A Context-Oriented Programming Approach to Dependency Hell

Yudai Tanabe
Tokyo Institute of Technology
yudaitnb@prg.is.titech.ac.jp

Tomoyuki Aotani
Tokyo Institute of Technology
aotani@c.titech.ac.jp

Hidehiko Masuhara
Tokyo Institute of Technology
masuhara@acm.org

ABSTRACT

Two or more incompatible versions of a library are sometimes needed in one software artifact, which is so-called dependency hell. One likely faces the problem if he or she uses two or more libraries that depend on the same library. In this paper, we propose versioned values to solve the problem. They allow us to have multiple versions of functions in a binary file. This gets rid of requiring two or more incompatible binaries. We develop a calculus λ_{VL} to discuss type safety in the case where definitions are available only in specific versions, which is a common and important nature of versioned programs.

KEYWORDS

Context-Oriented Programming, Contexts, Coeffects

ACM Reference Format:
Yudai Tanabe, Tomoyuki Aotani, and Hidehiko Masuhara. 2018. A Context-Oriented Programming Approach to Dependency Hell. In *10th International Workshop on Context-Oriented Programming (COP'18), July 16, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3242921.3242923

1 INTRODUCTION

Software applications and libraries are often versioned. Different versions may be (1) structurally incompatible [4, 20], i.e., two versions provide different sets of definitions such as classes, structures and functions, and/or (2) behaviorally incompatible [13], i.e., the same definitions of which behavior is different. An example of structural incompatibility is the class `AndroidHttpClient` for HTTP connections is available in Android API levels 8–22 but not in 23 and higher.[1] Another example is the class `Logger` in versions 1.2 and 2.10 of Log4j[2], a widely-used Java library for logging. It provides the method `getLogger` to get an instance of `Logger` in version 1.2 but not in 2.10. An example of behavioral incompatibility is the method `set` in the class `AlarmManager` in Android API level 19. It takes a `long` value that specifies the time to trigger an alarm but the scheduling strategy has been changed since Android API levels 19.[3] One has to use `setExact` in Android API levels 19 and later instead of `set` to achieve the behavior of `set` in Android API levels 18 and before.

The incompatibility causes two problems. First, if a newer version of a library than required is given to run an application, the application may not start or work correctly depending on the differences between the two versions. If some definitions are missing in the newer version, the application cannot start. If the behavior of necessary functions or methods is different, the behavior of the application goes unexpectedly. Second, two libraries that depend on the two different versions of the same library cannot co-exist in an application, which is known as *dependency hell.*[4] This is because from the viewpoint of a programming language two versions of definitions are not distinguishable if their names and signatures are the same.

In this paper, we consider context-oriented programming (COP) [8] as a solution to the problems. COP is a programming approach to modularizing context-dependent behavioral variations. COP languages and systems provide constructs and mechanisms to compose and abstract the variations, which can be activated and deactivated according to the computational context at runtime.

Our idea is to consider versions as contexts and bundle multiple versions of definitions in one library file such as a jar archive and a shared object. COP languages allow us to compose and abstract version-specific behavioral variations of definitions. This contributes to solving the problems attributable to behavioral incompatibility. Moreover, some advanced COP languages allow us to have baseless definitions [1, 10, 11] that are only available in some contexts. If we consider versions as contexts, baseless definitions allow us to add definitions in some versions that are not available in the initial version. This contributes to solving the problems attributable to structural incompatibility.

Unfortunately, COP is mainly studied as an extension of object-oriented programming (OOP). This means that we need classes and/or objects to have multiple versions of definitions in one program. It is desirable to avoid an OOP language as the base of our solution because dependency hell is not specific to OOP languages.

We propose *versioned values* as constructs to compose and abstract multiple versions of values and functions in languages that support first-class functions. We do not assume that the languages support classes and objects for OOP. Versioned values represent values and functions that are different among versions. Versioned values basically consist of pairs of a version number and a version-specific value. In other words, versioned values generalize the idea of objects with cursors in COP languages so that any values other than objects may depend on the version/context. For

[1] https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html#behavior-apache-http-client
[2] https://logging.apache.org/log4j/

[3] https://developer.android.com/about/versions/android-4.4.html#Behaviors
[4] https://en.wikipedia.org/wiki/Dependency_hell

COP'18, July 16, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5722-7/18/07…$15.00
https://doi.org/10.1145/3242921.3242923

example, we represent a versioned function that is `\x.x` in the first version and `\x.x+1` in the second version as `{1=\x.x, 2=\x.x+1}` basically. If versioned functions are applied to versioned values, the results are versioned values that are intuitively created by applying version-specific functions to version-specific values in a version-wise manner. For example, we get `{1=1, 2=3}` if we apply `{1=\x.x, 2=\x.x+1}` to `{1=1, 2=1}`.

To study our approach from the viewpoint of type safety, we develop a calculus λ_{VL} based on $\ell\mathcal{R}$PCF [3] that is a coeffect system [14] based on PCF [15]. A coeffect system can be considered as a framework for analyzing usage of resources that is called coeffects. The coeffects of λ_{VL} are sets of version numbers. In other words, the type system of λ_{VL} analyzes the versions of definitions that are necessary to have a "safe" program. Unfortunately, λ_{VL} does not support either data structures, classes or objects. Extending λ_{VL} to support them is left for the future work.

The rest of the paper is organized as follows. Section 1 shows a motivating example and discusses why COP techniques are not suitable. Section 3 Section 4 introduces versioned values and definitions and Section 4 gives the calculus λ_{VL}. Section 5 discusses related studies and techniques and Section 6 concludes the paper.

2 EXAMPLE: AVOIDING DEPENDENCY HELL

Dependency hell is one of the common problems/frustrations induced by incompatibility among versions of programs. In this section, we consider as an example a program that gets the number of connected physical monitors using GDK 3, a part of GTK+ 3 library.

2.1 Incompatibility in GDK 3

Table 1 shows availability of the two functions that gets the number of connected physical monitors namely

- `gdk_screen_get_n_monitors` and
- `gdk_display_get_n_monitors`.

The symbol yes denotes that the function is available; no denotes that the function is not available; and dep denotes that the function is deprecated. The version 3.22 of GDK deprecates and changes the function `gdk_screen_get_n_monitors` that gets the number of connected physical monitors. We should instead use `gdk_display_get_n_monitors` to get the number of connected physical monitors in the versions higher than or equal to 3.22.

If we consider deprecated functions unavailable, the version 3.22 is structurally incompatible with the version 3.20 because the former lacks `gdk_screen_get_n_monitors` that is available in the latter.

In the following sections, we consider for simplicity

- `gdk_screen_get_n_monitors` available in only the versions lower than 3.22 and
- `gdk_display_get_n_monitors` available in only the versions higher than or equal to 3.22.

2.2 COP approach

One solution to avoid dependency hell is to allow a library file to contain all versions of the library and to allow programs that depends on the library to use a version at a time at runtime. Considering versions as contexts, it can be achieved in a COP language naturally by having a layer for each version and by activating and deactivating layers so that at most one layer is active at a time.

For example, we can represent structural incompatibility of GDK 3 between the versions 3.9 and 3.22 in JCop [1] as follows.

```
class G{}
layer V3_20{
  int G.gdk_screen_get_n_monitors(){
    return /* the number of connected monitors */;
  }}
layer V3_22{
  int G.gdk_display_get_n_monitors(){
    return /* the number of connected monitors */;
  }}
```

The class `G` represents the set of functions that GDK 3 provides in a version. It has no methods by default, which means that GDK 3 provides no function if we do not specify any version. The layer `V3_20` adds the functions that are available in the version 3.20 of GDK 3 and here it adds `gdk_screen_get_n_monitors`. Similarly, the layer `V3_22` adds the functions that are available in the version 3.22 of GDK 3 and here it adds `gdk_display_get_n_monitors`.

Programs that use GDK 3 also use layers to import specific versions of the library. Using layer inheritance is one way to achieve that. A better way is using the `requires` clause proposed in Safe JCop [10] because we can represent importing specific versions of several libraries naturally. For example, the function that takes a screenshot for each monitor is implemented as follows.

```
layer ScreenshotV1 requires V3_20{
  int G.take_screenshots(){
    for( int i = 0;
         i < gdk_screen_get_n_monitors(); ++i){
      ...
    }}}
layer ScreenshotV2 requires V3_22{
  int G.take_screenshots(){
    for( int i = 0;
         i < gdk_display_get_n_monitors(); ++i){
      ...
    }}}
```

The layers `ScreenshotV1` and `ScreenshotV2` represent the programs that use the versions 3.20 and 3.22 of GDK 3, respectively. They define basically the same functions namely `take_screenshots` that take a screenshot in each monitor.

The applications that use the screenshot library select one version by activating `ScreenshotV1` or `ScreenshotV2` as follows.

```
with(new V3_20(), new ScreenshotV1()){
  new G().take_screenshots();
}
```

In this case, the application uses the version 3.20 of GDK 3.

2.3 Problem

A problem of the above COP approach is that we need a dummy class like `G` even though the programs and libraries do not need any class. A desirable solution in the above example must be independent of classes and objects.

GDK versions	gdk_screen_get_n_monitors	gdk_display_get_n_monitors
lower than 3.22	yes	no
higher than or equal to 3.22	dep	yes

Table 1: Availability and results of functions in GDK 3

Realizing the desirable solution is, however, not easy because, even if a function has several bodies depending on contexts, it is not clear how to select one body. In COP languages with classes and objects, selecting one method body for the context is easy because the receiver objects know the context. If we consider COP languages without classes and objects, there is no such value explicitly. In other words, we need a COP language that supports static partial methods that belongs to classes as static methods.

3 OUR APPROACH

We propose versioned values as a construct for COP without classes and objects. Versioned values are basically pairs of a version number and the value that is specific to the version.

In this section, we briefly explain versioned values in a hypothetical statically typed language VL where functions and versioned values are first-class values.

3.1 Versioned values

A versioned value consists of pairs of a version name/number and the value specific to the version and the default version. We say that a versioned value is available in the version v if the versioned value contains the version name/number v. For example, the following code creates a versioned value that is available in the versions V3_20 and V3_22, which are the version names that denote the versions 3.20 and 3.33 of GDK 3, and the version-specific values are 20 and 22, respectively. The default version is V3_22.

```
{ V3_20 = 20, V3_22 = 22 | V3_22 }
```

Types of versioned values show the versions where the version-specific values are available along with the types of the version-specific values. For example, the type of the value in the example above is Int!{V3_20,V3_22}, which means that the versioned value is available in the versions V3_20 and V3_22 and that a value of type Int is available in each version.

Functions that have different bodies among versions are represented as versioned values. For example, the function gdk_screen_get_n_monitors is defined as follows.

```
{ V3_20 = \x:unit./*the number of connected monitors*/
        | V3_20}
```

The type of the versioned value is unit→int!{V3_20}, which means that the versioned value is available in the version V3_20 and that a function that takes the unit value and returns an integer value is available in the version.

3.2 Versioned function application

Applying versioned functions to versioned values corresponds to invoking partial methods on objects in COP languages with classes and objects. What we want to do is, given a version name, to extract the function and the value that are associated with the version name from the versioned functions and values.

We consider the default versions in versioned values specifying the current version. To extract values from versioned values following to the default versions, we introduce a variation of let-bindings called contextual let-bindings. Let gdk_screen_get_n_monitors be a versioned function of type unit→int!{V3_20} defined as in the previous section and u be a versioned value of type unit!{V3_20,V3_22} defined as follows.

```
{ V3_20 = (), V3_22 = () | V3_22 }
```

To apply gdk_screen_get_n_monitors to u, we use contextual let-bindings as follows.

```
let !f = gdk_screen_get_n_monitors in
let !x = u in f x
```

The program first extracts the function in the version V3_20 from gdk_screen_get_n_monitors and binds f to the function. It then extracts the value in the version V3_22 from u and binds x to the value and appliques f to x.

3.3 Version-independent programs

If versioned functions and versioned values are available in several versions, it is desirable to have one program that applies the versioned functions to the versioned values in several versions. Suppose that gdk_screen_get_n_monitors is available in V3_20 and V3_22 for example. It is then desirable to have one program that computes the number of connected physical monitors in V3_20 and V3_22.

We achieve this by using the suspension operator ! and explicit extraction expression e.v where e is an expression that returns a versioned value and v is a version name. The suspension operator ! takes one expression and suspends the evaluation of the expression until the value is required. For example, the following program returns a suspended computation that returns an integer value in V3_20 and V3_22.

```
let !f = gdk_screen_get_n_monitors in
let !x = u in !(f x)
```

The explicit extraction expression e.v extracts the value in the version v from the versioned value returned by e. Here the default versions in e are overridden by the specified version v. For example, the following program computes the number of connected physical monitors in the version V3_20.

```
let !f = gdk_screen_get_n_monitors in
let !x = u in !(f x).V3_20
```

The default version of u is overridden by V3_22.

$$
\begin{array}{rrcl}
\text{(Terms)} & \mathtt{t}, \mathtt{u} & ::= & \mathtt{x} \mid \mathtt{()} \mid \mathtt{\backslash x.t} \mid \mathtt{t\ t} \mid \mathtt{\{}\overline{\mathtt{l=t}}\mid \mathtt{l}\}\mid \\
& & & \mathtt{t.l} \mid \mathtt{let\ !x\ =\ t\ in\ t} \mid \mathtt{!t} \mid \\
& & & \langle \overline{\mathtt{l=t}} \mid \mathtt{l} \rangle \\
\text{(Values)} & \mathtt{v} & ::= & \mathtt{()} \mid \mathtt{\backslash x.t} \mid \mathtt{\{}\overline{\mathtt{l=t}}\mid \mathtt{l}\} \mid \mathtt{!t} \\
\text{(Types)} & \mathtt{S}, \mathtt{T}, \mathtt{U} & ::= & \mathtt{()} \mid \mathtt{T} \to \mathtt{T} \mid \mathtt{!_r T} \\
\text{(Typing Env.)} & \Gamma, \Delta & ::= & \emptyset \mid \Gamma, \mathtt{x} : \mathtt{T} \mid \Gamma, \mathtt{x} : \mathtt{[T]_r} \\
\text{(Eval. Context)} & E & ::= & [] \mid E\ \mathtt{t} \mid E.\mathtt{l} \mid \mathtt{let\ !x=}E\ \mathtt{in\ t}
\end{array}
$$

Figure 1: Syntax of λ_{VL}

3.4 Abstracting version-dependent behavior

It is desirable if we can abstract version-dependent behavior because this allows us to develop version-independent programs. For example, `take_screenshots` abstracts version-dependent behavior that invokes `gdk_screen_get_n_monitors` in the version `V3_20` and `gdk_display_get_n_monitors` in the version `V3_22`. Programs that use `take_screenshots` are independent of any specific version.

In our language, we can abstract version-dependent behavior easily by using contextual let-bindings and versioned values. For example, the following program creates a versioned function that abstracts the version-dependent behavior that calls `gdk_screen_get_n_monitors` in `V3_20` and `gdk_display_get_n_monitors` in `V3_22`.

```
let !f = gdk_screen_get_n_monitors in
let !g = gdk_display_get_n_monitors in
{V3_20 = f (), V3_22 = g ()}
```

4 λ_{VL}: A CALCULUS FOR VERSIONED VALUES

In this section, we develop a core calculus of VL based on $\ell\mathcal{R}$PCF [3], which is an extension of the simply typed lambda calculus with contextual let-bindings and contextual types.

4.1 Context

The contexts in λ_{VL} denoted by the metavariable $\mathtt{R}$ and ranged over by the metavariable $\mathtt{r}$ are the sets of version numbers with the bottom denoted by $\bot$. We define a preordered semiring [7] $(\mathtt{R}, \leq, \oplus, \otimes, 0, 1)$ over the contexts as follows.

$$
0 = \bot \quad 1 = \{\} \qquad \bot \leq \mathtt{r} \qquad \frac{\mathtt{r}_1 \subseteq \mathtt{r}_2}{\mathtt{r}_1 \leq \mathtt{r}_2}
$$

$$
\mathtt{r}_1 \oplus \mathtt{r}_2 = \begin{cases} \mathtt{r}_1 & \text{if } \mathtt{r}_1 \text{ is } \bot \\ \mathtt{r}_2 & \text{if } \mathtt{r}_2 \text{ is } \bot \\ \mathtt{r}_1 \cup \mathtt{r}_2 & \text{otherwise} \end{cases}
$$

$$
\mathtt{r}_1 \otimes \mathtt{r}_2 = \begin{cases} \bot & \text{if at least either } \mathtt{r}_1 \text{ or } \mathtt{r}_2 \text{ is } \bot \\ \mathtt{r}_1 \cup \mathtt{r}_2 & \text{otherwise} \end{cases}
$$

$\mathtt{r}_1 \subseteq \mathtt{r}_2$ is the standard subset relation over sets and none of $\mathtt{r}_1$ and $\mathtt{r}_2$ is the bottom.

4.2 Syntax

Figure 1 shows the syntax of λ_{VL}. The metavariables $\mathtt{x}$ and $\mathtt{y}$ range over variables; $\mathtt{l}$, $\mathtt{m}$ and $\mathtt{n}$ range over version numbers; $\mathtt{r}$ ranges over *contexts* that are sets of version numbers with the bottom. We sometimes denote a sequence of version numbers, i.e., $\mathtt{l}_1, \cdots, \mathtt{l}_n$, by $\overline{\mathtt{l}}$. If the length of a sequence is not important or is clear from the context, we do not give it explicitly.

Reduction rules: $\boxed{\mathtt{t} \rightsquigarrow \mathtt{t}'}$

$$
(\mathtt{\backslash x.u})\ \mathtt{t} \rightsquigarrow [\mathtt{x} \mapsto \mathtt{t}]\mathtt{u} \quad \text{(E-Abs)}
$$

$$
\mathtt{let\ !x=!t\ in\ u} \rightsquigarrow [\mathtt{x} \mapsto \mathtt{t}]\mathtt{u} \quad \text{(E-LetB)}
$$

$$
\mathtt{let\ !x=}\{\overline{\mathtt{l=t}}\mid \mathtt{m}\}\ \mathtt{in\ u} \rightsquigarrow [\mathtt{x} \mapsto \langle \overline{\mathtt{l=t}} \mid \mathtt{m} \rangle]\mathtt{u} \quad \text{(E-LetP)}
$$

$$
\langle \overline{\mathtt{l=t}} \mid \mathtt{l}_i \rangle \rightsquigarrow \mathtt{t}_i\mathtt{@l}_i \quad \text{(E-Ver)}
$$

$$
(\mathtt{!t}).\mathtt{l} \rightsquigarrow \mathtt{t@l} \ \text{(E-ProjB)} \qquad \{\overline{\mathtt{l=t}}\mid \mathtt{m}\}.\mathtt{l}_i \rightsquigarrow \mathtt{t}_i\mathtt{@l}_i \ \text{(E-ProjV)}
$$

Evaluation rules: $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$
\frac{\mathtt{t} \rightsquigarrow \mathtt{u}}{E[\mathtt{t}] \longrightarrow E[\mathtt{u}]}
$$

Figure 2: Evaluation rules

A term in λ_{VL} is either a variable $\mathtt{x}$, unit value $\mathtt{()}$, lambda abstraction $\mathtt{\backslash x.t}$, application $\mathtt{t\ t}$, versioned values $\{\overline{\mathtt{l=t}}\mid \mathtt{l}_i\}$ where $\mathtt{l}_i$ is the default version and satisfies $\mathtt{l}_i \in \{\overline{\mathtt{l}}\}$, explicit extraction $\mathtt{t.l}$, contextual let-binding $\mathtt{let\ !x\ =\ t\ in\ t}$, lifting $\mathtt{!t}$, and versioned computation $\langle \overline{\mathtt{l=t}} \mid \mathtt{l}_i \rangle$ where $\mathtt{l}_i$ is the default version and satisfies $\mathtt{l}_i \in \{\overline{\mathtt{l}}\}$. The versioned computations represent terms of which evaluation in the default contexts is postponed. intuitively. We assume that versioned computations appear in only the intermediate terms in evaluation and do not appear in the users' code. A value is either the unit, a lambda abstraction, a versioned value or a lifting. We say a value is *proper* if it is the unit or a lambda abstraction. A type is either the unit type $\mathtt{()}$, a function $\mathtt{T} \to \mathtt{T}$ or a versioned type $\mathtt{!_r T}$.

We denote by Γ and Δ a typing environment. It is a set of typed variables which are either of the form $\mathtt{x} : \mathtt{T}$ called plain variables or $\mathtt{x} : \mathtt{[T]_r}$ called versioned variables. Versioned variables are a technical artifact useful to implicitly manage variables bound to versioned values. If we denote by $\mathtt{[\Gamma]}$ a versioned environment (a typing environment that only contains versioned variables) we can extend the operation $\oplus$ of the semiring to typing environments as follows.

$$
\begin{array}{rcll}
\emptyset \oplus \Gamma & = & \Gamma & \\
(\mathtt{x} : \mathtt{[A]_r}) \oplus (\mathtt{x} : \mathtt{[T]_{r'}}, \Gamma) & = & \mathtt{x} : \mathtt{[T]_{r \oplus r'}}, \Gamma & \\
(\mathtt{x} : \mathtt{[T]_r}) \oplus \Gamma & = & \mathtt{x} : \mathtt{[T]_r}, \Gamma & \text{if } \mathtt{x} \notin \Gamma \\
(\mathtt{x} : \mathtt{T}, \Gamma) \oplus \Delta & = & \mathtt{x} : \mathtt{T}, (\Gamma \oplus \Delta) & \text{if } \mathtt{x} \notin \Delta
\end{array}
$$

We denote $\mathtt{r}_1 \oplus \cdots \oplus \mathtt{r}_n$ and $\Gamma_1 \oplus \cdots \oplus \Gamma_n$ by $\Sigma_i^n \mathtt{r}$ and $\Sigma_i^n \Gamma_i$, respectively. We omit n for readability if it is clear from the context or is not important.

We denote by E an evaluation context. Each evaluation context is a term with a hole written as [] somewhere inside it. We write $E[\mathtt{t}]$ for the ordinary term obtained by replacing the hole in E with $\mathtt{t}$.

$$()@l \equiv () \qquad (\backslash x.t)@l \equiv \backslash x.(t@l) \qquad (!t)@l \equiv !t$$

$$(t\ u)@l \equiv (t@l)\ (u@l)$$

$$(\texttt{let !x=t in u})@l \equiv \texttt{let !x=(t@l) in (u@l)}$$

$$\{\overline{l=t}|l_i\}@l' \equiv \{\overline{l=t}|l_i\} \qquad (t.m)@l \equiv (t@l).m$$

$$\frac{n \in \{\overline{l}\}}{(\langle\overline{l=t}|m\rangle)@n \equiv \langle\overline{l=t}|n\rangle} \qquad \frac{n \notin \{\overline{l}\}}{(\langle\overline{l=t}|m\rangle)@n \equiv \langle\overline{l=t}|m\rangle}$$

Figure 3: Context substitution

4.3 Dynamic semantics

We give the rules for the small-step operational semantics of λ_{VL} in Figure 2. It basically follows the lazy-evaluation strategy, i.e., only functions t are evaluated to values to evaluate applications t u.

We have six reduction rules. E-Abs is the β-reduction rule. E-LetB and E-LetP reduce the contextual let-bindings let !x=v in t by replacing x in u with the value v. E-LetB replaces x with t if the value is a lifting !t. E-LetP translates v to a versioned computation and replaces x with it if the value is a versioned value. E-Ver reduces a versioned computation $\langle\overline{l=t}|l_i\rangle$ to t_i and substitutes the default versions within the subterms to l_i. The context substitution t@l is defined in Figure 3. It basically substitutes the default version in the every subterm of t except for the cases where t is either a lifting, versioned value, or versioned computation. The most interesting cases are versioned computations. $\langle\overline{l=t}|m\rangle@n$ becomes $\langle\overline{l=t}|n\rangle$ if $n \in \{\overline{l}\}$ and $\langle\overline{l=t}|m\rangle$ otherwise. The context substitution therefore ensures that E-Ver is always successful. E-ProjB and E-ProjV reduces v.l by extracting a term in the value v for the context l. If the value is a lifting !t, E-ProjB reduces the term to t where the default version is substituted to l. If the value is a versioned value, E-ProjV takes a term for the context l and substitute the default version with l.

Example 4.1. The following program simplifies the last program in the Section 3 and reduced as follows.

```
let !gdk_screen_get_n_monitors = {V3_20 = \x.x | V3_20} in
let !gdk_display_get_n_monitors = {V3_22 = \x.x | V3_22} in
{V3_20 = gdk_screen_get_n_monitors (),
 V3_22 = gdk_display_get_n_monitors () | V3_22}
```

⟶ (E-LetB)

```
let !gdk_display_get_n_monitors = {V3_22 = \x.x | V3_22} in
{V3_20 = [V3_20 = \x.x | V3_20] (),
 V3_22 = gdk_display_get_n_monitors () | V3_22}
```

⟶ (E-LetB)

```
{V3_20 = [V3_20 = \x.x | V3_20] (),
 V3_22 = [V3_22 = \x.x | V3_22] () | V3_22}
```

Let p be the versioned value. p.V3_22 is further evaluated as follows.

```
{V3_20 = [V3_20 = \x.x | V3_20] (),
 V3_22 = [V3_22 = \x.x | V3_22] () | V3_22}.V3_22
```

⟶ (E-ProjV)

```
[V3_22 = \x.x | V3_22] ()
```

$$x : T \vdash x : T \quad \text{(T-Var)} \qquad \emptyset \vdash () : () \quad \text{(T-Unit)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \backslash x.t : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t : T' \to T \qquad \Delta \vdash t' : T'}{\Gamma \oplus \Delta \vdash t\ t' : T} \quad \text{(T-App)}$$

$$\frac{\Gamma' \vdash t : T' \qquad \Gamma <: \Gamma' \qquad T' <: T}{\Gamma, [\Delta]_0 \vdash t : T} \quad \text{(T-Sub)}$$

$$\frac{\Gamma \vdash t_1 : !_r T' \qquad \Delta, x : [T']_r \vdash t_2 : T}{\Gamma \oplus \Delta \vdash \texttt{let !x = } t_1 \texttt{ in } t_2 : T} \quad \text{(T-Let)}$$

$$\frac{\Gamma, x : T' \vdash t : T}{\Gamma, x : [T']_1 \vdash t : T} \text{(T-Der)} \qquad \frac{[\Gamma] \vdash t : T}{r \otimes [\Gamma] \vdash !t : !_r T} \text{(T-Pr)}$$

$$\frac{\forall i.[\Gamma_i] \vdash t_i : T}{\Sigma_i\{l_i\} \otimes [\Gamma_i] \vdash \{\overline{l=t}|l_i\} : !_{\{\overline{l}\}} T} \quad \text{(T-Ver)}$$

$$\frac{\forall i.[\Gamma_i] \vdash t_i : T}{\Sigma_i[\Gamma_i] \vdash \langle\overline{l=t}|l_i\rangle : T} \text{(T-VerI)} \qquad \frac{\Gamma \vdash t : !_r T \qquad l \in r}{\Gamma \vdash t.l : T} \text{(T-Extr)}$$

Figure 4: Typing rules

$$T <: T \quad \text{(S-Refl)} \qquad \frac{T' <: T \qquad S <: S'}{T \to S <: T' \to S'} \text{(S-Arr)}$$

$$\frac{T <: T' \qquad r' \le r}{!_r T <: !_{r'} T'} \text{(S-Cdc)} \qquad \frac{T <: T' \qquad r' \le r}{[T]_r <: [T']_{r'}} \text{(S-Cdp)}$$

Figure 5: Subsumption

⟶ (E-Ver) \x.x () ⟶ (E-App) ()

4.4 Static semantics

The type system of λ_{VL} ensures that extractions never fail by maintaining the context of each subterm and bound variable. We say that the term t is well-typed if there exists some Γ and T such that $\Gamma \vdash t : T$ defined in Figure 4. The readers should consider each typing rule as "the term t requires the resources Γ and generates the resource T".

We explain the interesting last seven rules, though the first seven rules come from $\ell\mathcal{R}$PCF. T-Sub "weaken" not only the types of the terms but also the typing environments. The subsumption relation T <: S is given in Figure 5. From the view point of the contexts, a subtype is associated with larger contexts. For example, $[T]_r$ is a subtype of $[T]_{r'}$ if $r' \le r$. Therefore, T-Sub intuitively allows terms to require resources available in more contexts and generate resources available in fewer contexts than they actually do. T-Let adds versioned variables to the typing environments. T-Der converts a non-contextual type to a contextual type. It follows the intuition that resources are available in any contexts if they are free from any contexts. T-Pr and T-Ver intuitively repeat over contexts the requirements/generations of the resources for/of the given terms.

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \backslash x.x : () \to ()}}{\emptyset \vdash \{V3\_20 = \ldots | V3\_20\} : !_{\{V3\_20\}}() \to ()}\ A}{\emptyset \vdash p : !_{\{V3\_20, V3\_22\}}()}$$

Subtree A

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \backslash x.x : () \to ()}}{\emptyset \vdash \{V3\_22 = \ldots | V3\_22\} : !_{\{V3\_22\}}() \to ()}\ B}{f : [() \to ()]_{\{V3\_20\}} \vdash \texttt{let !g = ... in ...} : !_{\{V3\_20, V3\_22\}}()}$$

Subtree B

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : () \to () \vdash f : () \to ()}}{f : [() \to ()]_{\{\}} \vdash f : () \to ()} \quad \emptyset \vdash () : ()}{f : [() \to ()]_{\{\}} \vdash f\ () : ()} \quad \ldots}{f : [() \to ()]_{\{V3\_20\}}, g : [() \to ()]_{\{V3\_22\}} \vdash \{\ldots\} : !_{\{V3\_20, V3\_22\}}()}$$

Figure 6: Type derivation of Example 4.1

T-VerI is similar to T-Ver but it does not repeat the requirements and generations. This is because each versioned computation represents a computation in only the default version. T-Extr ensures that values in the specified contexts are available and extracts the types from the versioned types.

Example 4.2. The type of the program given in Example 4.1 is $!_{\{V3\_20, V3\_22\}}()$. The derivation is as follows, where p, f, and g are the program, `gdk_screen_get_n_monitors`, and `gdk_display_get_n_monitors`, respectively. Note that the type system successfully allows f and g to appear in the body of the versioned value. The type system successfully understands that for example `gdk_screen_get_n_monitors` is available and used in only `V3_20`.

Example 4.3. If we swap the use of `gdk_screen_get_n_monitors` and `gdk_display_get_n_monitors` in Example 4.1 as follows, the program is never well-typed.

```
let !gdk_screen_get_n_monitors = {V3_20 = \x.x | V3_20} in
let !gdk_display_get_n_monitors = {V3_22 = \x.x | V3_22} in
{V3_20 = gdk_display_get_n_monitors (),
 V3_22 = gdk_screen_get_n_monitors () | V3_22}
```

This is because it is not possible to have a subtree that corresponds to B in Figure 6.

4.5 Properties

This section proves that our type system is sound.

Lemma 4.4. *If* $\Gamma, x : S \vdash t : T$ *and* $\Delta \vdash u : S$ *then* $\Gamma \oplus \Delta \vdash [x \mapsto u]t : T$.

Proof. By induction on the derivation of $\Gamma, x : S \vdash t : T$. □

The following lemma generalizes the Lemma 2 in [7] but the proof depends on the fact that $\oplus$ and $\otimes$ are idempotent, i.e., $r \oplus r = r$ and $r \otimes r = r$.

Lemma 4.5. *If* $\Gamma, x : [S]_s \vdash t : T$ *and* $[\Delta] \vdash u : S$ *then* $\Gamma \oplus \Sigma_i(s_i \otimes [\Delta_i]) \vdash [x \mapsto u]t : T$ *where* $\Sigma_i s_i = s$ *and* $\Sigma_i[\Delta_i] = [\Delta]$.

Proof. By induction on the derivation of $\Gamma, x : S \vdash t : T$. □

Lemma 4.6. *If* $\Gamma \vdash t : T$ *and* $t \rightsquigarrow u$ *then* $\Gamma \vdash u : T$

Proof. By induction on the derivation of $\Gamma \vdash t : T$ with the lemmas 4.4 and 4.5. □

Theorem 4.7. *If* $\Gamma \vdash t : T$ *and* $t \longrightarrow u$ *then* $\Gamma \vdash u : T$.

Proof. By induction on the structure of the context E with the lemma 4.6. □

Theorem 4.8. *If* $\emptyset \vdash t : T$ *then (1)* t *is a value or (2) there is some* s *such that* $t \longrightarrow u$.

Proof. By induction on the derivation of $\Gamma \vdash t : T$. □

5 RELATED WORK

Because a coeffect system is basically a technique to analyze resource usage [14], any work on the resource usage analysis [9] is relevant to the work generally. Such work, however, lacks versioned values. Only the availability of each function depends on the states of the resources and its behavior never changes.

Programming techniques for software product lines [2, 16, 17] are closely related to the work because we can consider program evolution as program extension. For example, delta-oriented programming [17] modularizes modifications to programs such as adding classes and methods by using so-called delta modules. Although composing delta modules with the base program is usually static, i.e., fixed at runtime, there are also some studies that allow changing the compositions at runtime [5, 6]. They allow objects to live in multiple compositions of delta modules as COP languages, but they treat states of such objects seriously.

Studies on multi-stage programming languages such as MetaML [19] also relate to the work. For example, we can consider liftings, contextual let-bindings, and extractions as brackets, escapes and the `run`, respectively. In the type systems, environment classifiers [12, 18] in λ^{α} annotate variables with the stages where they are available, which is similar to the context-dependent variables in λ_{VL} and $\ell\mathcal{R}$PCF. We are not yet sure whether it is possible to develop a calculus for VL based on λ^{α} and its descendants or not. Studying the relevance is left for our future work.

6 CONCLUSIONS AND FUTURE WORK

We proposed a programming language VL to allow libraries to have multiple versions of functions and users to select one specific version. We developed a core calculus λ_{VL} of VL and proved its soundness.

Our future work includes integration with data types, experiments and efficient implementation. In particular, efficient implementation seems challenging because our semantics employs the lazy-evaluation strategy.

We are not sure whether in the evaluation of `t.l` every context-dependent computation in `t` runs in the context `l` or not, although this is very important in λ_{VL}. If it holds, we can omit the last context substitution rule in Figure 3. Addressing this is another direction of our future work.

REFERENCES

[1] Malte Appeltauer, Robert Hirschfeld, and Jens Linckeb. 2013. Declarative layer composition with the JCop programming language. *Journal of Object Technology* 12, 2 (2013). https://doi.org/10.5381/jot.2013.12.2.a4

[2] Jan Bosch. 2001. Software Product Lines: Organizational Alternatives. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*. IEEE Computer Society, Washington, DC, USA, 91–100. http://dl.acm.org/citation.cfm?id=381473.381482

[3] AloÃŕs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Programming Languages and Systems (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

[4] Bradley E. Cossette and Robert J. Walker. 2012. Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, 55:1–55:11. https://doi.org/10.1145/2393596.2393661

[5] Ferruccio Damiani, Luca Padovani, and Ina Schaefer. 2012. A Formal Foundation for Dynamic Delta-oriented Software Product Lines. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE '12)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2371401.2371403

[6] Ferruccio Damiani and Ina Schaefer. 2011. Dynamic Delta-oriented Programming. In *Proceedings of the 15th International Software Product Line Conference, Volume 2 (SPLC '11)*. ACM, New York, NY, USA, 34:1–34:8. https://doi.org/10.1145/2019136.2019175

[7] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining Effects and Coeffects via Grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 476–489. https://doi.org/10.1145/2951913.2951939

[8] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented programming. *Journal of Object Technology* 7, 3 (2008), 125–151. https://doi.org/10.5381/jot.2008.7.3.a4

[9] Atsushi Igarashi and Naoki Kobayashi. 2005. Resource Usage Analysis. *TOPLAS* 27, 2 (2005), 264–313.

[10] Hiroaki Inoue and Atsushi Igarashi. 2015. A Sound Type System for Layer Subtyping and Dynamically Activated First-Class Layers. In *Programming Languages and Systems (Lecture Notes in Computer Science)*. Springer, Cham, 445–462. https://doi.org/10.1007/978-3-319-26529-2_24

[11] Tetsuo Kamina, Tomoyuki Aotani, Hidehiko Masuhara, and Atsushi Igarashi. 2018. Method safety mechanism for asynchronous layer deactivation. *Sci. Comput. Program.* 156 (2018), 104–120. https://doi.org/10.1016/j.scico.2018.01.006

[12] Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science)*, Atsushi Igarashi (Ed.), Vol. 10017. 271–291. https://doi.org/10.1007/978-3-319-47958-3_15

[13] S. Mostafa, R. Rodriguez, and X. Wang. 2017. A Study on Behavioral Backward Incompatibility Bugs in Java Software Libraries. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 127–129. https://doi.org/10.1109/ICSE-C.2017.101

[14] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-dependent Computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 123–135. https://doi.org/10.1145/2628136.2628160

[15] G. D. Plotkin. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5, 3 (Dec. 1977), 223–255. https://doi.org/10.1016/0304-3975(77)90044-5

[16] Christian Prehofer. 1997. Feature-oriented programming: A fresh look at objects. In *ECOOP'97*. 419–443.

[17] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-oriented Programming of Software Product Lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10)*. Springer-Verlag, Berlin, Heidelberg, 77–91. http://dl.acm.org/citation.cfm?id=1885639.1885647

[18] Walid Taha and Michael Florentin Nielsen. 2003. Environment Classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 26–37. https://doi.org/10.1145/604131.604134

[19] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1 (Oct. 2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

[20] L. Xavier, A. Brito, A. Hora, and M. T. Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 138–147. https://doi.org/10.1109/SANER.2017.7884616