A Language-Based Version Control System for Python

Luís Carvalho
NOVA LINCS, NOVA School of Science and Technology, Caparica, Portugal

João Costa Seco
NOVA LINCS, NOVA School of Science and Technology, Caparica, Portugal

Abstract

We extend prior work on a language-based approach to versioned software development to support versioned programs with mutable state and evolving method interfaces. Unlike the traditional approach of mainstream version control systems, where a textual diff represents each evolution step, we treat versions as programming elements. Each evolution step, merge operation, and version relationship is represented explicitly in a multifaceted code representation. This provides static guarantees for safe code reuse from previous versions and forward and backwards compatibility between versions, allowing clients to use newly introduced code without needing to refactor their program manually. By lifting versioning to the language level, we pave the way for tools that interact with software repositories to have more insight into a system's behavior evolution. We instantiate our work in the Python programming language and demonstrate its applicability regarding common evolution and refactoring patterns found in different versions of popular Python packages.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Program semantics

Keywords and phrases Software evolution, type theory

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.9

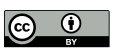
Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.3`

Funding This work is supported by EU Horizon Europe under Grant, Agreement no. 101093006 (TaRDIS), NOVA LINCS UIDB/04516/2020 (`https://doi.org/10.54499/UIDB/04516/2020`) and UIDP/04516/2020 (`https://doi.org/10.54499/UIDP/04516/2020`) with financial support of FCT.IP.

1 Introduction

The evolution of software systems is an essential aspect of software development and its life-cycle. 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 life-cycle of a software system. As the release cycles in the software development process become shorter [20] 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 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, which is largely rooted in the fact that version control systems (VCS, e.g. `git`, `svn`, `mercurial`) operate on *text* rather than *programs*, we extend prior work on a language-based VCS for a core functional language, Versioned Featherweight Java [5], by adding support for programs with mutable state and side effects, and versioning of class methods.



The current industry practices for software evolution advocate the use of version control systems. 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 [26], 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 [33, 27].

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 [23]) in which a developer may edit a snapshot and have the changes automatically committed with the appropriate version tags. We extend previous work [5] to provide a language-based VCS for Python programs, where versions and their relations are specified as code in the program. The features that strictly extend [5] are:

- A mechanism for defining transformations (which we call *lenses*) between method interfaces in different, related, versions. This allows the developer to specify how the evolution of a method interface is to be handled by clients, so that they do not have to manually adapt their code to account for the new definition.
- Support for mutability and side-effects. Featherweight Java (FJ) is a functional language and, as such, does not model side effects. In this work, we instantiate our core calculus in Python, a mainstream language with new challenges in comparison to FJ, particularly concerning mutability. To do so, we ensure that mutability and side-effects are well reflected when transitioning between versions with different state representations.

In this setting, the developer of a versioned program defines how the versions of the program relate to each other, and provides each evolution step as code, to define how clients should evolve between versions. As such, clients can then get new features from other versions without their code breaking, and without any need for manual refactoring: the evolution steps provided by the developer are used to adapt the code to the client version, so that they can use it without breaking.

The diagram in Figure 1 provides the intuition on the parallels between traditional version control systems and our approach. With the developer providing each evolution step as code, we allow clients to migrate automatically, thus removing the need for migration tools (which can also introduce bugs) to help clients do that. The traditional repository operations, such as committing a file, or merging two branches, are well supported in our setting, by defining the appropriate version graph. Finally, we provide a slicing procedure to allow developers to issue a release for a given version, without having to manually perform operations on the repository (e.g. backporting a security fix to another branch). The resulting slice, corresponding to a release for a given version, ensures that the release conforms to the versioning graph, i.e. it does not introduce unintended breaking changes.

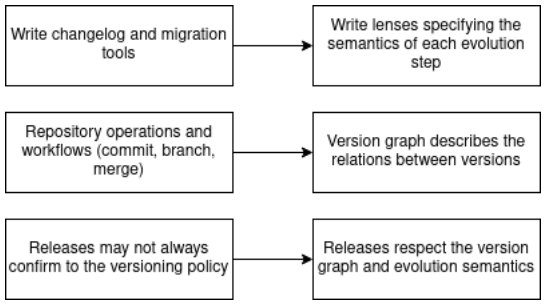


Figure 1 Diagram describing the parallels between version control systems and our approach.

The main contributions of this paper are as follows:

- Extending the work in [5], with method lenses and programs with state and side-effects.
- Instantiating the core calculus in Python to add support for versioned programs.
- Extending a type system for Python to account for versioned programs.
- Introducing a slicing compiler that can generate a projection of a versioned Python program for a single target version.

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

The remainder of this document is structured as follows: section 2 provides a running example to illustrate the ideas in this work; section 3 describes in detail our technical approach; section 4 evaluates the approach using public Python packages and provides the empirical results; section 5 discusses the related work in this space; section 6 discusses the limitations of this work and how we plan to address them; section 7 summarises our results.

2 Example

In Figure 2, we present examples of two Python programs: a versioned library program (Figure 2a) and a client program that uses a specific version of that library (Figure 2b). The versioned program (Figure 2a) contains a version graph describing how the different versions relate to each other (lines 1-3). The class `Name` of this program contains versioned definitions of methods, which are annotated with the version in which they are introduced.

In this example, there are three different versions of the library program, the class `Name`. The program starts with the definition of a version graph for class `Name`. Version `init` is the starting point of the example, and includes the definitions of fields `first` and `last`, and of methods `display` and `set_last`. Note that, in version `bugfix`, the developer introduces a new definition of method `display` (lines 15-17), but otherwise makes no changes. This new definition supersedes the previous one for clients running in the context of version `init`. This is indicated by the `replaces` relationship between versions `bugfix` and `init`.

Finally, in version `full` the developer introduces a new constructor (lines 9-11), where they change the internal state representation of the class (line 11), and a new method, `get_full_name` (lines 21-23). Contrary to version `bugfix`, these new definitions are only meant for clients that are specifically running in the context of version `full`, and does not affect clients in other versions. The code from version `init` is available in the new versions, allowing it to be safely reused.

```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.fname = 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.fname
```

(a) Example of a versioned Python class.

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

(b) Client code for version `full`.

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

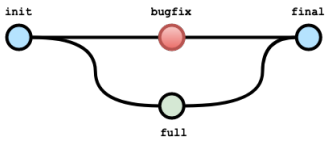
(c) Lenses for fields between different versions.

Figure 2 Client and library code.

The version graph (lines 1-3) is a set of version decorators defining a name for a new version and the type of relationship (`upgrades` or `replaces`) they have with other versions. The type of relationship describes how the new version affects the existing version graph, including versions introduced earlier: if the code is to be implicitly available for clients in previous (related) versions, then the type `replaces` is used. This ensures clients running in a previous version will have the new definitions available without having to update their code (e.g. bug or security fixes). Otherwise, if the new version is a backward-incompatible extension of a previous one (c.f. class inheritance), then the type `upgrades` is used. This ensures that definitions introduced in an `upgrade` version are only available for clients running explicitly in that version. These decorators can be provided by the developer (i.e. while developing the program) or, for existing programs, they can be (naively) inferred from the repository (e.g. the version graph for the repository in Figure 3a is depicted in Figure 3b).

The library program also contains several versioned programming elements, such as constructors (lines 5-11) and methods (lines 15-23), which are decorated with the version in which they are introduced.

The client program (Figure 2b) uses version `full` of the library. It uses the `__init__` method (line 3) introduced in version `full`; the `set_last` (line 4) and `display` (line 5) methods introduced in previous versions; and the attribute `fname`, which is the only field of class `Name` in version `full`.



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

```
1 @version('init')
2 @version('bugfix'
3          , replaces=['init'])
4 @version('full'
5          , upgrades=['init'])
6 @version('final'
7          , upgrades=['full']
8          , replaces=['bugfix'])
```

(b) Version graph inferred from a `git` repository.

Figure 3 Repository and its corresponding version graph.

To resolve methods for the client at version `full`, we perform a method lookup operation taking the version graph into account: the available definition of method `set_last` is the one introduced in version `init`; the definition of method `display` is the one from version `bugfix`, because it replaces the definition in version `init`. Given that these definitions are available at version `full`, clients should be able to safely use them in that context.

However, since these methods use a different internal representation of class `Name`, we can not use them as-is in version `full`, as that would result in an error since the class fields do not match. We propose that, instead of re-implementing all the methods missing from the new representation (of version `full`), the developer provides a mapping between the fields of versions `init` and `full`. This mapping, which we call a *get* lens (Figure 2c), represents the inner semantics of the evolution step.

A *get* lens is introduced by a method with a decorator of the form `@get(v,v',f)`, that maps how field `f` of version `v'` is derived from the state in version `v`. For instance, the lens from version `full` to version `init` of field `first` (Figure 2c, lines 4-8) defines how the field `first`, in version `init`, is obtained from the state of version `full`. Lenses are the building blocks for developers to express evolution steps in the form of code.

The versioning-aware method lookup policy coupled with lenses allows clients to use code from different versions at runtime in a way that respects the version graph and the evolution semantics specified by the developer.

It is also possible to obtain a static, self-contained, slice of a versioned program for a specific target version. This slice will include all code available throughout the version graph for that target version, using the lookup policy described earlier. Again, since this can include code that uses different internal state representations, the slicing procedure rewrites such expressions using the corresponding lenses so that they are correct in the context of the target version, as hinted at earlier.

For example, at version `init`, the available definition for method `display` is the one introduced in the `bugfix` version. As such, this definition must be included in the slice for version `init` (Listing 1). Since both versions `init` and `bugfix` share the same internal representation of class `Name` (version `bugfix` does not define its own class fields), in the slice for version `init` we do not need to rewrite the method `display`, as it already complies with the state of version `init`.

```
1 class Name:
2   def __init__(self, first: str, last: str):
3     self.first = first
4     self.last = last
5   def display(self):
6     return f'{self.last}, {self.first}'
```

Listing 1 Slice of a versioned Python class for version `init`.

At version `full`, the available definition for methods `display` and `set_last` are introduced in versions `bugfix` and `init`, respectively. These definitions must be included in the slice for version `full` (Listing 2). In this case, since version `full` introduces a new internal representation of class `Name` (by changing the fields from version `init`), then the definitions of methods `display` and `set_last` do not conform to this representation, as they are defined in the context of other versions. As such, we need to rewrite the methods using the corresponding lenses for fields `first` and `last` (Listing 2, lines 12-21), so that they match the context of version `full`.

```
1  class Name:
2    def __init__(self, full: str):
3      self.fname = full
4    def display(self):
5      return f'{self.lens_last()}, {self.lens_first()}'
6    def set_last(self, name: str):
7      __name = name
8      self.fname =
9        self.lens_full(first=self.lens_first(), last=__name)
10   def get_full_name(self):
11     return self.fname
12   def lens_full(self, first, last):
13     return f'{first} {last}'
14   def lens_first(self):
15     if ' ' in self.fname:
16       return self.fname.split(' ')[0]
17     return self.fname
18   def lens_last(self):
19     if ' ' in self.fname:
20       return self.fname.split(' ')[1]
21     return ''
```

Listing 2 Slice of a versioned Python class for version `full`.

Both the library and client programs are valid Python programs[1]. The library program can be fed as input to our compiler to extract a projection for a given version. The result of this is a valid Python program without any version annotations. The client program, when fed to an interpreter, is executed following the operational semantics described in this work for multi-version program execution.

3 Design

The ideas presented in this work are mainly language-agnostic, provided the language has support for objects and mutability. To implement these ideas, we instantiate this work in the Python programming language [1]. We chose Python for the following reasons:

- Being a mainstream and widely adopted language facilitates the evaluation of the approach using publicly available repositories of both software libraries and their corresponding clients.

[1] For brevity, the statement to import the decorators, which is necessary for the code to run, is omitted here

- In comparison with other mainstream languages that provide similar features (e.g. Java, C#), Python's less complex syntax usually results in simpler programs [22]. This, again, facilitates the empirical evaluation of the approach, as the code patterns will be simpler to grok.
- From an implementation point of view, Python's dynamic nature allows us to quickly prototype a solution that implements the ideas discussed in this work.

Our implementation works with a large subset of the Python language. Particularly, we do not provide semantics for the versioning of: async/await statements, yield statements, list comprehensions, and modules[2]. The implementation also works under the assumption that the program is typed, either manually or with the help of automated type inference tools [6, 17, 32, 21]. This requirement should not be deemed too restrictive, since the practice of providing type annotations in Python programs is becoming increasingly common [7].

On top of this, we provide a type-checker and a slicing procedure for versioned Python programs. The type-checker is an extension of `Pyanalyze`, a type-checker developed by Quora that also annotates the AST nodes with their corresponding types. We extend Pyanalyze with support for versioned lookup of fields and methods, ensuring the correct type are inferred. The type-checker ensures the soundness of the program against its version graph, and provides the following guarantees for well-typed programs:

- A client that follows the versioning policy, defined in the version graph, will never have their code break.
- If a method needs to be rewritten for a different, related, version, it will always succeed and never produce a type error.

The slicing procedure allows for the projection of code to a specific version, inspired by software product lines and other technical approaches, such as programming variability and CI/CD pipelines. This procedure is implemented on top of a rewriting mechanism to allow library developers to release the code targeting a specific version. For well-typed programs, the slicing procedure ensures that:

- All necessary code for the target version, according to the specification in the version graph, is included.
- All the code included from other, related versions, is well-typed in the context of the target versions, by applying state or method transformations when necessary (using lenses).
- Client code that targets the sliced projection will always type-check, even if the resulting slice includes code from other versions.

The remainder of this section is structured as follows:

- Section 3.1 describes how to define versioned elements in a program.
- Sections 3.2 and 3.3 describe the versioned lookup disciplines for fields and methods respectively.
- Section 3.4 describes the use of lenses, particularly how they affect the result of a slice for a version to ensure that it is well-typed in the presence of elements from different, related versions.
- Section 3.5 describes the rewriting procedure, which is crucial to ensure that code from different versions included in a slice is always well-typed.
- Section 3.6 describes the details of the slicing procedure, that allows library developers to produce the code that targets a specific version.

[2] Although we do not provide versioning semantics for these elements, they can still be used in a versioned program; however, they will not follow the semantics described here.

3.1 Versioned programming elements in Python

To add support for versioning elements in Python programs, we provide the following class and function decorators[3]. The motivation for using decorators is that they are enough to implement the semantics described here, without requiring changes to the language syntax:

```
1 @version(<version>, <replaces>, <upgrades>)
2 @at(<version>)
3 @get(<from>, <at>, <name>)
4 @run(<version>)
```

Listing 3 Decorators for versioned Python programs.

These decorators allow programmers to define new versions of a class (line 1) by providing a name (`<version>`) and the relation to other versions, if any (`<replaces>` and `<upgrades>`); to introduce class methods in a version (line 2); to define class lenses that map how a field or a method (`<name>`), defined in the context of a version (`<at>`), is mapped to the context of another version (`<from>`) (line 3); and to indicate that a function should run in the context of some version (`<version>`) (line 4).

The type-checker ensures the soundness of the decorators presented here. It ensures that all version references are defined; that the version graph is acyclic; that the attribute specified in a lens (`<name>`) exists in the context of its `<at>` version, if it corresponds to a field name, and that it exists on both versions, if it corresponds to a method name; that the return type of a `get` lens matches the type of its corresponding field, or the type of its corresponding method (`<name>`); and that a class method defined at some version (`<version>`) is type-checked against the context of that same version (namely, when resolving field and method types).

The `@run` decorator defines the operational semantics to provide an environment for multi-version program execution. Additionally, we provide static semantics for versioned program slicing, so that developers are able to extract a static projection of code for a specific version. This is based on the concepts of class field and method lookup, version lenses (provided by the developer), and program slicing. We present these concepts in greater detail in the following sections.

3.2 Class field lookup

When type-checking a method of a class defined at some version `v`, we might encounter field access expressions (e.g. `return self.f`). To check such expressions, we need to know 1) if the field exists in that version of the class and 2) what its type is. Since fields can be defined across multiple versions of the same class, the standard (syntactic) field lookup discipline will not yield the correct field set for a given version.

As such, to be able to type-check a program, we need to define a lookup policy for class fields at a given version ($fields(C, v)$), so that we know which fields are or are not available at that version, and what their types are.

In our setting, a version (v) of a class (C) may either redefine its fields (e.g. by introducing or removing a field) or inherit the fields of related versions. The version(s) in which the fields of v are defined are called the *base versions* of v. Each version in the graph has one or more corresponding base versions for a given class ($bases(C, v)$).

[3] For readers unfamiliar with Python, decorators are a method of applying a transformation to a function or a class. Except for the `run` decorator, none changes the decorated function or class (i.e. they act as syntactic hints to infer the version context.)

A field is defined in a version of a class by assigning, in any method at that version, a value to an attribute of the method caller[4] (this is the first parameter of the method, typically named `self`).

This is expressed in rule (Fields-At) (Figure 4). We start by collecting methods available at version v ($methods(C, v)$) and selecting only those tagged for version v ($at(m) = v$, where at performs a syntactic lookup of the method version decorator). Then, we collect all parameters to this method ($parameters(m)$), and inspect the method's body to check if there is an assignment to an attribute of the first parameter ($A_0.f = e \in body(m)$), which corresponds the class instance, and, if so, we collect it's type (T). Finally, we check all previous related versions (W) to ensure that, if f is defined in a previous version, its type is different than the type defined at v ($C \vdash_w f : T' \wedge T \neq T'$). If the type is the same, the field is considered to be inherited, and not explicitly defined at v.

Note that, in this and all subsequent inference rules, we use some helper functions (at, $args$, $parameters$, $upgrades$, $replaces$, and $body$) that perform standard syntactic lookup of nodes in the AST (e.g. $parameters$ returns all parameters for a given method definition).

For instance, in Figure 2a (lines 7 and 8), fields `first` and `last` are introduced in version `init`, in the constructor of that version. Note that the field is only considered to be introduced in this version (as opposed to inherited) if it is not a field, with the same type, of any parent version. We make a small exception (not expressed in rule (Fields-At) for brevity) for the constructor: there, a developer can redefine fields with the same name and type from other related versions[5].

To lookup the bases of a version `v`, we start by collecting the fields defined explicitly at that version given the procedure described earlier ($fields\_at(C, v)$). If this set is not empty, then the base of version v is simply itself (rule (Base-Self)). Otherwise ((rule (Bases))), the bases of `v` are the union of bases from the versions it upgrades and replaces (W), using the same lookup logic.

Finally, to lookup the (versioned) fields for a class C in version v ($fields(C, v)$), we collect the union of fields in all base versions of v (rules (Fields), (Fields-Self)).

Note that, in this setting, these lookup rules are different from those presented in [5]. In particular, in this work, a version of a class can have *multiple* base versions, as opposed to a single one: as such, the lookup logic for fields is also slightly different, since we can lookup fields on multiple base versions.

In the example of Figure 2a, the base version of `bugfix`, in which no fields are explicitly defined, is `init`; the base version of `full` is itself. If we were to add a new version to this program, `final`, that merges versions `full` and `bugfix`

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

then the base versions of `final` would be versions `full` (the base of itself) and `init` (the base of `bugfix`); its fields would be the union of all fields in these versions (`first`, `last`, and `fname`).

The base versions are used in all typing and reduction rules to ensure that the version graph is respected when resolving class fields..

[4] This follows the approach of most Python type-checkers, such as MyPy.
[5] This is to account for the *semantic* (as opposed to syntactic) evolution of a field, when its name and type are still preserved.

$$\frac{\begin{array}{c} m \in methods(C,v) \quad at(m) = v \quad A = parameters(m) \\ A_0.f = e \in body(m) \quad \Gamma \vdash_v e : T \quad W = upgrades(v) \cup replaces(v) \\ \forall_{w \in W} : f \in fields\_at(C,w) \Rightarrow C \vdash_w f : T' \wedge T \neq T' \end{array}}{f \in fields\_at(C,v)} \text{ (Fields-At)}$$

$$\frac{\#fields\_at(C,v) \neq 0}{v \in bases(C,v)} \text{ (Base-Self)} \qquad \frac{\begin{array}{c} \#fields\_at(C,v) = 0 \\ W = upgrades(v) \cup replaces(v) \end{array}}{\forall_{w \in W} : bases(C,w) \subset bases(C,v)} \text{ (Bases)}$$

$$\frac{bases(C,v) \neq \{\ v\ \} \quad W = bases(C,v)}{\forall_{w \in W} : fields(C,w) \subset fields(C,v)} \text{ (Fields)}$$

$$\frac{bases(C,v) = \{\ v\ \}}{fields(C,v) = fields\_at(C,v)} \text{ (Fields-Self)}$$

Figure 4 Inference rules for field and base version lookup.

3.3 Method lookup

Similar to class fields, class methods can be defined across multiple versions of the same class. As such, the standard method lookup discipline will not yield the correct definition of a method for a given version, since there can be multiple definitions with the same name across different versions. Consider the following (abstract) example of a class with three related versions. Version 1 introduces a definition for methods `n` and `m`. The other versions introduce a definition for method `m`:

```
@version('1')
@version('2', upgrades=['1'])
@version('2.1', replaces=['2'])
class C:
  @at('1')
  def n(self): ...
  @at('1')
  def m(self, x): ...
  @at('2')
  def m(self, x): ...
  @at('2.1')
  def m(self, y): ...
```

Listing 4 Evolution of a method between versions.

The intuition here is that clients running at version 1 should use the definition of `m` introduced in that version, since there is no definition of `m` that replaces the one from version 1. Clients at version 2 should use the definition of `m` introduced at version 2.1, since this is declared as a replacement over version 2; and clients at version 2.1 should use the definition of `m` introduced in that version. For method `n`, all versions use the definition from version 1, which is local to version 1 and inherited in version 2 (and, subsequently, in version 2.1).

To comply with the version graph, we must also define a lookup policy for class methods at a given version, as illustrated above. The reader may notice that the definition of `m` introduced in version 2.1 – which should be available to clients in version 2 – has a different interface from the local definition of `m` at version 2 (parameter `x` is renamed to `y`). Intuitively, this means that client code is written for version 2, which may call method `m` using keyword

arguments (i.e. `C().m(x=...)`), which type-checks against the local interface, cannot simply use the new definition, as that would introduce a type error (no parameter named `x`, missing parameter `y`).

As such, the lookup of a method `m` at version `v` must return both the interface (to comply with clients targeting v) and its implementation (to comply with new definitions introduced in replacement versions). The lookup discipline for methods works in the following order:

Local definition. Search for a local definition of `m` introduced at version `v`. If any is found, that definition corresponds to the interface and implementation of `m` for version `v`.

Parent definition. If no local definition was found, search all versions that `v` either upgrades or replaces for a definition of `m`. If any is found, that definition corresponds to the interface and implementation of `m` for version `v`. If there are multiple matches, they must be the same definition. Otherwise, a conflict occurs, and the program is not well-typed.

Replacement definition. Finally, search all replacement versions of `v` for an implementation of `m`. If no interface was found yet (either locally or inherited from a parent version), it means `m` was introduced in a replacement version, so that interface is the one available to clients at version `v`. In any case, if there are multiple matches, they must be the same definition. Otherwise, a conflict occurs and the program is not well-typed.

This lookup policy is illustrated in the example above, where, for version 2, the interface of `m` is the one from the local definition at 2, and the implementation is from the definition at version 2.1. Later in this section, we describe how to use one interface with a different implementation. For now, it's important to retain that the lookup of methods must respect the version graph, and take into account how the client code is typed (i.e., against which interface). For method `n`, the interface and implementation at version 2 is the one inherited from version 1.

This lookup policy is used to type-check function calls at a given version; to detect missing lenses between methods of different versions, when the interface differs from the interface of the implementation (e.g. method `m` at version 2 in the previous example); to detect conflicts in the version graph; to select method definitions when providing a slice for a target version; and to find the code to execute at runtime.

3.4 Version lenses

The lookup policy for class methods, described earlier, allows for a version `v` of a class to use methods introduced in another (parent or replacement) version `t`. In such cases, there are two situations where we need to pay special attention:

- The implementation provided by the lookup policy is defined at another version, `t`, which has a different state representation (i.e. different base versions from `v`).
- The interface provided by the lookup policy is different than the interface of the implementation, and the signatures of the interfaces differ (e.g. method `m` in version 2 of the previous example).

In the first case, since the state representation of the class is different, the method body may not comply with the representation of version `v`: this is illustrated in Figure 2a, where method `display`, available for version `full`, complies with the state of another version (`init`). Intuitively, this means that, to use such implementations, we must introduce a mapping between the fields used in the method body and the fields of the target version (in this case, version `full`), otherwise we would introduce type errors.

In the second case, since the interface of the method is different from the interface available for clients at version `v`, we can not simply use the new definition, since client code is typed against a different interface (as such, doing so would introduce a type error). Intuitively, this means that, to use the new implementation, while preserving the type correctness of clients at `v`, we must introduce a mapping between the two interfaces.

These mappings, called *lenses*, for fields and methods, are described in more detail in the remainder of this subsection.

3.4.1 Field lenses

Consider again the example in Figure 2a, where the interface and implementation of method `display` for version `full` is from version `bugfix` (Figure 2a, line 17). In version `full`, class `Name` has no `first` nor `last` field, so this definition of method `display` is not well-typed in version `full`. For a client in version `full` to correctly use this code, we need to rewrite the method body, so that it complies with the desired state.

The type system requires that the developer defines the necessary lenses at version `full` for the fields at version `init` (Figure 2c). Intuitively, these lenses express how each field evolved from the state of version `full`. In this case, the lenses are simple: split the full name on a whitespace, if any, and return the corresponding component (if it exists). Later, when projecting the code for version `full`, the lenses are used to rewrite field expressions in the method body so that they are well-typed in the context of version `full`.

Each field lens is a standard class method, annotated with a `@get` decorator, of the form `@get(<at>, <from>, <name>)` (Listing 3, line 3), with a single argument (the method caller, `self`). In practical terms, the implementation of the lens answers the question: "In the context of version `<at>`, how do I represent the field `<name>` of version `<from>`?". The type system ensures that the body of a lens is type-checked in the context of its `<at>` version (in this case, version `full`); that `<name>` is a field in version `<at>`; and that the return type of the lens matches the type of field `<name>` in version `<at>` (in this case, `str`).

Field lenses are a suitable mechanism for modelling common software evolution patterns, such as renaming a field, changing its type, or refactoring its representation (as in the example of Figure 2a, where we join both fields in a single one).

Evolution patterns that mostly concern text manipulation (as opposed to program semantics), are typically considered breaking changes (e.g. changing the name of a field in a class). In our setting, these patterns are well supported, as it is always possible for the developer to express such a pattern in the form of a lens. If they do so, then these patterns can be applied successfully without introducing breaking changes.

For instance, a lens to rename a field (`f`, to `t`) from version `1` to `2` is expressed by:

```
1 @get('1', '2', 't')
2 def rename_f_t(self):
3   return self.f
4 @get('2', '1', 'f')
5 def rename_t_f(self):
6   return self.t
```

and allows clients in version `1` to use code from version `2`, thus making it a non-breaking change; and for code in version `2` to reuse code from version `1`, so that the developer does not need to rewrite methods that use field `f`. In these cases, the lenses can be synthesised with the help of editor tools, instead of manually implemented by developers (c.f. refactoring tools in Bides)[6].

[6] This is discussed in greater detail in section 6.

Evolution patterns that concern program semantics, such as changing the type of a field, are not always possible to model with a lens. For instance, in the previous example (Figure 2c), we showed how to model such a pattern (refactoring two fields into one).

However, consider the case where we want to refactor a list (in version `1`) into a dictionary (in version `2`, that replaces `1`). Assuming the semantics of the program dictate that the elements of the list correspond to the values in the dictionary, then we can devise a lens that maps the dictionary to the list (e.g. `return list(self.data.values())`, where `data` is the dictionary), which allows the developer to reuse code from version `1` while working in the context of version version `2`. But what about the other way around? If we want to map the list to a dictionary, it may not be possible to infer the keys[7] (for instance, if they are provided by the client when creating the dictionary).

In such cases, the type system detects an error: the `replaces` relationship between the two versions defined in the graph implies that clients in version `1` should be able to use all code from version `2` – but without a lens this is not possible.

In such cases, where the developer can not define a lens for a field (e.g. `data`), then the code does not comply with the version graph, as the lens is missing. Intuitively, this indicates that the developer introduced a breaking change from version `1` to version `2`, so clients can not migrate automatically. To fix this, the developer must change the version graph and use the `upgrades` relationship between both versions instead.

This typing discipline reflects the nature of a breaking change when evolving class fields, as it prevents the developer from issuing such a change in a replacement release, which, in the absence of a lens, would make the client code crash. Instead, by using the `upgrades` relationship, the developer instructs clients to adapt manually, by migrating to the new version and then type checking their code in that context, correcting manually for any errors.

3.4.2 Method lenses

Similar to field lenses, method lenses map how (possibly different) interfaces of the same method evolve between different, related versions.

Consider again the example in Listing 4, where method `m` is refactored in version `2.1` by renaming parameter `x` to `y`.

In this case, the `replaces` relationship implies that clients in version `2`, whose code is written using the interface from that version (i.e. with parameter `x`), should be able to use the new implementation of `m` automatically, without their code breaking:

```
1  @at('2')
2  def client():
3    return C().m(x=...)
```

Since the client code is written against the interface defined at version `2`, to use the new implementation, the type system requires that the developer define a method lens at version `2` for method `m` at version `2.1`. This lens expresses how the method evolved from one version to the other, so that clients can safely use this new definition without rewriting their code. Later, when projecting the code for version `2`, the lenses are used to rewrite method definitions, so that they conform to the interface of version `2` while using the implementation from version `2.1`. The following is an example of a method lens that renames argument `x` to `y`, while otherwise preserving the semantics:

[7] Whether this is possible or not depends on the program's intended semantics.

```
1 @get('1', '2', 'm')
2 def lens_m(self, f: Callable[[C, P], T], x: P) -> T:
3     return f(y=x)
```

Listing 5 Method lens to rename a parameter

A method lens is a standard class method, annotated with a `@get` decorator (Listing 3, line 3). The body of each lens function is type-checked in the context of the `<from>` version. The type system ensures that the method (`<name>`) is available in both versions (`<from>` and `<at>`), and that the return type of the lens function (`T'`) matches the return type of method `<name>` in version `<from>`.

A method lens takes a reference to the instance object (first parameter, `self`); a reference to the method definition in version `<from>` (`f`), whose signature matches the type of method `<name>` in version `<at>` (in the example, the signature of `f` corresponds to the type of `m` at version `2.1`); and all positional and keyword arguments that method `<name>` takes in version `<at>` (in this case, `x`). The parameter `f` is to aid the developer statically expressing how the calls map between the two versions.

In the above example, `f` is a reference to the definition of `m` in version `2`. As such, the developer can express how a method call from a client in version `1` maps to the corresponding method in version `2`, in this case by calling `f` and passing `x` as the value to `y`.

Method lenses are a suitable mechanism for modelling common software evolution patterns, such as adding/removing/reordering parameters, changing the type of parameters, and changing a method's return type.

Evolution patterns that mostly concern text manipulation (as opposed to program semantics), are typically considered breaking changes (e.g. changing the name of a parameter in a method). In our setting these patterns are well supported, as it is always possible for the developer to express such a pattern in form of a lens, as hinted in the previous example. If they do so, then these patterns can be applied successfully without introducing breaking changes for clients.

The same approach can be used for methods that evolve semantically, not just textually. Consider the following example, of a method that returns a boolean, and is refactored to return `0` instead of `True` and `1` instead of `False` (the implementation details are omitted here as they are not relevant):

```
1 @at('2')
2 def m(self) -> bool: ...
3 @at('2.1')
4 def m(self) -> int: ...
```

Again, the `replaces` relationship between version `2` and `2.1` indicates the developer wants clients in version `2` to use the refactored method introduced in `2.1`. As such, the type system requires that they provide a lens expressing how the result of a call to the new implementation, defined in version `2.1`, can be mapped to the type of the interface, introduced in version `2`. The lens takes as parameter `f` a function that returns an `int`, matching the definition in version `2.1`, and returns a value of type `bool`, matching the definition at version `2`:

```
1 @get("2", "2.1", "m")
2 def lens_m(self, f: Callable[[C], int]) -> bool:
3   return f() == 0
```

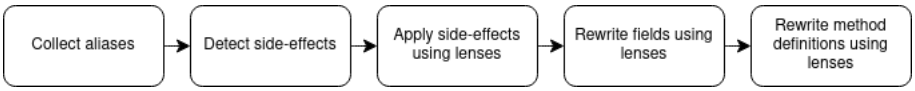


Figure 5 Diagram describing the pipeline of the rewriting procedure.

A client using method `m` in version `2`, such as:

```
@at('2')
def client():
  return not C().m()
```

can use the new code from version `2.1` without refactoring their code, since the mapping from `int` to `bool` (the expected return type for clients in version `2`) is provided by the lens.

Similar to field lenses, notice how, when a method evolves semantically, it may not always be possible to devise a lens that models the evolution step, depending on the semantics of the change. Once again, in those cases, to fix the typing error (missing lens), the developer must change the version graph, and define the new version as an upgrade, indicating to client that they must migrate manually and refactor their code to adjust to the new interface.

Method lenses allow clients to use new code without any need for manually refactoring, so developers can express versioning workflows with typical breaking changes, such as changing method signatures, and automatically have clients complying with the versioning policy established in the respective version graph.

3.5 Rewriting procedure

The concept of lenses, described in the previous subsection, allows clients to use code from a version different than the one their code is targeting, even in the presence of different state representations (i.e. class fields) and different method interfaces, respecting the evolution pattern specified by the developer. To do so, we must rewrite the methods that are defined in different versions, so that they conform to the client's version context.

This section describes the procedure to rewrite methods defined at some version `v` so that they match the context of the client version `t`. By doing so, we allow clients to use the new methods without introducing type errors, accounting for side effects and ensuring they are preserved across different version contexts.

The diagram in Figure 5, illustrates the pipeline of the rewriting procedure to rewrite a definition of method `m` from version `v` (where it is defined) to version `t` (which the client is running). Below, we describe each step of this pipeline in detail.

3.5.1 Collecting aliases

In Python, object references are passed by value. When we assign the value of an instance field to a variable, if the variable is mutated, the changes will be reflected back in the instance field. As such, to (statically) detect side-effects on instance fields, we first need to keep track of aliases (to fields) in a given method. Consider the following example of a method `m` at version `v` that appends an element to a list field:

```
@at('v')
def m(self):
  x = self.f
  x.append(1)
```

In this example, simple static analysis would not be enough to detect that `self.f` is mutated (since the `append` method is called on a variable, not on a field). So, we need to known that `x` is an alias to a class field (in this case, `f`). To do so, we perform static analysis on assignment statements to collect the aliases of class fields in scope in each method.

From the assignment statement in the above example, we can infer that `x` is an alias to the field `f` of the method caller's class. In some cases, we can also detect if assigning the result of other types of expressions such as a variable, the result of a function call, a list index expression, and so on, also results in a reference to a mutable object field.

We collect all aliases of mutable object fields in a method to use in the next steps, so that we can ensure that the side effects in the code for version `v` are correctly applied when we rewrite it for version `t`.

3.5.2 Detecting side-effects

Now that we have collected all aliases to fields within a method, we can start performing static analysis to detect side-effects. This is crucial to ensure that side-effects on fields of other versions are correctly carried over to the target version, `t`, to which we are rewriting the code. Conversely, detecting cases where no side-effects are produced, avoids having redundant rewrites of such expressions.

The intuition here is the following: we will be using field lenses (in this case, from version `v` to `t`) to rewrite fields (this is detailed further in this section); but, since field lenses are pure functions (i.e. they do not mutate the object calling them), the side effects would be lost if we simply replaced the field by its corresponding lenses – in which case, the side effect would apply to the result of the lens, but not to the current state representation of the class in version `t`.

Consider the following example of method `m` defined at version `v`, where the call to method `pop` will have a side effect on the state of the object (by removing the first element of the list stored in field `f`).:

```
1 @at('v')
2 def m(self):
3   x = self.f.pop()
```

We detect side effects to object fields by identifying expressions where the field is passed as a mutable reference to a method[8](in this case, method `pop`), so that we can preserve them when rewriting.

To do so, we start by extracting the field to a (new) local variable, rewriting its occurrence in the assignment, and then assigning back to the field the value of the local variable:

```
1 @at('v')
2 def m(self):
3   _x = self.f
4   _x.pop()
5   self.f = _x
```

This logic is expressed in the following rules. In rule (Rw-Field-Call), we rewrite method calls that mutate the calling object. We start by collecting all methods available in the target version t ($methods(C, t)$), and selecting those defined at a version other than t (i.e.

[8] It is not always possible to do so by static analysis. For instance, the functions from the Python standard library, such as `pop`, are compiled from C code, which we can not analyse statically. In these cases, we naively assume that function mutates its arguments.

those that need rewriting). Then, for each method, we inspect its body to check if there is a method called on an object ($obj.m'(A) \in body(m)$, where A are the arguments passed in the call). Finally, we check if method m' mutates its caller: this is given by the helper function $mutates(T, m', P_0)$, where T is the object's type and P_0 is the first parameter of method m' (i.e. its caller). If so, we need to rewrite the method call. As shown in the previous example, we start by declaring a new, unused, variable ($x = fresh()$) and assign the object to it ($x = obj$). Then, we call the method on this variable, passing the same arguments rewritten for the context of version t ($x.m'(A')$, where A' is the result of rewriting arguments A). Finally, we assign the value of x back to the object.

In rule (Rw-Field-Args) (Figure 6), we rewrite method calls that mutate their arguments. Similar to the previous rule, we start by collecting all methods available in the target version t ($methods(C, t)$), and selecting those defined at a version other than t (i.e. those that need rewriting). Then, for each method, we inspect its body to check if there is a method called on an object ($obj.m'(A) \in body(m)$, where A are the arguments passed in the call)[9]. Finally, we select all arguments which are mutated by m' (A'), and we create a fresh variable for each of these arguments (X). To rewrite the call, we start by assigning to each fresh variable the current value of its corresponding argument ($x_i = A'_i$). Then, we call the method, replacing each (mutated) argument with its corresponding variable ($\{^{A'}/_{X}\}$). Finally, we assign to the arguments the value of their corresponding variable (which was mutated in the method call).

$$\frac{\begin{array}{c} m \in methods(C,t) \quad v = at(m) \quad v \neq t \\ obj.m'(A) \in body(m) \quad \Gamma \vdash_v obj : T \quad P = parameters(m') \\ mutates(T, m', P_0) \quad A \ {}_v\!\rightsquigarrow_t A' \quad x = fresh() \end{array}}{obj.m'(A) \ {}_v\!\rightsquigarrow_t \ x = obj; x.m'(A'); obj = x} \text{ (Rw-Field-Call)}$$

$$\frac{\begin{array}{c} m \in methods(C,t) \quad v = at(m) \quad v \neq t \quad obj.m'(A) \in body(m) \\ \Gamma \vdash_v obj : T \quad A' = \{\ a \in A \mid mutates(T, m', a)\ \} \\ X = \{\ fresh() \mid a' \in A'\ \} \end{array}}{obj.m'(\overline{a}) \ {}_v\!\rightsquigarrow_t \ \overline{x_i = A'_i}; obj.m'(\{^{A'}/_{X}\}); \overline{A'_i = x_i}} \text{ (Rw-Field-Args)}$$

Figure 6 Rules to rewrite fields.

Notice how the code is still typed for the context of version `v`. This procedure is used whenever object fields (or aliases) are passed as mutable arguments to functions, as described earlier, and also across all language statements, such as loops, return, try-raise, and so on. For example, consider the following example, where an object field is mutated as a condition of an if statement:

```
1  @at('v')
2  def m(self):
3    if self.f.pop():
4      return True
```

To rewrite this statement, we create two new references: one for the object's fields (`_x`, as described earlier); and another for the condition value of the if statement (`_y`). Finally, to

[9] Note that this rule also applies for functions (e.g. `sort`, which sorts a list in place) and not just methods. For brevity, that case is elided here, although it follows the same logic

apply the side effects of the method call (`pop`), we assign the value of this reference back to the object's field, before executing the if statement. This ensures the semantics of the (original) code in version `v` are preserved:

```
@at('v')
def m(self):
  _x = self.f
  _y = _x.pop()
  self.f = _x
  if _y:
    return True
```

By the end of this step we should have all side effects to fields expressed as simple assignments of the form `self.f = _v`, where `_v` is the variable holding the value after side-effects are applied.

3.5.3 Rewriting assignments to fields

Following up on the previous step, we now have all side effects expressed as assignments to fields. To rewrite the assignments to the target version `t`, we use the corresponding lenses.

The intuition here is the following: when the developer defines a lens (from `v` to `t`) for a field (`f`), the lens expresses how to compute the value of the field given the state of the object at version `t`. As such, any fields of `v` that appear in the lens will be affected by an assignment to field `f` in the context of version `v`.

Consider the example of method `set_last` (Figure 2a), that assigns a value to field `last`. Since this method is available at version `full`, we must have a way to express how a change to field `last`, in version `init`, affects the state of this version. This is called a *put* lens.

By analysing the lenses from version `init` to version `full` we see that field `last` appears in the lens for field `fname`:

```
@get('init', 'full', 'fname')
def lens_full(self):
  return f"{self.first} {self.last}"
```

In practice, this indicates that the value of the field `fname`, in version `full`, is affected by the value of field `last`, in version `init`: if we run the code for this lens, replacing `self.last` with the value that we are assigning to the field, we obtain the matching side effect in field `fname` that results from the assignment. As such, the developer need not provide a definition for a *put* lens, as we can synthesise one from its corresponding *get* lens.

To do so, we add a parameter for each field referenced in the *get* lens, and replace the field reference in the lens body with the (matching) function parameter. The synthesised *put* lens for field `fname` at version `full` is:

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

To rewrite an assignment for field `f`, we start by synthesising all necessary *put* lenses. These are synthesise from the corresponding *get* lenses defined at version `v`, for any field (`f'`) from version `t`, that make a reference to field `f` in their body (rule (Synth-Put-Lenses)). This ensures that a change to field `f`, in version `v`, has its side effects applied to fields `f'` of version `t`.

In the previous example there was only one lens in version `init` using field `last`, so that is the one that is synthesised; in cases where there is more than one, we synthesised all necessary put lenses and unfold the original assignment into multiple assignments, each using a *put* lens for the affected fields. For instance, in Figure 2c, in the lenses from version `full` to version `init`, field `fname` appears in two lenses. This means that both lenses would be needed to correctly apply side effects to field `fname` when rewriting such an assignment to version `init` (which would reflect on fields `first` and `last`, using the same logic).

Now that we have the necessary *put* lenses synthesised, we can use them to rewrite the assignment. For example, to rewrite method `set_last` for version `full`, we can use the synthesised *put* lens (`lens_full`) to apply the side effects resulting from the assignment. To do so, we pass the assigned value (`name`) as the argument to the corresponding field (`last`). To all other unaffected field parameters (i.e. `first`), we pass their current value (in the context of the version where the method is defined, i.e. `self.f` for field `f`).

Finally, to ensure that this code is well-typed in the context of version `t`, we replace all field references (in this case, `self.first`) using the corresponding *get* lens (Figure 2c, line 2), as described in the next subsection. This is expressed in rule (Rw-Assignment), where we start by detecting assignments to fields in the method's body ($obj.f = e$), then we rewrite the right-hand side to match the context of version t (e), and finally collect all *put* lenses that affect field `f` (P). To rewrite the assignment, for each field of t affected by the assignment (F_i''), we assign the result of its corresponding put lens (P_i), passing the rewritten value for all unaffected fields ($F_i = F_i'$), and the rewritten assigned value for the assigned field ($f = e'$). The translation of the assignment for version `t` is then:

```
1  def set_last(self, name):
2    self.fname = self.lens_full(first=self.lens_first(), last=name)
```

$$\frac{\begin{array}{c} m \in methods(C,t) \quad v = at(m) \quad obj.f = e \in body(m) \\ \Gamma \vdash_v obj : T \quad e \ {}_v\!\rightsquigarrow_t e' \quad F = \{\ f_i \mid f_i \in fields(C,v) \wedge f_i \neq f\ \} \\ F' = \{\ f_i \ {}_v\!\rightsquigarrow_t f_i' \mid f_i \in F\ \} \\ F'', P = \{\ f', put\_lenses(T,v,t,f) \mid f' \in fields(C,t)\ \} \end{array}}{obj.f = e \ {}_v\!\rightsquigarrow_t \overline{obj.F_i'' = obj.P_i(F_i = F_i', f = e')}} \text{ (Rw-Assignment)}$$

3.5.4 Rewriting field references

As described earlier (section 2), if the method at version `v` contains expressions of references to class fields (e.g. `obj.f`), we need to rewrite these expressions, using the corresponding *get* lens, so they comply with the state of version `t`.

In this step, we do not account for field references to which the previous cases apply as those are already compliant with the target version (i.e. assignments, aliases, and function call arguments).

For example, in Figure 2a, the implementation of method `display` for version `full` is defined in the context of another version, `bugfix`. This definition makes references to fields (`first` and `last`) that are not defined in the context of version `full`. As such, we need to rewrite these references so that they comply with the state of version `full`.

To do so, we replace field occurrences with a call to their corresponding *get* lens provided by the developer (Figure 2c, lines 1-10). This ensures that the code is well-typed in the context of the target version `full`, and that it respects the evolution semantics described by the developer in the implementation of the lenses (rule (Rw-Field)):

```
1 def display(self):
2   return f'{self.lens_last()}, {self.lens_first()}'
```

$$\frac{m \in methods(C,t) \quad v = at(m) \quad obj.f \in body(m) \\ \Gamma \vdash_v obj : T \quad f \in fields(T,v) \quad l = lens(T,t,v,f)}{obj.f \ {}_v\!\rightsquigarrow_t \ obj.l()} \ (\textsc{Rw-Field})$$

3.5.5 Rewriting method definitions

As described earlier, method lenses allow clients to use new method interfaces and implementations without any need for manually refactoring, allowing developers to express versioning workflows with typical breaking changes and making them non-breaking. This mechanism applies to cases where the method signature or semantics have changed between versions.

Consider again the example in Listing 4 with two implementations of the same method, `m`, where a parameter is renamed from `x` to `y`, a typical breaking change. The semantics of this change is expressed in the lens provided by the developer for this method (Listing 5). With this lens, we can allow the clients of version `2` to use the definition of `m` introduced in version `2.1` without refactoring their code.

To do so, we use the method lens to rewrite the implementation of method `m` when extracting a slice for version `2`. The intuition here is that we want to preserve the interface of version `2`, since that is what the client code is written against, while using the implementation of version `2.1`, which the developer introduced as a replacement for the old definition.

To rewrite the definition using the method lens, we start by adding the new definition of method `m` (from version `2.1`) to the program and renaming it, so that we don't introduce a conflict (line 2). Then, we rewrite the body of this method, according to the rewriting procedure described in this subsection, so that it complies with the state of `2`. Then, we rewrite the lens function, by removing the parameter `f` (line 4) and replacing it in the body with a reference to the (renamed) method from version `2.1` (line 5). Finally, we rewrite the body of method `m` in version `2` to make a call to the method lens (line 7).

```
1 class C:
2   def __v2_1_m(self, y: int) -> int: ...
3   def lens_m(self, x: int) -> int:
4     return self.__v2_1_m(y=x)
5   def m(self, x: int):
6     return lens_m(x=x)
```

The rewriting of method definitions across different versions ensures that clients can write their code against the old interface while taking advantage of the new implementation. This concludes the presentation of the rewriting procedure, which we will use to produce a slice of the program that targets a specific version, as described in the following subsection.

3.6 Program slicing

We propose a slicing procedure, built on top of the rewriting procedure described earlier, applying static program transformations, to extract code for a specific target version from a versioned program. The result should include all code available in that version according to the versioning policy provided by the version graph.

The intuition for the slicing procedure is to be able to project code for a specific release, similar to techniques used in software product lines or variability programming settings.

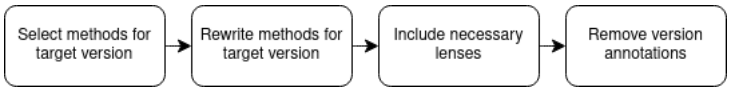


Figure 7 Diagram describing the pipeline of the slicing procedure.

At its core, the slicing procedure relies on the rewriting procedure to ensure that any code that is reused from other versions is safe to use in the context of the target version `t`, respecting the evolution semantics specified by the developer in the lenses.

Now that we have defined our rewriting procedure, we can use it to produce a slice for a target version `t`. In Figure 7, we present a diagram with the pipeline of the slicing procedure. Below, we describe each step of this pipeline in detail:

Select methods. We start by selecting the method definitions (their interface and corresponding implementation, as defined by the lookup policy described earlier) of `C` that are available at version `t`, according to the versioning policy of the version graph and the lookup functions described in subsection 3.3. This is expressed in rule (Sl-Methods).

$$\frac{M = methods(C, t)}{M \subset methods(slice(C, t))} \text{ (SL-Methods)}$$

Rewrite methods. Given the methods selected in the previous step, we rewrite those that either 1) have an implementation defined at a (base) version other than `t` or 2) the developer has provided a lens for (i.e. the methods whose semantics have changed). We rewrite these methods using the procedure described earlier so that they match the context of the target version `t`.

Select lenses. Since the rewriting procedure may need lenses to rewrite expressions for the context of `t`, we will need to include those (and only those) in the final result. We collect all necessary *get* and *put* lenses (rules (Sl-Get-Lenses) and (Sl-Put-Lenses)) that are required for the rewriting procedure, rewrite them to match the context of version `t`, and include them in the resulting slice for this version. Rule (Sl-Get-Lenses) expresses the logic for including *get* lenses: we check each method (m) of the target version (t) that is defined in a different version (v) and, if its body contains a field access expression on an object ($obj.f$) of type T, and a lens is defined between versions v and t of class T for field f ($lens(T, t, v, f)$), we include it in the final slice of T. Rule (Sl-Put-Lenses) expresses the logic for including *put* lenses: we check each method (m) of the target version (t) that is defined in a different version (v) and, if its body contains a field assignment expression on an object ($obj.f = e$) of type T we iterate over all (get) lenses for fields of version t in class T (L), and finally we include the ones where field f is used (L_f).

$$\frac{\begin{array}{c} m \in methods(C, t) \quad v = at(m) \quad v \neq t \quad obj.f \in body(m) \\ \Gamma \vdash_v obj : T \quad l = lens(T, t, v, f) \quad C' = slice(T, t) \end{array}}{l \in methods(C')} \text{ (Sl-Get-Lenses)}$$

$$\frac{\begin{array}{c} m \in methods(C, t) \quad v = at(m) \quad v \neq t \quad obj.f = e \in body(m) \\ \Gamma \vdash_v obj : T \quad L = \{\, lens(T, v, t, f') \mid f' \in fields(T, t) \,\} \\ L_f = \{\, l \mid l \in L \wedge self.f \in l \,\} \quad C' = slice(T, t) \end{array}}{L_f \subset methods(T')} \text{ (Sl-Put-Lenses)}$$

$$\frac{\begin{array}{c} l = lens(C, v, t, f) \\ F = \{\, f \mid s \in body(l) \wedge s = self.f \,\} \quad l' = l\{\, args = F \,\} \end{array}}{put\_lens(C, v, t, f) = l'} \text{ (Synth-Put-Lenses)}$$

Remove version annotations. Finally, to produce the slice for version `t`, we remove any version annotations that may exist, so that the end result is a standard Python program that can be fed to the interpreter to be executed.

In Listing 2, we present the slice of the program in Figure 2a for version `full`. The resulting slice includes the fields and methods of version `full` as defined by the lookup policies described earlier, and any lenses that are necessary to rewrite statements from other versions (e.g. lines 5 and 9).

4 Evaluation

In this section, we empirically evaluate the applicability of our approach by answering the following research questions:

RQ1. Can library developers mitigate the occurrence of breaking changes in common evolution patterns?

RQ2. Can clients update a library dependency without having to manually refactor their code to account for breaking changes?

4.1 Evaluation design

To setup the evaluation, we started by gathering a set of publicly available Python software libraries with at least two major version releases, `v` and `t`. We opted for popular packages, since, given their widespread adoption, these are likely to affect a higher number of clients. We also took care to select packages with different kinds of changes such as renaming methods, fields, or changing method signatures, to better illustrate the applicability of our approach.

For each library, `L`, we start by defining the version graph. Since we are trying to turn major versions into minor (i.e. so that clients can upgrade transparently without breaking), we define the later version as a replacement of the previous (major) version:

```
1 @version('v')
2 @version('t', replaces=['v'])
```

Then, we select the commit tagged for versions `v` and `t` and add the respective version annotations (`at('v')`, `at('t')`) to the methods in each commit. Now that we have the version graph defined and all elements annotated with their corresponding versions, we run the type-checker to detect any missing lenses. We implement the missing lenses, if possible, according to the description stated in the migration guide for `L`, which corresponds to the semantics the developers intend for each change.

At this stage, we should have a program that type-checks against its version graph (again, if the lenses are possible to implement). Finally, we extract a slice of library `L` for version `v`. Now, we can evaluate the applicability of our approach in two ways:

Using client code. In some cases, we were able to use a client program (`C`) to check if the slice of `L` for version `v` conforms to the migration semantics the library developer defined. To do so, we select the commit of `C` that targets version `v` of `L` and type-check this against the slice of `L` for version `v`. If the program type-checks, the approach is validated.

Guided by examples in migration guide. As the reader may have understood by now, this evaluation requires a bit of manual labour to setup. This is expected, since the ideas described in this work are more suitable to be applied throughout the development cycle, instead of applied to existing codebases. As a result, it was not feasible to validate some libraries against existing client code. In those cases, we simply used the examples stated in the migration guide (or modelled them ourselves if there are none), and validated the approach in the same way described in the previous point.

4.2 Evaluation results

We conducted our experiments using the libraries listed in Table 1, using client code where possible. The table shows the selected library, the start and target versions we chose, the client used to validate the approach (if any), the number of breaking changes[10], and how many we were able to successfully model.

Table 1 Libraries and clients selected for the experimental evaluation.

Library	Start version	Target version	Client	Changes	Successful
tensorflow	1	2	gpt-2	19	14
emoji	1.7	2	ntfy	1	1
metaapi-python-sdk	22	23	—	2	2
netbox	3.5	3.6	—	5	4
twillio-python	7	8	—	17	14

The following is a summary of the results we obtained for each library:

tensorflow. Out of 19 breaking changes that affected the `gpt-2` package, forcing its developers to migrate manually to support version 2.0 of `tensorflow`, we were able to model 14 successfully. The changes we were unable to model relate to the refactors of `tensforflow` between the two versions, that essentially force the developer to restructure (and not just rewrite) their code – and our approach does not provide any mechanism for specifying such changes (i.e. clients must always migrate manually). The most relevant example in this case study is the removal of the `tf.multinomial` method. The migration guide points client developers to use another method instead, `tf.random.categorical`. This change can be modelled in our approach by providing a method lens for the `tf.multinomial` method from version `1` to version `2`, and implement the lens to use the `tf.random.categorical` method instead.

emoji. Version 2 of the `emoji` package introduces 2 breaking changes, one of which affects the client package `ntfy`. This change involves removing a boolean parameter, `use_aliases`, which defaults to `False`, from method `emojize`. In version 2, client developers should pass `language='alias'` instead of `use_aliases=True`. Our approach is able to model this successfully, by defining a method lens for `emojize` that passes the appropriate value to `language` depending on the value of `use_aliases`. As such, the client package does not need to manually refactor to use the new version.

metaapi-python-sdk. The 2 breaking changes introduced in version `23` of this package involve the rename of a method (`enableMetastatsHourlyTarification` is renamed to `enableMetaStatsApi`), and the rename of a field (`metastatsHourlyTarificationEnabled` is renamed to `metastatsApiEnabled`). Both are supported in our setting and can be successfully implemented, by using a method and a field lens respectively, to allow clients to migrate without refactor.

netbox. Out of the 5 breaking changes introduced in version `3.6`, we were able to model 4. The change we were unable to model concerns a dependency (PostgreSQL) that must be upgraded. Since we do not yet support versioning of modules, this is not possible in our setting. The remainder of the changes involve: renaming a field (`device_role` field on the `Device` class is renamed to `role`); changing the name and type of a field

[10] When determining the number of breaking changes, we ignored some which fall outside of the scope of this work, particularly when concerning external dependencies.

(field `choices` from the `CustomField` class is renamed to `choice_set`, and its type is changed from a dictionary to `CustomFieldChoiceSets`); removing fields from a class (fields `napalm_driver` and `napalm_args` are removed from the `Platform` class); and changing the return type of a method (reports and scripts are returned within a `results` list). All of these were successfully modelled in our setting.

twillio-python. Out of the 17 breaking changes introduced in version `8`, we were able to model 14. The changes we were unable to model concern the renaming of classes: class `ConversationsGrant` is replaced by `VoiceGrant`; and class `IpMessagingGrant` is replaced by `ChatGrant`). We can not model such cases since our type system restricts method lenses (in this case, the constructor method `__init__`) to return the same type as the original definition[11]. The remainder of the changes involve renaming methods (12 instances) and changing the signature of a method, by removing a parameter (2 instances). All of these were successfully modelled in our setting.

4.3 Evaluation answers

From the results presented in the previous section, we answer the research questions with:

RQ1. Yes, library developers can mitigate (and in some cases, eliminate) the occurrence of breaking changes using our approach.

RQ2. Yes, in most cases clients can update without refactoring their code manually.

5 Related work

Program slicing. In his seminal paper, Weiser [31] describes program slicing as a method for automatically decomposing a program, starting from a subset of its behaviour, and reducing it to a minimal form which still produces that behaviour. This technique is employed in many software engineering activities such as debugging, testing, maintenance and parallelization.

Komondoor et al. [16] propose using slicing to identify duplication in source code, by using program dependence graphs and program slicing to find clones (instances of duplicated code) that are then displayed to the programmer. In the same thread, Gupta et al. [11] suggest a new approach for locating faulty code, with the use of a delta debugging algorithm to identify a minimal failure-inducing input which is then used to compute a forward dynamic slice that is intersected with the statements in the backward dynamic slice of the erroneous output, to compute a failure-inducing chop.

More recently, Maras et al. [18] have applied program slicing to extract the code implementing a certain behaviour for a client-side web application, based on a web page dependency graph. Maruyama et al. [19] propose a slicing mechanism to extract code changes necessary to construct a particular class member of a Java program, based on the history of past code changes which are represented by edit operations recorded on source code of a program, helping programmers avoid replaying edit operations that are non-essential to the construction of class members they are analysing.

To the best of our knowledge, ours is the first attempt to use slicing techniques to handle program variability and versioning.

Update programming. Erwig and Ren [9], Apel and Hutchins [3] introduce an extension to Haskell that supports update programming, where a program is an abstract data type whose building blocks are language terms. They provide a mechanism to script changes in

[11] This is detailed later on, in section 6.

programs, creating new terms and changing existing ones. Hazelnut [24] is a core calculus that builds on typed "holes" and a gradual type theory that features a type system for expressions with holes and a language of edit actions ensuring that every edit state has static meaning. Both these approaches allows for progressive program construction, as well as giving semantic meaning to incomplete code. We maintain the history of programming versions, well-formed by construction, instead of defining semantics for partial programs [25]. Such history is a guide to the program slicing procedure in VFJ, unlike others where an edit calculus is needed to understand changes ([19]).

Delta-oriented programming. Schaefer et al. [28] introduce DOP, a programming language for designing software product lines based on the concept of program deltas. The implementation of software product lines is divided into a core module, comprising a complete valid product, and a set of delta modules, changes to be applied to the core module to target other products/variations. The language further ensures that all product variations are well typed.

Multiversion systems analysis. The analysis of multiversion systems is usually a project management activity that tries to detect change patterns in the code, and assessing risks of interference between development threads that may result in the introduction of vulnerabilities [14], code repetition [15] and maintenance hurdles [4, 13, 8, 30], and the other difficulties in the management of multiple versions [10, 34, 12, 29]. Our approach acts preventively by detecting illegal evolution steps in the development history and also complements update and delta oriented programming approaches [2, 9, 28] by recording a modification history and allowing (legal) branching in the code base.

6 Limitations and future work

Support for versioned modules. Currently, we do not support versioning of modules. In doing so, we would be able to 1) declared versioned elements at the module level (e.g. functions, constants, variables) and 2) defined versioned imports of packages (i.e. at some version `v`, we want to import version `t` of package `p`). The main challenge is devising a syntax for declaring a module-level version graph (since we use decorators, which are only valid for classes and functions), and devising a syntax for versioned imports.

Structural typing for lenses. Currently, the type system requires that method lenses return the same type (or a subtype) of the original method definition. This forbids us from, for example, defining a lens to rename a class `C` to `D` (which would be reflected on the lens of `__init__` method of `C`, by returning an object of type `D`). Since our type checker uses nominal sub-typing, this is not possible (since `D` is not syntactically declared as a subtype of `C`). As such, we intend to define a structural sub-typing discipline for method lenses.

Inlining for lenses. The rewriting procedure for methods and fields replaces their occurrences with calls to the corresponding lenses. However, from the experiments we conducted, it's clear that these are, more frequently than not, single line expressions (e.g. when renaming a method argument, or renaming a field). To declutter and optimize the resulting slice, we intend to implement an `@inline` decorator for lenses whose body is a single return statement, to indicate that the lens can be inlined instead of rewriting to a function call.

Version-aware development environment. From a user experience perspective, we believe this approach is not yet suitable for adoption. As such, we intend on implementing tools for a version-aware development environment that would automate most of the common refactoring practices (e.g. moving a method, renaming a field). We are working on an extension for VS-Code to do so, and also plan on extending `rope`, a refactoring library for Python, to account for refactoring of versioned programs. Ideally, the developer would

apply the refactor from the extension in the IDE, and the versioned program would be changed accordingly to include the proposed refactor (for instance, renaming a method would introduce a new definition and its corresponding lens).

7 Conclusions

We build on prior work that presents a language-based approach for a version control system incorporating semantic knowledge of the evolution steps in the code and allowing code sharing and reuse across versions of a software product. We extend it with support for method transformations, and for state and side-effects in an imperative setting.

We instantiate this approach in a large subset of the Python programming language, and demonstrate its applicability by evaluating it against different versions of popular Python packages. We show that this approach is suitable for capturing common software evolution steps, rich versioning workflows, and streamlining the delivery of a snapshot for a given version. We provide a type system to detect conflicts and unintended breaking changes, that operates on a semantic level on top of the entire version graph and its classes, and a slicing compiler to extract the Python code targeting a single version.

References

1. ast - Abstract Syntax Trees, 2023. URL: `https://docs.python.org/3/library/ast.html`.
2. Edward Amsden, Ryan Newton, and Jeremy Siek. Editing Functional Programs Without Breaking Them. In *IFL 2014*, 2014.
3. Sven Apel and Delesley Hutchins. A calculus for uniform feature composition. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(5):1–33, 2008.
4. Keith H Bennett and Václav T Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, 2000.
5. Luís Carvalho and João Costa Seco. Deep semantic versioning for evolution and variability. In *PPDP 2021*, pages 1–13, 2021.
6. Siwei Cui et al. PYInfer: Deep Learning Semantic Type Inference for Python Variables, 2021. arXiv:2106.14316.
7. Luca Di Grazia et al. The evolution of type annotations in python: an empirical study. In *ESEC/FSE 2022*, pages 209–220. ACM, 2022.
8. S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 2001.
9. Martin Erwig and Deling Ren. A rule-based language for programming software updates. In *Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming - RULE '02*, Pittsburgh, Pennsylvania, 2002.
10. T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 2000.
11. Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference On Automated Software Engineering - ASE '05*, page 263, Long Beach, CA, USA, 2005. ACM Press. `doi:10.1145/1101908.1101948`.
12. P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *2013 35th International Conference on Software Engineering (ICSE)*, May 2013. `doi:10.1109/ICSE.2013.6606607`.
13. C. Izurieta and J. M. Bieman. How Software Designs Decay: A Pilot Study of Pattern Evolution. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007.

14 J. Kim, Y. K. Malaiya, and I. Ray. Vulnerability Discovery in Multi-Version Software Systems. In *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, 2007.

15 Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, 2006.

16 Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Patrick Cousot, editors, *Static Analysis*, volume 2126, pages 40–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. doi:10.1007/3-540-47764-0_3.

17 Li Li et al. Scalpel: The python static analysis framework. *arXiv preprint*, 2022. arXiv:2202.11840.

18 Josip Maras, Jan Carlson, and Ivica Crnkovic. Client-side web application slicing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 504–507, Lawrence, KS, USA, 2011. IEEE. doi:10.1109/ASE.2011.6100110.

19 Katsuhisa Maruyama, Eijiro Kitsu, Takayuki Omori, and Shinpei Hayashi. Slicing and replaying code change history. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 246–249. ACM, 2012.

20 Stuart McIlroy et al. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store, 2016.

21 Raphaël Monat et al. Static type analysis by abstract interpretation of python programs. In *ECOOP 2020*, 2020.

22 Kashif Munawar and Muhammad Shumail Naveed. The impact of language syntax on the complexity of programs: A case study of java and python. *Int. J. Innov. Sci. Technol*, 4:683–695, 2022.

23 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A Hammer. Live functional programming with typed holes. In *Proceedings of the ACM on Programming Languages*, volume 3, pages 1–32. ACM New York, NY, USA, 2019.

24 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages*, 2019.

25 Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. *ACM SIGPLAN Notices*, 2017.

26 Tom Preston-Werner. Semantic Versioning 2.0.0, 2023. URL: https://www.semver.org.

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

28 Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

29 Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A vm-centric approach. *SIGPLAN Not.*, 44(6):1–12, June 2009. doi:10.1145/1543135.1542478.

30 Rick Wash, Emilee Rader, Kami Vaniea, and Michelle Rizor. Out of the loop: How automated software updates cause unintended security consequences. In *10th Symposium On Usable Privacy and Security ({SOUPS} 2014)*, 2014.

31 Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

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

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

34 Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 2005.