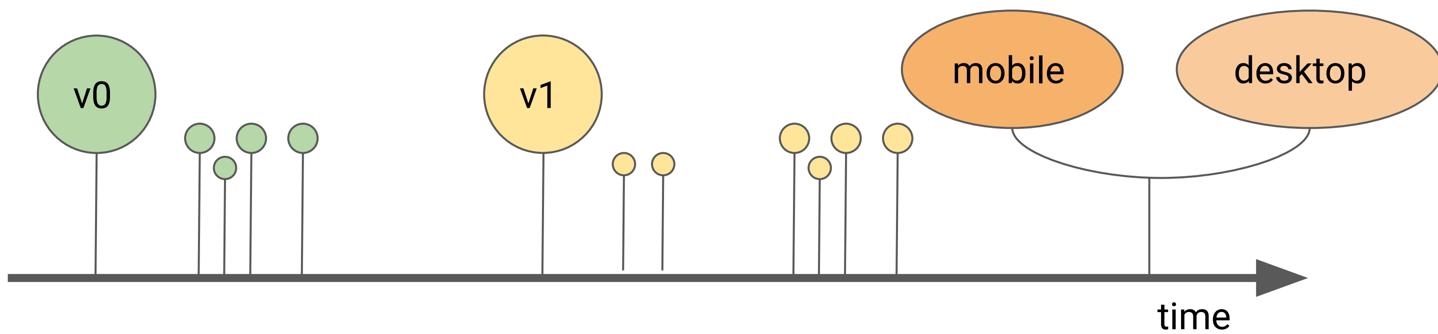
typing the evolution of variational software

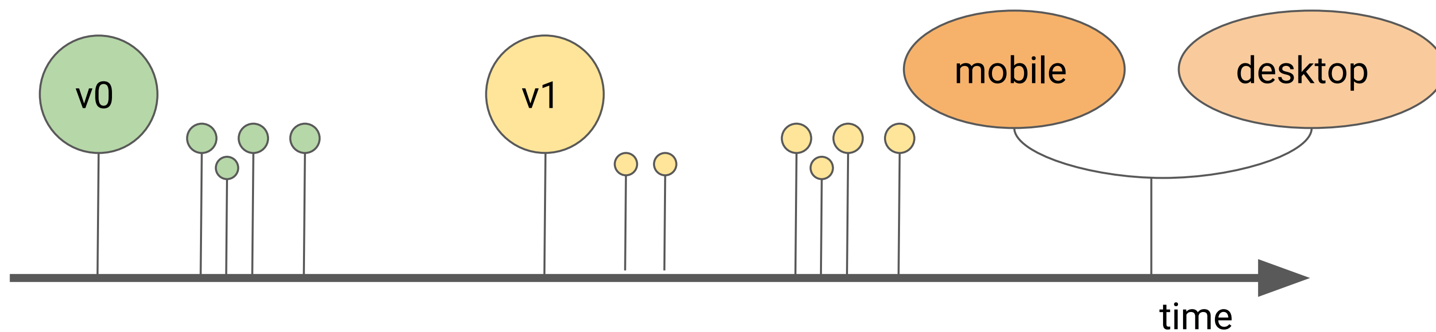
Luís Carvalho João Costa Seco Jácome Cunha

TYPES 2018 - Braga, 18 June

context



context



software of Theseus: as the software evolves and its parts are replaced and upgraded, and new parts are introduced, is it still the same software?

https://en.wikipedia.org/wiki/Ship_of_Theseus

context

software extensions

- bug fixes
- backwards compatibility
- architectual changes
  - adding new classes, types, traits, etc.

Fix coding style: Shareable is a metaclass.
master (#2918) v1.4.7
jdetrey committed on May 8 1 parent 4f7c1c9 commit

Showing 1 changed file with 7 additions and 7 deletions.

14 beets/util/artresizer.py

```
@@ -151,15 +151,15 @@ class Shareable(type):
151 151        lazily-created shared instance of ``MyClass`` while calling
152 152        ``MyClass()`` to construct a new object works as usual.
153 153        """
154      -    def __init__(self, name, bases, dict):
155      -        super(Shareable, self).__init__(name, bases, dict)
156      -        self._instance = None
    154  +    def __init__(cls, name, bases, dict):
```

mbsync: fix updating album with invalid first track MBID

MBID of recording could become invalid after merging. The existing code always copies metadata from first track after updating. But for albums with invalid track MBID that happens to be the first track, MusicBrainz changes won't be applied to whole album, only whose tracks with valid MBID. This is particularly annoying since those changes are actually displayed for every `beet mbsync` run, but never get applied.

Fix this issue by finding any track that get MusicBrainz updates, and apply it to whole album.

master (#2920) v1.4.7
bjin committed on May 9 1 parent 7e0fb

Showing 2 changed files with 6 additions and 1 deletion.

5 beetsplug/mbsync.py

```
@@ -152,10 +152,13 @@ def albums(self, lib, query, move, pretend, write
152 152            with lib.transaction():
153 153                autotag.apply_metadata(album_info, mapping)
154 154                changed = False
    155  +              # Find any changed item to apply MusicBrainz changes t
```

context

- multiple targets
- different licensing levels
- service-based architectures
- multiple live versions of service endpoints supporting older client devices

time/product orientation

Versions

For the most up to date information on historical versions of the Twitter Ads API, please re

Version	Path	Introduction Date	Deprecated D
3.0	/3/	February 1, 2018	
2.0	/2/	July 10, 2017	February 1, 20
1.0	/1/	March 31, 2016	July 7, 2017
		February 21, 2013	N/A

Eclipse IDE for Java and DSL Developers
347 MB - Downloaded 3,693 Times

Windows 32-bit 64-bit
Mac Cocoa 64-bit
Linux 32-bit 64-bit

motivation

problems

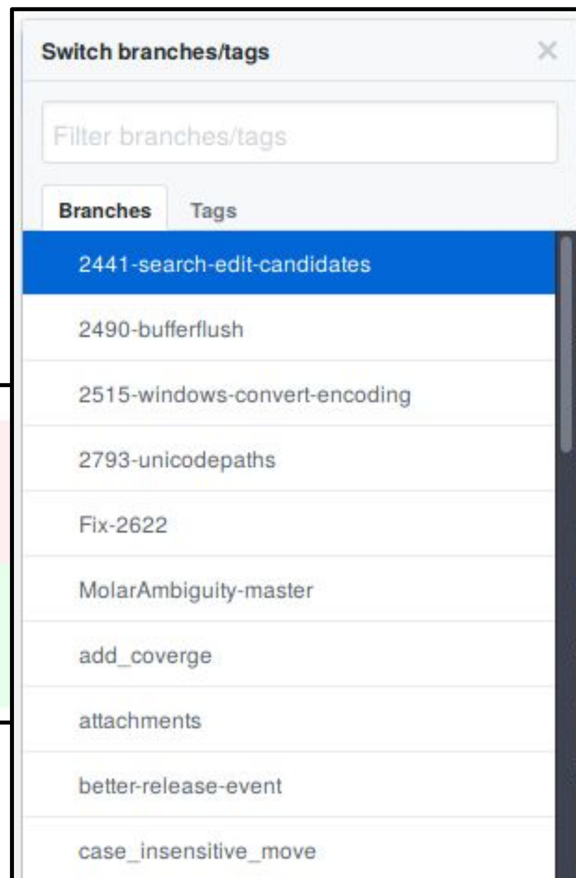
- related versions share a significant amount of code
- no true guaranties about soundness of code sharing and evolution
- ad-hoc co-existence of versions and transitions between versions

motivation

problems

- each commit represents an evolution step
- each branch represents a different version
- each merge establishes a sound co-existence and transition between two versions

```
39  39
40      -   @patch('beets.autotag.hooks.album_for_mbid')
41      -   @patch('beets.autotag.hooks.track_for_mbid')
42      -   def test_update_library(self, track_for_mbid, album_for_mbid):
43      -       album_for_mbid.return_value = \
    40  +   @patch('beets.autotag.mb.album_for_id')
    41  +   @patch('beets.autotag.mb.track_for_id')
    42  +   def test_update_library(self, track_for_id, album_for_id):
    43  +       album_for_id.return_value = \
```

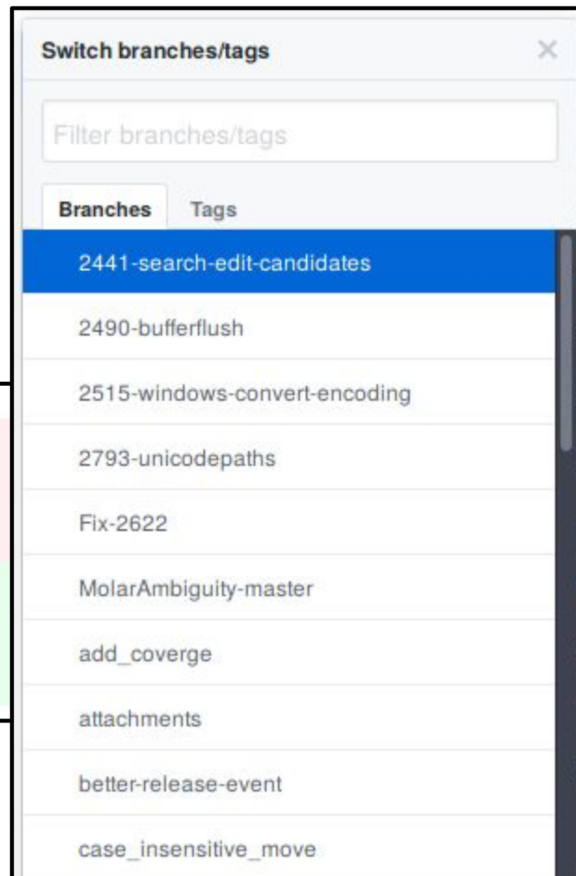


motivation

problems

~ git + branching + merging within the core language

```
39  39
40     -    @patch('beets.autotag.hooks.album_for_mbid')
41     -    @patch('beets.autotag.hooks.track_for_mbid')
42     -    def test_update_library(self, track_for_mbid, album_for_mbid):
43     -        album_for_mbid.return_value = \
    40 +    @patch('beets.autotag.mb.album_for_id')
    41 +    @patch('beets.autotag.mb.track_for_id')
    42 +    def test_update_library(self, track_for_id, album_for_id):
    43 +        album_for_id.return_value = \
```



technical approach principles

- software development tools based on lightweight type-based verification
- types ensure evolution, like git, but sound with branching & merging
- we extend FeatherweightJava (Igarashi et al.) with
  - native support for versions in the codebase
    - base versions
    - single versions
  - support for lenses that specify how the state transition between one version and another is performed
  - type system that ensures evolution is sound

technical approach

example

v1

```
class Post extends Object {
  Boolean priv, draft;
  Post(Boolean priv, Boolean draft) {
    this.priv  = priv;
    this.draft = draft;
  }
  Post active() {
    return new Post(false, false);
  }
  Boolean isPrivate() {
    return this.priv;
  }
}
```

technical approach

example

v1

```
class Post extends Object {
  Boolean priv, draft;
  Post(Boolean priv, Boolean draft) {
    this.priv  = priv;
    this.draft = draft;
  }
  Post active() {
    return new Post(false, false);
  }
  Boolean isPrivate() {
    return this.priv;
  }
}
```

change the variable representing the status to a String instead of two Booleans.

what do we need to refactor?

technical approach

example

change the variable representing the status to a String instead of two Booleans.

v1

```
class Post extends Object {
  Boolean priv, draft;
  Post(Boolean priv, Boolean draft) {
    this.priv  = priv;
    this.draft = draft;
  }
  Post active() {
    return new Post(false, false);
  }
  Boolean isPrivate() {
    return this.priv;
  }
}
```

refactoring →

v2

```
class Post extends Object {
  String st;
  Post(String st) {
    this.st = st;
  }
  Post active() {
    return new Post("active");
  }
  Boolean isPrivate() {
    return this.st.equals("private");
  }
  String status() {
    return this.st;
  }
}
```

technical approach

example

do we need to refactor everything?

v1

```
class Post extends Object {
  Boolean priv, draft;
  Post(Boolean priv, Boolean draft) {
    this.priv  = priv;
    this.draft = draft;
  }
  Post active() {
    return new Post(false, false);
  }
  Boolean isPrivate() {
    return this.priv;
  }
}
```

refactoring →

v2

```
class Post extends Object {
  String st;
  Post(String st) {
    this.st = st;
  }
  Post active() {
    return new Post("active");
  }
  Boolean isPrivate() {
    return this.st.equals("private");
  }
  String status() {
    return this.st;
  }
}
```

technical approach

example

do we need to refactor everything?
what about defining how the state transitions from v1 to v2, and then take advantage of the existing implementation?

v1

```
class Post extends Object {
  Boolean priv, draft;
  Post(Boolean priv, Boolean draft) {
    this.priv  = priv;
    this.draft = draft;
  }
  Post active() {
    return new Post(false, false);
  }
  Boolean isPrivate() {
    return this.priv;
  }
}
```

refactoring →

v2

```
class Post extends Object {
  String st;
  Post(String st) {
    this.st = st;
  }
  Post active() {
    return new Post("active");
  }
  Boolean isPrivate() {
    return this.st.equals("private");
  }
  String status() {
    return this.st;
  }
}
```

technical approach

example

```
version 1
class Post extends Object {
  Boolean priv@1, draft@1;
  Post@1(Boolean priv, Boolean draft) {
     this.priv  = priv;
     this.draft = draft;
  }
  Post active@1() {
     return new Post(false, false);
  }
  Boolean isPrivate@1() {
     return this.priv;
  }
}
```

declare version 1

tag fields, constructor, and methods as belonging to v1

technical approach

example

```
version 1
version 2 extends 1
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
      this.priv  = priv;
      this.draft = draft;
  }
  Post@2(String st) {
      this.st = st;
  }
  Post active@1() {
      return new Post(false, false);
  }
  Boolean isPrivate@1() {
      return this.priv;
  }
  String status@2() {
      return this.st;
  }
}
```

add declaration for version 2

add needed fields, constructor, and methods for v2 - but keep the existing ones from v1!

technical approach

example

```
version 1
version 2 extends 1
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
      this.priv  = priv;
      this.draft = draft;
      this@2(priv?"private":(draft?"draft":"active"));
  }
  Post@2(String st) {
      this.st = st;
      this@1(st.equals("private"),st.equals("draft"));
  }
  Post active@1() {
      return new Post(false, false);
  }
  Boolean isPrivate@1() {
      return this.priv;
  }
  String status@2() {
      return this.st;
  }
}
```

add lenses to the constructors describing how state transitions from one version to the other

no need to re-implement the methods from v1

technical approach

example

```
version 1
version 2 extends 1
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
      this.priv  = priv;
      this.draft = draft;
      this@2(priv?"private":(draft?"draft":"active"));
  }
  Post@2(String st) {
      this.st = st;
      this@1(st.equals("private"),st.equals("draft"));
  }
  Post active@1() {
      return new Post(false, false);
  }
  Boolean isPrivate@1() {
      return this.priv;
  }
  String status@2() {
      return this.st;
  }
}
```

the entire codebase is versioned and so, at each point in time, the developer only sees a slice of the codebase corresponding to the version they are developing

technical approach

example

```
version 1
class Post extends Object {
  Boolean priv, draft;
  Post(Boolean priv, Boolean draft) {
      this.priv  = priv;
      this.draft = draft;
      this@2(priv?"private":(draft?"draft":"active"));
  }
  Post active() {
      return new Post(false, false);
  }
  Boolean isPrivate() {
      return this.priv;
  }
}
```

the entire codebase is versioned and so, at each point in time, the developer only sees a slice of the codebase corresponding to the version they are developing

technical approach

example

```
version 2 extends 1
class Post extends Object {
  String st;
   Post(String st) {
     this.st = st;
     this@1(st.equals("private"),st.equals("draft"));
  }
  String status() {
     return this.st;
  }
}
```

the entire codebase is versioned and so, at each point in time, the developer only sees a slice of the codebase corresponding to the version they are developing

technical approach

example

```
version 1
version 2 extends 1
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
     this.priv  = priv;
     this.draft = draft;
     this@2(priv?"private":(draft?"draft":"active"));
  }
  Post@2(String st) {
     this.st = st;
     this@1(st.equals("private"),st.equals("draft"));
  }
  Post active@1() {
     return new Post(false, false);
  }
  Boolean isPrivate@1() {
     return this.priv;
  }
  String status@2() {
     return this.st;
  }
}
```

at any time, the developer may look at a different version and examine the code for that version (~ `git checkout`)

technical approach

example

```
version 1
version 2 extends 1
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
      this.priv  = priv;
      this.draft = draft;
      this@2(priv?"private":(draft?"draft":"active"));
  }
  Post@2(String st) {
      this.st = st;
      this@1(st.equals("private"),st.equals("draft"));
  }
  Post active@1() {
      return new Post(false, false);
  }
  Boolean isPrivate@1() {
      return this.priv;
  }
  String status@2() {
      return this.st;
  }
}
```

any expression runs in a given context, provided by the version in which it is to be executed

technical approach

example

```
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
       this.priv  = priv;
       this.draft = draft;
this@2(priv?"private":(draft?"draft":"active"));
}
  Post@2(String st) {
       this.st = st;
this@1(st.equals("private"),st.equals("draft"));
}
  Post active@1() {
       return new Post(false, false);
  }
  Boolean isPrivate@1() {
       return this.priv;
  }
  String status@2() {
       return this.st;
  }
}
```

derivation

```
@2(new Post("draft").active().status())
```

technical approach

example

```
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
       this.priv  = priv;
       this.draft = draft;
this@2(priv?"private":(draft?"draft":"active"));
}
  Post@2(String st) {
       this.st = st;
this@1(st.equals("private"),st.equals("draft"));
}
  Post active@1() {
       return new Post(false, false);
  }
  Boolean isPrivate@1() {
       return this.priv;
  }
  String status@2() {
       return this.st;
  }
}
```

derivation

```
@2(new Post("draft").active().status())
```

technical approach

example

```
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
       this.priv  = priv;
       this.draft = draft;
this@2(priv?"private":(draft?"draft":"active"));
}
  Post@2(String st) {
       this.st = st;
this@1(st.equals("private"),st.equals("draft"));
}
  Post active@1() {
       return new Post(false, false);
  }
  Boolean isPrivate@1() {
       return this.priv;
  }
  String status@2() {
       return this.st;
  }
}
```

derivation

```
@2(new Post("draft").active().status())

@2(@1(new Post(false, false)).status())
```

technical approach

example

```
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
       this.priv  = priv;
       this.draft = draft;
this@2(priv?"private":(draft?"draft":"active"));
}
  Post@2(String st) {
       this.st = st;
this@1(st.equals("private"),st.equals("draft"));
}
  Post active@1() {
       return new Post(false, false);
  }
  Boolean isPrivate@1() {
       return this.priv;
  }
  String status@2() {
       return this.st;
  }
}
```

derivation

```
@2(new Post("draft").active().status())

@2(@1(new Post(false, false)).status())
```

technical approach

example

```
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
       this.priv  = priv;
       this.draft = draft;
this@2(priv?"private":(draft?"draft":"active"));
}
  Post@2(String st) {
       this.st = st;
this@1(st.equals("private"),st.equals("draft"));
}
  Post active@1() {
       return new Post(false, false);
  }
  Boolean isPrivate@1() {
       return this.priv;
  }
  String status@2() {
       return this.st;
  }
}
```

derivation

```
@2(new Post("draft").active().status())

@2(@1(new Post(false, false)).status())

@2(new Post("active").status())
```

technical approach

example

```
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
       this.priv  = priv;
       this.draft = draft;
this@2(priv?"private":(draft?"draft":"active"));
}
  Post@2(String st) {
       this.st = st;
this@1(st.equals("private"),st.equals("draft"));
}
  Post active@1() {
       return new Post(false, false);
  }
  Boolean isPrivate@1() {
       return this.priv;
  }
  String status@2() {
       return this.st;
  }
}
```

derivation

```
@2(new Post("draft").active().status())

@2(@1(new Post(false, false)).status())

@2(new Post("active").status())
```

technical approach

example

```
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
       this.priv  = priv;
       this.draft = draft;
this@2(priv?"private":(draft?"draft":"active"));
}
  Post@2(String st) {
       this.st = st;
this@1(st.equals("private"),st.equals("draft"));
}
  Post active@1() {
       return new Post(false, false);
  }
  Boolean isPrivate@1() {
       return this.priv;
  }
  String status@2() {
       return this.st;
  }
}
```

derivation

```
@2(new Post("draft").active().status())

@2(@1(new Post(false, false)).status())

@2(new Post("active").status())

@2("active")
```

technical approach

example

```
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
       this.priv  = priv;
       this.draft = draft;
this@2(priv?"private":(draft?"draft":"active"));
}
  Post@2(String st) {
       this.st = st;
this@1(st.equals("private"),st.equals("draft"));
}
  Post active@1() {
       return new Post(false, false);
  }
  Boolean isPrivate@1() {
       return this.priv;
  }
  String status@2() {
       return this.st;
  }
}
```

derivation

```
@2(new Post("draft").active().status())

@2(@1(new Post(false, false)).status())

@2(new Post("active").status())

@2("active")
```

technical approach

example

```
class Post extends Object {
  Boolean priv@1, draft@1;
  String st@2;
  Post@1(Boolean priv, Boolean draft) {
       this.priv  = priv;
       this.draft = draft;
this@2(priv?"private":(draft?"draft":"active"));
}
  Post@2(String st) {
       this.st = st;
this@1(st.equals("private"),st.equals("draft"));
}
  Post active@1() {
       return new Post(false, false);
  }
  Boolean isPrivate@1() {
       return this.priv;
  }
  String status@2() {
       return this.st;
  }
}
```

derivation

```
@2(new Post("draft").active().status())

@2(@1(new Post(false, false)).status())

@2(new Post("active").status())

@2("active")

"active"
```

language

syntax

$$P ::= \overline{V}\ \overline{L}\ e$$

$$V ::= version\ v \mid version\ v\ extends\ v'$$

$$L ::= class\ C\ extends\ D\ \{\ \overline{C\ f@v;}\ \overline{K}\ \overline{M}\ \}$$

$$K ::= C@v(\overline{C\ f})\{\ super(\overline{f});\ \overline{this@v(\overline{e});}\ \overline{this.f = f;}\ \}$$

$$M ::= C\ m@v(\overline{C\ x})\{\ return\ e;\ \}$$

$$e ::= x \mid e.f \mid e.m(\overline{e}) \mid new\ C(\overline{e}) \mid @v(e) \mid this$$

language

syntax: versions and classes

$$P ::= \overline{V}\ \overline{L}\ e$$

$$V ::= version\ v \mid version\ v\ extends\ v'$$

$$L ::= class\ C\ extends\ D\ \{\ \overline{C\ f@v;}\ \overline{K}\ \overline{M}\ \}$$

$$K ::= C@v(\overline{C\ f})\{\ super(\overline{f});\ \overline{this@v(\overline{e});}\ \overline{this.f = f;}\ \}$$

$$M ::= C\ m@v(\overline{C\ x})\{\ return\ e;\ \}$$

$$e ::= x \mid e.f \mid e.m(\overline{e}) \mid new\ C(\overline{e}) \mid @v(e) \mid this$$

language

syntax: versions and classes

$$P ::= \overline{V}\ \overline{L}\ e$$

$$V ::= version\ v \mid version\ v\ extends\ v'$$

$$L ::= class\ C\ extends\ D\ \{\ \overline{C\ f@v;}\ \overline{K}\ \overline{M}\ \}$$

$$K ::= C@v(\overline{C\ f})\{\ \ super(\overline{f});\ \overline{this@v(\overline{e});}\ \overline{this.f = f;}\ \}$$

$$M ::= C\ m@v(\overline{C\ x})\{\ return\ e;\ \}$$

$$e ::= x \mid e.f \mid e.m(\overline{e}) \mid new\ C(\overline{e}) \mid @v(e) \mid this$$

language

syntax: versions and classes

$$P ::= \overline{V}\ \overline{L}\ e$$

$$V ::= version\ v \mid version\ v\ extends\ v'$$

$$L ::= class\ C\ extends\ D\ \{\ \overline{C\ f@v};\ \overline{K}\ \overline{M}\ \}$$

$$K ::= C@v(\overline{C\ f})\{\ super(\overline{f});\ \overline{this@v(\overline{e})};\ \overline{this.f = f;}\ \}$$

$$M ::= C\ m@v(\overline{C\ x})\{\ return\ e;\ \}$$

$$e ::= x \mid e.f \mid e.m(\overline{e}) \mid new\ C(\overline{e}) \mid @v(e) \mid this$$

language

syntax: versions and classes

$$P ::= \overline{V}\ \overline{L}\ e$$

$$V ::= version\ v \mid version\ v\ extends\ v'$$

$$L ::= class\ C\ extends\ D\ \{\ \overline{C\ f@v;}\ \overline{K}\ \overline{M}\ \}$$

$$K ::= C@v(\overline{C\ f})\{\ \ super(\overline{f});\ \overline{this@v(\overline{e});}\ \overline{this.f = f;}\ \}$$

$$M ::= C\ m@v(\overline{C\ x})\{\ return\ e;\ \}$$

$$e ::= x \mid e.f \mid e.m(\overline{e}) \mid new\ C(\overline{e}) \mid @v(e) \mid this$$

language

syntax: versions and classes

$$P ::= \overline{V}\ \overline{L}\ e$$

$$V ::= version\ v \mid version\ v\ extends\ v'$$

$$L ::= class\ C\ extends\ D\ \{\ \overline{C\ f@v;}\ \overline{K}\ \overline{M}\ \}$$

$$K ::= C@v(\overline{C\ f})\{\ super(\overline{f});\ \overline{this@v(\overline{e});}\ \overline{this.f = f;}\ \}$$

$$M ::= C\ m@v(\overline{C\ x})\{\ return\ e;\ \}$$

$$e ::= x \mid e.f \mid e.m(\overline{e}) \mid new\ C(\overline{e}) \mid @v(e) \mid this$$

language

syntax: versions and classes

$$P ::= \overline{V}\ \overline{L}\ e$$

$$V ::= version\ v \mid version\ v\ extends\ v'$$

$$L ::= class\ C\ extends\ D\ \{\ \overline{C\ f@v;}\ \overline{K}\ \overline{M}\ \}$$

$$K ::= C@v(\overline{C\ f})\{\ super(\overline{f});\ \overline{this@v(\overline{e});}\ \overline{this.f = f;}\ \}$$

$$M ::= C\ m@v(\overline{C\ x})\{\ return\ e;\ \}$$

$$e ::= x \mid e.f \mid e.m(\overline{e}) \mid new\ C(\overline{e}) \mid @v(e) \mid this$$

language

types

$$\Gamma \vdash_v x : \Gamma(x)$$

$$\frac{\Gamma \vdash_v e_0 : C_0 \quad fields_v(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash_v e_0.f_i : C_i}$$

$$\frac{\Gamma \vdash_v e_0 : C_0 \quad mtype_v(m, C_0) = \overline{D} \to C \quad \Gamma \vdash_v \overline{e} : \overline{C} \quad \overline{C} <: \overline{D}}{\Gamma \vdash_v e_0.m(\overline{e}) : C}$$

$$\frac{fields_v(C) = \overline{D}\ \overline{f} \quad \Gamma \vdash_v \overline{e} : \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash_v new\ C(\overline{e}) : C}$$

$$\frac{\overline{M}\ OK\ in\ C \qquad \overline{K}\ OK\ in\ C}{class\ C\ extends\ D\ \{\ \overline{C\ f\ v};\ \overline{K}\ \overline{M}\ \}\ OK}$$

extension of the FWJ typing rules by enriching the context with version *v*

language

types: version

$$\frac{\Gamma \vdash_{v'} e : C}{\Gamma \vdash_{v} @v'(e) : C}$$

if *e* has type C in version *v'*, then
@v'(e) has type C in version *v*

language

types: constructor

$$\frac{\begin{array}{c} class\ C\ extends\ D\ \{\ ...\ \} \\ fields_v(D) = \overline{D}\ \overline{g} \qquad fields_v(C) = \overline{D}\ \overline{g}, \overline{C}\ \overline{f} \\ fields_{v'_i}(C) = \overline{C_i}\ \overline{x_i} \qquad \overline{D}\ \overline{g}, \overline{C}\ \overline{f} \vdash_v \overline{e_i} : \overline{G_i} \quad \overline{G_i} <: \overline{C_i} \quad i = 1..n \end{array}}{C@v(\overline{D}\ \overline{g}, \overline{C}\ \overline{f})\{\ \ super(\overline{g});\ \overline{this@v'(\overline{e});}\ this.\overline{f} = \overline{f};\ \}\ OK\ in\ C}$$

language

types: constructor

$$\frac{\begin{array}{c} \mathit{class}\ C\ \mathit{extends}\ D\ \{\ \dots\ \} \\ \mathit{fields}_v(D) = \overline{D}\ \overline{g} \qquad \mathit{fields}_v(C) = \overline{D}\ \overline{g}, \overline{C}\ \overline{f} \\ \mathit{fields}_{v'_i}(C) = C_i\ \overline{x_i} \qquad D\ \overline{g}, C\ f \vdash_v \overline{e_i} : G_i \quad G_i <: C_i \quad i = 1..n \end{array}}{C@v(\overline{D}\ \overline{g}, \overline{C}\ \overline{f})\{\ \ \mathit{super}(\overline{g});\ \overline{\mathit{this}@v'(\overline{e});}\ \mathit{this}.\overline{f} = \overline{f};\ \}\ \mathit{OK\ in}\ C}$$

lenses must be well-typed within the constructor of the source version

language

types

$$\Gamma \vdash_v x : \Gamma(x)$$

$$\frac{\Gamma \vdash_v e_0 : C_0 \quad fields_v(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash_v e_0.f_i : C_i}$$

$$\frac{\Gamma \vdash_v e_0 : C_0 \quad mtype_v(m, C_0) = \overline{D} \to C \quad \Gamma \vdash_v \overline{e} : \overline{C} \quad \overline{C} <: \overline{D}}{\Gamma \vdash_v e_0.m(\overline{e}) : C}$$

$$\frac{fields_v(C) = \overline{D}\ \overline{f} \quad \Gamma \vdash_v \overline{e} : \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash_v new\ C(\overline{e}) : C}$$

$$\frac{\overline{M}\ OK\ in\ C \qquad \overline{K}\ OK\ in\ C}{class\ C\ extends\ D\ \{\ \overline{C\ f\ v};\ \overline{K}\ \overline{M}\ \}\ OK}$$

extension of the FWJ typing rules by enriching the context with version *v*

language

types

base version lookup:

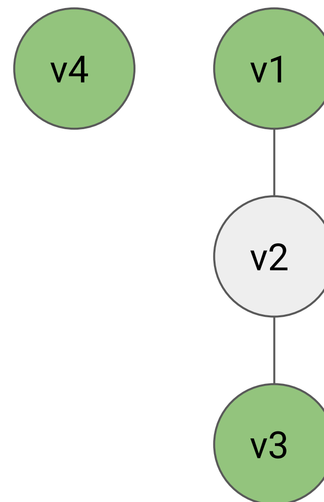
$$base_v(C) = lup\ \{v' \mid C@v'(...), v' \leq v\}$$

$base_1\ (C) = 1$

$base_2\ (C) = 1$

$base_3\ (C) = 3$

$base_4\ (C) = 4$



version hierarchy for class C
green means there is a constructor for that version

language

types

fields lookup:

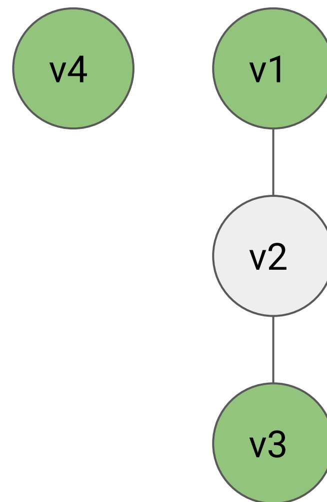
$$fields_v(C) = \overline{C}\ \overline{f} \ \cup fields_v(D)$$
$$\text{if} \ \ base_v(C) = v' \text{ and}$$
$$class\ C\ extends\ D\ \{\overline{C}\ \overline{f@v'};\ldots\}$$

$base_1\ (C) = 1$

$base_2\ (C) = 1$

$base_3\ (C) = 3$

$base_4\ (C) = 4$



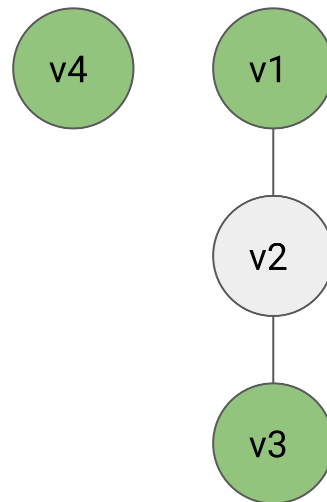
version hierarchy for class C
green means there is a constructor for that version

language

types

method type lookup:

$$mtype_v(m, C) = \overline{D} \to D \text{ where}$$
$$(\overline{D} \to D, v') =$$
$$\quad (\text{lup} \cdot \text{snd})\ \{(\overline{D} \to D, v') \mid D\ m@v'(\overline{D}\ \overline{x})\{return\ e_0;\} \text{ in } C, v' \leq v\}$$



version hierarchy for class C
green means there is a constructor for that version

language

semantics

$$\frac{fields_v(C) = \overline{C}\ \overline{f}}{new\ C(\overline{e}).f_i \xrightarrow{v} e_i}$$

$$\frac{e_0 \xrightarrow{v} e_0'}{e_0.f \xrightarrow{v} e_0'.f}$$

$$\frac{e_0 \xrightarrow{v} e_0'}{e_0.m(\overline{e}) \xrightarrow{v} e_0'.m(\overline{e})}$$

$$\frac{e \xrightarrow{v} e'}{@v(e) \xrightarrow{v'} @v(e')}$$

extension of the FWJ semantics by enriching the context with version *v*

language semantics: version upgrade

$$\frac{\overline{e'} = \underset{C:v \to v'}{\mathcal{L}}(\overline{e})}{@v(new\ C(\overline{e})) \xrightarrow{v'} new\ C(\overline{e'})}$$

$$\underset{C:v \to v'}{\mathcal{L}}(\overline{e}) \triangleq \overline{@v(e'\{\overline{e}/\overline{x}\})} \quad if \quad C@v(\overline{x})\{...; this@v_b(\overline{e'}); ...\}\ and\ v_b = base_{v'}(C)$$

$$\underset{C:v \to v}{\mathcal{L}}(\overline{e}) = \overline{e}$$

language

semantics: method call

$$\frac{mbody_v(m, C) = (\overline{x}.e_0, v')}{new\ C(\overline{e}).m(\overline{f}) \xrightarrow{v} @v'(e_0\{^{@v(\overline{f})}/_{\overline{x}}\}\{^{@v(new\ C(\overline{e}))}/_{this}\})}$$

method body lookup:

$mbody_v(m, C) = (\overline{x}.e_0, v')$ where

$(\overline{x}.e_0, v') =$

$\quad (\mathrm{lup} \cdot \mathrm{snd})\{(\overline{x}.e_0, v') \mid D\ m@v'(\overline{D}\ \overline{x})\{return\ e_0;\}\ \text{in}\ C, v' \leq v\}$

language

semantics: method call

$$\frac{mbody_v(m, C) = (\overline{x}.e_0, v')}{new\ C(\overline{e}).m(\overline{f}) \xrightarrow{v} @v'(e_0\{^{@v(\overline{f})}/_{\overline{x}}\}\{^{@v(new\ C(\overline{e}))}/_{this}\})}$$

method body lookup:

$mbody_v(m, C) = (\overline{x}.e_0, v')$ where

$(\overline{x}.e_0, v') =$

$\quad (\mathrm{lup} \cdot \mathrm{snd})\{(\overline{x}.e_0, v') \mid D\ m@v'(\overline{D}\ \overline{x})\{return\ e_0;\}\ \text{in}\ C, v' \leq v\}$

language

semantics: method call

$$\frac{mbody_v(m, C) = (\overline{x}.e_0, v')}{new\ C(\overline{e}).m(\overline{f}) \xrightarrow{v} @v'(e_0\{^{@v(\overline{f})}/_{\overline{x}}\}\{^{@v(new\ C(\overline{e}))}/_{this}\})}$$

method body lookup:

$mbody_v(m, C) = (\overline{x}.e_0, v')$ where

$(\overline{x}.e_0, v') =$

$\quad (\text{lup} \cdot \text{snd})\{(\overline{x}.e_0, v') \mid D\ m@v'(\overline{D}\ \overline{x})\{return\ e_0;\}\ \text{in}\ C, v' \leq v\}$

language

semantics: method call

$$\frac{mbody_v(m, C) = (\overline{x}.e_0, v')}{new\ C(\overline{e}).m(\overline{f}) \xrightarrow{v} @v'(e_0\{^{@v(\overline{f})}/_{\overline{x}}\}\{^{@v(new\ C(\overline{e}))}/_{this}\})}$$

method body lookup:

$mbody_v(m, C) = (\overline{x}.e_0, v')$ where

$(\overline{x}.e_0, v') =$

$\quad (\text{lup} \cdot \text{snd})\{(\overline{x}.e_0, v') \mid D\ m@v'(\overline{D}\ \overline{x})\{return\ e_0;\}\ \text{in}\ C, v' \leq v\}$

conclusions

- type- & language-based approach to code evolution and software product lines management
- extend a core language (Featherweight Java) with a typing discipline that ensures the sound evolution of state and code
  - analyse the codebase as a whole
  - keep all versions well-typed
  - ensure that version/state transformations are checked
  - detect illegal version context crossings
  - standard subject reduction results
- execute all versions simultaneously with seamless transitions
- slice versions/variants from the whole codebase