Semantic Versioning for Python Programs

Luís Carvalho
la.carvalho@campus.fct.unl.pt
NOVA University Lisbon
Portugal

Abstract

We propose a language-based approach to software versioning. Unlike the traditional approach of mainstream version control systems, where each evolution step is represented by a textual diff, we treat versions as programming elements. Each evolution step, merge operation, and version relationship, is represented explicitly in the program. This provides compile time guarantees for safety code reuse from previous versions, as well as forward and backwards compatibility between versions, allowing clients to use newly introduced code without needing to refactor their program. By lifting the versioning to the language level, we pave the way for tools that interact with software repositories to have more insight regarding the evolution of the software semantics.

CCS Concepts: • **Theory of computation** → **Type theory**; **Program semantics**.

Keywords: Software evolution, type theory

ACM Reference Format:
Luís Carvalho. 2023. Semantic Versioning for Python Programs. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '23), October 22–27, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3618305.3623589

1 Motivation

The evolution of software systems is an essential aspect of software development and its lifecycle. As requirements from stakeholders change, software systems must evolve to conform to such changes. These changes may include bug fixing, implementing new features, porting code to new hardware, updating business requirements, and other tasks that represent activities in the lifecycle of a software system. As the release cycles in the software development process become shorter [8] the overhead of managing multiple versions increases. On the one hand, software maintainers have to reason more frequently about what changes to backport and whether the changes they introduced break existing client code. On the other hand, the stakeholders of a product (e.g. developers of a client application) will need to consider more frequently whether or not to upgrade. Integrating a release with breaking changes leads to runtime errors and requires manual intervention, while missing a critical update may lead to software vulnerabilities.

Given the manual effort required for semantic software versioning, largely rooted in the fact that VCS operate on *text* rather than *programs*, we propose a language-based approach for providing a version control system.

2 Problem

The current industry practices for software evolution advocate the use of version control systems (e.g. `git`, `svn`, `mercurial`). However, while very good at managing *changes* to information, VCS give no semantic meaning to each program delta (*diff*). Each evolution step is typically defined by 1) the new code it introduces and 2) an accompanying natural language message describing the change. To issue a new version of the software, the developer includes one or more of these steps and generates a *changelog*, naming the version according to some convention.

A widely adopted convention is Semantic Versioning [1], where the increment of each version identifier denotes the nature of the introduced changes. Clients can then define their update policy according to this convention. There is no guarantee that the code effectively follows the versioning policy (e.g. the developer unknowingly introduced an unexpected breaking change), thus VCS allow for inconsistent versions to be committed, and still require work from developers in identifying the kind of changes made [12, 14].

In this work, we embed the versioning in the programming language, so that the developer specifies as code what each delta is, and also how it relates to other versions. This allows for (formal) verification on if and how the different versions interact with each other; it allows for clients to use newly introduced code without changing their existing program; it is well-suited for targeting software for a specific version, producing artifacts containing only the necessary code; for providing a version-aware development environment (drawing inspiration from [11]) in which a developer may edit a snapshot and have the changes automatically committed with the appropriate version tags.

SPLASH Companion '23, October 22–27, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0384-3/23/10.
https://doi.org/10.1145/3618305.3623589

We expect this approach to guarantee compile time safety of code reuse from previous snapshots, as well as ensuring forward and backward compatibility between related versions through type-safe state transformation functions.

3 Approach

We propose a language-based approach to providing a certifying version control system for Python programs, where versions and their relations are specified as code in the program. This work is an extension of our previous work in a core functional language, Versioned Featherweight Java [3].

In Fig. 1, we provide two examples of such programs. The library program (Fig. 1a) comprises a version graph, where versions are marked with modes capturing the different types of software evolution, which we can infer from its repository (e.g. the version graph for the repository in Fig. 1d is depicted in Fig. 1d); versioned programming elements, such as constructors (lines 5-11) and methods (lines 15-23), decorated with the version in which they were introduced; and *get* lenses (Fig. 1c) in the form of `@get(v,v',f)` that map how field `f` of version `v'` is derived from the state in version `v`. Finally, Fig. 1f shows the client program using library code for version `full`, which is a valid Python program that when executed follows the semantics described in this work.

We provide static semantics for versioned program slicing and manipulation, dynamic semantics for multi-version program execution, and a type system for versioned programs based on the following crucial aspects.

Class fields lookup Each version in the graph has exactly one corresponding base version, where its fields are defined. In the example of Fig. 1a, the base version of `bugfix`, in which no fields are explicitly defined, is `init`; the base version of `full` is itself. This lookup discipline is used in all typing and reduction rules involving class fields.

Methods lookup Method definitions are looked up based on the relationship between versions. First, since we have a version that replaces version `init`, lookups in version `init` will occur first on the replacement version and only after in itself. An upgrade version, on the other hand, has no impact on the lookups in the version it upgrades. Lookups in the `full` version will occur first on version `full`, and then on the `init` version, following the same discipline described above. As an example, a lookup on version `full` for method `display` will yield the definition introduced in version `bugfix`.

Crossing version contexts A key aspect of this approach is allowing the crossing of version contexts. Whenever the lookup functions, for fields and methods, returns a version other than the current context, we need to translate the object from one version to the other using lenses, which are mappings of an object's field from one version to another. Consider the example of running method `display` on version `full`: to correctly use the code from version `bugfix`, we need to map the fields `first` and `last` for version `full`, since they do not exist in that version. The mappings are applied according to lens expression for each field defined in the classes of a module (Fig. 1c).

Assignment and side effects Python programs usually involve side effects through assignment to class fields. Consider method `set_last`, defined for version `init`: to rewrite this method for version `full`, we must track what fields are affected in version `full` by changing field `last`. To do so, we statically analyze all `get` lenses from `init` to `full` to detect where this field is accessed; we can conclude that changing field `last` will have an effect on field `fullname`, according to the code in `lens_full` (Fig. 1c, lines 1-3). To rewrite the expression `self.last = last` for version `full`, we need to propagate the side effects to the corresponding fields (`fullname`). We start by storing the right-hand side of the assignment in a fresh variable (`__name`). Then, we synthesize a *put* lens from its corresponding *get* lens, by replacing fields within the lens body with (matching) function parameters (Fig. 1b); then, we use the *put* lens to update the value of the affected field (Fig. 1e, line 8); we pass the right-hand side of the assignment as argument for its corresponding field (left-hand side, `last`), and the rewritten field expressions as argument to all other unaffected fields.

Program slicing We propose a slicing procedure, built on top of a rewriting algorithm applying static program transformations (using lenses) whenever it is necessary to cross version contexts. To slice a program for version `t`, we rewrite all method definitions available at `t`, given by the lookup functions, from their version (`v`) to the target version. In Fig. 1e, we present the slice for version `full`. The resulting slice includes: fields and methods returned by the lookup functions (note how the replacement definition of `display`, introduced in version `bugfix`, is selected); the necessary *get* lenses for field access (Fig. 1c, `lens_full`), and the synthesized *put* lenses for field assignment (omitted in the figure for brevity). This procedure produces the source code with the necessary definitions targeted for version `full`.

We believe this approach is suitable for capturing software evolution steps. However, it is limited since there is no way to specify how a method evolves from one version to the other (e.g. changes in result, parameter order). We plan on extending our work with such a mechanism. For the design, we chose annotations because they allow us to still have valid Python programs, which helps in bootstrapping the prototype. We plan on conducting usability studies to find good compromises on the design of the language, and on the workflow for developers using this approach.

4 Evaluation Methodology

We will validate our approach with the following methodology. We start by gathering a corpus of Python software repositories that are used as libraries. Since our approach requires typed programs, we add any missing type annotations with the help of automated type inference tools [4, 7, 9, 13]. This may involve manual labor, even though the practice of

```python
@version('init')
@version('bugfix', replaces=['init'])
@version('full', upgrades=['init'])
class Name:
    @at('init')
    def __init__(self, first: str, last: str):
        self.first = first
        self.last = last
    @at('full')
    def __init__(self, full: str):
        self.fullname = full
    @at('init')
    def display(self):
        return f'{self.first}, {self.last}'
    @at('bugfix')
    def display(self):
        return f'{self.last}, {self.first}'
    @at('init')
    def set_last(self, name: str):
        self.last = name
    @at('full')
    def get_full_name(self):
        return self.fullname
```

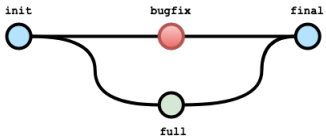
(a) Example of a versioned Python class.

```python
@put('init', 'full', 'fullname')
def lens_full(self, first, last):
    return f'{first} {last}'
```

(b) Synthesized lens for assignment on a single field between different versions.

```python
@get('init', 'full', 'fullname')
def lens_full(self):
    return f'{self.first} {self.last}'
@get('full', 'init', 'first')
def lens_first(self):
    if ' ' in self.fullname:
        return self.fullname.split(' ')[0]
    return self.fullname
@get('full', 'init', 'last')
def lens_last(self):
    if ' ' in self.fullname:
        return self.fullname.split(' ')[1]
    return ''
```

(c) Lenses for fields between different versions.




```python
@version('init')
@version('bugfix', replaces=['init'])
@version('full', upgrades=['init'])
@version('final', upgrades=['full'], replaces=['bugfix'])
```

(d) Example of a git repository with two commits (`init`, `bugfix`), one branch (`full`), and one merge (`final`).

```python
class Name:
    def __init__(self, full: str):
        self.fullname = full
    def display(self):
        return f'{self.lens_last()},
          {self.lens_first()}'
    def set_last(self, name: str):
        __name = name
        self.fullname = self.lens_full(
            first=self.lens_first(),
            last=__name)
    def get_full_name(self):
        return self.fullname
```

(e) Slice of a versioned Python class for version `full`.

```python
@run('full')
def main():
    n = Name('Bob Dylan')
    n.set_last('Marley')
    print(n.display())
```

(f) Client code running in version `full`.

Figure 1. Elements of a versioned Python program.

providing type annotations in Python programs is becoming increasingly common [5]. We analyze the repository history and select tags that meaningfully represent the various types of evolution of the program (e.g. introducing minor and breaking changes). From this set, we generate the corresponding version graph (e.g. Fig. 1d). Then, we run the type checker to detect any missing lenses and implement them. We will use Large Language Models [2, 6, 10] to bootstrap this process and then (manually) adjust for any errors. Finally, we need to add version annotations to the client program, so we select a version of the library we want to use (e.g. version `t`) and mark the dependency in the client with that version. The evaluation is two-fold:

Version graph validation We will evaluate if this approach is able to detect the unintended malformedness of the version graph. We manually select commits that introduce related versions and try to synthesize the required lenses: if not possible, the type checking of the program against its version graph fails, meaning the version graph is wrongly specified.

Client evolution without refactoring We will evaluate if it is possible for clients to use newly introduced code without any refactoring on their part. We checkout the client code to the commit before they refactored their code for version `t`; then, we extract a slice of the client code for version `t`; we compare this slice with the (original) refactored client code. The approach is validated if the behavior of both programs is equivalent. A simple verification method for this is running the client's tests against the resulting slice and check that they pass.

References

[1] 2023. Semantic Versioning 2.0.0. www.semver.org

[2] Jacob Austin et al. 2021. Program synthesis with large language models. arXiv:2108.07732.

[3] Luís Carvalho and João Costa Seco. 2021. Deep semantic versioning for evolution and variability. In *PPDP 2021*. 1–13.

[4] Siwei Cui et al. 2021. PYInfer: Deep Learning Semantic Type Inference for Python Variables. arXiv:2106.14316.

[5] Luca Di Grazia et al. 2022. The evolution of type annotations in python: an empirical study. In *ESEC/FSE 2022*. ACM, 209–220.

[6] Naman Jain et al. 2022. Jigsaw: Large languag*f*e models meet program synthesis. In *Proceedings of the 44th ICSE*. 1219–1231.

[7] Li Li et al. 2022. Scalpel: The python static analysis framework. (2022). arXiv:2202.11840.

[8] Stuart McIlroy et al. 2016. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. , 1346–1370 pages.

[9] Raphaël Monat et al. 2020. Static type analysis by abstract interpretation of Python programs. In *ECOOP 2020*.

[10] Erik Nijkamp et al. 2022. Codegen: An open large language model for code with multi-turn program synthesis. (2022). arXiv:2203.13474.

[11] Cyrus Omar et al. 2019. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages* 3, 1–32.

[12] S. Raemaekers, A. Van Deursen, and J. Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158.

[13] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 607–618.

[14] Lyuye Zhang et al. 2022. Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing. arXiv:2209.00393.

Received 2023-07-21; accepted 2023-08-10