A Step toward Programming with Versions in Real-World Functional Languages

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

Luthfun Anshar Lubis
luthfanlubis@prg.is.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Tomoyuki Aotani
t.aotani@gmail.com
Mamezou Holdings Co., LTD.
Tokyo, Japan

Hidehiko Masuhara
masuhara@acm.org
Tokyo Institute of Technology
Tokyo, Japan

ABSTRACT

λ_{VL} is a core calculus based on the concept of programming with versions that supports multiple versions of program definitions and values inherently in the semantics of a language. However, since λ_{VL} was not designed as a surface language, its complex syntax and semantics only provide primitive constructs to manipulate versioned values. In order to realize the programming with versions concept in a real-world language, we propose a compilation method for functional languages through λ_{VL} and discuss how real-world programs can be written in a Haskell-like functional language with versions.

CCS CONCEPTS

• **Theory of computation → Type theory**; **Linear logic**; • **Software and its engineering → Software configuration management and version control systems**; **Software libraries and repositories**.

KEYWORDS

Program evolution, dependencies, software product lines, Graded modal types, Coeffects

ACM Reference Format:
Yudai Tanabe, Luthfun Anshar Lubis, Tomoyuki Aotani, and Hidehiko Masuhara. 2022. A Step toward Programming with Versions in Real-World Functional Languages. In *COP 2022: International Workshop on Context-Oriented Programming and Advanced Modularity (collocated with ECOOP) (COP '22), June 7, 2022, Berlin, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3570353.3570359

1 INTRODUCTION

Updates are an essential part of the software life cycle. Through updates, developers improve maintainability, fix bugs, optimize performance, and provide new features. Units of updates are called packages, each of which is distinguished by version numbers.

Some updates are incompatible to their previous versions, which can cause problems with the client software. The situation becomes more complicated in software development with many packages some of which depend some others. Sometimes, two incompatible versions of a package are required from different packages in a system, which is not possible to build, or link, or load in most programming languages.

Many existing techniques seek to avoid using two versions of a package with the same name. Semantic versioning, for example, enforces versioning conventions to indicate incompatible changes and helps developers avoid updating to incompatible versions. Package name mangling changes the package name so that two versions can be used simultaneously. This technique is used in advanced package managers such as cargo[1] and npm [2], but it has the problem of treating the two versions as completely different ones.

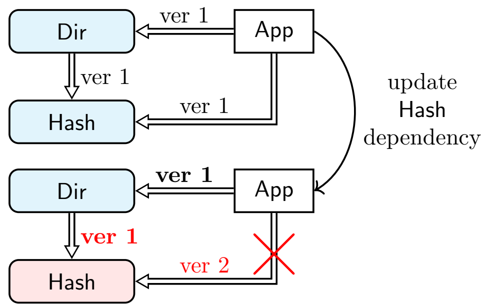
Despite the proliferation of such versioning rules and techniques, updating remains a burden on package users. Even in ecosystems that adopt semantic versioning, one-third of minor updates introduce at least one breaking change [17]. Updating dependent packages to incompatible versions is an extra burden on package users, making them reluctant to update dependent packages [1]. As a result, adoption of new versions in the downstream is much slower than the evolution of packages in the upstream [12].

Programming with versions [22, 23] is a proposal to embrace incompatible updates by allowing to use multiple versions of programming elements simultaneously. λ_{VL} is a core calculus based on the proposal, where a *versioned value* represents multiple possibilities of a value (including a function value) computed in different versions. Application of a versioned value to a versioned (functional) value performs computation with pairs of matching versions. The type system guarantees the existence of a matching version in every sub-components in the program.

λ_{VL} is a mere calculus, not a fully-fledged language. Programming directly in λ_{VL} is infeasible because (1) a unique syntax derived from substructural languages, which is difficult to understand (2)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
COP '22, June 7, 2022, Berlin, Germany
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9986-9/22/06…$15.00
https://doi.org/10.1145/3570353.3570359

[1] If two or more packages have a common dependency on an incompatible version, cargo will build two separate copies of the dependency [8].
[2] npm builds dependencies graph as dependency trees. If multiple versions of a package with the same name are indirectly needed, npm allows each version as a nested dependency. This strategy bloats code size, so newer versions have mechanisms to reduce redundant dependencies. [13]



App directly depends on Dir and Hash. Dir depends on Hash. After an update, App requires the two distinct versions of Hash.

Figure 1: App dependencies before (top) and after (bottom) the update.

exposed versions as a version label in the code, so developers need to understand the type system to write safe programs.

This paper proposes a method of compiling real-world functional language through λ_{VL}. This allows developers to write programs in their favorite functional language while enjoying the benefits of programming with versions; *i.e.*, simultaneous use of multiple versions and strict checking of version inconsistency.

The compilation consists of two steps: compilation of Haskell-like functional language into GrMini and bundling top-level definitions as a versioned record. The first step compiles each version of a program in a Haskell-subset to GrMini [14], a subset of λ_{VL}. This compilation is based on the Girard's translation [9] from the simply-typed lambda calculus into the linear calculus, and eliminates the complicated syntax from the surface language for λ_{VL}. The second step bundles all versions of each top-level definition into a single versioned record. This translation is performed for each corresponding symbol and uses version information as a version label. This method makes it possible to omit version information from the surface program and allows programmers to construct λ_{VL} programs more easily.

As an extension of this translation technique, we discuss a method for extracting a specific version of the Haskell-subset program from a λ_{VL} program. This method uses type information to translate a λ_{VL} program into a version-specified extraction. We expect the extracted program preserves the original semantics of the original Haskell-subset program. It is also possible to reuse an existing runtime to run programs retrieved in this manner.

The rest of this paper is organized as follows. Section 2 revisits λ_{VL} with a motivating example. Section 3 demonstrates how to compile a Haskell program through λ_{VL} by using concrete examples. Section 4 discusses further developments expected under programming with versions. Section 5 shows related work. Finally, section 6 concludes the paper.

2 PROGRAMMING WITH VERSIONS

2.1 Motivating Example

Before explaining the main ideas behind programming with versions, we explain a small example to illustrate the essence of incompatibility problems. Here, we consider a scenario in which a

```
1 -- App version 1
2 import Dir (exists)
3 import Hash (makeHash)
4
5 main :: ()
6 main () = let str = getArg () in
7           let digest = makeHash str in
8           if exists digest then print "Found"
9                            else error "Not found"
```

```
1 -- Dir version 1
2 import Hash (match)
3
4 exists :: String -> Boolean
5 exists hash =
6   let filelist = getFileList () in
7   foldLeft
8     (\(acc, fn) -> acc || match fn hash)
9     false
10    filelist
```

```
1 -- Hash version 1
2 makeHash :: String -> String
3 makeHash str = (* generate hash based on MD5 *)
4
5 match :: String -> String -> Boolean
6 match str hash = (makeHash str) == hash
```

Figure 2: App, Dir, and Hash modules *before* the update.

```
1 -- Hash version 2
2 makeHash :: String -> String
3 makeHash str = (* generate hash based on SHA-3 *)
4
5 match :: String -> String -> Boolean
6 match str hash = (makeHash str) == hash
```

Figure 3: Hash module *after* the update.

breaking change is introduced to a dependent package during the development of an application App.

The top of figure 1 shows the App dependencies. App is a file explorer and provides hash-based file search. This feature is developed using the system library Dir and the cryptographic library Hash. Note that Dir also depends on Hash.

App, Dir, and Hash in version 1 are shown in Figure 2. The pseudo-code is written in a Haskell-like language. Hash defines a function `makeHash` to generate a hash value from a given string using the MD5 algorithm. The `match` function determines whether the first argument string and the second argument hash are equal under the relation of the function `makeHash`.

Dir defines a function `exists` that determines if there exists a file with a file name equal to the given hash.

App uses the makeHash defined in Hash to convert a string given from standard input into a hash. Then, using `exists` defined in Dir, it prints Found if such file exists; otherwise, it results in an error `Not Found`. Assuming that there is a file with the required name in the directory, thus, the executable of App with the dependency shown in on top of Figure 1 will print Found on the standard output.

Now, the App developer updated Hash from version 1 to version 2 due to security concerns, as shown in Figure 3. Figure 3 shows the updated version of Hash. Version 2 of Hash uses SHA-3 as the new hashing algorithm.

After the update, App dependencies are changed as shown at the bottom of Figure 1. In the updated dependency, it is important to note that Dir continues to use version 1 of Hash, so that App and Dir require different Hash versions. The situation in Dir can occur for a variety of reasons. For example, Dir has already abandoned its maintenance, or perhaps other functions in Dir must continue to use the functionality provided by version 1 of Hash.

This update makes no changes to the App code but causes problems with App. When multiple versions of the same package are required, build systems reject the program. Even if the programs were successfully compiled with a technique that allows use of multiple versions, such as name mangling, the program would result in an error with an output `Not found`.

This unexpected output is due to the difference between the two versions required for Hash: the App uses version 2 of `makeHash` in line 7 of App. In line 7 of App, `makeHash` from Hash version 2 generates a hash value with SHA-3, and the value is assigned to `digest`. On the other hand, `exists` uses version 1 of `match` (including `makeHash`) to determine hash equivalence, so `exists` compares hashes generated by two completely different algorithms, SHA-3 and MD-5. As a result, in line 8 of App, `exists digest` evaluates to `false` against the expected behavior.

This example suggests that we should take care of consistency of versions in the use of values produced by packages. As long as the results of `Hash` in two versions were used independently, there would be no problem. However, it would be semantically incorrect if those results were compared each other.

Therefore, the developers use build tools with dependency analysis functionalities to avoid the simultaneous use of multiple versions in actual development. Such a tool will collect and analyze the dependencies for each package and reject the combination of packages where incompatible versions are needed simultaneously, as in the bottom of figure 1.

As a side issue, there is also a problem called version-locking [16]. In the situation shown in Figure 2, the only way to update Hash is to wait for Dir to support version 2 of Hash or to abandon Dir and rewrite the same functionality from scratch yourself. For more complex software with many dependencies, a huge amount of work is required to update a single package. The fact that all interoperator must use compatible versions of a package is a threat to software reuse. Indeed, many developers are unwilling to update dependencies unless there is a significant update [1].

2.2 Lambda VL

2.2.1 Versioned Values. Programming with versions in λ_{VL} proposes to use multiple versions simultaneously. To achieve this, λ_{VL} provides *versioned values*, constructs made of version record $\{\overline{l_i = t_i}\}$[3] representing a program that holds the values of multiple versions.

l_i is a *version label* identifying a particular version combination. For simplicity, we assume that there are only two version variations of Dir, namely l_1 for version 1 and l_2 for version 2. However, in the actual development, each module has a version space, so it is necessary to consider the mapping from the label to the combinations of versions for each module, such as $l_1 \mapsto$ {version 1 of Module A, version 2 of Module B}.

For example, in λ_{VL}, the makeHash in Hash can be rewritten as follows.

$$
\begin{aligned}
&\mathit{makeHash} = \\
&\qquad \{l_1 = \lambda\, \mathit{str}.\ \text{(* generate hash based on MD5 *)}, \\
&\qquad\ \ l_2 = \lambda\, \mathit{str}.\ \text{(* generate hash based on SHA-3 *)}\}
\end{aligned}
$$

This *makeHash* is a versioned value, and both versions of `makeHash` definitions are bundled into a version record. In this way, a client program that uses Hash can refer to both program versions by the variable *makeHash*.

Another way to construct a versioned value is by *promotion* $[t]$, which lifts a normal term to a versioned value. For example, $[1]$ is a versioned value with the Int value 1 available for all versions. Terms lifted by promotion do not have version labels, but the type system infers which versions are available.

In λ_{VL}, programs are constructed by function application. To apply a function of a versioned value, such as *makeHash*, the versioned value must be passed as an argument. For function application between versioned values, λ_{VL} provides *contextual let-binding* $\textbf{let}\ [x] = t\ \textbf{in}\ t$. For example, a part of line 8 of the `main` function can be rewritten as follows.

$$
\textbf{let}\ [\mathit{digest}] = [\mathit{makeHash}\ \mathit{str}]\ \textbf{in}\ [\mathit{exists}\ \mathit{digest}] \tag{1}
$$

The program initially binds the variable *digest* to the versioned value constructed by *makeHash str*. Here, the result of the evaluation of *makeHash str* is the versioned value, which is evaluated in the context of multiple versions. That is, *makeHash str* constructs a program with the possibility of computing both the hash value in MD5 corresponding to version 1 (l_1) and the hash value in SHA-3 corresponding to version 2 (l_2). Programs constructed in this way will be evaluated in the context of a particular version by the *extraction* described later.

λ_{VL} preserves all possible versions until a particular version is extracted. The `match` function that `exists` calls (and `makeHash` used inside it) in App is a versioned value available in both versions 1 and 2 and does not depend on a specific version. As a result, *exists* is also a function of a versioned value that can be evaluated in both versions 1 and 2. Finally, the program constructs a computation that can interpret the filename `digest` as a value generated by the hash with MD5 for version 1 and SHA-3 for version 2.

[3] Precisely speaking, the original notation of versioned record $\{\overline{l_i = t_i} \mid l_k\}$ has an additional element called a *default version* l_k, a label that indicates in which version context the term will be evaluated. However, this is not an essential issue in this paper and will therefore be omitted.

2.2.2 Extraction. To extract a specific version of a constructed program, λ_{VL} provides an *extraction* by specifying a label for the versioned value. All evaluation of versioned value can proceed only after an extraction. For example, extracting the program (1) with l_1 and l_2 from the previous program is as follows.

$$\textbf{let } [digest] = [makeHash\ str] \textbf{ in } [exists\ digest].l_1$$
$$\textbf{let } [digest] = [makeHash\ str] \textbf{ in } [exists\ digest].l_2$$

Both programs evaluate to *true*.

$$\begin{aligned} & \textbf{let } [digest] = [makeHash\ str] \textbf{ in } [exists\ digest].l_1 \\ \to^* & \ [exists\ (makeHash\ str)].l_1 \\ \to^* & \ exists\ (\text{* MD5 hash (ver 1) *}) \\ \to & \ true \end{aligned}$$

Note that version 1 of *match* and *makeHash* in *exists* are extracted in the evaluation of the second to third lines.

Even if we use multiple versions of a program within a single program, there will be no confusion between different hash generation algorithms because extraction allows programs with multiple possible versions to be evaluated in the context of one consistent version.

2.3 The Lambda VL Type System

2.3.1 Type of a Versioned Value. The λ_{VL} type system computes the available versions for a program constructed by a function application.

For example, the type system assigns the following type to *makeHash*.

$$\begin{aligned} & makeHash : \Box_{\{l_1,l_2\}}(\mathsf{String} \to \mathsf{String}) \\ & makeHash = \\ & \qquad \{l_1 = \ \lambda\, str.\ (\text{* generate hash based on MD5 *}), \\ & \qquad\ l_2 = \ \lambda\, str.\ (\text{* generate hash based on SHA-3 *})\} \end{aligned}$$

The type of a versioned value is given by a set of version labels for which the value is available, in addition to the usual type. We call such a set of version labels *version resources*. Here, *makeHash* has the type String → String and is available in versions 1 (l_1) and 2 (l_2).

Similarly, the type system assigns the following type to *exists*.

$$\begin{aligned} & exists : \Box_{\{l_1,l_2\}}(\mathsf{String} \to \mathsf{Boolean}) \\ & exists = \{l_1 = \ \lambda\, str.\ \cdots, l_2 = \ \lambda\, str.\ \cdots\} \end{aligned}$$

For the function application, The type system computes an intersection of version resources of all subterms. In this way, the type system knows which labels are consistent with its programs. For example, the type system assigns the following types to the program (1).

$$\begin{aligned} & \textbf{let } [digest] = [makeHash\ str] \\ & \ \textbf{in } [exists\ digest] : \Box_{\{l_1,l_2\}}\mathsf{Bool} \end{aligned} \quad (2)$$

Since *makeHash* and *exists* are available in both l_1 and l_2, all subterms in this program are also available in versions 1 (l_1) and 2 (l_2). Thus, the type system indicates that this program is also available in versions 1 (l_1) and 2 (l_2).

$$\begin{aligned} t ::= \ & \underbrace{x \mid t_1\, t_2 \mid \lambda x.t}_{\lambda\text{-terms}} \mid \underbrace{\ n\ }_{\text{constructors}} \mid \\ & \underbrace{[t] \mid \textbf{let } [x] = t_1 \textbf{ in } t_2}_{\text{coeffect terms}} \mid \\ & \underbrace{\{\overline{l_i = t_i}\} \mid t.l \mid \langle \overline{l = t} \mid l_i \rangle}_{\text{versioned terms}} \end{aligned} \quad \text{(terms)}$$

$$A, B ::= \underbrace{\mathsf{Int}}_{\text{Integer}} \mid \underbrace{A \to B}_{\text{function types}} \mid \underbrace{\Box_r A}_{\text{versioned types}} \quad \text{(types)}$$

Figure 4: The λ_{VL} syntax.

2.3.2 Type of an Extraction. For extraction, the type system determines if the specified label is available. For example, the following program is well-typed because l_1 and l_2 are both available for the program (1) (see typing (2)).

$$\textbf{let } [digest] = [makeHash\ str] \textbf{ in } [exists\ digest].l_1 : \mathsf{Bool}$$
$$\textbf{let } [digest] = [makeHash\ str] \textbf{ in } [exists\ digest].l_2 : \mathsf{Bool}$$

Once evaluated for a particular version by extraction, a program loses its version resource. This is because extraction is defined as a destructor of versioned values.

In contrast, the following program is rejected by the type system.

$$\textbf{let } [digest] = [makeHash\ str] \textbf{ in } [exists\ digest].l_3 : (rejected)$$

This is because *makehash* and *exists* do not have l_3 definitions. Since the λ_{VL} type system keeps track of which versions are available for each variable in the type environment, it is possible to inform the developer why this program cannot be evaluated in the context of l_3.

2.4 Desirable Properties of Surface Language

2.4.1 Problems in Lambda VL Programming. As you can see from the previous examples, programming directly in λ_{VL} is not easy. Difficulties in λ_{VL} programming arise from the following points:

- *Complex syntax derived from substructural language.* λ_{VL} is a language defined as an extension of the coeffect calculus ℓRPCF [3] and GrMini [14], and has a difficult syntax derived from substructural language, as shown in Figure 4. Terms related to version resources, such as promotion $[t]$ and contextual let-bindings $\textbf{let } [x] = t_1 \textbf{ in } t_2$, require programmers to understand the λ_{VL} type system and prevent them from implementing the logic on which the developer wants to focus.
- *Versions exposed to surface language.* As shown in Figure 4, the λ_{VL} program includes versions as part of label-dependent terms such as versioned record $\{\overline{l_i = t_i}\}$ and extraction $t.l$. Version labels are a cross-cutting concern in λ_{VL} for all modules and correspond with each module's version. However, since hundreds of modules are used in actual development, it is difficult for a programmer to know which label corresponds to which definition in each module.

$$t ::= \underbrace{x \mid t_1\, t_2 \mid \lambda x.t}_{\lambda\text{-terms}} \mid \underbrace{n}_{\text{constructors}} \mid \underbrace{[t] \mid \textbf{let}\ [x] = t_1\ \textbf{in}\ t_2}_{\text{coeffect terms}} \quad \text{(terms)}$$

$$A, B ::= \underbrace{\mathsf{Int}}_{\text{Integer}} \mid \underbrace{A \to B}_{\text{function types}} \mid \underbrace{\Box_r A}_{\text{versioned types}} \quad \text{(types)}$$

Figure 5: The GrMini (a subset of λ_{VL}) syntax.

2.4.2 Usual functional language as a surface language for Lambda VL. To mitigate the two difficulties listed above, we propose using an ordinary functional language as a surface language and a method of mechanically translating λ_{VL}-specific terms using externally defined version information.

This proposal is based on the following intuitions. First, since new versions are usually released for each package in an existing programming language, it is feasible to determine which version of the symbol is available. For example, before and after the update, the `Hash` modules in Figures 2 and 3 provide functions `makeHash` and `match`. These programs are distinct, and there is no program belongs to both versions. Therefore, we expect it is possible to assign each version of the definition to a label mechanically .

Furthermore, since the λ_{VL} type system knows the available versions of all programs by semantic analysis, it is also feasible to assign the required labels to extraction. For example, program (2) is indicated to be available with labels l_1 and l_2 by the type system. Although we need to prioritize those two labels, we should be able to specify the appropriate label among these options.

3 COMPILATION

In this section, we provide examples of the compilation from a Haskell-like functional language to λ_{VL}. The compilation consists of two steps:

(1) Compilation into GrMini, a subset of λ_{VL}
(2) Bundling with version labels

Furthermore, we examine a method of compiling them back to Haskell-like functional language based on λ_{VL} semantics.

3.1 Compilation into GrMini

This compilation is based on Girard's translation [9] of simply-typed lambda calculus to intuitionistic linear calculus. Orchard [14] note that any term and type derivation of a simply typed lambda calculus can be compiled to GrMini, as shown in Figure 5.

Girard's translation is the following syntax-directed translations.

Definition 3.1 (Girard's translation). *Let A and B be metavariables over types of simply-typed lambda calculus. The translation from the simply-typed lambda calculus to GrMini is described as follows.*

$$[\![A]\!] \equiv A$$
$$[\![A \to B]\!] \equiv \Box_r [\![A]\!] \to [\![B]\!]$$

$$[\![x]\!] \equiv x$$
$$[\![\lambda x.t]\!] \equiv \lambda x'.\textbf{\textit{let}}\ [x] = x'\ \textbf{\textit{in}}\ [\![t]\!]$$
$$[\![t\ s]\!] \equiv [\![t]\!]\ [[\![s]\!]]$$

All occurrences of $A \to B$ will be replaced with $\Box_r A \to B$ using the suitable version resource $r \in \mathcal{R}$, and replaced all lambda abstractions and function applications by using contextual let-binding and promotion. For each version resource $r \in \mathcal{R}$, the appropriate version resource will be inferred later by the type inference.

To give the readers a better understanding, we will illustrate the translation process using a simple multi-versioned functional program, as shown in Figure 7. Suppose there are two modules called `Main` and `M`. The `main` function defined in the `Main` module is translated to the following GrMini program.

$$main : \mathsf{Int}$$
$$main = id\ [n]$$

Here, the argument n of the function application is promoted.

Similarly, the `id` in `M` version 1 is translated into the GrMini program as follows:

$$id : \Box_r \mathsf{Int} \to \mathsf{Int}$$
$$id = \lambda n'.\textbf{let}\ [n] = n'\ \textbf{in}\ n$$
$$n : \mathsf{Int}$$
$$n = 1$$

where r is a resource variable that will later be instantiated into a set of appropriate version labels by type inference. The same translation can be applied to `M` version 2.

3.2 Bundling with Version Labels

Next, the top-level elements of each module are bundled using a versioned record. At this time, externally defined version information is used as version information.

Here, the versions to be considered are `M` versions 1 and 2, so the following labels are generated for them.

$$\texttt{M version 1} \rightsquigarrow l_1, \qquad \texttt{M version 2} \rightsquigarrow l_2$$

Next, we use these version labels to bundle the top-level elements. For example, the top-level symbols id, n of module M are translated into the following λ_{VL} program.

$$id : \Box_{\{l_1,l_2\}} (\Box_r \mathsf{Int} \to \mathsf{Int})$$
$$id = \{l_1 = \lambda n'.\textbf{let}\ [n] = n'\ \textbf{in}\ n, l_2 = \lambda n'.\textbf{let}\ [n] = n'\ \textbf{in}\ n\}$$
$$n : \Box_{\{l_1,l_2\}} \mathsf{Int}$$
$$n = \{l_1 = 1, l_2 = 2\}$$

The labels are tagged with corresponding version definitions for id and n. A version 1 definition of id and n is stored as an element of the versioned record tagged with l_1; likewise, version 2 is tagged with l_2.

Reduction rules

$$\frac{}{[t].l \rightsquigarrow t@l}\ (\text{E-EX1}) \qquad \frac{}{\{\overline{l = t} \mid m\}.l_i \rightsquigarrow t_i@l_i}\ (\text{E-EX2}) \qquad \frac{}{\langle \overline{l = t} \mid l_i \rangle \rightsquigarrow t_i@l_i}\ (\text{E-VERI})$$

Default version overwriting rules

$$n@l \equiv n \qquad (\lambda x.t)@l \equiv \lambda x.(t@l) \qquad (t\,u)@l \equiv (t@l)\,(u@l)$$

$$\textbf{let}\ [x] = t_1\ \textbf{in}\ t_2@l \equiv \textbf{let}\ [x] = (t_1@l)\ \textbf{in}\ (t_2@l)$$

$$[t]@l \equiv [t@l] \qquad \{\overline{l = t} \mid l_i\}@l' \equiv \{\overline{l = t} \mid l_i\} \qquad (t.l)@l' \equiv (t@l').l$$

$$\frac{l' \in \{\bar{l}\}}{\langle \overline{l = t} \mid l_i \rangle@l' \equiv \langle \overline{l = t} \mid l' \rangle} \qquad \frac{l' \notin \{\bar{l}\}}{\langle \overline{l = t} \mid l_i \rangle@l' \equiv \langle \overline{l = t} \mid l_i \rangle}$$

Figure 6: λ_{VL}dynamic semantics (excerpt)

```
1 -- Main
2 main :: Int
3 main = id n
```

```
1 -- M version 1
2 id :: Int -> Int
3 id n = n
4 n :: Int
5 n = 1
```

```
1 -- M version 2
2 id :: Int -> Int
3 id n = n
4 n :: Int
5 n = 2
```

Figure 7: Simple example program written in Haskell subset.

Instead of translating into a versioned record, the *main* function abstracts the return value with a promotion. The resulting *main* function is as follows.

$$main : \Box_s \text{Int}$$
$$main = [id\ [n]]$$

Here we use the version resource variable s will be inferred later by type inference.

This main function results in a series of concatenated by contextual let-binding, just as a normal program is a series of let-binding with values provided by an external module.

$$\begin{aligned} main &: \Box_{\{l_1,l_2\}} \text{Int} \\ main &= \textbf{let}\ [id'] = id\ \textbf{in}\ \textbf{let}\ [n'] = n\ \textbf{in}\ [id\ [n]] \\ &\equiv \textbf{let}\ [id'] = \{l_1 = \lambda n'.\textbf{let}\ [n] = n'\ \textbf{in}\ n, \\ &\qquad\qquad l_2 = \lambda n'.\textbf{let}\ [n] = n'\ \textbf{in}\ n\}\ \textbf{in} \\ &\quad\ \textbf{let}\ [n'] = \{l_1 = 1, l_2 = 2\}\ \textbf{in}\ [id'\ [n']] \end{aligned}$$

The type system infers the version resources of the *main* function by using those of *id* and *n*. The type of the *main* function indicates that it is available for labels l_1 and l_2.

The extraction of the *main* function with each label evaluates to 1 and 2, respectively.

$$main.l_1 \longrightarrow^* 1$$
$$main.l_1 \longrightarrow^* 2$$

These results are equivalent to those obtained by selecting versions 1 and 2 of module `M` in Figure 7 and evaluating `main`.

Although the developer will need to assign some priority between l_1 and l_2, now it is possible to evaluate a program by choosing from multiple versions, using only normal functional programs as input.

3.3 Dispatch a Specific Version of Programs

Before performing reductions, λ_{VL} removes all versioned records and translates them into intermediate terms called *version-specified records* $\langle \overline{l_i = t_i} \mid l_k \rangle$. An additional element l_k indicates in which version the version-specified record is evaluated. This l_k is overwritten by the propagation from an extraction according to the default version overwriting rule shown in Figure 6. Here we assume that the initial label is the latest version (l_2).

For example, the evaluation of $main.l_1$ is as follows.

$$\begin{aligned} & main.l_1 \\ &\equiv \textbf{let}\ [id'] = \{l_1 = \cdots, l_2 = \cdots \mid l_2\}\ \textbf{in} \\ &\quad\ \textbf{let}\ [n'] = \{l_1 = 1, l_2 = 2 \mid l_2\}\ \textbf{in}\ [id'\ [n']].l_1 \\ &\longrightarrow^* \langle l_1 = \cdots, l_2 = \cdots \mid l_2 \rangle@l_1\ [\langle l_1 = 1, l_2 = 2 \mid l_2 \rangle]@l_1 \\ &\longrightarrow^* \langle l_1 = \cdots, l_2 = \cdots \mid l_1 \rangle\ [\langle l_1 = 1, l_2 = 2 \mid l_1 \rangle] \end{aligned}$$

The two versioned records are both evaluated in the context of l_1, so each internal l_1 definition is extracted.

$$\begin{aligned} & \langle l_1 = \cdots, l_2 = \cdots \mid l_1 \rangle\ [\langle l_1 = 1, l_2 = 2 \mid l_1 \rangle] \\ &\longrightarrow^* (\lambda n'.\textbf{let}\ [n] = n'\ \textbf{in}\ n)\ [1] \end{aligned}$$

Recall that we applied the translation into a versioned record only to the top-level definition. The extraction converts the outermost versioned record to a version-specified record for all subterms. In other words, this program does not have a single λ_{VL}-specific term and is written in the GrMini, which is shown in Figure 5.

We can obtain a normal functional language program by performing a reverse compilation from GrMini to a simply-typed lambda

calculus on this GrMini program. The reverse compilation is similarly based on Girard's translation, but this one is simpler than the forward translation.

$$[\![(\lambda n'.\textbf{let } [n] = n' \textbf{ in } n)\ [1]]\!]^{-1} \equiv (\lambda n.n)\ 1$$

The resulting program is a normal functional program so that we can run it on a runtime of an existing functional language.

4 FURTHER WORK

4.1 Adapting Old Version into New Version

The current λ_{VL} considers all values with different versions incompatible, but we hope it can be improved to allow for a compatible update. In many cases, even if there is a breaking change in a part of the package, most other parts will remain unchanged and compatible. Therefore, it is too conservative to assume that all values from different versions are incompatible.

One possible solution is an automatic insertion of adapter functions. λ_{VL} knows the version of every expression, it knows what should be updated by a package change. Suppose there is a mechanism for the developer to specify an adapter to use the value evaluated in the older version in the newer version. In that case, the adapter can automatically convert the value when a version mismatch is detected. This solution applies to a wider range of programs than existing adaptation techniques such as text substitution.

Another solution is to consider compatibility in the type system. For example, when version 1 and version 2 of function f are the same, a version 2 of f should accept a version 1 value as an argument. This improvement would be supported by extending the typing algorithm to use compatibility information pre-registered by the package developer.

4.2 Reformalization Based on Macros-Dispatch Mechanisms

The translation described in section 3 translates a usual functional program to a record format of λ_{VL} and then again back to the original functional language based on the results of type inference. This transformation is similar to the macro-dispatching mechanisms such as multi-stage programming [20, 21]. The semantics of λ_{VL} can be separated into two stages, generating a program and evaluating the program. It can be improved based on the formalization of multi-stage programming. Similarities with multi-stage programming have also been noted from the perspective of the coeffect calculus [14].

5 RELATED WORK

5.1 Software Product Lines

Several studies for software product lines [2, 15, 18] regard the evolution of a program as an extension of the program. For example, delta-oriented programming provides a mechanism, called a delta module, to modularize program modifications. Developers can combine core modules with delta modules to build software with specific configurations. Since a packaging system with expression-level dependency information is essential to our research, we may be able to refer to such an implementation technique.

5.2 Adaptation Techniques

Adaptation techniques help client code to be connected to a version of a library that is incompatible with the older one. The simplest approach is to compare between the old and new version of a source code and find patterns of substitution. Other approaches generate replacement rules based on structural similarities [5, 24] and extract API replacement patterns from a code base that has been migrated [19].

Another approach lets the library maintainer generate replacement rules. Some techniques [7, 10] require library maintainer to record refactorings made to the source code and generates refactoring scripts for providing it to library users. Another technique [4] requires the library maintainer provides annotations that describes how to update client code. These techniques are reported to provide correct code recommendations on average in only less than 20% of cases [6].

6 CONCLUSION

Lam *et al.* [11] pointed out that the lack of tool support for breaking changes forces developers to pay a great attention to compatibility. Programming with versions brings versions, traditionally package identifiers, into a programming language and allows expressions to be tagged with versions. Versions are still a compatibility agreement from the package provider to the package user. Programming with versions allows better management and analysis of compatibility information in a more granular context.

Our long-term goal is to bring the benefits of programming with versions into developer's familiar programming languages. The proposed compilation technique from an existing functional program to λ_{VL} terms makes one step towards the goal.

ACKNOWLEDGMENTS

We thank the members of the Programming Research Group at the Tokyo Institute of Technology for valuable discussions. Especially, Youyou Cong provided feedback on the earlier versions of the paper. This work was supported by JSPS KAKENHI grant numbers JP18H03219, 20K21790, and 22J14382.

REFERENCES

[1] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache Community Upgrades Dependencies: An Evolutionary Study. *Empirical Software Engineering* 20, 5 (Oct. 2015), 1275–1317. https://doi.org/10.1007/s10664-014-9325-9

[2] Jan Bosch. 2001. Software Product Lines: Organizational Alternatives. In *Proceedings of the 23rd International Conference on Software Engineering* (Toronto, Ontario, Canada) *(ICSE '01)*. IEEE Computer Society, USA, 91–100.

[3] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

[4] Chow and Notkin. 1996. Semi-automatic update of applications in response to library changes. In *1996 Proceedings of International Conference on Software Maintenance*. IEEE, New York, USA, 359–368. https://doi.org/10.1109/ICSM.1996.565039

[5] Bradley Cossette, Robert Walker, and Rylan Cottrell. 2014. Using Structural Generalization to Discover Replacement Functionality for API Evolution. https://doi.org/10.11575/PRISM/10182

[6] 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* (Cary, North Carolina) *(FSE '12)*. Association

for Computing Machinery, New York, NY, USA, Article 55, 11 pages. https://doi.org/10.1145/2393596.2393661
[7] Danny Dig and Ralph Johnson. 2006. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* 18, 2 (2006), 83–107. https://doi.org/10.1002/smr.328 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.328
[8] The Rust Foundation. 2022. Dependency Resolution - The Cargo Book. https://doc.rust-lang.org/cargo/reference/resolver.html
[9] Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50, 1 (Jan. 1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4
[10] J. Henkel and A. Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE, New York, USA, 274–283. https://doi.org/10.1109/ICSE.2005.1553570
[11] Patrick Lam, Jens Dietrich, and David J. Pearce. 2020. *Putting the Semantics into Semantic Versioning.* Association for Computing Machinery, New York, NY, USA, 157–179. https://doi.org/10.1145/3426428.3426922
[12] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *2013 IEEE International Conference on Software Maintenance, ICSM.* IEEE, New York, USA, 70–79. https://doi.org/10.1109/ICSM.2013.18
[13] Inc. npm. 2022. How npm2 Works - How npm Works Docs. https://npm.github.io/how-npm-works-docs/npm2/how-npm2-works.html
[14] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (July 2019), 30 pages. https://doi.org/10.1145/3341714
[15] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the 11th European Conference on Object-Oriented Programming (Lecture Notes in Computer Science, Vol. 1241).* Springer-Verlag, Berlin, Heidelberg, 419–443. https://doi.org/10.1007/BFb0053389
[16] Tom Preston-Werner. 2013. Semantic Versioning 2.0.0. http://semver.org
[17] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2014. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation.* IEEE, New York, USA, 215–224. https://doi.org/10.1109/SCAM.2014.30
[18] 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* (Jeju Island, South Korea) *(SPLC'10).* Springer-Verlag, Berlin, Heidelberg, 77–91.
[19] Thorsten Schäfer, Jan Jonas, and Mira Mezini. 2008. Mining Framework Usage Changes from Instantiation Code. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08).* Association for Computing Machinery, New York, NY, USA, 471–480. https://doi.org/10.1145/1368088.1368153
[20] 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
[21] 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
[22] Yudai Tanabe, Tomoyuki Aotani, and Hidehiko Masuhara. 2018. A Context-Oriented Programming Approach to Dependency Hell. In *Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-time Composition* (Amsterdam, Netherlands) *(COP '18).* ACM, New York, NY, USA, 8–14. https://doi.org/10.1145/3242921.3242923
[23] Yudai Tanabe, Luthfan Anshar Lubis, Tomoyuki Aotani, and Hidehiko Masuhara. 2021. A Functional Programming Language with Versions. *The Art, Science, and Engineering of Programming* 6, 1 (jul 2021), 5:1–5:30. https://doi.org/10.22152/programming-journal.org/2022/6/5
[24] Wei Wu. 2011. Modeling Framework API Evolution as a Multi-objective Optimization Problem. In *2011 IEEE 19th International Conference on Program Comprehension.* IEEE, New York, USA, 262–265. https://doi.org/10.1109/ICPC.2011.43