np

arr = np.array(['␣', 'a', ''])
print(np.nonzero(arr))
```

```
@ Running nonzero.py with numpy 1.26.4
(array([1]),)
@ Running nonzero.py with numpy 2.0.0
(array([0, 1]),)
```

`numpy.linalg.lstsq`

The function returns the least squares solution to a linear matrix equation. The default value of the `rcond` (cut-off ratio) parameter in `lstsq` was changed in NumPy 2.0.0. This change introduces a subtle incompatibility: while most inputs yield the same output regardless of the NumPy version, inputs with elements near machine precision can produce different results depending on the NumPy version. The following example illustrates such a case.

```
import numpy as np

a = np.zeros((10**2, 2))
a[0, 0] = 1
a[m-1, 1] = 2.22e-16
b = np.zeros(m)
b[m-1] = 1

x, res, rank, s = np.linalg.lstsq(a, b)
print(...)
```

```
@ Running linalg_lstsq.py with numpy 1.26.4
Solution with default rcond: [0.0000000e+00 4.5045045e
    +15]
Residuals: [4.93038066e-32]
Rank: 2
Singular values: [1.00e+00 2.22e-16]
@ Running linalg_lstsq.py with numpy 2.0.0
Solution with default rcond: [0. 0.]
Residuals: []
Rank: 1
Singular values: [1.00e+00 2.22e-16]
```

`numpy.loadtxt` **and** `numpy.genfromtxt`

The functions provide readers for simly formatted files. Default encoding for these functions was changed in NumPy 2.0.0. Previously, these two functions selected `encoding=bytes` as the default parameter, but starting from version 2.0.0, it has been changed to `encoding=string`. As a result, programs that expect custom converters assuming a byte value will be broken by the update.

```
import numpy as np
import io
def custom_converter(byte_string):
  return float(byte_string.decode('utf-8'))

data = b"1.1\n2.2\n3.3\n"
with open('data.txt', 'wb') as f:
  f.write(data)

# Load the data using loadtxt with the custom converter
try:
  data = np.loadtxt('data.txt', converters={0:
      custom_converter})
  print(f"Data␣loaded␣successfully:␣{data}")
except Exception as e:
  print(f"An␣error␣occurred:␣{e}")
```

```
@ Running loadtxt_genfromtxt.py with numpy 1.26.4
Data loaded successfully: [1.1 2.2 3.3]
@ Running loadtxt_genfromtxt.py with numpy 2.0.0
An error occurred: could not convert string '1.1' to
    float64 at row 0, column 1.
```

A.2 Dynamically Switching NumPy Versions

This section describes the reproduction of the motivating examples from Section 2 in actual Python programs. The complete source code and instrucations to reproduce the results of this paper are available on the GitHub repository (`https://github.com/prg-titech/use-multi-versions`).

A.2.1 Installing Multiple NumPy Versions from Sources

For example, to install numpy version 1.26.4 into a directory named numpy-1.26.4 using pip on a Linux OS, use the following command.

```
$ mkdir numpy-1.26.4
$ pip donwload numpy==1.26.4
$ pip install numpy-1.26.4- ... .whl -t numpy-1.26.4
```

A.2.2 Simultaneouslly Using Multiple NumPy Versions in Code

The following `load_numpy` function dynamically loads a specified version of NumPy. It takes a string representing the version, sets the appropriate NumPy path, and removes any cached instances of NumPy from `sys.modules`. The function then temporarily modifies the system path to include the specified version's path installed in the last section and imports the NumPy module from its initialization file. Finally, `load_numpy` returns the module object for the specified version of NumPy.

```
# version_dispatch.py
def load_numpy(version):
    if version == '1.26.4':
        numpy_path = os.path.abspath('numpy-1.26.4')
    elif version == '2.0.0':
        numpy_path = os.path.abspath('numpy-2.0.0')
    else:
        raise ValueError(f"Unsupported␣numpy␣version:␣{
            version}")

    # Clear cache
    if 'numpy' in sys.modules:
        del sys.modules['numpy']
    for mod_name in list(sys.modules):
        if mod_name.startswith('numpy'):
```

```
            del sys.modules[mod_name]

    # Set environment pathes
    original_path = sys.path.copy()
    sys.path.insert(0, numpy_path)

    try:
        numpy_init_path = os.path.join(numpy_path, 'numpy', '__init__.py')
        spec = importlib.util.spec_from_file_location("numpy", numpy_init_path)
        if spec is None:
            raise ImportError(f"Cannot find numpy module in {numpy_path}")

        numpy = importlib.util.module_from_spec(spec)
        spec.loader.exec_module(numpy)
    finally:
        # Restore sys.path
        sys.path = original_path
```

The following program shows the full version of the program shown in Figure 2. The implementation of the pole placement problem (`place_poles` and `my_place_poles`) has been simplified, as it is not the focus of this section. Using `./version_dispatch.py`, which defines the `load_numpy` function described in the previous subsection, `place_poles` is evaluated with NumPy 1.26.4 on line 26, and `my_place_poles` is evaluated with NumPy 2.0.0 on line 27. Finally, the results of the two functions are compared on line 29.

As mentioned in Section 2, despite `place_poles` and `my_place_poles` being identical implementations except for the NumPy version they use, the result evaluates to `False`.

```
1 from version_dispatch import load_numpy
2
3 # SciPy
4 class SciPy():
5   def place_poles(self, A, B, desired_poles):
6     np = load_numpy('1.26.4')
7     res = np.linalg.solve(A, B)
8     return res
9
10 # User Program
11 def my_place_poles(A, B, desired_poles):
12   np = load_numpy('2.0.0')
13   res = np.linalg.solve(A, B)
14   return res
15
16 def main():
17   np = load_numpy('2.0.0')
18   A = np.array(
19     [ [[3, 1], [1, 2]]
20     , [[2, 1], [1, 3]] ])
21   B = np.array(
22     [ [9, 8]
23     , [7, 10] ])
24   desired_poles = np.array([-1.0, -2.0])
25
26   expect = SciPy().place_poles(A,B,desired_poles).tolist()
27   actual = my_place_poles(A,B,desired_poles).tolist()
28
29   test = np.array_equal(expect, actual) # => False
30
31 main()
```

A.3 Programs Used for Simple Benchmarks

`sort` performs a merge sort on a Python list of 1000 elements, `is_prime` uses a simple algorithm to determine the primality of 128456903, `fib` recursively computes the 20th Fibonacci number, and `insert` inserts one thousand `Node` instances to a binary tree.

A.3.1 insert

```
class Node!1():
  def __init__(self, value):
      self.value = value
      self.left = None
      self.right = None

  def insert_right(self, v):
      if self.right == None:
          self.right = Node!1(v)
      else:
          self.right.insert(v)

  def insert_left(self, v):
      if self.left == None:
          self.left = Node!1(v)
      else:
          self.left.insert(v)

  def insert(self, v):
      if(self.value <= v):
          self.insert_right(v)
      else:
          self.insert_left(v)

root = Node!1(5)
a = [...] # Array of 1000 elements, random numbers between 1 and 10000
for i in a:
    root.insert(i)
```

A.3.2 sort

```
def sort(list):
  if len(list) < 1:
      return []
  elif len(list) == 1:
      return list
  pivot = list[0]
  lower_list = []
  upper_list = []
  middle_list = []

  for item in list:
      if item < pivot:
          lower_list.append(item)
      elif item > pivot:
          upper_list.append(item)
      else:
          middle_list.append(item)

  sorted_lower_list = sort(lower_list)
  sorted_upper_list = sort(upper_list)

  return sorted_lower_list + middle_list + sorted_upper_list

a = [...] # Array of 1000 elements, random numbers between 1 and 10000
sort(a)
```

A.3.3 prime

```
def is_prime(n):
  if n <= 1:
      return False
  if n == 2 or n == 3:
      return True
  if n % 2 == 0 or n % 3 == 0:
      return False
  return is_prime_recursive(n, 5)

def is_prime_recursive(n, i):
    if i * i > n:
        return True
    if n % i == 0 or n % (i + 2) == 0:
        return False
    return is_prime_recursive(n, i + 6)

is_prime(128456903)
```

A.3.4 fib

```
def fib(n):
  if n<=2:
      return 1
  else:
      return fib(n-1) + fib(n-2)

fib(20)
```

Satsuki Kasuya He is a graduate student in Department of Mathematical and Computing Science, School of Computing, Institute of Science Tokyo (formerly Tokyo Institute of Technology). He received B.S. degree from Tokyo Institute of Technology in 2024. He is interested in programming languages.

Yudai Tanabe He is an assistant professor in Department of Mathematical and Computing Science, School of Computing, Institute of Science Tokyo (formerly Tokyo Institute of Technology.) He received B.S., M.S., and D.S. degrees from Tokyo Institute of Technology in 2018, 2020, and 2023 respectively. Formerly, he was a JSPS Research Fellow at Tokyo Institute of Technology from 2022 until 2023, a program-specific researcher at Kyoto University from 2023 until 2024, and an assistant professor in Department of Mathematical and Computing Science, School of Computing, Tokyo Institute of Technology until September 2024. His research interest is programming language and software engineering, especially in programming language theory, type systems, and software maintenance.

Hidehiko Masuhara He is a Professor of Mathematical and Computing Science, School of Computing, Institute of Science Tokyo (formerly Tokyo Institute of Technology.) He received his B.S., M.S., and Ph.D. in Computer Science from Department of Information Science, the University of Tokyo in 1992, 1994, and 1999, respectively, and served as an assistant professor, lecturer, and associate professor at Department of Graphics and Computer Science, Graduate School of Arts and Sciences, the University of Tokyo from 1995 until 2013. He was the Dean of the School of Computing of Tokyo Institute of Technology from 2022 until 2024. His research interest is programming languages, especially in aspect- and context-oriented programming, partial evaluation, computational reflection, meta-level architectures, parallel/concurrent computing, and programming environments.