Compilation Semantics for a Programming Language with Versions

Yudai Tanabe[1(✉)], Luthfan Anshar Lubis[2], Tomoyuki Aotani[3], and Hidehiko Masuhara[2]

[1] Kyoto University, Kyoto, Japan
yudaitnb@fos.kuis.kyoto-u.ac.jp
[2] Tokyo Institute of Technology, Tokyo, Japan
luthfanlubis@prg.is.titech.ac.jp, masuhara@acm.org
[3] Sanyo-Onoda City University, Yamaguchi, Japan
aotani@rs.socu.ac.jp

Abstract. *Programming with versions* is a paradigm that allows a program to use multiple versions of a module so that the programmer can selectively use functions from both older and newer versions of a single module. Previous work formalized λ_{VL}, a core calculus for programming with versions, but it has not been integrated into practical programming languages. In this paper, we propose VL, a Haskell-subset surface language for λ_{VL} along with its compilation method. We formally describe the core part of the VL compiler, which translates from the surface language to the core language by leveraging Girard's translation, soundly infers the consistent version of expressions along with their types, and generates a multi-version interface by bundling specific-version interfaces. We conduct a case study to show how VL supports practical software evolution scenarios and discuss the method's scalability.

Keywords: Type system · Type inference · Version control system

1 Introduction

Updating dependent software packages is one of the major issues in software development. Even though a newer version of a package brings improvements, it also brings the risk of breaking changes, which can make the entire software defective.

We argue that this issue originates from the principle of most programming languages that only allow the use of one version of a package at a time. Due to this principle, developers are faced with the decision to either update to a new, improved version of a package that requires many changes or to remain with an older version. The problem gets worse when a package is indirectly used. This dilemma often results in delays in adopting upgrades, leading to stagnation in software development and maintenance [2,16].

Programming with versions [15,28,29,31] is a recent proposal that allows programming languages to support multiple versions of programming elements

C.-K. Hur (Ed.): APLAS 2023, LNCS 14405, pp. 3–23, 2023.
https://doi.org/10.1007/978-981-99-8311-7_1

at a time so that the developer can flexibly cope with incompatible changes. λ_{VL} is the core calculus in which a *versioned value* encapsulates multiple versions of a value (including a function value). The λ_{VL} type system checks the consistency of each term so that a value produced in a version is always passed to functions in the same version. The calculus and the type system design are based on coeffect calculus [3,20].

While λ_{VL} offers the essential language constructs to support multiple versions in a program, the language is far from practical. For example, with multiple versions of a module, each version of the function must be manually represented inside a versioned value (i.e., a record-like expression). λ_{VL} is as simple as lambda calculus, yet it has a verbose syntax due to the coeffect calculus. In short, there are aspects of versioning in λ_{VL} that a surface language compiler can automate.

We propose the functional language VL as a surface language for λ_{VL} along with its compilation method. In VL, a function name imported from an external module represents a multi-version term, where each occurrence of the function name can reference a different version of the function. The VL compiler translates a program into an intermediate language VLMini, a version-label-free variant of λ_{VL}, determines the version for each name occurrence based on a type and version inference algorithm, and translates it back into a version-specialized Haskell program. VL also offers the constructs to explicitly control versions of expressions, which are useful to keep using an older version for some reason.

This paper presents the following techniques in VL: (a) *an application of Girard's translation* for translating VL into VLMini, (b) *the bundling* for making a top-level function act as a versioned value, and (c) *a type and version inference algorithm* for identifying the version of each expression with respect to the λ_{VL} type system. Finally, we prove the soundness of the inference system and implement a VL compiler. Code generation converts a VL program into a version-specialized Haskell program using the solution obtained from z3 [18].

Paper Organization. Section 2 introduces incompatibility issues and fundamental concepts in programming with versions with λ_{VL} and VL. Section 3 introduces bundling and Girard's transformation. Section 4 presents an algorithmic version inference for VL. Section 5 features an implementation of VL, and Sect. 6 introduces a case study that simulates an incompatible update made in a Haskell library. Finally, Sect. 7 discusses further language development and concludes the paper by presenting related work and a conclusion.

2 Overview

2.1 Motivating Example

First, we will explain a small example to clarify incompatibility issues. Consider a scenario where an incompatible change is made to a dependent package. Figure 1 shows the package dependencies in a file explorer **`App`** based on a hash-based file search. This function is developed using the system library **`Dir`** and the cryptography library **`Hash`**. For simplicity, we equate packages and modules here

```
module App where
import Dir
import Hash
main () =
  let s = getArg ()
      digest = mkHash s in
  if exists digest
    then print "Found"
    else error "Not found"
```

```
module Dir where
import Hash
-- version 1.0.0
exists hash =
  let fs = getFiles () in
  foldLeft
    (\(acc, f) ->
      acc || match f hash)
    false fs
```

version 2.0.0

```
module Hash where
-- version 2.0.0
mkHash s = {- SHA-3 hash -}
match s hash =
    mkHash s == hash
```

version 1.0.0

```
module Hash where
-- version 1.0.0
mkHash s = {- MD5 hash -}
match s hash =
    mkHash s == hash
```

The `exists` provided from **`Dir`** (which depends on version 1 of **`Hash`**) expects an MD5 hash as an argument. However, after the dependency update of **`App`** on **`Hash`**, the value assigned to `digest` is a SHA-3 hash.

Fig. 1. Minimal module configuration before and after the dependency update causing an error due to inconsistency expected to the dependent package.

(each package consists of a single module), and we only focus on the version of **`Hash`**. The pseudocode is written in a Haskell-like language.

Before its update, **`App`** depends on version 1.0.0 of **`Hash`** (denoted by $\dashrightarrow$). The **`App`**'s `main` function implements file search by a string from standard input using `mkHash` and `exists`. The function `mkHash` is in version 1.0.0 of **`Hash`**, and it generates a hash value using the MD5 algorithm from a given string. **`Hash`** also provides a function `match` that determines if the argument string and hash value match under `mkHash`. The function `exists` is in version 1.0.0 of **`Dir`**, which is also dependent on version 1.0.0 of **`Hash`**, and it determines if a file with a name corresponding to a given hash exists.

Due to security issues, the developer of **`App`** updated **`Hash`** to version 2.0.0 (denoted by $\longrightarrow$). In version 2.0.0 of **`Hash`**, SHA-3 is adopted as the new hash algorithm. Since **`Dir`** continues to use version 1.0.0 of **`Hash`**, **`App`** needs two different versions of **`Hash`**. Various circumstances can lead to this situation: **`Dir`** may have already discontinued maintenance, or functions in **`Dir`**, other than `exists`, might still require the features provided by version 1.0.0 of **`Hash`**.

Although the update does not modify **`App`**, it causes errors within **`App`**. Even if a file with an input filename exists, the program returns `Not Found` error contrary to the expected behavior. The cause of the unexpected output lies in the differences between the two versions required for `main`. In line 6 of **`App`**, an SHA-3 hash value is generated by `mkHash` and assigned to `digest`. Since `exists` evaluates hash equivalence using MD5, `exists digest` compares hashes generated by different algorithms, evaluating to `false`.

This example highlights the importance of version compatibility when dealing with functions provided by external packages. Using different versions of **`Hash`** in separate program parts is fine, but comparing results may be semantically incorrect. Even more subtle changes than those shown in Fig. 1 can lead to significant errors, especially when introducing side effects or algorithm modifications that break the application's implicit assumptions. Manually managing version compatibility for all external functions is unfeasible.

In practical programming languages, dependency analysis is performed before the build process to prevent such errors, and package configurations requiring multiple versions of the same package are rejected. However, this approach tends towards conservative error reporting. In cases where a core package, which many other libraries depend on, receives an incompatible change, no matter how minuscule, it requires coordinated updates of diverse packages across the entire package ecosystem [2,29,32].

2.2 λ_{VL}

λ_{VL} [28,29] is a core calculus designed to follow the principles: (1) enabling simultaneous usage of multiple versions of a package, (2) ensuring version consistency within a program. λ_{VL} works by encapsulating relevant terms across multiple versions into a record-like term, tagged with a label indicating the specific module version. Record-like terms accessible to any of its several versions are referred to as *versioned values*, and the associated labels are called *version labels*.

Version Labels. Figure 2 shows the syntax of λ_{VL}. Given modules and their versions, the corresponding set of version labels characterizes the variation of programs of a versioned value. In λ_{VL}, version labels are implicitly generated for all external module-version combinations, in which M_i is unique, with the universal set of these labels denoted by $\mathcal{L}$. Specifically, in the example illustared in Fig. 1, $\mathcal{L} = \{l_1, l_2\}$ and $l_1 = \{$**`Hash`** $= 1.0.0$, **`Dir`** $= 1.0.0\}, l_2 = \{$**`Hash`** $=$ $2.0.0$, **`Dir`** $= 1.0.0\}$. The size of $\mathcal{L}$ is proportional to V^M where M is the number of modules and V is the maximum number of versions.

Syntax of $\boldsymbol{\lambda_{\mathrm{VL}}}$. λ_{VL} extends $\ell\mathcal{R}$PCF [3] and GrMini [20] with additional terms that facilitate introducing and eliminating versioned values. Versioned values can be introduced through versioned records $\{\overline{l_i = t_i}\}$ and promotions $[t]$. A versioned record encapsulates related definitions $t_1, \ldots, t_n$ across multiple versions

λ_{VL} **syntax**

$$t ::= n \mid x \mid t_1\, t_2 \mid \lambda x.t \mid \textbf{let}\ [x] = t_1\ \textbf{in}\ t_2 \mid u.l \mid \langle \overline{l_i = t_i} \mid l_k \rangle \mid u \qquad \text{(terms)}$$

$$u ::= [t] \mid \{\overline{l_i = t_i}\} \quad \text{(versioned values)} \qquad r ::= \bot \mid \{\,\overline{l_i}\,\} \quad \text{(resources)}$$

$$A, B ::= \mathsf{Int} \mid A \rightarrow B \mid \square_r A \quad \text{(types)} \qquad \mathcal{L} \ni l ::= \{\overline{M_i = V_i}\} \quad \text{(version labels)}$$

M_i and V_i are metavariables over module names and versions of M_i, respectively.

Fig. 2. The syntax of λ_{VL}.

and their version labels $l_1, \ldots, l_n$. For instance, the two versions of `mkHash` in Fig. 1 can be bundled as the following version record.

$$\mathit{mkHash} \quad := \quad \begin{array}{l} \{l_1 = \lambda s.\texttt{\{- make MD5 hash -\}}, \\ \ l_2 = \lambda s.\texttt{\{- make SHA-3 hash -\}}\} \end{array}$$

In λ_{VL}, programs are constructed via function application of versioned values. A function application of `mkHash` to the string `s` can be written as follows.

$$\mathit{app} \quad := \quad \begin{array}{l} \textbf{let}\ [\mathit{mkHash}'] = \mathit{mkHash}\ \textbf{in} \\ \textbf{let}\ [s] = [\text{“}\texttt{compiler.vl}\text{”}]\ \textbf{in}\ [\mathit{mkHash}'\ s] \end{array}$$

This program (*app* hereafter) makes a hash for the string “`compiler.vl`" and is available for both l_1 and l_2. The contextual let-binding $\textbf{let}\ [x] = t_1\ \textbf{in}\ t_2$ provides the elimination of version values by binding a versioned value for t_1 to x, thus making it accessible in t_2. Promotion $[x]$ offers an alternative way to introduce versioned values, making any term t act as a versioned value.

The evaluation of terms t_i stored in a versioned value $\{\overline{l_i = t_i}\}$ and $[t]$ is postponed until a specific version label is later specified. To proceed with a postponed evaluation of a versioned value, we use extraction $u.l_k$. Extraction specifies one versioned label l_k for the versioned value u and recursively extracts the inner term t_k corresponding to l_k from $\{l_i = t_i\}$, and t from $[t]$ as follows.

$$\begin{array}{rcl} \mathit{app}\#l_1 & := & \textbf{let}\ [\mathit{mkHash}'] = \mathit{mkHash}\ \textbf{in} \\ & & \textbf{let}\ [s] = [\text{“}\texttt{compiler.vl}\text{”}]\ \textbf{in}\ [\mathit{mkHash}'\ s].l_1 \\ & \longrightarrow^* & (\lambda s.\texttt{\{- make MD5 hash -\}})\ \text{“}\texttt{compiler.vl}\text{”} \\ & \longrightarrow & \texttt{4dcb6ebe3c6520d1f57c906541cf3823} \end{array}$$

Consequently, $\mathit{app}\#l_1$ evaluates into an MD5 hash corresponding to l_1.

Type of Versioned Values. The type of a versioned value is expressed as $\square_r A$, assigning a set of version labels r, called *version resources*, to a type A. Intuitively, the type of a versioned value represents the versions available to that versioned value. For example, *mkHash* and *app* are typed as follows.

$$\mathit{mkHash} : \square_{\{l_1, l_2\}}\,(\mathsf{String} \rightarrow \mathsf{String}) \qquad \mathit{app} : \square_{\{l_1, l_2\}}\,(\mathsf{String} \rightarrow \mathsf{String})$$

```
module App where
import Dir
import Hash
main () =
  let s = getArg ()
      digest = mkHash s in
  if exists digest
    then print "Found"
    else error "Not found"
```

Bundled **Dir**

```
module Dir where
-- version 1.0.0
import Hash
exists hash =
  let fs = getFiles () in
  foldLeft
    (\(acc, f) ->
      acc || match f hash)
    false fs
```

Bundled **Hash**

```
module Hash where
-- version 2.0.0
mkHash s = {- SHA-3 hash -}
match s hash =
    mkHash s == hash
```

```
module Hash where
-- version 1.0.0
mkHash s = {- MD5 hash -}
match s hash =
    mkHash s == hash
```

The versions of each external module are bundled. Programs using a bundled module can refer to the definitions of all versions of the bundled module.

Fig. 3. The programs in Fig. 1 in VL.

The types have $\{l_1, l_2\}$ as their version resource, illustrating that the versioned values have definitions of l_1 and l_2. For function application, the type system computes the intersection of the version resource of subterms. Since the promoted term is considered to be available in all versions, the version resource of the entire function application indicates $\{l_1, l_2\} = \{l_1, l_2\} \cap \mathcal{L}$.

For extractions, the type system verifies if the version resource contains the specified version as follows.

$$app\#l_1 \;:\; \mathsf{String} \to \mathsf{String} \qquad app\#l_3 \;:\; (\textit{rejected})$$

Assuming $\mathcal{L} = \{l_1, l_2, l_3\}$, $app\#l_3$ is rejected by type checking because the version resource of *app* does not contain l_3. Conversely, $app\#l_1$ is well-typed, but note that the resultant type lost its version resource. It is attributed to the design principle that it could be used in other versions upon extraction.

The λ_{VL} type system incorporates the notion of version consistency in addition to the standard notions of preservation and progress. Proofs of these theorems can be found in Appendix C [30].

2.3 Programming with Versions in VL

Our contributions enjoy the benefits of programming with versions on a λ-calculus-based functional language VL. To achieve this, we develop a compilation method between lambda calculus and VLMini, a version-label free variant

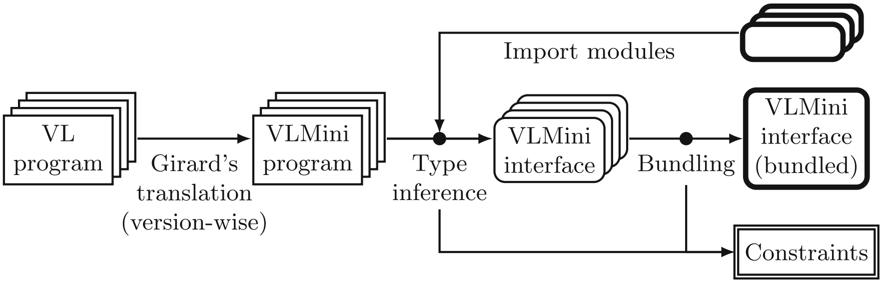

Fig. 4. The translation phases for a single module with multiple versions.

of λ_{VL}, and a version inference algorithm to infer the appropriate version of expressions.

In VL, (1) all versions are available for every module, and (2) the version of each expression is determined by expression-level dependency analysis. This approach differs from existing languages that determine one version for each dependent package. Figure 3 shows how the programs in Fig. 1 are interpreted in VL. The VL compiler bundles the interfaces of multiple versions and generates a cross-version interface to make external functions available in multiple versions. The VL type system enforces version consistency in `main` and selects a newer version if multiple versions are available. Thus it gives the version label $\{$**`Hash`** $= 2.0.0$, **`Dir`** $= 1.0.0\}$ to dependent expressions of `main`. As a result, since **`Hash`** version referenced from **`Dir`** is no longer limited to 1.0.0, `exists digest` is evaluated using SHA-3 under the context of **`Hash`** version 2.0.0.

Furthermore, VL provides *version control terms* to convey the programmer's intentions of versions to the compiler. For example, to enforce the evaluation in Fig. 3 to MD5, a programmer can rewrite line 7 of **`App`** as follows.

```
7  if ver [Hash=1.0.0] of (exists digest)
```

The program dictates that `exists digest` is evaluated within the context of the **`Hash`** version 1.0.0. Consequently, both `mkHash` and `match`, which depend on `exists digest`, are chosen to align with version 1.0.0 of **`Hash`**. Moreover, VL provides **`unversion`** `t`. It eliminates the dependencies associated with term `t`, facilitating its collaboration with other versions under the programmer's responsibility, all while maintaining version consistency within its subterm. Thus, VL not only ensures version consistency but also offers the flexibility to control the version of a particular part of the program.

3 Compilation

The entire translation consists of three parts: (1) *Girard's translation*, (2) an *algorithmic type inference*, and (3) *bundling*. Figure 4 shows the translation process of a single module. First, through Girard's translation, each version of the VL program undergoes a version-wise translation into the VLMini program. Second, the type inference synthesizes types and constraints for top-level symbols. Variables imported from external modules reference the bundled interface generated in the subsequent step. Finally, to make the external variables act as multi-version expressions, bundling consolidates each version's interface into one VLMini interface. These translations are carried out in order from downstream of the dependency tree. By resolving all constraints up to the main module, the appropriate version for every external variable is determined.

It is essential to note that the translations focus on generating constraints for dispatching external variables into version-specific code. While implementing versioned records in λ_{VL} presents challenges, such as handling many version labels and their code clones, our method is a constraint-based approach in VLMini that enables static inference of version labels without their explicit declaration.

In the context of coeffect languages, constraint generation in VL can be seen as the automatic generation of type declarations paired with resource constraints. Granule [20] can handle various resources as coeffects, but it requires type declarations to indicate resource constraints. VL restricts its resources solely to the version label set. This specialization enables the automatic collection of version information from external sources outside the codebase.

3.1 An Intermediate Language, VLMini

Syntax of VLMini. Figure 5 shows the syntax of VLMini. VLMini encompasses all the terms in λ_{VL} except for versioned records $\{\overline{l_i = t_i}\}$, intermediate term $\langle \overline{l_i = t_i} \mid l_k \rangle$, and extractions $t.l_k$. As a result, its terms are analogous to those in $\ell\mathcal{R}$PCF [3] and GrMini [20]. However, VLMini is specialized to treat version resources as coeffects. We also introduce data constructors by introduction $C\, t_1, ..., t_n$ and elimination $\mathbf{case}\ t\ \mathbf{of}\ \overline{p_i \mapsto t_i}$ for lists and pairs, and version control terms $\mathbf{unversion}\ t$ and $\mathbf{version}\ \{\overline{M_i = V_i}\}\ \mathbf{of}\ t$. Here, contextual-let in λ_{VL} is a syntax sugar of lambda abstraction applied to a promoted pattern.

$$\mathbf{let}\ [x] = t_1\ \mathbf{in}\ t_2 \triangleq (\lambda[x].t_2)\, t_1$$

Types, version labels, and version resources are almost the same as λ_{VL}. Type constructors are also added to the type in response to the VLMini term having a data constructor. The remaining difference from λ_{VL} is type variables α. Since VLMini is a monomorphic language, type variables act as unification variables; type variables are introduced during the type inference and are expected to be either concrete types or a set of version labels as a result of constraint resolution. To distinguish those two kinds of type variables, we introduce kinds κ. The kind Labels is given to type variables that can take a set of labels $\{\overline{l_i}\}$ and is used to distinguish them from those of kind Type during algorithmic type inference.

Constraints. The lower part of Fig. 5 shows constraints generated through bundling and type inference. Dependency constraints comprise *variable dependencies* and *label dependencies* in addition to propositional formulae. Variable

VLMini syntax (w/o data constructors and version control terms)

$$
\begin{array}{llll}
t ::= n \mid x \mid t_1\, t_2 \mid \lambda p.t \mid [t] & \text{(terms)} & \Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]_r & \text{(contexts)} \\
p ::= x \mid [x] & \text{(patterns)} & \Sigma ::= \emptyset \mid \Sigma, \alpha : \kappa & \text{(type variable kinds)} \\
A, B ::= \mathsf{Int} \mid \alpha \mid A \rightarrow B \mid \Box_r A & \text{(types)} & R ::= - \mid r & \text{(resource contexts)} \\
\kappa ::= \mathsf{Type} \mid \mathsf{Labels} & \text{(kinds)} & &
\end{array}
$$

Extended with data constructors

$$
\begin{array}{llll}
t ::= \ldots \mid C\,\overline{t_i} \mid \textbf{case } t \textbf{ of } \overline{p_i \mapsto t_i} & \text{(terms)} & A, B ::= \ldots \mid K\,\overline{A_i} & \text{(types)} \\
p ::= \ldots \mid C\,\overline{p_i} & \text{(patterns)} & K ::= (,) \mid [,] & \text{(type constructors)} \\
C ::= (,) \mid [,] & \text{(constructors)} & &
\end{array}
$$

Extended with version control terms

$$
t ::= \ldots \mid \textbf{version } \{\overline{M_i = V_i}\} \textbf{ of } t \mid \textbf{unversion } t \qquad \text{(terms)}
$$

VLMini constraints

$$
\mathcal{C} ::= \underbrace{\top \mid \mathcal{C}_1 \wedge \mathcal{C}_2 \mid \mathcal{C}_1 \vee \mathcal{C}_2}_{\text{propositional formulae}} \mid \underbrace{\alpha \preceq \alpha'}_{\text{variable dependencies}} \mid \underbrace{\alpha \preceq \mathcal{D}}_{\text{label dependencies}} \qquad \text{(dependency constraints)}
$$

$$
\mathcal{D} ::= \langle\!\langle \overline{M_i = V_i} \rangle\!\rangle \qquad \text{(dependent labels)}
$$

$$
\Theta ::= \top \mid \Theta_1 \wedge \Theta_2 \mid \{A \sim B\} \qquad \text{(type constraints)}
$$

Fig. 5. The syntax of VLMini.

dependencies $\alpha \sqsubseteq \alpha'$ require that if a version label for α' expects a specific version for a module, then α also expects the same version. Similarly, label dependencies $\alpha \preceq \langle\!\langle \overline{M_i = V_i} \rangle\!\rangle$ require that a version label expected for α must be V_i for M_i. For example, assuming that versions 1.0.0 and 2.0.0 exist for both modules **A** and **B**, the minimal upper bound set of version labels satisfying $\alpha \preceq \langle\!\langle \mathbf{A} \mapsto 1.0.0 \rangle\!\rangle$ is $\alpha = \{\{\mathbf{A} = 1.0.0, \mathbf{B} = 1.0.0\}, \{\mathbf{A} = 1.0.0, \mathbf{B} = 2.0.0\}\}$. If the constraint resolution is successful, α will be specialized with either of two labels. Θ is a set of type equations resolved by the type unification.

3.2 Girard's Translation for VLMini

We extend Girard's translation between VL (lambda calculus) to VLMini following Orchard's approach [20].

$$
[\![n]\!] \equiv n \qquad [\![x]\!] \equiv x \qquad [\![\lambda x.t]\!] \equiv \lambda[x].[\![t]\!] \qquad [\![t\ s]\!] \equiv [\![t]\!]\ [[\![s]\!]]
$$

The translation replaces lambda abstractions and function applications of VL by lambda abstraction with promoted pattern and promotion of VLMini, respectively. From the aspect of types, this translation replaces all occurrences of $A \rightarrow B$ with $\Box_r A \rightarrow B$ with a version resource r. This translation inserts a syntactic annotation $[*]$ at each location where a version resource needs to be addressed. Subsequent type inference will compute the resource at the specified location and produce constraints to ensure version consistency at that point.

The original Girard's translation [11] is well-known as a translation between the simply-typed λ-calculus and an intuitionistic linear calculus. The approach involves replacing every intuitionistic arrow $A \to B$ with $!A \multimap B$, and subsequently unboxing via let-in abstraction and promoting during application [20].

3.3 Bundling

Bundling produces an interface encompassing types and versions from every module version, allowing top-level symbols to act as multi-version expressions. During this process, bundling reviews interfaces from across module versions, identifies symbols with the same names and types after removing $\square_r$ using Girard's transformation, and treats them as multiple versions of a singular symbol (also discussed in Sect. 7). In a constraint-based approach, bundling integrates label dependencies derived from module versions, ensuring they align with the version information in the typing rule for versioned records of λ_{VL}.

For example, assuming that the id that takes an Int value as an argument is available in version 1.0.0 and 2.0.0 of **M** as follows:

$$id : \square_{\alpha_1}(\square_{\alpha_2}\mathsf{Int} \to \mathsf{Int}) \mid \mathcal{C}_1 \qquad \text{(version 1.0.0)}$$

$$id : \square_{\beta_1}(\square_{\beta_2}\mathsf{Int} \to \mathsf{Int}) \mid \mathcal{C}_2 \qquad \text{(version 2.0.0)}$$

where α_1 and α_2 are version resource variables given from type inference. They capture the version resources of id and its argument value in version 1.0.0. $\mathcal{C}_1$ is the constraints that resource variables of version 1.0.0 will satisfy. Likewise for β_1, β_2, and $\mathcal{C}_2$. Since the types of id in both versions become $\mathsf{Int} \to \mathsf{Int}$ via Girard's translation, they can be bundled as follows:

$$\begin{aligned} id : \square_{\gamma_1}(\square_{\gamma_2}\mathsf{Int} \to \mathsf{Int}) \mid \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \Big(&(\gamma_1 \preceq \langle\!\langle \mathtt{M} = 1.0.0 \rangle\!\rangle \wedge \gamma_1 \preceq \alpha_1 \wedge \gamma_2 \preceq \alpha_2) \\ &\vee (\gamma_1 \preceq \langle\!\langle \mathtt{M} = 2.0.0 \rangle\!\rangle \wedge \gamma_1 \preceq \beta_1 \wedge \gamma_2 \preceq \beta_2) \Big) \end{aligned}$$

where γ_1 and γ_2 are introduced by this conversion for the bundled id interface, with label and variable dependencies that they will satisfy. γ_1 captures the version resource of the bundled id. The generated label dependencies $\gamma_1 \preceq \langle\!\langle \mathtt{M} = 1.0.0 \rangle\!\rangle$ and $\gamma_1 \preceq \langle\!\langle \mathtt{M} = 2.0.0 \rangle\!\rangle$ indicate that id is available in either version 1.0.0 or 2.0.0 of **M**. These label dependencies are exclusively[1] generated during bundling. The other new variable dependencies indicate that the id bundled interface depends on one of the two version interfaces. The dependency is made apparent by pairing the new resource variables with their respective version resource variable for each version. These constraints are retained globally, and the type definition of the bundled interface is used for type-checking modules importing id.

[1] In the type checking rules for **version** l **of** t, type inference exceptionally generates label dependencies. Please see Appendix B.4 [30].

VLMini pattern type synthesis $\boxed{\Sigma, R \vdash p : A \rhd \Gamma; \Sigma'; \Theta; \mathcal{C}}$

$$\frac{}{\Sigma; - \vdash x : A \rhd x : A; \Sigma; \top; \top}\ (\text{pVar}) \qquad \frac{}{\Sigma; r \vdash x : A \rhd x : [A]_r; \Sigma; \top; \top}\ ([\text{pVar}])$$

$$\frac{\Sigma, \alpha : \mathsf{Labels}, \beta : \mathsf{Type}; \alpha \vdash x : \beta \rhd \Delta; \Sigma'; \Theta; \mathcal{C}}{\Sigma; - \vdash [x] : A \rhd \Delta; \Sigma'; \Theta \wedge \{A \sim \Box_\alpha \beta\}; \mathcal{C}}\ (\text{p}\Box)$$

VLMini type synthesis (excerpt) $\boxed{\Sigma; \Gamma \vdash t \Rightarrow A; \Sigma'; \Theta; \mathcal{C}}$

$$\frac{x : A \in \Gamma}{\Sigma; \Gamma \vdash x \Rightarrow A; \Sigma; x : A; \top; \top}\ (\Rightarrow_{\text{LIN}}) \qquad \frac{x : [A]_r \in \Gamma}{\Sigma; \Gamma \vdash x \Rightarrow A; \Sigma; x : [A]_1; \top; \top}\ (\Rightarrow_{\text{GR}})$$

$$\frac{\Sigma_1, \alpha : \mathsf{Type}; - \vdash p : \alpha \rhd \Gamma'; \Sigma_2; \Theta_1 \quad \Sigma_2; \Gamma, \Gamma' \vdash t \Rightarrow B; \Sigma_3; \Delta; \Theta_2; \mathcal{C}}{\Sigma_1; \Gamma \vdash \lambda p.t \Rightarrow \alpha \to B; \Sigma_3; \Delta \backslash \Gamma'; \Theta_1 \wedge \Theta_2; \mathcal{C}}\ (\Rightarrow_{\text{ABS}})$$

$$\frac{\begin{array}{c}\Sigma_1 \vdash [\Gamma \cap \mathsf{FV}(t)]_{\mathsf{Labels}} \rhd \Gamma' \quad \Sigma_1; \Gamma' \vdash t \Rightarrow A; \Sigma_2; \Delta; \Theta; \mathcal{C}_1 \\ \Sigma_3 = \Sigma_2, \alpha : \mathsf{Labels} \quad \Sigma_3 \vdash \alpha \sqsubseteq_c \Gamma' \rhd \mathcal{C}_2\end{array}}{\Sigma_1; \Gamma \vdash [t] \Rightarrow \Box_\alpha A; \Sigma_3; \alpha \cdot \Delta; \Theta; \mathcal{C}_1 \wedge \mathcal{C}_2}\ (\Rightarrow_{\text{PR}})$$

VLMini constraints generation $\boxed{\Sigma \vdash \alpha \sqsubseteq_c \Gamma \rhd \mathcal{C}}$

$$\frac{}{\Sigma \vdash \alpha \sqsubseteq_c \emptyset \rhd \top}\ (\emptyset) \qquad \frac{\Sigma \vdash \alpha \sqsubseteq_c \Gamma \rhd \mathcal{C}}{\Sigma \vdash \alpha \sqsubseteq_c (x : [A]_r, \Gamma) \rhd (\alpha \preceq r \wedge \mathcal{C})}\ (\alpha)$$

Fig. 6. VLMini algorithmic typing.

4 Algorithmic Type Inference

We develop the algorithmic type inference for VLMini derived from the declarative type system of λ_{VL} [28,29]. The type inference consists of two judgments: *type synthesis* and *pattern type synthesis*. The judgment forms are similar to Gr [20], which is similarly based on coeffect calculus. While Gr provides type-checking rules in a bidirectional approach [8,9] to describe resource constraint annotations and performs unifications inside the type inference, VLMini only provides synthesis rules and unification performs after the type inference. In addition, Gr supports user-defined data types and multiple computational resources, while VLMini supports only built-in data structures and specializes in version resources. The inference system is developed to be sound for declarative typing in λ_{VL}, with the proof detailed in Appendix D [30]. Type synthesis takes type variable kinds Σ, a typing context Γ of term variables, and a term t as inputs. Type variable kinds Σ are added to account for distinct unification variables for types and version resources. The synthesis produces as outputs a type A, type variable kinds Σ', type constraints Θ, and dependency constraints $\mathcal{C}$. The type variable kinds Σ and Σ' always satisfy $\Sigma \subseteq \Sigma'$ due to the additional type variables added in this phase.

Pattern type synthesis takes a pattern p, type variable kinds Σ, and resource environment R as inputs. It synthesizes outputs, including typing context Γ, type variable kinds Σ', and type and dependency constraints Θ and $\mathcal{C}$. Pattern type synthesis appears in the inference rules for λ-abstractions and case expressions. It generates a typing context from the input pattern p for typing λ-bodies and branch expressions in case statements. When checking a nested promoted pattern, the resource context R captures version resources inside a pattern.

4.1 Pattern Type Synthesis

Pattern type synthesis conveys the version resources captured by promoted patterns to the output typing context. The rules are classified into two categories, whether or not it has resources in the input resource context R. The base rules are pVar, p$\square$, while the other rules are resource-aware versions of the corresponding rules. The resource-aware rules assume they are triggered within the promoted pattern and collect version resource r in the resource context.

The rules for variables pVar and [pVar] differ in whether the variable pattern occurs within a promoted pattern. pVar has no resources in the resource context because the original pattern is not inside a promoted pattern. Therefore, this pattern produces typing context $x : A$. [pVar] is for a variable pattern within the promoted pattern, and a resource r is recorded in the resource context. The rule assigns the collected resource r to the type A and outputs it as a versioned assumption $x : [A]_r$.

The rules for promoted patterns p$\square$ propagate version resources to the subpattern synthesis. The input type A is expected to be a versioned type, so the rule generates the fresh type variables α and β, then performs the subpattern synthesis considering A as $\square_\alpha\beta$. Here, the resource α captured by the promoted pattern is recorded in the resource context. Finally, the rule unifies A and $\square_\alpha\beta$ and produces the type constraints Θ' for type refinement.

4.2 Type Synthesis

The algorithmic typing rules for VLMini, derived from declarative typing rules for λ_{VL}, are listed in Fig. 6. We explain a few important rules in excerpts.

The rule $\Rightarrow_{\text{ABS}}$ generates a type variable α, along with the binding pattern p of the λ-abstraction generating the typing context Γ'. Then the rule synthesizes a type B for the λ-body under Γ', and the resulting type of the λ-abstraction is $\alpha \to B$ with the tentatively generated α. With the syntax sugar, the type rules of the contextual-let are integrated into $\Rightarrow_{\text{ABS}}$. Instead, λ-abstraction does not just bind a single variable but is generalized to pattern matching, which leverages pattern typing, as extended by promoted patterns and data constructors.

The rule $\Rightarrow_{\text{PR}}$ is the only rule that introduces constraints in the entire type inference algorithm. This rule intuitively infers consistent version resources for the typing context Γ. Since we implicitly allow for weakening, we generate a constraint from Γ' that contains only the free variables in t, produced by *context grading* denoted as $[\Gamma]_{\mathsf{Labels}}$. Context grading converts all assumptions in the input environment into versioned assumptions by assigning the empty set for the assumption with no version resource.

Finally, the rule generates constraints from Γ' and a fresh type variable α by constraints generation defined in the lower part of Fig. 6. The rules assert that the input type variable α is a subset of all the resources of the versioned assumptions in the input environment Γ. The following judgment is the simplest example triggered by the type synthesis of $[f\, x]$.

$$r : \mathsf{Labels}, s : \mathsf{Labels} \vdash \alpha \sqsubseteq_c f : [\mathsf{Int} \to \mathsf{Int}]_r, x : [\mathsf{Int}]_s \rhd \alpha \preceq r \wedge \alpha \preceq s$$

The inputs are type variable α and the type environment $(f : [\mathsf{Int} \to \mathsf{Int}]_r, x : [\mathsf{Int}]_s)$. In this case, the rules generate variable dependencies for r and s, each resource of the assumptions, and return a constraint combined with $\wedge$.

4.3 Extensions

Version Control Terms. The rule for **version** l **of** t uses the same trick as $(\Rightarrow_{\mathrm{PR}})$, and generates label dependencies from the input environment Γ to $\langle\langle l \rangle\rangle$. Since **version** l **of** t only instructs the type inference system, the resulting type is the same as t. **unversion** t removes the version resource from the type of t, which is assumed to be a versioned value. We extend Girard's translation so that t is always a versioned value. Since a new resource variable is given to the term by the promotion outside of **unversion**, the inference system guarantees the version consistency inside and outside the boundary of **unversion**. The list of the rules is provided in Appendix B.4 [30].

Data Structures. To support data structures, Hughes et al. suggest that coeffectful data types are required to consider the interaction between the resources inside and outside the constructor [13]. They introduce the derivation algorithm for *push* and *pull* for an arbitrary type constructor K to address this.

```
push : ∀{a b: Type, r: Labels}. (a,b)[r] -> (a[r],b[r])
push [(x, y)] = ([x], [y])
pull : ∀{a b: Type, m n: Labels}. (a[n],b[m]) -> (a,b)[n⊓m]
pull ([x], [y]) = [(x, y)]
```

Following their approach, we developed inference rules for pairs and lists. When a data structure value p is applied to a function f, the function application $f\, p$ is implicitly interpreted as $f\ (pull\, p)$. As a dual, a pattern match for a data structure value **case** p **of** $\overline{p_i \mapsto t_i}$ is interpreted as **case** $(push\, p)$ **of** $\overline{p_i \mapsto t_i}$. Appendix B.5 [30] provides the complete set of extended rules.

5 Implementation

We implement the VL compiler[2] on GHC (v9.2.4) with haskell-src-exts[3] as its parser with an extension of versioned control terms, and z3 [18] as its constraint

[2] https://github.com/yudaitnb/vl.

[3] https://hackage.haskell.org/package/haskell-src-exts.

Table 1. Availability of functions in hmatrix before and after tha update.

version	`join`	`vjoin`	`udot`, `sortVector`, `roundVector`
< 0.15	available	undefined	undefined
≥ 0.16	deleted	available	available

solver. The VL compiler performs the code generation by compiling VLMini programs back into λ-calculus via Girard's translation and then translating them into Haskell ASTs using the version in the result version labels.

Ad-hoc Version Polymorphism via Duplication. The VL compiler replicates external variables to assign individual versions to homonymous external variables. Duplication is performed before type checking of individual versions and renames every external variable along with the type and constraint environments generated from the import declarations. Such ad hoc conversions are necessary because VLMini is monomorphic, and the type inference of VLMini generates constraints by referring only to the variable's name in the type environment. Therefore, assigning different versions to homonymous variables requires manual renaming in the preliminary step of the type inference. A further discussion on version polymorphism can be found in Sect. 7.

Constraints Solving with z3. We use sbv[4] as the binding of z3. The sbv library internally converts constraints into SMT-LIB2 scripts [1] and supplies it to z3. Dependency constraints are represented as vectors of symbolic integers, where the length of the vector equals the number of external modules, and the elements are unique integers signifying each module's version number. Constraint resolution identifies the expected vectors for symbolic variables, corresponding to the label on which external identifiers in VL should depend. If more than one label satisfies the constraints, the default action is to select a newer one.

6 Case Study and Evaluation

6.1 Case Study

We demonstrate that VL programming achieves the two benefits of programming with versions. The case study simulated the incompatibility of hmatrix,[5] a popular Haskell library for numeric linear algebra and matrix computations, in the VL module **`Matrix`**. This simulation involved updating the applications **`Main`** depending on **`Matrix`** to reflect incompatible changes.

Table 1 shows the changes introduced in version 0.16 of hmatrix. Before version 0.15, hmatrix provided a `join` function for concatenating multiple vectors.

[4] https://hackage.haskell.org/package/sbv-9.0.

[5] https://github.com/haskell-numerics/hmatrix/blob/master/packages/base/CHANGELOG.

```
module Main where
import Matrix
import List
main = let
  vec = [2, 1]
  sorted = sortVector vec
  m22 = join -- [[1,2],[2,1]]
          (singleton sorted)
          (singleton vec)
  in determinant m22
-- error: version inconsistent
```

```
module Main where
import Matrix
import List
main = let
  vec = [2, 1]
  sorted = unversion
              (sortVector vec)
  m22 = join -- [[1,2],[2,1]]
          (singleton sorted)
          (singleton vec)
  in determinant m22 -- ->⋆ -3
```

Fig. 7. Snippets of `Main` before (left) and after (right) rewriting.

The update from version 0.15 to 0.16 replaced `join` with `vjoin`. Moreover, several new functions were introduced. We implement two versions of **`Matrix`** to simulate backward incompatible changes in VL. Also, due to the absence of user-defined types in VL, we represent `Vector a` and `Matrix a` as `List Int` and `List (List Int)` respectively, using **`List`**, a partial port of `Data.List` from the Haskell standard library.

We implement **`Main`** working with two conflicting versions of **`Matrix`**. The left side of Fig. 7 shows a snippet of **`Main`** in the process of updating **`Matrix`** from version 0.15.0 to 0.16.0. `main` uses functions from both versions of **`Matrix`** together: `join` and `sortVector` are available only in version 0.15.0 and 0.16.0 respectively, hence **`Main`** has conflicting dependencies on both versions of **`Matrix`**. Therefore, it will be impossible to successfully build this program in existing languages unless the developer gives up using either `join` or `sortVector`.

- **Detecting Inconsistent Version**: VL can accept **`Main`** in two stages. First, the compiler flags a version inconsistency error. It is unclear which **`Matrix`** version the `main` function depends on as `join` requires version 0.15.0 while `sortVector` requires version 0.16.0. The error prevents using such incompatible version combinations, which are not allowed in a single expression.
- **Simultaneous Use of Multiple Versions**: In this case, using `join` and `sortVector` simultaneously is acceptable, as their return values are vectors and matrices. Therefore, we apply **`unversion`** `t` for t to collaborate with other versions. The right side of Fig. 7 shows a rewritten snippet of **`Main`**, where `sortVector vec` is replaced by **`unversion`** (sortVector vec). Assuming we avoid using programs that depend on a specific version elsewhere in the program, we can successfully compile and execute `main`.

6.2 Scalability of Constraint Resolution

We conducted experiments on the constraint resolution time of the VL compiler. In the experiment, we duplicated a VL module, renaming it to `#mod`

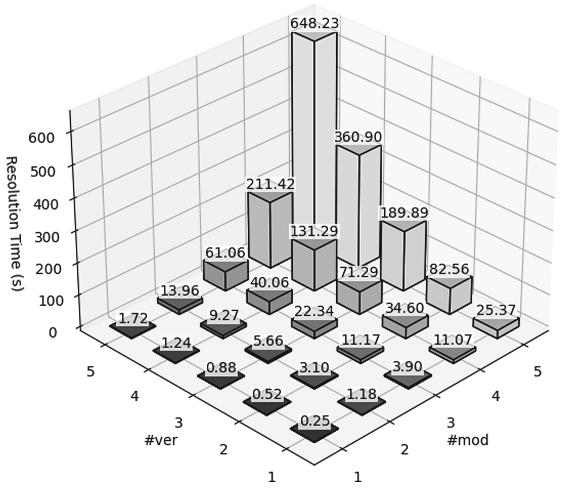

Fig. 8. Constraint resolution time for the duplicated `List` by `#mod` × `#ver`.

like `List_i`, and imported each module sequentially. Every module had the same number of versions, denoted as `#ver`. Each module version was implemented identically to `List`, with top-level symbols distinguished by the module name, such as `concat_List_i`. The experiments were performed ten times on a Ryzen 9 7950X running Ubuntu 22.04, with `#mod` and `#ver` ranging from 1 to 5.

Figure 8 shows the average constraint resolution time. The data suggests that the resolution time increases polynomially (at least square) for both `#mod` and `#ver`. Several issues in the current implementation contribute to this inefficiency: First, we employ sbv as a z3 interface, generating numerous redundant variables in the SMT-Lib2 script. For instance, in a code comprising 2600 LOC (with `#mod` = 5 and `#ver` = 5), the VL compiler produces 6090 version resource variables and the sbv library creates SMT-Lib2 scripts with approximately 210,000 intermediate symbolic variables. Second, z3 solves versions for all AST nodes, whereas the compiler's main focus should be on external variables and the subterms of `unversion`. Third, the current VL nests the constraint network, combined with $\vee$, `#mod` times at each bundling. This approach results in an overly complex constraint network for standard programs. Hence, to accelerate constraint solving, we can develop a more efficient constraint compiler for SMT-Lib2 scripts, implement preprocess to reduce constraints, and employ a greedy constraint resolution for each module.

7 Related Work, Future Work, and Conclusion

Managing Dependency Hell. Mainstream techniques for addressing dependency hell stand in stark contrast to our approach, which seeks to manage dependencies at a finer granularity. *Container* [17] encapsulates each application with all its dependencies in an isolated environment, a container, facilitating multiple library versions to coexist on one physical machine. However, it does not handle internal dependencies within the container. *Monorepository* [10,21] versions logically distinct libraries within a single repository, allowing updates across multiple libraries with one commit. It eases testing and bug finding but can lower the system modularity.

Toward a Language Considering Compatibility. The next step in this research is to embed compatibility tracking within the language system. The current VL considers different version labels incompatible unless a programmer uses `unversion`. Since many updates maintain backward compatibility and change only minor parts of the previous version, the existing type system is overly restrictive.

To illustrate, consider Fig. 3 again with more version history. The module `Hash` uses the MD5 algorithm for `mkHash` and `match` in the 1.x.x series. However, it adopts the SHA-3 algorithm in version 2.0.0, leaving other functions the same. The hash by `mkHash` version 1.0.1 (an MD5 hash) aligns with any MD5 hash from the 1.x.x series. Therefore, we know that comparing the hash using `match` version 1.0.0 is appropriate. However, the current VL compiler lacks mechanisms to express such compatibility in constraint resolution. The workaround involves using `unversion`, risking an MD5 hash's use with `match` version 2.0.0.

One promising approach to convey compatibilities is integrating semantic versioning [22] into the type system. If we introduce semantics into version labels, the hash generated in version 1.0.1 is backward compatible with version 1.0.0. Thus, by constructing a type system that respects explicitly defined version compatibilities, we can improve VL to accept a broader range of programs.

It is important to get reliable versions to achieve this goal. Lam et al. [14] emphasize the need for tool support to manage package modifications and the importance of analyzing compatibility through program analysis. *Delta-oriented programming* [24–26] could complement this approach by facilitating the way modularizing addition, overriding, and removal of programming elements and include application conditions for those modifications. This could result in a sophisticated package system that provides granular compatibility information.

Such a language could be an alternative to existing technologies for automatic update, collectively known as *adoptation*. These methods generate replacement rules based on structural similarities [5,33] and extract API replacement patterns from migrated code bases [27]. Some techniques involve library maintainers recording refactorings [7,12] and providing annotations [4] to describe how to update client code. However, the reported success rate of these techniques is less than 20% on average [6].

Supporting Type Incompatibility. One of the apparent problems with the current VL does not support *type incompatibilities*. VL forces terms of different versions to have the same type, both on the theoretical (typing rules in λ_{VL})

and implementation (bundling in VLMini) aspects. Supporting type incompatibility is important because type incompatibility is one of the top reasons for error-causing incompatibilities [23]. The current VL is designed in such a way because it retains the principle that equates the types of promotions and versioned records in λ_{VL}, easing the formalization of the semantics.

A promising approach to address this could be to decouple version inference from type inference and develop a version inference system on the polymorphic record calculus [19]. The idea stems from the fact that versioned types $\square_{\{l_1,l_2\}}A$ are structurally similar to record types $\{l_1 : A,\, l_2 : A\}$ of $\Lambda^{\forall,\bullet}$. Since $\Lambda^{\forall,\bullet}$ allows different record-element types for different labels and has concrete inference algorithms with polymorphism, implementing version inference on top of $\Lambda^{\forall,\bullet}$ would also make VL more expressive.

Adequate Version Polymorphism. In the current VL, there is an issue that the version label of top-level symbols in imported modules must be specified one, whereas users can select specific versions of external variables using **`unversion`** within the importing module. Consider using a generic function like `List.concat` in Fig. 7. If it is used in one part of the program within the context of **`Matrix`** version 1.0.0, the solution of the resource variable of `List.concat` version 1.0.0 becomes confined to $\{$**`Matrix`** $= 1.0.0,$ **`List`** $= \ldots\}$. As a result, it is impossible to utilize `List.concat` version 1.0.0 with **`Matrix`** version 2.0.0 elsewhere in the program. This problem becomes apparent when we define a generic module like a standard library.

It is necessary to introduce full-version polymorphism in the core calculus instead of duplication to address this problem. The idea is to generate a type scheme by solving constraints for each module during bundling and instantiate each type and resource variable at each occurrence of an external variable. Such resource polymorphism is similar to that already implemented in Gr [20]. However, unlike Gr, VLMini provides a type inference algorithm that collects constraints on a per-module basis, so we need the well-defined form of the principal type. This extension is future work.

Conclusion. This paper proposes a method for dependency analysis and version control at the expression level by incorporating versions into language semantics, which were previously only identifiers of packages. This enables the simultaneous use of multiple versions and identifies programs violating version consistency at the expression level, which is impossible with conventional languages.

Our next step is to extend the version label, which currently only identifies versions, to *semantic versions* and to treat the notion of compatibility with language semantics. Like automatic updates by modern build tools based on semantic versioning, it would be possible to achieve incremental updates, which would be done step-by-step at the expression level. Working with existing package managers to collect compatibility information at the expression level would be more feasible to realize the goal.

References

1. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK), vol. 13, p. 14 (2010)
2. Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., Panichella, S.: How the apache community upgrades dependencies: an evolutionary study. Empir. Softw. Eng. **20**(5), 1275–1317 (2015). https://doi.org/10.1007/s10664-014-9325-9
3. Brunel, A., Gaboardi, M., Mazza, D., Zdancewic, S.: A core quantitative coeffect calculus. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 351–370. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_19
4. Chow, Notkin: Semi-automatic update of applications in response to library changes. In: 1996 Proceedings of International Conference on Software Maintenance, pp. 359–368. IEEE, New York, USA (1996). https://doi.org/10.1109/ICSM.1996.565039
5. Cossette, B., Walker, R., Cottrell, R.: Using structural generalization to discover replacement functionality for API evolution (2014). https://doi.org/10.11575/PRISM/10182, https://prism.ucalgary.ca/handle/1880/49996
6. Cossette, B.E., Walker, R.J.: 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 2012, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2393596.2393661
7. Dig, D., Johnson, R.: How do APIs evolve? A story of refactoring. J. Softw. Maint. Evol. Res. Pract. **18**(2), 83–107 (2006). https://doi.org/10.1002/smr.328, https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.328
8. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. SIGPLAN Not. **48**(9), 429–442 (2013). https://doi.org/10.1145/2544174.2500582
9. Dunfield, J., Krishnaswami, N.R.: Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. In: Proceedings of ACM Programming Language, vol. 3(POPL) (2019). https://doi.org/10.1145/3290322
10. Durham Goode: Facebook Engineering: Scaling Mercurial at Facebook (2014). https://code.fb.com/core-data/scaling-mercurial-at-facebook/
11. Girard, J.Y.: Linear logic. Theor. Comput. Sci. **50**(1), 1–102 (1987). https://doi.org/10.1016/0304-3975(87)90045-4
12. Henkel, J., Diwan, A.: Catchup! Capturing and replaying refactorings to support API evolution. In: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005, pp. 274–283. IEEE, New York, USA (2005). https://doi.org/10.1109/ICSE.2005.1553570
13. Hughes, J., Vollmer, M., Orchard, D.: Deriving distributive laws for graded linear types. In: Dal Lago, U., de Paiva, V. (eds.) Proceedings Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Online, 29–30 June 2020. Electronic Proceedings in Theoretical Computer Science, vol. 353, pp. 109–131. Open Publishing Association (2021). https://doi.org/10.4204/EPTCS.353.6
14. Lam, P., Dietrich, J., Pearce, D.J.: Putting the Semantics into Semantic Versioning, pp. 157–179. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3426428.3426922

15. Lubis, L.A., Tanabe, Y., Aotani, T., Masuhara, H.: Batakjava: an object-oriented programming language with versions. In: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, pp. 222–234. SLE 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3567512.3567531
16. McDonnell, T., Ray, B., Kim, M.: An empirical study of API stability and adoption in the Android ecosystem. In: 2013 IEEE International Conference on Software Maintenance, ICSM, pp. 70–79. IEEE, New York, USA (2013). https://doi.org/10.1109/ICSM.2013.18
17. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux J. **239**, 2 (2014)
18. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer, Heidelberg (2008)
19. Ohori, A.: A polymorphic record calculus and its compilation. ACM Trans. Program. Lang. Syst. **17**(6), 844–895 (1995). https://doi.org/10.1145/218570.218572
20. Orchard, D., Liepelt, V.B., Eades, H., III.: Quantitative program reasoning with graded modal types. Proc. ACM Program. Lang. **3(ICFP)**, 1–30 (2019). https://doi.org/10.1145/3341714
21. Potvin, R., Levenberg, J.: Why google stores billions of lines of code in a single repository. Commun. ACM **59**(7), 78–87 (2016). https://doi.org/10.1145/2854146
22. Preston-Werner, T.: Semantic versioning 2.0.0 (2013). http://semver.org
23. Raemaekers, S., van Deursen, A., Visser, J.: Semantic versioning and impact of breaking changes in the maven repository. J. Syst. Softw. **129**, 140–158 (2017). https://doi.org/10.1016/j.jss.2016.04.008, http://www.sciencedirect.com/science/article/pii/S0164121216300243
24. Schaefer, I., Bettini, L., Damiani, F.: Compositional type-checking for delta-oriented programming. In: Proceedings of the Tenth International Conference on Aspect-Oriented Software Development, pp. 43–56. AOSD 2011, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1960275.1960283, https://doi.org/10.1145/1960275.1960283
25. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15579-6_6
26. Schaefer, I., Damiani, F.: Pure delta-oriented programming. In: Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, pp. 49–56. FOSD 2010, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1868688.1868696
27. Schäfer, T., Jonas, J., Mezini, M.: Mining framework usage changes from instantiation code. In: Proceedings of the 30th International Conference on Software Engineering, pp. 471–480. ICSE 2008, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1368088.1368153
28. Tanabe, Y., Aotani, T., Masuhara, H.: 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, pp. 8–14. COP 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3242921.3242923

29. Tanabe, Y., Lubis, L.A., Aotani, T., Masuhara, H.: A functional programming language with versions. Art, Sci. Eng. Programm. **6**(1), 5:1–5:30 (2021). https://doi.org/10.22152/programming-journal.org/2022/6/5, https://doi.org/10.22152%2Fprogramming-journal.org%2F2022%2F6%2F5
30. Tanabe, Y., Lubis, L.A., Aotani, T., Masuhara, H.: Compilation semantics for a programming language with versions (2023). https://doi.org/10.48550/arXiv.2310.00298
31. Tanabe, Y., Lubis, L.A., Aotani, T., Masuhara, H.: A step toward programming with versions in real-world functional languages. In: Proceedings of the 14th ACM International Workshop on Context-Oriented Programming and Advanced Modularity, pp. 44–51. COP 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3570353.3570359
32. Tolnay, D.: The semver trick (2017). https://github.com/dtolnay/semver-trick
33. Wu, W.: Modeling framework API evolution as a multi-objective optimization problem. In: 2011 IEEE 19th International Conference on Program Comprehension, pp. 262–265. IEEE, New York, USA (2011). https://doi.org/10.1109/ICPC.2011.43